LogicalFactChecker: Leveraging Logical Operations for Fact Checking with Graph Module Network

Wanjun Zhong[1]*, **Duyu Tang**[2], **Zhangyin Feng**[4]*, **Nan Duan**[2], **Ming Zhou**[2]
Ming Gong[3], **Linjun Shou**[3], **Daxin Jiang**[3], **Jiahai Wang**[1] **and Jian Yin**[1]
[1] The School of Data and Computer Science, Sun Yat-sen University.
Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou, P.R.China
[2] Microsoft Research [3] Microsoft Search Technology Center Asia [4] Harbin Institute of Technology
{zhongwj25@mail2,wangjiah@mail,issjyin@mail}.sysu.edu.cn
{dutang,nanduan,mingzhou,migon,lisho,djiang}@microsoft.com
zyfeng@ir.hit.edu.cn

Abstract

Verifying the correctness of a textual statement requires not only semantic reasoning about the meaning of words, but also symbolic reasoning about logical operations like *count*, *superlative*, *aggregation*, etc. In this work, we propose LogicalFactChecker, a neural network approach capable of leveraging logical operations for fact checking. It achieves the state-of-the-art performance on TABFACT, a large-scale, benchmark dataset built for verifying a textual statement with semi-structured tables. This is achieved by a graph module network built upon the Transformer-based architecture. With a textual statement and a table as the input, LogicalFactChecker automatically derives a program (a.k.a. logical form) of the statement in a semantic parsing manner. A heterogeneous graph is then constructed to capture not only the structures of the table and the program, but also the connections between inputs with different modalities. Such a graph reveals the related contexts of each word in the statement, the table and the program. The graph is used to obtain graph-enhanced contextual representations of words in Transformer-based architecture. After that, a program-driven module network is further introduced to exploit the hierarchical structure of the program, where semantic compositionality is dynamically modeled along the program structure with a set of function-specific modules. Ablation experiments suggest that both the heterogeneous graph and the module network are important to obtain strong results.

1 Introduction

Fact checking for textual statements has emerged as an essential research topic recently because of the unprecedented amount of false news and rumors spreading through the internet (Thorne et al., 2018; Chen et al., 2019; Goodrich et al., 2019; Nakamura et al., 2019; Kryściński et al., 2019; Vaibhav et al., 2019). Online misinformation may manipulate people's opinions and lead to significant influence on essential social events like political elections (Faris et al., 2017). In this work, we study fact checking, with the goal of automatically assessing the truthfulness of a textual statement.

Table

Year	Venue	Winner	Score
2005	Arlandastad	David Patrick	272
2004	Arlandastad	Matthew King	270
2003	Falsterbo	Titch Moore	273
2002	Halmstad	Thomas Besancenez	279

Statement In 2004, the score is less than 270.

Label REFUTED

Program $less(\ hop(\ filter\_eq(\ Year;\ 2004);Score);\ 270)$

Figure 1: An example of table-based fact checking. Given a statement and a table as the input, the task is to predict the label. Program reflects the underlying meaning of the statement, which should be considered for fact checking.

The majority of previous studies in fact checking mainly focused on making better use of the meaning of words, while rarely considered symbolic reasoning about logical operations (such as "*count*", "*superlative*", "*aggregation*"). However, modeling logical operations is an essential step towards the modeling of complex reasoning and semantic compositionality. Figure 1 shows a motivating example for table-based fact checking, where the evidence used for verifying the statement comes from a semi-structured table. We can see that correctly verifying the statement "*In 2004, the score is less than 270*" requires a system to not only discover the connections between tokens in the statement and the table, but more importantly understand the meaning of logical operations and how they interact in a structural way to form a whole. Under this

* Work done while this author was an intern at Microsoft Research.

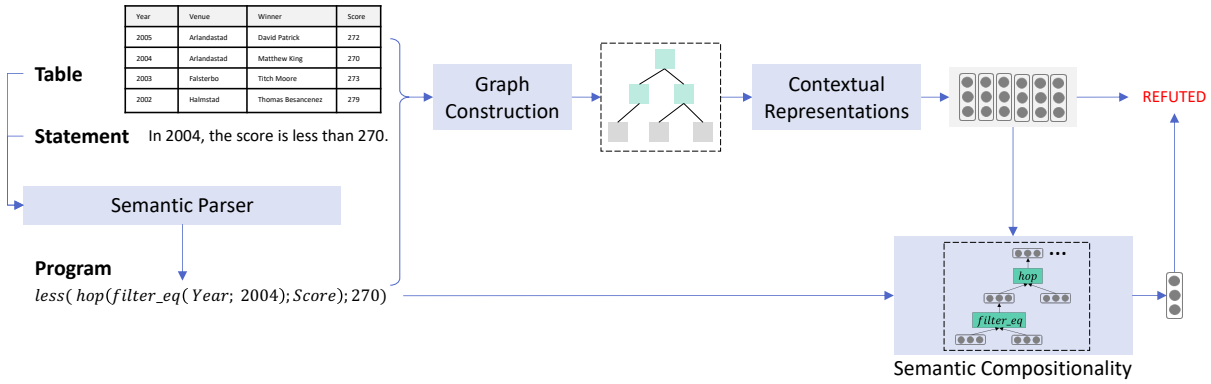

Figure 2: An overview of our approach LogicalFactChecker. It includes a semantic parser to generate program (§ 3.5), a graph construction mechanism (§ 3.2), a graph-based contextual representation learning for tokens (§ 3.3) and a semantic composition model over the program by neural module network (§ 3.4).

consideration, we use table-based fact checking as the testbed to investigate how to exploit logical operations in fact checking.

In this paper, We present LogicalFactChecker, a neural network approach that leverages logical operations for fact checking when semi-structured tables are given as evidence. Taking a statement and a table as the input, it first derives a program, also known as the logical form, in a semantic parsing manner (Liang, 2016). Then, our system builds a heterogeneous graph to capture the connections among the statement, the table and the program. Such connections reflect the related context of each token in the graph, which are used to define attention masks in a Transformer-based (Vaswani et al., 2017) framework. The attention masks are used to learn graph-enhanced contextual representations of tokens[1]. We further develop a program-guided neural module network to capture the structural and compositional semantics of the program for semantic compositionality. (Socher et al., 2013; Andreas et al., 2015). Graph nodes, whose representations are computed using the contextual representations of their constituents, are considered as arguments, and logical operations are considered as modules to recursively produce representations of higher level nodes along the program.

Experiments show that our system outperforms previous systems and achieves the state-of-the-art verification accuracy. The contributions of this paper can be summarized as follows:

- We propose LogicalFactChecker, a graph-based neural module network, which utilizes logical operations for fact-checking.
- Our system achieves the state-of-the-art performance on TABFACT, a large-scale and benchmark dataset for table-based fact checking.
- Experiments show that both the graph-enhanced contextual representation learning mechanism and the program-guided semantic compositionality learning mechanism improve the performance.

2 Task Definition

We study the task of table-based fact checking in this paper. This task is to assess the veracity of a statement when a table is given as evidence. Specifically, we evaluate our system on TABFACT (Chen et al., 2019), a large benchmark dataset for table-based fact checking. With a given semi-structured table and a statement, systems are required to perform reasoning about the structure and content of the table and assess whether the statement is "*ENTAILED*" or "*REFUTED*" by the table. The official evaluation metric is the accuracy for the two-way classification (*ENTAILED*/*REFUTED*). TABFACT consists of 118,439 statements and 16,621 tables from Wikipedia. More details about the dataset are given in Appendix A.

3 LogicalFactChecker: Methodology

In this section, we present our approach LogicalFactChecker, which simultaneously considers the meaning of words, inner structure of tables and programs, and logical operations for fact-checking. One way to leverage program information is to use standard semantic parsing methods, where automatically generated programs are directly executed on

[1]Here, tokens includes word pieces in the statement, table column names, table row names, table cells, and the program.

tables to get results. However, TABFACT does not provide annotated programs. This puts the problem in a weak-supervised learning setting, which is one of the major challenges in the semantic parsing field. In this work, we use programs in a soft way that programs are represented with neural modules to guide the reasoning process between a textual statement and a table.

Figure 2 gives an overview of our approach. With a statement and a corresponding table, our system begins with program generation, which synthesizes a program. Then, we build a heterogeneous graph for capturing the inner structure of the input. With the constructed graph, we incorporate a graph-based attention mask into the Transformer for learning graph-enhanced token representations. Lastly, we learn the semantic compositionality by developing a program-guided neural module network and make the final prediction.

This section is organized as follows. We first describe the format of the program (§ 3.1) for a more transparent illustration. After that, the graph construction approach (§ 3.2) is presented first, followed by a graph-enhanced contextual representation learning mechanism (§ 3.3). Moreover, we introduce how to learn semantic compositionality over the program by neural module network (§ 3.4). At last, we describe how to synthesize programs by our semantic parsing model (§3.5).

3.1 Program Representation

Before presenting the technical details, we first describe the form of the program (also known as logical form) for clearer illustrations.

With a given natural language statement, we begin by synthesizing the corresponding semantic representation (LISP-like program here) using semantic parsing techniques. Following the notation defined by Chen et al. (2019), the functions (logical operations) formulating the programs come from a fixed set of over 50 functions, including "*count*" and "*argmax*", etc. The detailed description of the functions is given in Appendix C. Each function takes arguments of predefined types like string, number, bool or sub-table as input. The programs have hierarchical structure because the functions can be nested. Figure 3 shows an example of a statement and a generated program, accompanying with the derivation of the program and its semantic structure. The details of the generation of a program for a textual statement are introduced in § 3.5.

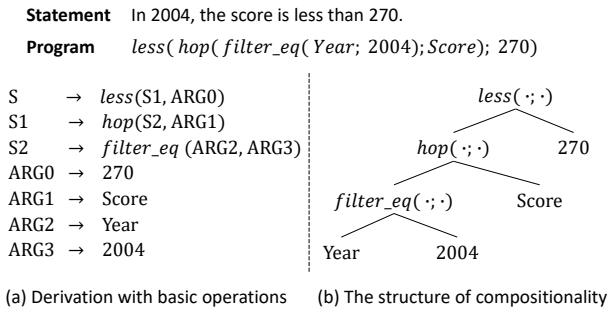

Figure 3: An example of a program with its semantic structure and derivation with basic logical operations.

3.2 Graph Construction

In this part, we introduce how to construct a graph to explicitly reveal the inner structure of programs and tables, and the connections among statements and them. Figure 4 shows an example of the graph. Specifically, with a statement, a table and a program, our system operates in the following steps.

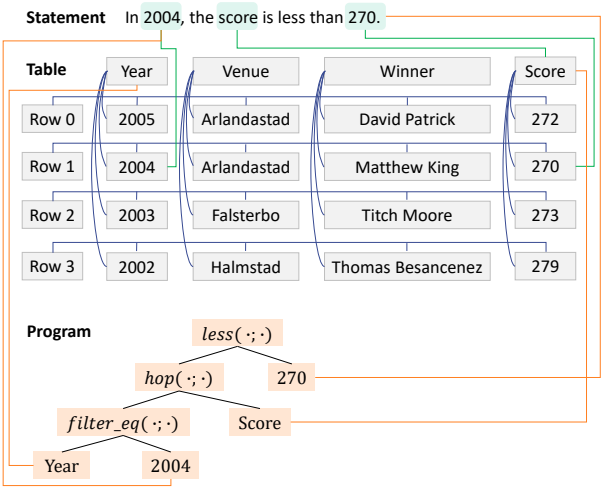

Figure 4: An example of the constructed graph.

- For a table, we define nodes as columns, cells, and rows, which is partly inspired by the design of the graph for table-based question answering (Müller et al., 2019). As shown in Figure 4, each cell is connected to its corresponding column node and row node. Cell nodes in the same row are fully-connected to each other.

- Program is a naturally structural representation consisting of functions and arguments. In the program, functions and arguments are represented as nodes, and they are hierarchically

connected along the structure. Each node is connected to its direct parents and children. Arguments are also linked to corresponding column names of the table.

- By default, in the statement, all tokens are the related context of each other, so they are connected. To further leverage the connections from the statement to the table and the program, we add links for tokens which are linked to cells or columns in the table, and legitimate arguments in the program.

After these processes, the extracted graph not only maintains the inner-structure of tables and programs but also explores the connections among aligned entities mentioned in different contents.

3.3 Graph-Enhanced Contextual Representations of Tokens

We describe how to utilize the graph structure for learning graph-enhanced contextual representations of tokens [2]. A simple way to learn contextual representations is to concatenate all the contents[3] as a single string and use the original attention mask in Transformer, where all the tokens are regarded as the contexts for each token. However, this simple way fails to capture the semantic structure revealed in the constructed graph. For example, according to Figure 4, the content "*2004*" exists in the statement, program and table. These aligned entity nodes for "*2004*" should be more related with each other when our model calculate contextual representations. To address this problem, we use the graph structure to re-define the related contexts of each token for learning a graph-enhanced representation.

Specifically, we present a graph-based mask matrix for self-attention mechanism in Transformer. The graph-based mask matrix G is a 0-1 matrix of the shape $N \times N$, where N denotes the total number of tokens in the sequence. This graph-based mask matrix records which tokens are the related context of the current token. G_{ij} is assigned as 1 if token j is the related context of token i in the graph and 0 otherwise.

Then, the constructed graph-based mask matrix will be feed into BERT (Devlin et al., 2018) for learning graph-enhanced contextual representations. We use the graph-based mask to control the contexts that each token can attend in the self-attention mechanism of BERT during the encoding process. BERT maps the input $\boldsymbol{x}$ of length T into a sequence of hidden vectors as follows.

$$h(\boldsymbol{x}) = [h(\boldsymbol{x})_1, h(\boldsymbol{x})_2, \cdots, h(\boldsymbol{x})_T] \quad (1)$$

These representations are enhanced by the structure of the constructed graph.

3.4 Semantic Compositionality with Neural Module Network

In the previous subsection, we describe how our system learns the graph-enhanced contextual representations of tokens. The process mentioned above learns the token-level semantic interaction. In this subsection, we make further improvement by learning logic-level semantics using program information. Our motivation is to utilize the structures and logical operations of programs for learning logic-enhanced compositional semantics. Since the logical operations forming the programs come from a fixed set of functions, we design a modular and composable network, where each logical operation is represented as a tailored module and modules are composed along the program structure.

We first describe how we initialize the representation for each entity node in the graph (§ 3.4.1). After that, we describe how to learn semantic compositionality based on the program, including the design of each neural module (§ 3.4.2) and how these modules are composed recursively along the structure of the program (§ 3.4.3).

3.4.1 Entity Node Representation

In a program, entity nodes denote a set of entities (such as "*David Patrick*") from input contexts while function nodes denote a set of logical operations (such as"*filter_equal*"), both of which may contain multiple words/word-pieces. Therefore, we take graph-enhanced contextual representations as mentioned in §3.3 to initialize the representations of entity nodes. Specifically, we initialize the representation h_e of each entity node e by averaging the projected hidden vectors of each words contained in e as follows:

$$h_e = \frac{1}{n}\sum_{i=0}^{n} relu(\boldsymbol{W_e} h(\boldsymbol{x})_{p_e^i}) \quad (2)$$

where n denotes the total number of tokens in the span of entity e, p_e^i denotes the position of the i^{th}

[2] In this work, tokens include word pieces in the statement, column names and row names and contents of cells in the table, and function names in the program

[3] All the contents indicate texts in the concatenated sequence of the linearized table, the statement, and the sequence of the linearized program.

token, $\boldsymbol{W_e} \in \boldsymbol{R}^{F \times D}$ is a weight matrix, F is the dimension of feature vectors of arguments, D is the dimension of hidden vectors of BERT and $relu$ is the activation function.

3.4.2 Modules

In this part, we present function-specific modules, which are used as the basic computational units for composing all the required configurations of module network structures.

Inspired by the neural module network (Andreas et al., 2015) and the recursive neural network (Socher et al., 2013), we implement each module with the same neural architecture but with different function-specific parameters. All the modules are trained jointly. Each module corresponds to a specific function, where the function comes from a fixed set of over 50 functions described before. In a program, each logical operation has the format of FUNCTION$(ARG0, ARG1, ...)$, where each function may have variable-length arguments. For example, the function hop has 2 arguments while the function $count$ has 1 argument. To handle variable-length arguments, we develop each module as follows. We first calculate the composition for each function-argument pair and then produce the overall representation via combining the representations of items.

The calculation for each function-argument pair is implemented as matrix-vector multiplication, where each function is represented as a matrix and each argument is represented as a vector. This is inspired by vector-based semantic composition (Mitchell and Lapata, 2010), which states that matrix-vector multiplication could be viewed as the matrix modifying the meaning of vector. Specifically, the output y_m of module m is computed with the following formula:

$$y_m = \frac{1}{N^m} \sum_{i=0}^{N^m} \sigma(\boldsymbol{W_m} v_i + b_m) \qquad (3)$$

where $\boldsymbol{W_m} \in \boldsymbol{R}^{F \times F}$ is a weight matrix and b_m is a bias vector for a specific module m. N^m denotes the number of arguments of module m, and each $v_i \in \boldsymbol{R}^F$ is the feature vector representing the i^{th} input. σ is the activation function.

Under the aforementioned settings, modules can compose into a hierarchical network determined by the semantic structure of the parsed program.

3.4.3 Program-Guided Semantic Compositionality

In this part, we introduce how to compose a program-guided neural module network based on the structure of programs and predefined modules. Taking the structure of the program and representations of all the entity nodes as the input, the composed neural module network learns the compositionality of the program for the final prediction. Figure 5 shows an example of a composed network based on the structure of the program.

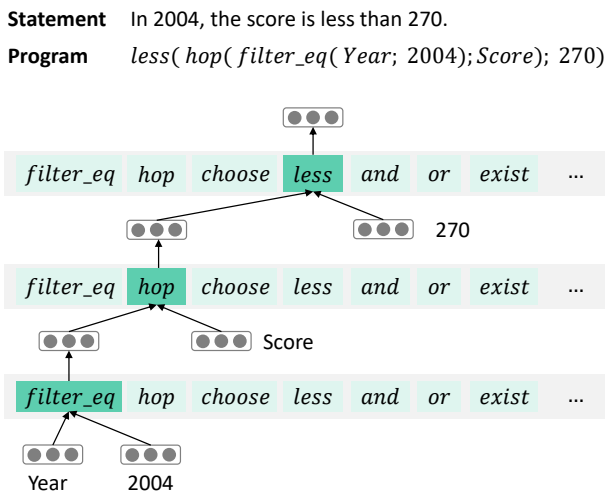

Figure 5: An example of neural module network.

Along the structure of the program, each step of compositionality learning is to select a module from a fixed set of parameterized modules defined in § 3.4.2 and operate on it with Equation 3 to dynamically generate a higher-level representation. The above process will be operated recursively until the output of the top-module is generated, which is denoted as y_m^{top}.

After that, we make the final prediction by feeding the combination of y_m^{top} and the final hidden vector $h(\boldsymbol{x})_T$ from § 3.3 through an MLP (Multilayer Perceptron) layer. The motivation of this operation is to retain the complete semantic meaning of the whole contexts because some linguistic cues are discarded during the synthesizing process of the program.

3.5 Program Generation

In this part, we describe our semantic parser for synthesizing a program for a textual statement. We tackle the semantic parsing problem in a weakly-supervised setting (Berant et al., 2013; Liang et al., 2017; Misra et al., 2018), since the ground-truth program is not provided.

Model	Val	Test	Test (simple)	Test (complex)	Small Test
Human Performance	-	-	-	-	92.1
Majority Guess	50.7	50.4	50.8	50.0	50.3
BERT classifier w/o Table	50.9	50.5	51.0	50.1	50.4
Table-BERT (Horizontal-S+T-Concatenate)	50.7	50.4	50.8	50.0	50.3
Table-BERT (Vertical-S+T-Template)	56.7	56.2	59.8	55.0	56.2
Table-BERT (Vertical-T+S-Template)	56.7	57.0	60.6	54.3	55.5
Table-BERT (Horizontal-S+T-Template)	66.0	65.1	79.0	58.1	67.9
Table-BERT (Horizontal-T+S-Template)	66.1	65.1	79.1	58.2	68.1
LPA-Voting w/o Discriminator	57.7	58.2	68.5	53.2	61.5
LPA-Weighted-Voting w/ Discriminator	62.5	63.1	74.6	57.3	66.8
LPA-Ranking w/ Discriminator	65.2	65.0	78.4	58.5	68.6
LogicalFactChecker (program from LPA)	71.7	71.6	**85.5**	64.8	74.2
LogicalFactChecker (program from Seq2Action)	**71.8**	**71.7**	85.4	**65.1**	**74.3**

Table 1: Performance on TABFACT in terms of label accuracy (%). The performances of Table-BERT and LPA are reported by Chen et al. (2019). Our system is abbreviated as LogicalFactChecker, with program generated via our Sequence-to-Action model and baseline (i.e. LPA), respectively. T, S indicate the table, the statement and + means the order of concatenation. In the linearization of tables, Horizontal (Vertical) refers to the horizontal (vertical) order for concatenating the cells. Concatenate (Template) means concatenating the cells directly (filling the cells into a template). In LPA settings, (Weighted) Voting means assigning each program with (score-weighted) equal weight to vote for the final result. Ranking means using the result generated by the top program ranked by the discriminator.

As shown in Figure 3, a program in TABFACT is structural and follows a grammar with over 50 functions. To effectively capture the structure of the program and also generate legitimate programs following a grammar in the generation process, we develop a sequence-to-action approach, which is proven to be effective in solving many semantic parsing problems (Chen et al., 2018; Iyer et al., 2018; Guo et al., 2018). The basic idea is that the generation of a program tree is equivalent to the generation of a sequence of action, which is a traversal of the program tree following a particular order, like depth-first, left-to-right order. Specifically, our semantic parser works in a top-down manner in a sequence-to-sequence paradigm. The generation of a program follows an ASDL grammar (Yin and Neubig, 2018), which is given in Appendix C. At each step in the generation phase, candidate tokens to be generated are only those legitimate according to the grammar. Parent feeding (Yin and Neubig, 2017) is used for directly passing information from parent actions. We further regard column names of the table as a part of the input (Zhong et al., 2017) to generate column names as program arguments.

We implement the approach with the LSTM-based recurrent network and Glove word vectors (Pennington et al., 2014) in this work, and the framework could be easily implemented with Transformer-based framework. Following Chen et al. (2019), we employ the label of veracity to guide the learning process of the semantic parser. We also employ programs produced by LPA (Latent Program Algorithm) for comparison, which is provided by Chen et al. (2019).

In the training process, we train the semantic parser and the claim verification model separately. The training of semantic parser includes two steps: candidate search and sequence-to-action learning. For candidate search, we closely follow LPA by first collecting a set of programs which could derive the correct label and then using the trigger words to reduce the number of spurious programs. For learning of the semantic parser, we use the standard way with back propagation, by treating each (claim, table, positive program) as a training instance.

4 Experiments

We evaluate our system on TABFACT (Chen et al., 2019), a benchmark dataset for table-based fact checking. Each instance in TABFACT consists of a statement, a semi-structured Wikipedia table and a label ("ENTAILED" or "REFUTED") indicates whether the statement is supported by the table or

not. The primary evaluation metric of TABFACT is label accuracy. The statistics of TABFACT are given in Appendix A. Detailed hyper-parameters for model training are given in Appendix B for better reproducibility of experiments.

We compare our system with following baselines, including the textual matching based baseline Table-BERT and semantic parsing based baseline LPA, both of which are developed by Chen et al. (2019).

- Table-BERT tackles the problem as a matching problem. It takes the linearized table and the statement as the input and employs BERT to predict a binary class.

- Latent Program Algorithm (LPA) formulates the verification problem as a weakly supervised semantic parsing problem. With a given statement, it operates in two step: (1) latent program search for searching executable program candidates and (2) transformer-based discriminator selection for selecting the most consistent program. The final prediction is made by executing the selected program.

4.1 Model Comparison

In Table 1, we compare our model (LogicalFactChecker) with baselines on the development set and test set. It is worth noting that complex test set and simple test set are partitioned based on its collecting channel, where the former involves higher-order logic and more complex semantic understanding. As shown in Table 1, our model with programs generated by Sequence-to-Action model, significantly outperforms previous systems with 71.8% label accuracy on the development set and 71.7% on the test set, and achieves the state-of-the-art performance on the TABFACT dataset.

4.2 Ablation Study

We conduct ablation studies to evaluate the effectiveness of different components in our model.

Model	Label Acc. (%)	
	Val	Test
LogicalFactChecker	71.83	71.69
-w/o Graph Mask	70.06	70.13
-w/o Compositionality	69.62	69.61

Table 2: Ablation studies on the development set and the test set.

As shown in Table 2, we evaluate LogicalFactChecker under following settings: (1) removing the graph-based mask described in § 3.3 (the first row); (2) removing the program-guided compositionality learning mechanism described in § 3.4 (the second row).

Table 2 shows that, eliminating the graph-based mask drops the accuracy by 1.56% on test set. Removing the program-guided compositionality learning mechanism drops the accuracy by 2.08% on test set, which reflects that the neural module network plays a more important role in our approach. This observation verifies that both mechanisms are beneficial for our task.

4.3 Case Study

We conduct a case study by giving an example shown in Figure 6. From the example, we can see that our system synthesizes a semantic-consistent program of the given statement and make the correct prediction utilizing the synthesized program. This observation reflects that our system has the ability to (1) find a mapping from the textual cues to a complex function (such as the mapping from "*most points*" to function "*argmax*") and (2) derive the structure of logical operations to represent the semantic meaning of the whole statement.

Table

Position	Pilot	Country	Points
1	Sebastian Kawa	Poland	69
2	Carlos Rocca	Chile	55
3	Mario Kiessling	Germany	47
4	Uli Schwenk	Germany	40

Statement The country with the most points is Poland.

Program $eq(\ Poland; hop(\ argmax(\ Points); Country))$

Predict ENTAILED

Figure 6: A case study of our approach.

4.4 Error Analysis

We randomly select 400 instances and summarize the major types of errors, which can be considered as future directions for further study.

The **dominant** type of errors is caused by the misleading programs generated by the semantic parser. As shown in the example in Figure 7 (a), the semantic parser fails to generate a semantically correct program because it lacks the external knowledge about the date in the table and the "new year eve" in the statement. The **second** type of errors is caused by semantic compositionality, even though

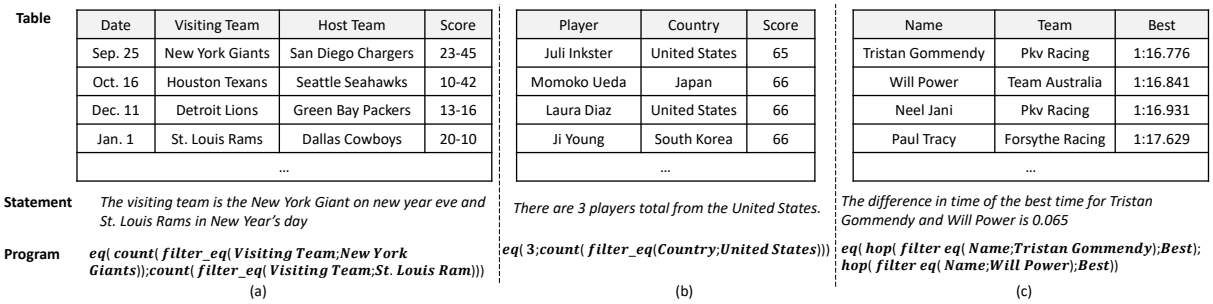

Figure 7: Examples of error types, including (a) predicting a wrong program because of the lack of background knowledge, (b) predicting a correct program but predicting a wrong label, and (c) that the logical operations required to understand the statement is not covered in the grammar.

programs are correctly predicted. As shown in Figure 7 (b), the program involves operations requiring complex reasoning, like counting the exact number of rows. Potential ways to alleviate this problem is to design more function-specific modules like Andreas et al. (2015). The **third** type of errors is caused by the coverage of the logical operations we used. In this work, we follow Chen et al. (2019) and use exactly the same functions. However, as shown in 7 (c), understanding this statement requires the function of *difference_time*, which is not covered by the current set.

5 Related Work

There is a growing interest in fact checking in NLP with the rising importance of assessing the truthfulness of texts, especially when pre-trained language models (Radford et al., 2019; Zellers et al., 2019; Keskar et al., 2019) are more and more powerful in generating fluent and coherent texts. Previous studies in the field of fact checking differ in the genres of supporting evidence used for verification, including natural language (Thorne et al., 2018), semi-structured tables (Chen et al., 2019), and images (Zlatkova et al., 2019; Nakamura et al., 2019).

The majority of previous works deal with textual evidence. FEVER (Thorne et al., 2018) is one of the most influential datasets in this direction, where evidence sentences come from 5.4 million Wikipedia documents. Systems developed on FEVER are dominated by pipelined approaches with three separately trained models, i.e. document retrieval, evidence sentence selection, and claim verification. There also exist approaches (Yin and Roth, 2018) that attempt to jointly learn evidence selection and claim verification. More recently, the second FEVER challenge (Thorne et al., 2019) is built for studying adversarial attacks in fact checking[4]. Our work also relates to fake news detection. For example, Rashkin et al. (2017) study fact checking by considering stylistic lexicons, and Wang (2017) builds LIAR dataset with six fine-grained labels and further uses meta-data features. There is a fake news detection challenge[5] hosted in WSDM 2019, with the goal of the measuring the truthfulness of a new article against a collection of existing fake news articles before being published. There are very recent works on assessing the factual accuracy of the generated summary in neural abstractive summarization systems (Goodrich et al., 2019; Kryściński et al., 2019), as well as the use of this factual accuracy as a reward to improve abstractive summarization (Zhang et al., 2019).

Chen et al. (2019) recently release TABFACT, a large dataset for table-based fact checking. Along with releasing the great dataset, they provide two baselines: Table-BERT and LPA. Table-BERT is a textual matching based approach, which takes the linearized table and statement as inputs and states the veracity. However, Table-BERT fails to utilize logical operations. LPA is a semantic parsing based approach, which first synthesizes programs by latent program search and then ranks candidate programs with a neural-based discriminator. However, the ranking step in LPA does not consider the table information. Our approach simultaneously utilizes the logical operations for semantic compositionality and the connections among tables, programs, and statements. Results show that our approach achieves the state-of-the-art performance on TABFACT.

[4] http://fever.ai/

[5] https://www.kaggle.com/c/fake-news-pair-classification-challenge/

6 Conclusion

In this paper, we present LogicalFactChecker, a neural network based approach that considers logical operations for fact checking. We evaluate our system on TABFACT, a large-scale benchmark dataset for verifying textual statements over semi-structured tables, and demonstrate that our approach achieves the state-of-the-art performance. LogicalFactChecker has a sequence-to-action semantic parser for generating programs, and builds a heterogeneous graph to capture the connections among statements, tables, and programs. We utilize the graph information with two mechanisms, including a mechanism to learn graph-enhanced contextual representations of tokens with graph-based attention mask matrix, and a neural module network which learns semantic compositionality in a bottom-up manner with a fixed set of modules. We find that both graph-based mechanisms are beneficial to the performance, and our sequence-to-action semantic parser is capable of generating semantic-consistent programs.

Acknowledge

Wanjun Zhong, Jiahai Wang and Jian Yin are supported by the National Natural Science Foundation of China (U1711262, U1611264,U1711261,U1811261,U1811264, U1911203), National Key R&D Program of China (2018YFB1004404), Guangdong Basic and Applied Basic Research Foundation (2019B1515130001), Key R&D Program of Guangdong Province (2018B010107005). The corresponding author is Jian Yin.

References

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. 2015. Deep compositional question answering with neural module networks. *ArXiv*, abs/1511.02799.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544.

Bo Chen, Le Sun, and Xianpei Han. 2018. Sequence-to-action: End-to-end semantic graph generation for semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777, Melbourne, Australia. Association for Computational Linguistics.

Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2019. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Robert Faris, Hal Roberts, Bruce Etling, Nikki Bourassa, Ethan Zuckerman, and Yochai Benkler. 2017. Partisanship, propaganda, and disinformation: Online media and the 2016 us presidential election.

Ben Goodrich, Vinay Rao, Peter J Liu, and Mohammad Saleh. 2019. Assessing the factual accuracy of generated text. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 166–175. ACM.

Daya Guo, Duyu Tang, Nan Duan, Ming Zhou, and Jian Yin. 2018. Dialog-to-action: conversational question answering over a large-scale knowledge base. In *Advances in Neural Information Processing Systems*, pages 2942–2951.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. 2019. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*.

Wojciech Kryściński, Bryan McCann, Caiming Xiong, and Richard Socher. 2019. Evaluating the factual consistency of abstractive text summarization. *arXiv preprint arXiv:1910.12840*.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. 2017. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *ACL*.

Percy Liang. 2016. Learning executable semantic parsers for natural language understanding. *arXiv preprint arXiv:1603.06677*.

Dipendra Misra, Ming-Wei Chang ad Xiaodong He, and Wen tau Yih. 2018. Policy shaping and generalized update equations for semantic parsing from denotations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brusses, Belgium. Association for Computational Linguistics.

Jeff Mitchell and Mirella Lapata. 2010. Composition in distributional models of semantics. *Cognitive science*, 34(8):1388–1429.

Thomas Müller, Francesco Piccinno, Massimo Nicosia, Peter Shaw, and Yasemin Altun. 2019. Answering conversational questions on structured data without logical forms. *arXiv preprint arXiv:1908.11787*.

Kai Nakamura, Sharon Levy, and William Yang Wang. 2019. r/fakeddit: A new multimodal benchmark dataset for fine-grained fake news detection. *arXiv preprint arXiv:1911.03854*.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8).

Hannah Rashkin, Eunsol Choi, Jin Yea Jang, Svitlana Volkova, and Yejin Choi. 2017. Truth of varying shades: Analyzing language in fake news and political fact-checking. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2931–2937.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 455–465.

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. 2018. Fever: a large-scale dataset for fact extraction and verification. *arXiv preprint arXiv:1803.05355*.

James Thorne, Andreas Vlachos, Oana Cocarascu, Christos Christodoulopoulos, and Arpit Mittal. 2019. The FEVER2.0 shared task. In *Proceedings of the Second Workshop on Fact Extraction and VERification (FEVER)*, pages 1–6, Hong Kong, China. Association for Computational Linguistics.

Vaibhav Vaibhav, Raghuram Mandyam Annasamy, and Eduard Hovy. 2019. Do sentence interactions matter? leveraging sentence level representations for fake news classification. *arXiv preprint arXiv:1910.12203*.

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

William Yang Wang. 2017. " liar, liar pants on fire": A new benchmark dataset for fake news detection. *arXiv preprint arXiv:1705.00648*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720*.

Wenpeng Yin and Dan Roth. 2018. Twowingos: A two-wing optimization strategy for evidential claim verification. *arXiv preprint arXiv:1808.03465*.

Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. 2019. Defending against neural fake news. *arXiv preprint arXiv:1905.12616*.

Yuhao Zhang, Derek Merck, Emily Bao Tsai, Christopher D Manning, and Curtis P Langlotz. 2019. Optimizing the factual correctness of a summary: A study of summarizing radiology reports. *arXiv preprint arXiv:1911.02541*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

Dimitrina Zlatkova, Preslav Nakov, and Ivan Koychev. 2019. Fact-checking meets fauxtography: Verifying claims about images. *arXiv preprint arXiv:1908.11722*.

A Statistic of TABFACT

Split	#Sentence	Table	Avg. Row	Avg. Col
Train	92,283	13,182	14.1	5.5
Val	12,792	1,696	14.0	5.4
Test	12,779	1,695	14.2	5.4

Table 3: Basic statistics of Train/Val/Test split in the dataset

B Training Details

In this part, we describe the training details of our experiments. As described before, the semantic parser and statement verification model are trained separately.

We first introduce the training process of the semantic parser. Both training and validation datasets are created in a same way as described in § 3.5. Specifically, each pair of data is labeled as true or false. Finally, the training dataset contains 495,131 data pairs, and the validation dataset contains 73,792 data pairs. We implement the approach with the LSTM-based recurrent network and use the following set of hyper parameters to train models: hidden size is 256, learning rate is 0.001, learning rate decay is 0.5, dropout is 0.3, batch size is 150. We use glove embedding to

initialize embedding and use Adam to update the parameters. We use beam search during inference and set beam size as 15. We use BLEU to select the best checkpoint by validation scores.

Then we introduce the training details of statement verification model. We employ cross-entropy loss as the loss function. We apply AdamW as the optimizer for model training. In order to directly compare with Table-BERT, we also employ BERT-Base as the backbone of our approach. The BERT network and neural module network are trained jointly. We set learning rate as 1e-5, batch size as 8 and set max sequence length as 512. The training time for one epoch is 1.2 hours by 4 P40 GPUs. We set the dimension of entity node representation as 200.

C ASDL-Grammar

In this part, we introduce the ASDL grammar (Yin and Neubig, 2018) we apply for synthesizing the programs in Seq2Action model. The definition of functions mainly follows Chen et al. (2019). Details can be found in following two pages.[6]

[6]The function "*filter_eq*" contains three arguments (subtable, column_name, value), but we ignore the first argument in the running example for a clearer illustration.

Composite Type	Constructor	Fields
OutBool	Bool	pr_bool bool
	none	OutStr str
	only	OutRow row
	zero	OutNum num
	after	OutRow row1, OutRow row2
	before	OutRow row1, OutRow row2
	first	OutRow row1, OutRow row2
	second	OutRow row1, OutRow row2
	third	OutRow row1, OutRow row2
	fourth	OutRow row1, OutRow row2
	fifth	OutRow row1, OutRow row2
	last	OutRow row1, OutRow row2
	greater	OutNum num1, OutNum num2
	less	OutNum num1, OutNum num2
	eq	OutStr str1, OutStr str2
	not_eq	OutStr str1, OutStr str2
	and	OutBool bool1, OutBool bool2
	within	OutRow row, pr_header header, OutStr str
	not_within	OutRow row, pr_header header, OutStr str
	all_eq	OutRow row, pr_header header, OutStr str
	all_not_eq	OutRow row, pr_header header, OutStr str
	all_less	OutRow row, pr_header header, OutNum num
	all_less_eq	OutRow row, pr_header header, OutNum num
	all_greater	OutRow row, pr_header header, OutNum num
	all_greater_eq	OutRow row, pr_header header, OutNum num
OutRow	Row	pr_row row
	top	OutRow row
	bottom	OutRow row
	argmax	OutRow row, pr_header header
	argmin	OutRow row, pr_header header
	filter_eq	OutRow row, pr_header header, OutStr str
	filter_not_eq	OutRow row, pr_header header, OutStr str
	filter_less	OutRow row, pr_header header, OutNum num
	filter_greater	OutRow row, pr_header header, OutNum num
	filter_greater_eq	OutRow row, pr_header header, OutNum num
	filter_less_eq	OutRow row, pr_header header, OutNum num

OutNum	Num	pr_number num
	count	OutRow row
	half	OutRow row
	one_third	OutRow row
	inc_num	OutNum num
	uniq	OutRow row, pr_header header
	avg	OutRow row, pr_header header
	sum	OutRow row, pr_header header
	max	OutRow row, pr_header header
	min	OutRow row, pr_header header
	diff	OutNum num1, OutNum num2
	add	OutNum num1, OutNum num2
OutStr	Str	pr_str str
	hop	OutRow row, pr_header header
	most_freq	OutRow row, pr_header header
OutNone	dec_num	OutNum num