Verifying Text Summaries of Relational Data Sets

Saehan Jo
Cornell University
sj683@cornell.edu

Immanuel Trummer
Cornell University
itrummer@cornell.edu

Weicheng Yu
Cornell University
wy248@cornell.edu

Xuezhi Wang
Google Research
xuezhiw@google.com

Cong Yu
Google Research
congyu@google.com

Daniel Liu
Cornell University
dl596@cornell.edu

Niyati Mehta
Cornell University
nbm44@cornell.edu

ABSTRACT

We present a novel natural language query interface, the AggChecker, aimed at text summaries of relational data sets. The tool focuses on natural language claims that translate into an SQL query and a claimed query result. Similar in spirit to a spell checker, the AggChecker marks up text passages that seem to be inconsistent with the actual data. At the heart of the system is a probabilistic model that reasons about the input document in a holistic fashion. Based on claim keywords and the document structure, it maps each text claim to a probability distribution over associated query translations. By efficiently executing tens to hundreds of thousands of candidate translations for a typical input document, the system maps text claims to correctness probabilities. This process becomes practical via a specialized processing backend, avoiding redundant work via query merging and result caching. Verification is an interactive process in which users are shown tentative results, enabling them to take corrective actions if necessary. We tested our system on 53 publicly available articles containing 392 claims. Our tool revealed erroneous claims in roughly a third of test cases. Also, AggChecker compares favorably against several automated and semi-automated fact checking baselines.

ACM Reference Format:
Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Verifying Text Summaries of Relational Data Sets. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300074

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300074

1 INTRODUCTION

We present a tool for verifying text summaries of relational data sets. Our tool resembles a spell checker and marks up claims that are believed to be erroneous. We focus on natural language claims that can be translated into an SQL query and a claimed query result. More precisely, we focus on claims that are translated into aggregation queries on data subsets. Hence the name of our system: AggChecker. Our analysis shows that this claim type is at the same time very common and error-prone. The following example illustrates the concept.

In contrast to prior work [22], our focus is not mostly on adversarial fact checking. We also want to support text authors (e.g., data journalists or scientists) or third persons, collaborating with authors (e.g, a lector or reviewer), in creating accurate data summaries. The motivation for using our tool is similar to the motivation for using a spell checker in those cases. However, publishing erroneous numbers can in some cases have more serious consequences than spelling mistakes (e.g., corrections or even retractions, erroneous numbers might even have legal consequences if they appear in business reports). Hence, the need for specialized verification tools. Note that some of our assumptions, e.g. having access to the data set associated with text, are motivated by this focus (even though we also report results on determining suitable data sets via our tool later in this paper).

Example 1. Consider the passage *"There were only four previous lifetime bans in my database - three were for repeated substance abuse"* taken from a 538 newspaper article [13]. It contains two claims that translate into the SQL queries SELECT Count(*) FROM nflsuspensions WHERE Games = 'indef' (with claimed result 'four') and SELECT Count(*) FROM nflsuspensions WHERE Games = 'indef' AND Category = 'substance abuse, repeated offense' (with claimed result 'three') on the associated data set. Our goal is to automatically translate text to queries, to evaluate

Table 1: AggChecker: primary design choices and underlying motivation.

Component	Design Choice	Goal
Text Analysis	Keyword-based	High-recall heuristic
Processing Engine	Batch-Optimized	High-throughput verification
User Interface	Interactive	Integrate user feedback
Claim Checker	Probabilistic Model	Leverage heterogeneous feedback
	Expectation Maximization Learning	Exploit Semantic Correlation

those queries, and to compare the evaluation result against the claimed one.

Internally, the system executes the following, simplified process to verify a claim. First, it tries to translate the natural language claim into an SQL query reflecting its semantics. Second, it executes the corresponding query on the relational database. Third, it compares the query result against the value claimed in text. If the query result rounds to the text value then the claim has been verified. Otherwise, the claim is considered erroneous. Color markup indicates the verification result to users. Additionally, users may obtain information on the verification process and can take corrective actions if necessary (similar to how users correct erroneous spell checker markup).

The most challenging step is of course the translation of a natural language claim into an SQL query. Among the challenges we encountered when studying real-world test cases are the following. First, the claim sentence itself is often missing required context. This context can only be found when analyzing preceding paragraphs or headlines (assuming a hierarchical input text document). Second, claim sentences often contain multiple claims which make it hard to associate sentence parts to claims. Third, claim sentences are often long and contain parts which do not immediately correspond to elements in the associated query. This makes it hard to map the claim sentence parse tree to a similar SQL query tree. Fourth, data sets often contain entries (e.g., abbreviations) that are not found immediately in the claim text. Altogether, this makes it hard to map claim sentences unambiguously to database elements.

The high-level design of AggChecker is motivated by those challenges. We acknowledge that translating text claims to queries, based on text analysis alone, is inherently unreliable. Hence, we seek to exploit additional signals to reduce our uncertainty in translation. First, as opposed to natural language querying, we are not only given a query but also a claimed, numerical result. The probability that the result of a random query matches a claimed number is typically low. Also, correct claims are in practice more likely than incorrect claims for the type of claim we are considering (as evaluated in more detail later). This makes query candidates whose result matches the claimed value more likely as claim translations. Of course, to exploit this signal, we first need to execute the candidate query. Second, as we analyze in more detail later, claims in the same text document are often similar. Authors tend to use the same aggregation functions, and similar aggregates and predicates, in their claims throughout a document. Hence, if we can translate a few claims with high confidence, it can help us to translate the others. Third, if all else fails, it might be necessary to get help from users. The goal is of course to limit user intervention to the most difficult cases (similar to a spell checker, which needs corrective action only very occasionally). Also, we want to make the most out of the user's time by "transferring" feedback we obtain for one claim to others.

AggChecker is designed to exploit various signals in claim to query translation. Table 1 gives an overview of its components, the primary design choices made in each component, and the motivation for doing so. In order to exploit additional signals in translation, we first need to obtain a space of possible query translations for each given claim. As ranking and filtering steps follow, this space can be large but it needs to contain the correct query to enable a successful translation. Hence, we use a high-recall (but low precision) heuristic for analyzing the input text and matching claim text to database entries. This heuristic is based on keyword matching and the input document structure (we exploit document structure to associate claims with keywords from other document parts). As discussed before, a first signal comes from executing query candidates and comparing their results to claimed values. To enable us to verify the large query candidate space, resulting from the first stage, we use an execution engine that is tailored for executing batches of similar queries with high throughput. To enable users to take corrective actions if necessary, AggChecker features an interactive user interface that exploits partial verification results to minimize overheads for users (e.g., by showing likely translations for single-click feedback).

AggChecker exploits heterogeneous features for claim to query translation. We need to integrate feedback from different sources in a principled manner to come to a tentative verification result. We use a probabilistic model to do so, integrating feedback from text analysis, query executions, user feedback, and (as discussed next), semantic

correlations between claims. We assume (and experimentally verify later) that documents typically have a common theme, which can be represented by an a-priori probability distribution over query fragments (e.g., aggregation functions, specific columns in the data set, etc.). Knowing the document theme helps in query translations and knowing (some) query translations helps in inferring the document theme. This circular dependency motivates an iterative expectation-maximization approach, in which we infer document theme and likely translations at once, thereby exploiting semantic correlations between claims.

Our contributions lie in the high-level design of the system (which is novel and tailored to our specific scenario), as well as in the design of each single component.

We evaluated our system on a variety of real-world test cases, containing 392 claims on relational data sets. Our test cases cover diverse topics and derive from various sources, reaching from Wikipedia to New York Times articles. We generated ground truth claim translations by hand and contacted the article authors in case of ambiguities. We identified a non-negligible number of erroneous claims, many of which are detected by our system. We compare against baseline systems and perform a user study. The user study demonstrates that users verify documents significantly faster via AggChecker than via standard query interfaces.

In summary, our original scientific contributions are the following:

- We introduce the problem of translating natural language claims on relational data to SQL queries, without using prior training or manual annotations.
- We propose a first corresponding system whose design is tailored to the particularities of our scenario.
- We compare our system against baselines in fully automated checking as well as in a user study.

The remainder of this paper is organized as follows. We formalize our problem model in Section 2. Next, we give an overview of our system in Section 3. The following three sections describe specific components of our system: keyword matching, probabilistic reasoning, and massive-scale candidate query evaluations. After that, we present experimental results in Section 7. Finally, we compare against related work in fact-checking and natural language query interfaces in Section 8. In the appendix, we provide more experimental results and list all our test cases.

2 PROBLEM STATEMENT

We introduce our problem model and related terminology. We generally assume a scenario where we have a relational *Database* together with a natural language *Text* summarizing it. The relational database might be optionally associated with a data dictionary (mapping database elements such as columns and values to text descriptions). The text document may be semi-structure, i.e. it is organized as a hierarchy of sections and subsections with associated headlines. Also, the text contains *Claims* about the database.

Definition 1. A **Claim** is a word sequence from the input text stating that evaluating a query q on the associated database D yields a rounded result e. We focus on SQL queries with numerical results ($e \in \mathbb{R}$). We call q also the *Matching Query* or *Ground Truth Query* with regards to the claim. A claim may be a sentence part or a sentence (one sentence may contain multiple claims). A claim is *Correct* if there is an admissible rounding function $\rho : \mathbb{R} \rightarrow \mathbb{R}$ such that the rounded query results equals the claimed value (i.e., $\rho(q(D)) = e$).

We currently consider rounding to any number of significant digits as admissible. The approach presented in the next sections can be used with different rounding functions as well. We focus on *Simple Aggregate Queries*, a class of claim queries defined as below.

Definition 2. A **Simple Aggregate Query** is an SQL query of the form SELECT Fct(Agg) FROM T1 E-JOIN T2 ... WHERE C1 = V1 AND C2=V2 AND ..., calculating an aggregate over an equi-join between tables connected via primary key-foreign key constraints. The where clause is a conjunction of unary equality predicates.

Claims of this format are very popular in practice and at the same time error-prone (see Section 7). Currently, we support the following aggregation functions: Count, Count Distinct, Sum, Average, Min, Max, Percentage, and Conditional Probability[1] (we plan to gradually extend the scope). The ultimate goal would be to perform purely *Automatic Aggregate-Checking* (i.e., given a text document and a database, identify claims automatically and decide for each one whether it is correct). This would however require near-perfect natural language understanding which is currently still out of reach. Hence, in this paper, we aim for *Semi-Automatic Aggregate-Checking* in which we help users to verify claims without taking them out of the loop completely.

Definition 3. Given input $\langle T, D \rangle$, a text T and a database D, the goal of **Semi-Automatic Aggregate-Checking** is to identify claims and to map each claim c to a probability distribution Q_c over matching queries. This probability distribution can be exploited by a corresponding user interface to quickly verify text in interaction with the user. The quality of a corresponding approach can be measured based on how

[1]For conditional probability, we assume that the first predicate is the condition and the rest form the event. That is, (SELECT ConditionalProbability(Agg) FROM T1 E-JOIN T2 ... WHERE C1 = V1 AND C2=V2 AND ...) = (SELECT Count(Agg) FROM T1 E-JOIN T2 ... WHERE C1 = V1 AND C2=V2 AND ...) * 100 / (SELECT Count(Agg) FROM T1 E-JOIN T2 ... WHERE C1 = V1).

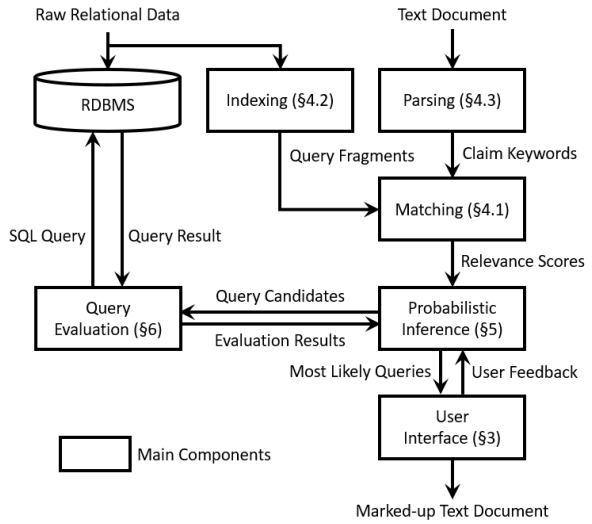


Figure 1: Overview of AggChecker system.

often the top-x likely query candidates in Q_c contain the matching query.

3 SYSTEM OVERVIEW

Figure 1 shows an overview of the AggChecker system. The input to the AggChecker consists of two parts: a relational data set and a text document, optionally enriched with HTML markup highlighting the text structure. The text contains claims about the data. Our goal is to translate natural language claims into pairs of SQL queries and claimed query results. The process is semi-automated and relies on user feedback to resolve ambiguities. Finally, we enrich the input text with visual markup, identifying claims that are inconsistent with the data.

For each newly uploaded data set, we first identify relevant query fragments (see Figure 2(c)). The system focuses on *Simple Aggregate Queries* as defined in Section 2. Query fragments include aggregation functions, aggregation columns, or unary equality predicates that refer to columns and values in the data set. We associate each query fragment with keywords, using names of identifiers within the query fragment as well as related keywords that we identify using WordNet [11, 35]. We index query fragments and the associated keywords via an information retrieval engine (we currently use Apache Lucene [17]).

Next, we parse the input text using natural language analysis tools such as the Stanford parser [33]. We identify potentially check-worthy text passages via simple heuristics and rely on user feedback to prune spurious matches. Then, we associate each claim with a set of related keywords (see Figure 2(d)). We use dependency parse trees as well as the document structure to weight those keywords according to their relevance. We query the information retrieval engine, indexing query fragments, using claim keywords as queries. Thereby we obtain a ranked set of query fragments for each claim.

Query fragments with relevance scores form one out of several inputs to a probabilistic model. This model maps each text claim to a probability distribution over SQL query candidates, representing our uncertainty about how to translate the claim (see Figure 2(e)). The model considers the document structure and assumes that claims in the same document are linked by a common theme. The document theme is represented via model parameters capturing the prior probabilities of certain query properties. We infer document parameters and claim distributions in an iterative expectation-maximization approach. Furthermore, we try to resolve ambiguities in natural language understanding via massive-scale evaluations of query candidates. The AggChecker uses evaluation strategies such as query merging and caching to make this approach practical (we currently use Postgres [19] to evaluate merged queries). We typically evaluate several tens of thousands of query candidates to verify one newspaper article.

Example 2. Figure 2 provides a concrete running example demonstrating the inputs and outputs of the main components. Figure 2(a) depicts the raw relational data where query fragments and their associated keywords are extracted as in Figure 2(c). Figure 2(b) illustrates a text passage from a 538 newspaper article [13]. It contains three claimed results (colored in blue) where we focus on the claimed result 'one' in this example. In Figure 2(d), we extract relevant keywords for this claimed result and weigh them based on the text structure. Then, we calculate relevance scores for pairs of query fragments and claims based on their keywords. The probabilistic model takes into account the relevance scores as well as two other inputs to infer the probability distribution over query candidates. Figure 2(e) captures this concept. First, 'Keyword Probability' is derived from the relevance scores. Second, 'Prior Probability' encapsulates the model parameters that embrace all claims in the text document in a holistic fashion. Third, green or red color under 'Evaluation Result' shows whether the query result matches the value claimed in text. Lastly, 'Refined Probability' illustrates the final probability distribution over query candidates, considering all three inputs. After expectation-maximization iterations have converged, the system verifies the claim according to the query with the highest probability. We provide more detailed explanations in each section where all following examples refer to this figure.

After an automated verification stage, the system shows tentative verification results to the user. Claims are colored based on their probability of being erroneous (see Figure 3(a)).

name	games	category	...
A. Karras	14	personal conduct	
A. Jones	20	personal conduct	
L. Brazill	indef	substance abuse, repeated offense	
G. Miller	4	PEDs	
...			...

(a) Raw relational data.

There were only four previous lifetime bans in my database — three were for repeated substance abuse, one was for gambling.
...

(b) Text document. Claims colored in blue. We focus on claimed result 'one'.

Fragment	Keywords
Count(*)	total, number, ...
Sum(Games)	sum, total, game, ...
⋮	
Category = 'PEDs'	category, PED, drug, ...
Games = 'indef'	game, Indef, indefinite, ...
⋮	

(c) Query fragments.

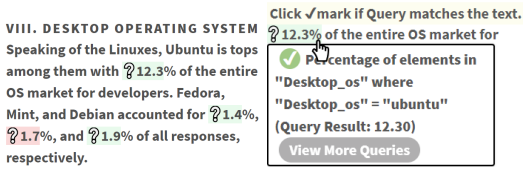
Keyword	Source	Weight
gambling	Claim Sentence	1.0
abuse	Claim Sentence	0.5
⋮		
lifetime	Previous Sentence	0.1
bans	Previous Sentence	0.1
⋮		

(d) Keywords and their weights (specific to claimed result 'one').

Query Candidates	Prior Probability	Keyword Probability	Evaluation Result	Refined Probability
SELECT Count(*) WHERE Games = 'indef' AND Category = 'gambling'	0.057	0.008	1	**0.197**
SELECT Count(*) WHERE Category = 'gambling'	0.137	0.012	5	0.059
⋮				
SELECT Percentage(Category) WHERE Category = 'gambling'	0.011	0.012	1.86	0.006
SELECT Sum(Games) WHERE Category = 'substance abuse'	0.011	0.006	78398	0.002
⋮				

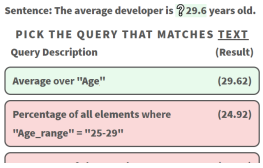
(e) Probability distribution and evaluation results of query candidates. Green indicates a match between query result and claimed result while red indicates the opposite (specific to claimed result 'one').

Figure 2: Running example of AggChecker system.

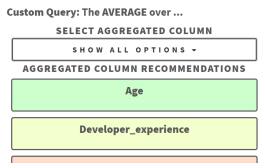


(a) Marked up claims after initial processing.

(b) Query description shown upon hovering.



(c) Select query from top-5 likely candidates.



(d) Construct query by selecting fragments.

Figure 3: Screenshots from AggChecker interface.

Users can hover over a claim to see a natural language description of the most likely query translation (see Figure 3(b)) and may correct the system if necessary. Alternatively, users may pick among the top-k most likely query candidates (see Figure 3(c)) or assemble the query from query fragments with high probability (see Figure 3(d)).

4 KEYWORD MATCHING

In the first processing phase, we extract query fragments and claim keywords from the inputs and match them together to calculate relevance scores.

4.1 Keyword Matching Overview

We calculate relevance scores for pairs of claims and query fragments. The higher the relevance score, the more likely the fragment to be part of the query matching the claim. We consider aggregation functions, aggregation columns, and predicate parts as query fragments. Given an input database, we can infer all potentially relevant query fragments (i.e., we introduce an equality predicate fragment for each literal in the database, an aggregation column fragment for each column containing numerical values etc.). Furthermore, we can associate query fragments with relevant keywords (e.g., the name of a literal, as well as the name of the containing column and synonyms for a fragment representing an equality predicate).

On the other side, we can associate each claim in the input text with relevant keywords, based on the document structure. Having query fragments and claims both associated with keyword sets, we can use methods from the area of information retrieval to calculate relevance scores for specific pairs of query fragments and claims. For instance, we use Apache Lucene in our current implementation, indexing keyword sets for query fragments and querying with claim-specific keyword sets. While keyword-based relevance scores are inherently imprecise, they will form one out of several input signals for the probabilistic model described in the next section. The latter model will associate each claim with a probability distribution over query candidates.

4.2 Indexing Query Fragments

When loading a new database, we first form all potentially relevant query fragments. Function IndexFragments (we describe its implementation without providing pseudo-code) traverses the database in order to form query fragments that could be part of a claim query. We consider three types of query fragments: aggregation functions, aggregation columns, and equality predicates. All aggregation function specified in the SQL standard are potentially relevant (we could easily add domain-specific aggregation functions). We consider all

```
1: // Extract keywords for claim c from text T.
2: function ClaimKeywords(c, T)
3:     // Initialize weighted keywords
4:     K ← ∅
5:     // Add keywords in same sentence
6:     for word ∈ c.sentence do
7:         weight ← 1/TreeDistance(word, c)
8:         K ← K ∪ {⟨word, weight⟩}
9:     end for
10:    // Add keywords of sentences in same paragraph
11:    m ← min{1/TreeDistance(k, c)|k ∈ c.sentence}
12:    K ← K ∪ {⟨k, 0.4m⟩|k ∈ c.prevSentence}
13:    K ← K ∪ {⟨k, 0.4m⟩|k ∈ c.paragraph.firstSentence}
14:    // Add keywords in preceding headlines
15:    s ← c.containingSection
16:    while s ≠ null do
17:        K ← K ∪ {⟨k, 0.7m⟩|k ∈ s.headline.words}
18:        s ← s.containingSection
19:    end while
20:    return K
21: end function
```

Algorithm 1: Extracts a set of keywords for a claim.

numerical columns in any table of the database as aggregation columns (in addition, we consider the "all column" * as argument for count aggregates). Finally, we consider all equality predicates of the form $c = v$ where c is a column and v a value that appears in it.

We associate each query fragment with a set of relevant keywords. Keyword sets are indexed via an information retrieval engine (together with a reference to the corresponding fragment). We associate each standard SQL aggregation function with a fixed keyword set. The keywords for aggregation columns are derived from the column name and the name of its table. Column names are often concatenations of multiple words and abbreviations. We therefore decompose column names into all possible substrings and compare against a dictionary. Furthermore, we use WordNet to associate each keyword that appears in a column name with its synonyms. The keywords for an equality predicate of the form $c = v$ are derived from the column name c (and from the name of the containing table) as well as from the name of value v. Finally, the AggChecker also offers a parser for common data dictionary formats. A data dictionary associates database columns with additional explanations. If a data dictionary is provided, we add for each column the data dictionary description to its associated keywords.

4.3 Extracting Keywords from Text

Next, we associate each claim in the input text with a weighted set of keywords. More precisely, we iterate over each number in the input text that is likely to represent a claimed query

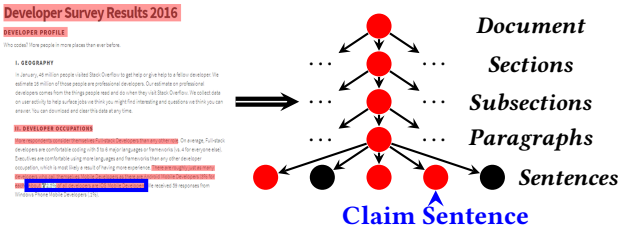


Figure 4: Keyword context of claim sentence: keyword sources in red, the claim sentence in blue.

result. We describe in Section 3 how they are identified. Algorithm 1 associates each such claim with weighted keywords, extracted from the containing text. First, we consider keywords in the claim sentence itself (i.e., the sentence in which the claimed result number is found). One sentence might contain multiple claims and we must decide what keywords are most relevant to one specific claim. For that, we construct a dependency parse tree of the claim sentence. We make the simplifying assumption that sentence parts are more closely related, the lower their distance (i.e., number of tree edges to traverse) is in the parse tree. Hence, for each numerical aggregate representing the result of a claim, we weight the surrounding keywords based on their distance from the numerical aggregate in the dependency tree (denoted by TreeDistance in Algorithm 1).

Considering keywords in the same sentence is often insufficient. In practice, relevant context is often spread over the entire text. We exploit the structure of the text document in order to collect potentially relevant keywords. Our current implementation uses HTML markup but the document structure could be easily derived from the output format of any word processor. We assume that the document is structured hierarchically into sections, sub-sections etc. For a given claim sentence, we "walk up" that hierarchy and add keywords in all headlines we encounter. In addition, we add keywords from the first and preceding sentences in the same paragraph. Figure 4 illustrates keyword sources for an example claim.

Example 3. To provide a concrete example, we refer to the paragraph in Figure 2(b). The second sentence contains two claimed results ('three' and 'one') that translate into queries of the form: SELECT Count(*) FROM T WHERE Games = 'indef' AND Category = V. We illustrate two difficulties associated with these claims.

First, there are two claims in one sentence. The system needs to distinguish keywords that are more relevant to each claim. Let's consider the keyword 'gambling'. According to the dependency parse tree of the second sentence, the distance from 'three' to 'gambling' is two while the distance from 'one' to 'gambling' is one. Then, we assign weights

by taking the reciprocal of the distance (see Figure 2(d)). This helps the system to understand that 'gambling' is more related to 'one' than 'three'.

Second, no keyword in the second sentence explicitly refers to the restriction Games = 'indef'. Rather, it can be implicitly inferred from the context where only the first sentence has the keywords 'lifetime bans'. Thereby, considering the keyword context of a claim sentence enables us to identify important and relevant keywords from other parts of the text. In Section 7, we conduct an experiment to measure the effect of keyword context (see Figure 8).

4.4 Constructing Likely Query Candidates

Having associated both, query fragments and claims, with keywords, we can map claims to likely query candidates. We indexed keyword sets associated with query fragments in an information retrieval engine. For a given claim, we use the associated keyword set to query that information retrieval engine. The returned results correspond to query fragments that are associated with similar keywords as the claim. Furthermore, each returned query fragment is associated with a relevance score, capturing how similar its keywords are to the claim-related keywords. Combining all returned query fragments in all possible ways (within the boundaries of the query model described in Section 2) yields the space of claim-specific query candidates. Each candidate is characterized by a single aggregation function fragment, applied to an aggregation column fragment, in the SQL select clause. In addition, each candidate is characterized by a set of unary equality predicates that we connect via a conjunction in the SQL where clause. The SQL from clause can be easily inferred from the other query components: it contains all tables containing any of the columns referred to in aggregates or predicates. We connect those tables via equi-joins along foreign-key-primary-key join paths.

5 PROBABILISTIC MODEL

We map each natural language claim to a probability distribution over matching SQL queries. Based on the most likely query for each claim, we can decide which claims are likely to be wrong and focus the user's attention on those.

5.1 Probabilistic Model Overview

Our probabilistic model is based on a fundamental property of typical text documents (we quantify this effect in Appendix A): text summaries tend to have a primary focus. The claims made in a text are not independent from each other but typically connected via a common theme. If we find out the common theme, mapping natural language claims to queries becomes significantly easier.

```
// Calculate for each claim in C a distribution over
// matching queries on database D using relevance
// scores S via expectation maximization.
function QueryAndLearn(D, C, S)
    // Initialize priors describing text document
    Θ ←Uniform
    // Iterate EM until convergence
    while Θ not converged yet do
        // Treat each factual claim
        for c ∈ C do
            // Calculate keyword-based probability
            Q_c ←TextProbability(S, Θ)
        end for
        // Refine probability via query evaluations
        Q ←RefineByEval({Q_c|c ∈ C}, C, D)
        // Update document-specific priors
        Θ ←Maximization({Q_c|c ∈ C})
    end while
    return {Q_c|c ∈ C}
end function
```

Algorithm 2: Learn document-specific probability distribution over queries and refine by evaluating query candidates.

We represent the common theme as a document-specific probability distribution over queries. We use that distribution as a prior when inferring likely queries for each claim. Beyond the prior distribution, the likelihood of queries depends on the keyword-based relevance scores that are associated with each claim (we described how those relevance scores can be calculated in the last section).

We face a circular dependency: if we had the document theme, we could use it as prior in our search for the most likely query for each claim. On the other side, if we had the most likely query for each claim, we could infer the document theme. This motivates an expectation-maximization approach [9] in which model parameters (describing here the document-specific query distribution) and values of latent variables (describing claim-specific query distributions) are iteratively refined, using tentative values for one of the two to infer estimates for the other.

Algorithm 2 describes how the AggChecker infers probability distributions over query candidates. Starting from a uniform document distribution (captured by parameter Θ whose precise components are described in the following), the algorithm iterates until convergence. In each iteration, a claim-specific probability distribution over query candidates is calculated for each claim, based on the relevance scores provided as input and the model parameters.

Strong evidence that a query candidate matches a natural language claim can be obtained by evaluating the query and comparing its result to the claimed one. However, evaluating queries on potentially large data sets may lead to significant

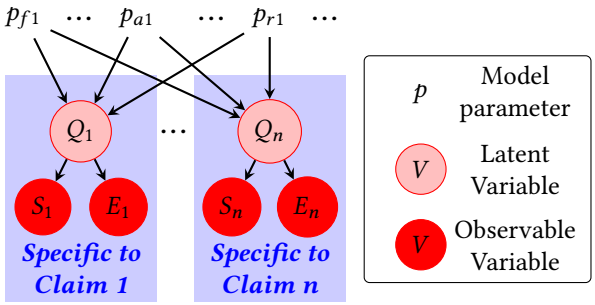


Figure 5: Simplified probabilistic model for query inference: parameters describe prior probabilities of query characteristics, claim queries (Q_c) are latent while relevance scores (S_c) and evaluation results (E_c) are observable.

processing overheads. Myriads of queries are possible for a given data set and we cannot execute all of them. This is why we use a preliminary, claim-specific query distribution to select promising query candidates for execution. In the next section, we describe efficient processing strategies enabling us to execute hundreds of thousands of query candidates during learning. The evaluation of promising candidates is encapsulated in Function RefineByEval in Algorithm 2. The result is a refined probability distribution over query candidates for a given claim that takes evaluation results into account. Finally, the model parameters are updated based on the claim-specific distributions. In the following subsections, we provide details on our probabilistic model. A simplified version of the model is illustrated in Figure 5.

5.2 Prior Probabilities

We need to keep that model relatively simple for the following reason: having more parameters to learn typically requires a higher number of iterations until convergence. In our case, each iteration requires expensive data processing and hence we cannot afford an elevated number of steps. We therefore introduce only parameters that describe the probability of certain coarse-grained query features:

$$\Theta = \langle p_{f1}, p_{f2}, \ldots, p_{a1}, p_{a2}, \ldots, p_{r1}, p_{r2}, \ldots \rangle \quad (1)$$

Here, p_{fi} is the prior probability of selecting the i-th aggregation function (such as an average or a sum), p_{ai} is the probability of selecting the i-th numerical column to aggregate, and p_{ri} is the probability that a restriction (i.e., equality predicate) is placed on the i-th column. We can only have a single aggregation function and a single column to aggregate over, the corresponding parameters must therefore sum up to one. This does not apply to the parameters describing the likelihood of a restriction as a query might restrict multiple columns at the same time. During the maximization step of Algorithm 2 (line 17), we simply set each component of Θ to the ratio of maximum likelihood queries with the corresponding property, scaled to the total number of claims in the document. For instance, for updating r_i, we divide the number of maximum likelihood queries (summing over all claims) placing a restriction on the i-th column by the number of claims.

Table 2: Changing priors during EM iterations until convergence (for the example in Figure 2).

Query Fragment	Initial Prior	...	Final Prior
Count(*)	0.025	...	0.150
Sum(Games)	0.025	...	0.012
...			
Games = (any value)	0.143	...	0.417
Category = (any value)	0.143	...	0.297
...			

Example 4. Table 2 shows the convergence of priors Θ for the example in Figure 2. Note that all three claims in this example have ground truth queries of the form: SELECT Count(*) FROM T WHERE Games = 'indef' (AND Category = V). In Table 2, priors successfully reflect this pattern after several EM iterations. For instance, the final priors imply the fact that a query with Count(*) is more likely to be the correct one among query candidates. An experimental result in Section 7 demonstrates that the system truly benefits from the prior probabilities (see Table 4).

5.3 Claim-specific Probability Distribution

Next, we show how to calculate claim-specific probability distributions over queries, assuming given values for the parameters above. We introduce random variable Q_c to model the query described in claim c. Hence, $\Pr(Q_c = q)$ is the probability that the text for claim c describes a specific query q. Variable Q_c depends on another random variable, S_c, modeling relevance scores for each query fragment. Those relevance scores are generated by an information retrieval engine, as discussed in the last section, based on keywords in claim text. If a query fragment has high relevance scores (i.e., many related keywords appear in the claim), the probability of Q_c for queries containing that fragment increases. Also, if a query evaluates to the claimed result, the probability that the claim describes this query should intuitively increase. We model by E_c evaluation results for a set of promising query candidates and Q_c depends on E_c. According to the Bayes rule, we obtain:

$$\Pr(Q_c|S_c, E_c) \propto \Pr(S_c \wedge E_c|Q_c) \cdot \Pr(Q_c) \quad (2)$$

$\Pr(Q_c)$ denotes prior query probabilities, derived from a document-specific theme. Assuming independence between relevance scores and evaluation results, we obtain:

$$\Pr(S_c \wedge E_c|Q_c) = \Pr(S_c|Q_c) \cdot \Pr(E_c|Q_c) \quad (3)$$

We assume independence between different query characteristics. This is a simplification (as certain aggregation functions might often be used with certain aggregation columns for instance), but modeling dependencies would require additional parameters and our prior remark about model complexity applies. Via independence assumptions, we obtain:

$$\Pr(S_c|Q_c) = \Pr(S_c^F|Q_c) \cdot \Pr(S_c^A|Q_c) \cdot \Pr(S_c^R|Q_c) \quad (4)$$

Variable S_c^F represents the relevance scores assigned to each aggregation function by the information retrieval engine, based on keywords surrounding claim c. By S_c^A, we denote relevance scores for aggregation columns, and by S_c^R scores for query fragments representing restrictions (i.e., equality predicates). The probability $\Pr(S_c^A|Q_c)$ is for instance the probability that we obtain relevance scores S_c^A, assuming that the text author describes query Q_c in claim c. Obtaining a high relevance score for a query fragment (be it aggregation function, column, or predicate) means that related keywords appear prominently in the claim text. Hence, query fragments that are part of the claim query should tend to receive higher relevance scores than the others.

We use a simple model that complies with the latter intuition: the probability to receive certain relevance scores is proportional to the relevance scores of the fragments appearing in the claim query. E.g., assume that claim query q aggregates over column $a \in A$ (where A designates the set of all candidate columns for aggregates in the database). We denote by $S_c^A(a)$ the relevance score for the query fragment representing that specific column a. We set $\Pr(S_c^A|Q_c = q) = S_c^A(a)/\sum_{a \in A} S_c^A(a)$, scaling relevance scores to the sum of relevance scores over all query fragments in the same category (i.e., aggregation columns in this case).

Correct claims are more likely than incorrect claims in typical text documents. Hence, if a query candidate evaluates to the claimed value, it is more likely to be the query matching the surrounding text. It is typically not feasible to evaluate all candidate queries on a database. Hence, we restrict ourselves to evaluating queries that have high probabilities based on relevance scores alone (line 14 in Algorithm 2). Let E_c be the evaluation results of promising queries for claim c. We set $\Pr(E_c|Q_c = q) = p_T$ if the evaluations E_c map query q to a result that rounds to the claimed value. We set $\Pr(E_c|Q_c = q) = 1 - p_T$ otherwise. Parameter p_T is hence the assumed probability of encountering true claims in a document. Different settings realize different tradeoffs between precision and recall (see Section 7).

We finally calculate prior query probabilities, assuming again independence between different query characteristics:

$$\Pr(Q_c) = \Pr(Q_c^F) \cdot \Pr(Q_c^A) \cdot \prod_{r \in R} \Pr(Q_c^r) \quad (5)$$

Prior probabilities of specific aggregation functions, aggregation columns, and restrictions ($\Pr(Q_c^F)$, $\Pr(Q_c^A)$, and $\Pr(Q_c^r)$) follow immediately from the model parameters Θ. In summary, let q be an SQL query with aggregation function f_q and aggregation column a_q, containing equality predicates restricting the i-th column to value $V_q(i)$ (with $V_q(i) = *$ if no restriction is placed on the i-th column). The probability $\Pr(Q_c = q)$ that query q is described in claim c is *proportional* to the product of the following factors: the prior probability of q appearing in the current document (i.e., $p_{f_q} \cdot p_{f_a} \cdot \prod_{i:V_q(i) \neq *} p_{r_i}$), the likelihood to receive the observed keyword-based relevance scores for q's query fragments (i.e., $S_c(f_q) \cdot S_c(a_q) \cdot \prod_i S_c(r_i = V_q(i))$ where $S_c()$ generally maps fragments to their scores for claim c), and p_T if the rounded query result matches the claim value (or $1 - p_T$ otherwise). We can use this formula to map each claim to a maximum likelihood query (line 12 in Algorithm 2). Note that it is not necessary to scale relevance scores since the scaling factor is constant for each specific claim. We only compare query candidates that have been selected for evaluation based on their keyword and prior-based probabilities alone (line 15 in Algorithm 2). Before providing further details on query evaluation, we present the following example to elaborate on the benefit of RefineByEval at line 15 in Algorithm 2.

Example 5. Let's again use the example with claimed result 'one' introduced in Figure 2. Without near-perfect natural language understanding, it is almost impossible to map the phrase *"lifetime bans"* to query fragment Games = 'indef'. Nevertheless, the learned priors during EM iterations will at least tell us that a restriction is usually placed on column Games (11 out of 13 claims in this article [13] have ground truth queries with restriction on Games). By evaluating many related query candidates, the system can find out that SELECT Count(*) FROM nflsuspensions WHERE Games = 'indef' AND Category = 'gambling' yields the same result as claimed in text (and largely no other queries do). Since we boost the probability of queries that evaluate to the claimed value, this query gets a higher refined probability (see Figure 2(e)). It is noteworthy to mention that this is possible only when the system has learned the correct priors reflecting the common theme of ground truth queries. Nevertheless, articles typically have some easy cases (i.e., claims with distinctive keywords nearby) where the system can correctly translate into queries. Then, the system can also cope with other more difficult cases as the information gained from easy cases spreads across claims through EM iterations. Thus, the system benefits from easy cases and successfully learns the correct priors. In summary, the overall effect of RefineByEval on our test cases is presented in Section 7 (see Table 4).

```
1: // Calculate aggregates S_FA on data D for row sets
2: // defined by predicates on columns G with non-zero
3: // probability according to query distribution Q.
4: function Cube(Q, D, G, S_FA)
5:     // Collect directly referenced tables
6:     T ← {Table(x)|⟨f, x⟩ ∈ S_FA ∨ x ∈ D}
7:     // Add tables and predicates on join paths
8:     T ← T ∪ {JoinPathTables(t1, t2)|t1, t2 ∈ T}
9:     J ← {JoinPathPreds(t1, t2)|t1, t2 ∈ T}
10:    // Collect relevant literals for any claim
11:    L ← {Literals(r)|r ∈ D ∧ ∃c ∈ C : Pr(l|Q_c) > 0}
12:    return ExecuteQuery(D, "
           SELECT S_FA, D FROM
               ( SELECT S_FA,InOrDefault(P, L)
               FROM T WHERE J ) CUBE BY P ")
13: end function
14: // Refine probabilities Q for claims C on data D.
15: procedure RefineByEval(Q, C, D)
16:    // Evaluate likely queries for each claim
17:    for c ∈ C do
18:        // Pick scope for query evaluations
19:        ⟨S_F, S_A, S_R⟩ ←PickScope(Q, C, D)
20:        // Initialize query evaluation results
21:        E ← ∅
22:        // Iterate over predicate column group
23:        for G ⊆ S_R : |G| = n_G(|S_R|) do
24:            // Form likely aggregates
25:            S_FA ← F × A
26:            // Exploit cache content
27:            for fa ∈ S_FA|IsCached(fa, G) do
28:                E ← E∪CacheGet(fa, G)
29:                S_FA ← S_FA \ {fa}
30:            end for
31:            // Generate missing results
32:            E ← E∪Cube(Q, D, G, S_FA)
33:            // Store in cache for reuse
34:            CachePut(S_FA, G, E)
35:        end for
36:        // Refine query probabilities
37:        Q_c ←Refine(Q_c, E)
38:    end for
39:    return Q
40: end procedure
```

Algorithm 3: Refine query probabilities by evaluations.

6 QUERY EVALUATION

In fact-checking, we can partially resolve ambiguities in natural language understanding by evaluating large numbers of query candidates. As we show in Section 7, the ability to process large numbers of query candidates efficiently turns out to be crucial to make fact-checking practical.

The expectation maximization approach presented in the previous section relies on a sub-function, RefineByEval, to refine query probabilities by evaluating candidates. Algorithm 3 implements that function. The input is an unrefined probability distribution over query candidates per natural language claim, as well as the claim set and the database they refer to. The output is a refined probability distribution, taking into account query evaluation results.

6.1 Prioritizing Query Evaluations

Ideally, we would be able to evaluate all query candidates with non-zero probability based on the initial estimates. By matching the results of those queries to results claimed in text, we would be able to gain the maximal amount of information. In practice, we must select a subset of query candidates to evaluate. To make the choice, we consider two factors. First, we consider the a-priori likelihood of the query to match the claim. Second, we consider the processing efficiency of evaluating many queries together.

Queries are characterized by three primary components in our model: an aggregation function, an aggregation column, and a set of predicates. In a first step (line 19 in Algorithm 3), we determine the evaluation scope as the set of alternatives that we consider for each query characteristic. To determine the scope, we use a cost model that takes into account the size of the database as well as the number of claims to verify. Function PickScope exploits the marginal probabilities of query characteristics. It expands the scope, prioritizing more likely alternatives, until estimated evaluation cost according to our cost model reaches a threshold.

6.2 Merging Query Candidates

As a naive approach, we could form all query candidates within the scope and evaluate them separately. This would however neglect several opportunities to save computation time by avoiding redundant work. First, alternative query candidates for the same claim tend to be quite similar. This means that the associated query plans can often share intermediate results, thereby amortizing computational overheads. Second, even the query candidates for different claims in the same document tend to be similar, thereby enabling us to amortize cost again. This relates to our observation (quantified in Section 7) that different claims in the same document are often semantically related. Finally, Algorithm 3 will be called repeatedly for the same document (over different iterations of the expectation maximization approach). In particular in later iterations, topic priors change slowly and likely query candidates for the same claim change only occasionally between iterations. This enables us to reuse results from previous iterations. Algorithm 3 exploits all three opportunities to avoid redundant work.

At the lowest layer, we use cube queries to efficiently calculate aggregates for different data subsets. Each data subset

is associated with one specific combination of query predicates. One cube query can therefore cover many alternative query candidates as long as their equality predicates refer to a small set of columns (the cube dimensions). Executing a cube query on a base table yields aggregates for each possible value combination in the cube dimension. The result set can be large if cube dimensions contain many distinct values. In the context of fact-checking, we are typically only interested in a small subset of values (the ones with non-zero marginal probabilities, meaning that corresponding matches are returned after keyword matching). We can however not filter the data to rows containing those values before applying the cube operator: this would prevent us from obtaining results for query candidates that do not place any predicate on at least one of the cube dimensions. Instead, we apply the cube operator to the result of a sub-query that replaces all literals with zero marginal probability by a default value (function InOrDefault). This reduces the result set size while still allowing us to evaluate all related query candidates. Note that evaluating multiple aggregates for the same cube dimensions in the same query is typically more efficient than evaluating one cube query for each aggregate. Hence, we merge as many aggregates as possible for the same dimensions into the same query.

6.3 Caching across Claims and Iterations

Furthermore, we avoid redundant computation by the use of a cache. The cache is accessed via functions IsCached, CacheGet, and CachePut with the obvious semantics. The cache persists across multiple iterations of the main loop in Algorithm 3 and across multiple invocations of Algorithm 3 for the same document (during expectation maximization). We avoid query evaluations if results are available in the cache and cache each generated result. We can choose the granularity at which results are indexed by the cache. Indexing results at a coarse granularity might force us to retrieve large amount of irrelevant data. Indexing results at a very fine granularity might create overheads when querying the cache and merging result parts. We found the following granularity to yield a good performance tradeoff: we index (partial) cube query results by a combination of one aggregation column, one aggregation function, and a set of cube dimensions. The index key does not integrate the set of relevant literals in the cube columns, although the set of literals with non-zero marginal probability may vary across different claims or iterations. We generate results for all literals that are assigned to a non-zero probability for any claim in the document (this set is equivalent to the set of literals in predicates returned during keyword matching). This simplifies indexing and is motivated by the observation that different claims tend to have high overlap in the sets of relevant literals for a given column.

To create more opportunities to share partial results, we cover the query scope via multiple cube queries, iterating over subsets of cube dimensions. Additionally, this prevents us from generating results for cube queries with an unrealistically high number of cube dimensions (e.g., we expect at most three predicates per claim in typical newspaper articles while the query scope may include several tens of predicate columns). On the other hand, we increase the total number of queries and generate redundant results. We use function $n_G(x)$ to pick the number of dimensions for each cube query. We chose $n_G(x) = \max(m, x-1)$ for our Postgres-based cube operator implementation where m is the maximal number of predicates per claim (we use $m = 3$). Function Cube in Algorithm 3 constructs the cube query (we use simplified SQL in the pseudo-code), executes it, and returns the result. It uses function Table to retrieve associated database tables for aggregates and predicates. It exploits join paths to identify connecting tables and join predicates. Our approach assumes that the database schema is acyclic.

7 EXPERIMENTAL EVALUATION

We evaluated the AggChecker on 53 real articles, summarizing data sets and featuring 392 claims. Those test cases range from New York Times and 538 newspaper articles to summaries of Internet surveys. Test case statistics and details on the test case collection process can be found in Appendix A. Using our tool, we were able to identify multiple erroneous claims in those articles, as confirmed by the article authors.

7.1 Evaluation Metrics

We utilize two different metrics to evaluate the performance of fact-checking systems, *top-k coverage* and *F1 score*. *Top-k coverage* is defined over a set of claims with respect to a positive integer k as follows. **Top-k coverage** is the percentage of claims for which the right query is within the top-k likely query candidates. *F1 score* measures a system's performance on identifying erroneous claims. We calculate the F1 score based on these definitions of *precision* and *recall*. **Precision** is the fraction of truly erroneous (according to ground truth) claims that the system has tentatively marked as erroneous. **Recall** is the fraction of claims identified by the system as erroneous among the total set of truly erroneous claims.

7.2 User Study

We performed an anonymized user study to see whether the AggChecker enables users to verify claims more efficiently than generic interfaces. We selected six articles from diverse sources (538, the New York Times, and Stack Overflow). We selected two long articles featuring more than 15 claims [12, 43] and four shorter articles with five to ten claims each [5, 14, 15, 42]. Users had to verify claims in those documents,

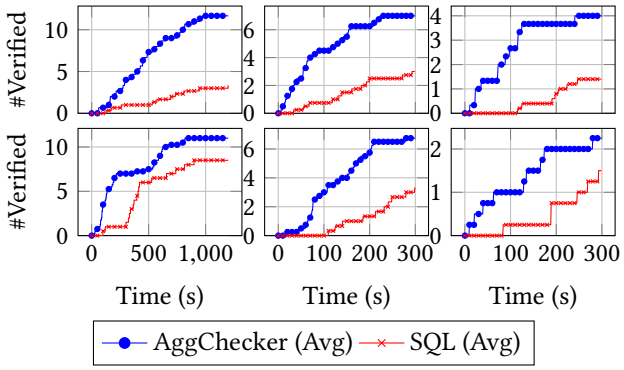


Figure 6: Number of correctly verified claims as a function of time, using different tools on different articles. Articles left-to-right then top-to-bottom: [43], [42], [14], [12], [5], [15].

Table 3: Verification by used AggChecker features.

Top-1 (1 click)	Top-5 (2 clicks)	Top-10 (3 clicks)	Custom
44.5%	38.1%	4.6%	12.8%

either by executing SQL queries on the associated database or by using the AggChecker. We gave a time limit of 20 minutes for each of the two longer articles and five minutes for each of the shorter ones. Users were alternating between tools and never verified the same document twice. We had eight participants, seven of which were computer science majors. We gave users a six minute tutorial for AggChecker.

Figure 6 reports the number of correctly verified claims for different articles and tools as a function of time. We collected timestamps from the AggChecker interface and analyzed SQL logs to calculate times for SQL queries. We count a claim as verified if the user selected the right query in the AggChecker interface or if the right query was issued via SQL. Clearly, users tend to be more efficient when using the AggChecker in each single case. Table 3 shows that this speedup is mainly due to our ability to automatically map claims to the right query in most cases (in 82.6% of cases, users selected one of the top-5 proposed queries).

7.3 Automated Checking Accuracy

We evaluate the AggChecker in fully automated mode. We demonstrate the impact of various design decisions on the text to query translation accuracy. Fully automated fact-checking is not our primary use case as user feedback can help to improve checking accuracy. Still, the first phase of fact-checking (inferring probability distributions over claim

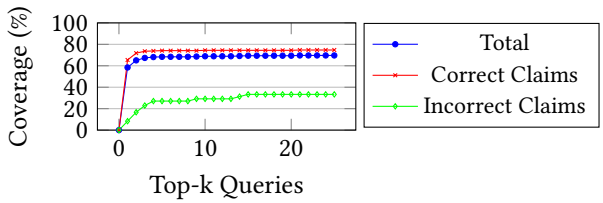


Figure 7: Percentage of claims for which correct queries were in the N most likely queries.

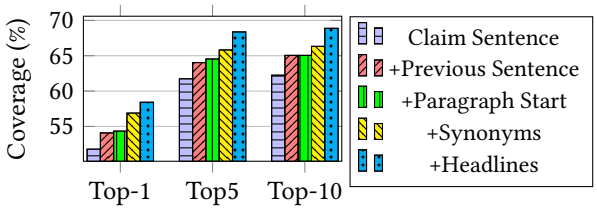


Figure 8: Top-k coverage as a function of keyword context.

Table 4: Top-k coverage versus probabilistic model.

Version	Top-1	Top-5	Top-10
Relevance scores S_c	10.7%	31.6%	41.1%
+ Evaluation results E_c	53.1%	64.8%	65.8%
+ Learning priors Θ	58.4%	68.4%	68.9%

queries) is purely automated and can be benchmarked separately. The higher the accuracy of the first phase, the smaller the number of corrective actions required from users.

To put the following results into perspective, note that our test cases *allow in average to form* 3.76×10^{10} *query candidates* that comply with our query model. Figure 7 shows top-k coverage of claims using fully automated verification. The most likely query (which is used for tentative fact-checking before user intervention) is in 58.4% of claims the right one. For 68.4% of the test claims, the right query is within the top-5 recommendations (each of those queries can be selected with only two clicks total by users).

Besides total coverage, Figure 7 reports top-k coverage for correct and incorrect claims (according to ground truth) separately. Clearly, coverage is higher for correct claims as matching the evaluation result of the query to the text value provides us with strong evidence. Note however that, even if we cannot recognize the right query for an incorrect claim, it will still often be marked as probably incorrect (since no matching query candidate can be found either).

Finally, we validate our design decisions by examining the factors that contribute towards higher accuracy as follows:

Table 5: Run time for all test cases.

Version	Total (s)	Query (s)	Speedup
Naive	2587	2415	-
+ Query Merging	151	39	×61.9
+ Caching	128	18	×2.1

Keyword context. Figure 8 illustrates the impact of keyword context on text to query translation coverage. Clearly, in particular for determining the most likely query, each keyword source considered by the AggChecker is helpful and improves top-k coverage.

Probabilistic model. Table 4 demonstrates the benefits of the probabilistic model. Compared to using keyword-based relevance scores alone (variables S_c), we successively improve top-k coverage by first integrating query evaluation results (variables E_c) and then document-specific priors (parameters Θ).

Effect of massive processing. We evaluate a large number of query candidates to refine verification accuracy. An efficient processing strategy is required to avoid excessive computational overheads. Table 5 demonstrates the impact of the optimizations discussed in Section 6 (we used a laptop with 16 GB RAM and a 2.5 GHZ Intel i5-7200U CPU running Windows 10). We report total execution times and also only the query processing times for fact-checking our entire set of test cases. Processing candidates naively yields query processing times of more than 40 minutes. Merging query candidates and caching query results yield accumulated processing time speedups of more than factor 129.9.

7.4 Comparison against Baselines

Table 6 compares all baselines in terms of precision, recall, F1 score, and run time. We also compare different variants of the AggChecker. We simplify, in multiple steps, different parts of the system and evaluate the performance. The first section of Table 5 analyzes the impact of the keyword context. In our main AggChecker version, we consider keywords from multiple sources beyond the claim sentence: keywords from the previous sentence, the first sentence in a paragraph, we consider synonyms of other keywords, and keywords from preceding headlines. In Table 5, we add those keyword sources successively, resulting in significant improvements in precision and F1 score. Next, we consider simplifications to our probabilistic model. Our main version considers random variables associated with keyword-based relevance scores (S_c), query evaluation results (E_c), and priors (Θ). We add those variables step by step, demonstrating significant improvements in F1 scores. Any simplification of our probabilistic model therefore worsens performance. Next, we analyze tradeoffs between processing time and result quality. We vary the number of query fragments, retrieved via Lucene, that are considered during processing. Considering higher number of fragments increases the F1 score but leads to higher processing overheads. We use 20 query fragments per claim in our main AggChecker version, realizing what we believe is the most desirable tradeoff between result quality and processing overheads. Altogether, our results demonstrate that each component of our system is important in order to achieve best performance.

Next, we compare against another system that focuses on automated fact checking. ClaimBuster [21, 22] is a recently proposed system for automated fact-checking of text documents. ClaimBuster supports a broader class of claims while we specialize to numerical aggregates. We demonstrate in the following that this specialization is necessary to achieve good performance for the claim types we consider. Still, both systems realize Pareto-optimal tradeoffs between generality and performance on numerical claims. ClaimBuster comes in multiple versions. ClaimBuster-FM matches input text against a database containing manually verified statements with truth values. ClaimBuster-FM returns the most similar statements from the database, together with similarity scores. We tried aggregating the truth values of the returned matches in two ways: ClaimBuster-FM (Max) uses the truth value of the statement with maximal similarity as final result. ClaimBuster-FM (MV) uses the weighted majority vote, weighting the truth value of each returned statement by its similarity score.

Another version of ClaimBuster (ClaimBuster-KB) transforms an input statement into a series of questions, generated by a question generation tool [23, 24]. Those questions are sent as queries to knowledge bases with natural language interfaces (e.g, Wolfram Alpha and Google Answers). The bottleneck however is that the required data for our test cases is not available in these generic knowledge bases. Nevertheless, a natural language query interface running on a database with all our data sets can be used as an alternative knowledge base for ClaimBuster-KB instead. To do so, we use NaLIR [28, 29], a recently proposed natural language database query interface. We cannot directly use NaLIR for fact-checking as its input format only allows natural language queries (not claims). Thus, we use the same question generation tool as ClaimBuster-KB to transform claims into question sequences and send them (including the original sentence) as queries to NaLIR. Then, we compare the results from NaLIR with the claimed value in text to see if there is a match on at least one of the queries. If so, we verify the claim as correct and if not, as wrong. Note that, using the original code of NaLIR, less than 5% of sentences are translated into SQL queries while others throw exceptions during the translation process. We extended the code to support a

Table 6: Comparison of AggChecker with baselines.

Tool	Recall	Precision	F1	Time
AggChecker - Keyword Context (Figure 8)				
Claim sentence	70.8%	29.3%	41.7%	-
+ Previous sentence	68.8%	31.1%	42.9%	-
+ Paragraph Start	70.8%	31.8%	43.9%	-
+ Synonyms	70.8%	34.3%	46.3%	-
+ Headlines (current version)	70.8%	36.2%	**47.9%**	-
AggChecker - Probabilistic Model (Table 4)				
Relevance scores S_c	93.8%	13.3%	23.3%	-
+ Evaluation results E_c	70.8%	32.7%	44.7%	-
+ Learning priors Θ (current version)	70.8%	36.2%	**47.9%**	-
AggChecker - Time Budget by Lucene Hits (Figure 11)				
# Hits = 1	79.2%	20.1%	32.1%	108s
# Hits = 10	70.8%	33.7%	45.6%	121s
# Hits = 20 (current version)	70.8%	36.2%	**47.9%**	128s
# Hits = 30	68.8%	36.3%	47.5%	133s
Baselines				
ClaimBuster-FM (Max)	34.1%	12.3%	18.1%	142s
ClaimBuster-FM (MV)	31.7%	15.9%	21.1%	142s
ClaimBuster-KB + NaLIR	2.4%	10.0%	3.9%	18733s
AggChecker Automatic	70.8%	36.2%	**47.9%**	**128s**

broader range of natural language queries (e.g., by implementing a more flexible method for identifying command tokens in parse trees) which increased the translation ratio to 42.1%. Still, only 13.6% of the translated queries return a single numerical value which can be compared with the claimed value in text.

In Table 6, the AggChecker outperforms the other baselines by a significant margin. The reasons vary across baselines. ClaimBuster-FM relies on manually verified facts in a repository. This covers popular claims (e.g., made by politicians) but does not cover "long tail" claims. We verified that the relatively high recall rate of ClaimBuster-FM is in fact due to spurious matches, the necessary data to verify those claims is not available.

Prior work on natural language query interfaces has rather focused on translating relatively concise questions that a user may ask. The claim sentences in our test data tend to be rather complex (e.g., multi-line sentences with multiple sentence parts). This makes it already hard to derive relevant questions for verification. Also, sentence parse tree and query tree tend to have a high edit distance (which hurts approaches such as NaLIR that assume high similarity). Further, many claims (30%) do not explicitly state the aggregation function (in particular for counts and sums) or there are multiple claims within the same sentence (29%). All of those challenges motivate specialized approaches for fact-checking from raw relational data.

Table 7: Properties of AggChecker and ClaimBuster.

	AggChecker	ClaimBuster
Task	Fact-checking	Claim identification, Fact-checking
Claims	Numerical	Generic
Data	Structured	Unstructured, Structured
Summary	Specialized to claims on numerical aggregates	Broader task and claim scope

8 RELATED WORK

ClaimBuster [21, 22] supports users in fact-checking natural language texts. ClaimBuster verifies facts by exploiting natural language fact checks prepared by human fact checkers, natural language query interfaces to existing knowledge bases, or textual Google Web query results. In short, Table 7 gives a comparison of the two systems.

We focus on facts (i.e., numerical aggregates) that are not explicitly given but can be derived from the input data set. Most prior work on fact-checking [4, 21, 22, 27, 37, 40, 41, 46–48, 50] assumes that entities a claim refers to are readily available in a knowledge base [3]. Prior work on argument mining [10, 20, 30–32, 38] identifies complex claims and supporting arguments in text. We link text claims to structured data instead. Prior work tests the robustness of claim queries against small perturbations [49, 51–53]. Techniques such as query merging and caching can be used in this context as well. However, as the claim SQL query is given as input, this line of work avoids the primary challenge that we address: the translation of natural language claims into SQL queries. The problem of translating natural language queries or keyword sets to SQL queries has received significant attention [1, 28, 29, 39]. Fact-checking differs as users specify queries together with claimed results, allowing new approaches. Also, we are operating on entire documents instead of single queries.

9 CONCLUSION

We introduced the problem of fact-checking natural language summaries of relational databases. We have presented a first corresponding approach, encapsulated into a novel tool called the AggChecker. We successfully used it to identify erroneous claims in articles from major newspapers.

ACKNOWLEDGMENT

This project was supported by Huawei.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. 2002. DBXplorer: a system for keyword-based search over relational databases. In *ICDE*. 5–16. https://doi.org/10.1109/ICDE.2002.994693

[2] Amazon. [n. d.]. https://www.mturk.com/mturk/welcome.

[3] Mevan Babakar and Will Moy. 2016. The state of Automated Factchecking. *Full Fact* (2016).

[4] Giovanni Luca Ciampaglia, Prashant Shiralkar, Luis Mateus Rocha, Johan Bollen, Filippo Menczer, and Alessandro Flammini. 2015. Computational fact checking from knowledge networks. *CoRR* abs/1501.03471 (2015).

[5] The New York Times Company. 2014. Looking for John McCain? Try a Sunday Morning Show. https://www.nytimes.com/2014/09/06/upshot/looking-for-john-mccain-try-a-sunday-morning-show.html

[6] The New York Times Company. 2014. Race in 'Waxman' Primary Involves Donating Dollars. https://www.nytimes.com/2014/04/24/upshot/race-in-waxman-primary-involves-donating-dollars.html

[7] The New York Times Company. 2017. The Upshot. https://www.nytimes.com/section/upshot/

[8] Wikipedia contributors. 2018. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/

[9] Chuong B Do and Serafim Batzoglou. 2008. What is the expectation maximization algorithm? *Nature Biotechnology* 26, 8 (2008), 897–899.

[10] Mihai Dusmanu, Elena Cabrio, and Serena Villata. 2017. Argument Mining on Twitter: Arguments, Facts and Sources. In *EMNLP*. 2317–2322.

[11] C. Fellbaum. 1998. *WordNet: an electronic lexical database*. MIT Press. https://books.google.com/books?id=Rehu8OOzMIMC

[12] FiveThirtyEight. 2014. 41 Percent Of Fliers Think You're Rude If You Recline Your Seat. https://fivethirtyeight.com/features/airplane-etiquette-recline-seat/

[13] FiveThirtyEight. 2014. The NFL's Uneven History Of Punishing Domestic Violence. https://fivethirtyeight.com/features/nfl-domestic-violence-policy-suspensions/

[14] FiveThirtyEight. 2015. Blatter's Reign At FIFA Hasn't Helped Soccer's Poor. https://fivethirtyeight.com/features/blatters-reign-at-fifa-hasnt-helped-soccers-poor/

[15] FiveThirtyEight. 2016. Hip-Hop Is Turning On Donald Trump. https://projects.fivethirtyeight.com/clinton-trump-hip-hop-lyrics/

[16] FiveThirtyEight. 2016. Sitting Presidents Give Way More Commencement Speeches Than They Used To. https://goo.gl/7nuGE9

[17] The Apache Software Foundation. 2017. Apache Lucene Core. https://lucene.apache.org/core/

[18] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 157 Languages. In *LREC*.

[19] PostgreSQL Global Development Group. 2017. PostgreSQL. https://www.postgresql.org/

[20] Ivan Habernal and Iryna Gurevych. 2017. Argumentation Mining in User-Generated Web Discourse. *Computational Linguistics* 43, 1 (2017), 125–179.

[21] Naeemul Hassan, Fatma Arslan, Chengkai Li, and Mark Tremayne. 2017. Toward automated fact-checking: detecting check-worthy factual claims by ClaimBuster. In *SIGKDD*. 1803–1812.

[22] Naeemul Hassan, Gensheng Zhang, Fatma Arslan, Josue Caraballo, Damian Jimenez, Siddhant Gawsane, Shohedul Hasan, Minumol Joseph, Aaditya Kulkarni, Anil Kumar Nayak, Vikas Sable, Chengkai Li, and Mark Tremayne. 2017. ClaimBuster: The first-ever end-to-end fact-checking system. *PVLDB* 10, 12 (2017).

[23] Michael Heilman and Noah A Smith. 2009. *Question generation via overgenerating transformations and ranking*. Technical Report. CMU, Language Technologies Institute.

[24] Michael Heilman and Noah A. Smith. 2010. Good question! Statistical ranking for question generation. In *NACL*. 609–617. http://www.aclweb.org/anthology/N10-1086

[25] ESPN Inc. 2017. FiveThirtyEight. http://fivethirtyeight.com/

[26] Andrew Kachites McCallum. 2002. MALLET: A machine learning for language toolkit. (01 2002).

[27] Georgi Karadzhov, Preslav Nakov, Lluís Màrquez, Alberto Barrón-Cedeño, and Ivan Koychev. 2017. Fully automated fact checking using external sources. *CoRR* abs/1710.00341 (2017).

[28] Fei Li and HV Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. *SIGMOD* (2014), 709–712. https://doi.org/10.1145/2588555.2594519

[29] Fei Li and H. V. Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *PVLDB* 8, 1 (2014), 73–84. http://www.vldb.org/pvldb/vol8/p73-li.pdf

[30] Marco Lippi and Paolo Torroni. 2015. Context-Independent Claim Detection for Argument Mining. In *IJCAI*. 185–191.

[31] Marco Lippi and Paolo Torroni. 2016. Argumentation mining: state of the art and emerging trends. *ACM Trans. Internet Techn.* 16, 2 (2016), 10:1–10:25. https://doi.org/10.1145/2850417

[32] Marco Lippi and Paolo Torroni. 2016. MARGOT: A web server for argumentation mining. *Expert Syst. Appl.* 65 (2016), 292–303. https://doi.org/10.1016/j.eswa.2016.08.050

[33] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL*. 55–60.

[34] Vox Media. 2017. Vox. https://www.vox.com/

[35] George A. Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[36] Jonas Mueller and Aditya Thyagarajan. 2016. Siamese Recurrent Architectures for Learning Sentence Similarity. In *AAAI*. 2786–2792.

[37] Ndapandula Nakashole and Tom M. Mitchell. 2014. Language-aware truth assessment of fact candidates. In *ACL*. 1009–1019.

[38] Andreas Peldszus and Manfred Stede. 2013. From argument diagrams to argumentation mining in texts: a survey. *IJCINI* 7, 1 (2013), 1–31. https://doi.org/10.4018/jcini.2013010101

[39] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Ozcan. 2016. ATHENA: An ontology-driven system for natural language querying over relational data stores. *VLDB* 9, 12 (2016), 1209–1220.

[40] Baoxu Shi and Tim Weninger. 2015. Fact checking in large knowledge graphs - a discriminative predicate path mining approach. *CoRR* abs/1510.05911 (2015).

[41] Baoxu Shi and Tim Weninger. 2016. Fact checking in heterogeneous information networks. In *WWW*. 101–102.

[42] Inc. Stack Exchange. 2015. 2015 Developer Survey. https://insights.stackoverflow.com/survey/2015/

[43] Inc. Stack Exchange. 2016. Developer Survey Results 2016. https://insights.stackoverflow.com/survey/2016/

[44] Inc. Stack Exchange. 2017. Developer Survey Results 2017. https://insights.stackoverflow.com/survey/2017/

[45] Inc. Stack Exchange. 2017. Stack Overflow Insights. https://insights.stackoverflow.com/survey/

[46] James Thorne and Andreas Vlachos. 2017. An extensible framework for verification of numerical claims. In *EACL*. 37–40.

[47] Andreas Vlachos and Sebastian Riedel. 2014. Fact Checking: Task definition and dataset construction. *ACL Workshop on Language Technologies and Computational Social Science*, 18–22.

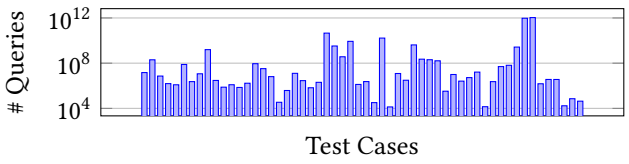


Figure 9: Number of possible query candidates per data set.

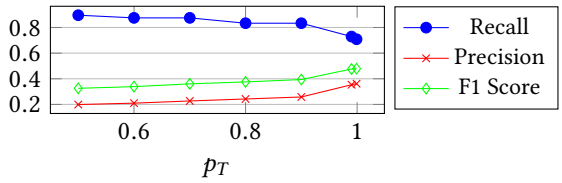


Figure 10: Parameter p_T versus recall and precision.

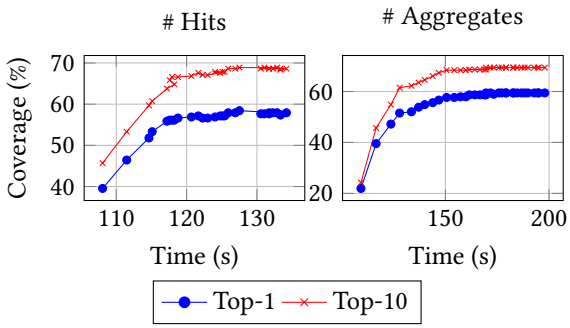


Figure 11: Top-k coverage versus processing overheads.

[48] Andreas Vlachos and Sebastian Riedel. 2015. Identification and verification of simple claims about statistical properties. In *EMNLP*. 2596–2601.
[49] Brett Walenz and Jun Yang. 2016. Perturbation analysis of database queries. *PVLDB* 9, 14 (2016), 1635–1646. http://www.vldb.org/pvldb/vol9/p1635-walenz.pdf
[50] William Yang Wang. 2017. "Liar, Liar Pants on Fire": A new benchmark dataset for fake news detection. In *ACL*. 422–426.
[51] You Wu, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. Toward computational fact-checking. *PVLDB* 7, 7 (2014), 589–600.
[52] You Wu, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2017. Computational fact checking through query perturbations. *TODS* 42, 1 (2017), 4:1–4:41. https://doi.org/10.1145/2996453
[53] You Wu, Brett Walenz, Peggy Li, Andrew Shim, Emre Sonmez, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. iCheck: computationally combating "lies, d-ned lies, and statistics". In *SIGMOD*. 1063–1066.

A DETAILS ON TEST CASES

We collected 53 publicly available articles summarizing data sets. All test cases will be made available online on the demo website. The most important criterion for our selection was that the article must unambiguously identify a tabular data set it refers to. Under that constraint, we aimed at collecting articles from different sources and authors, treating a variety of topics (covering for instance sports, politics or economy). Our goal was to obtain experimental results that are representative across a variety of domains and writing styles. We used newspaper articles from the New York Times [7], 538 [25], Vox [34], summaries of developer surveys on Stack Overflow [45], and Wikipedia [8] articles. The associated data sets range in size from a few kilobytes to around 100 megabytes. Most of them were stored in the .csv format. In a few instance, we removed free text comments, written before or after the actual table data, to obtain valid .csv format. We did however not apply any kind of pre-processing that could have simplified the fact-checking process (e.g., we did not change the original column or value names in the data nor did we change the data structure in any way).

Table 8 presents a selection of the erroneous claims we discovered in those test cases. We added comments on likely error causes that we received from the article authors. The primary challenge solved by the AggChecker is the mapping from text claims to SQL queries. This task becomes harder, the more queries can be formed according to our target query structure. Figure 9 shows the number of *Simple Aggregate Queries* (according to the definition in Section 2) that can be formed for our test data sets (the three Wikipedia articles reference total of six tables). Evidently, the number of queries is typically enormous, reaching for instance more than a trillion queries for the Stack Overflow Developer Survey 2017 [44] (this data set features more than 154 table columns).

To generate ground truth for the claims in our articles, we constructed corresponding SQL queries by hand, after a careful analysis of text and data. We contacted the article authors in case of ambiguities. The AggChecker currently supports claims about *Simple Aggregate Queries* (see Section 2). We identified 392 claims that comply with the supported format.

B AUTOMATED CHECKING ACCURACY

We present more experimental results on the automated checking accuracy with respect to different factors as follows:

Effect of parameter p_T. Parameter p_T is the assumed a-priori probability of encountering correct claims. Figure 10 shows that we obtain different tradeoffs between recall (i.e., percentage of erroneous claims spotted, based on the most likely query for each claim) and precision (i.e., percentage of claims correctly marked up as wrong) when varying it. Reducing p_T makes the system more "suspicious" and increase recall at the cost of precision. We empirically determined $p_T = 0.999$ to yield a good tradeoff for our test data set.

Effect of massive processing. Figure 11 shows the benefit of massive processing. We vary the number of hits to collect using Apache Lucene per claim (left) as well as the

Table 8: Examples for erroneous claims.

Erroneous Claim	Author Comment	Ground Truth SQL Query	Correct Value
There were only four previous lifetime bans in my database - **three** were for repeated substance abuse, one was for gambling. [13]	Yes – the data was updated on Sept. 22, and the article was originally published on Aug. 28. There's a note at the end of the article, but you're right the article text should also have been updated.	SELECT Count(*) FROM nflsuspensions WHERE Games = 'indef' AND Category = 'substance abuse, repeated offense'	4
Using their campaign fund-raising committees and leadership political action committees separately, the pair have given money to **64** candidates. [6]	I think you are correct in that it should be 63 candidates in the article, not 64.	SELECT CountDistinct(Recipient) FROM eshoopallone	63
13% of respondents across the globe tell us they are only self-taught. [43]	This was a rounding error/typo on our part – so yes, you're correct.	SELECT Percentage(Education) FROM stackoverflow2016 WHERE Education = 'I'm self-taught'	14

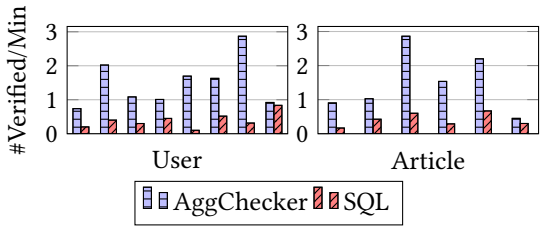


Figure 12: Number of claims verified per minute, grouped by user and by article. Articles left to right: [43], [12], [42], [14], [5], [15].

Table 9: Results of on-site user study.

Tool	Recall	Precision	F1 Score
AggChecker + User	100.0%	91.4%	95.5%
SQL + User	30.0%	56.7%	39.2%
GUI + User	23.1%	50.0%	31.6%

number of aggregation columns we consider during evaluation (right). Both affect run time (for all test case) as well as average top-k coverage. It turns out that a higher processing time budget results in greater coverage. On the other side, the figure shows that the increase in coverage diminishes as we evaluate more query candidates. Thus as mentioned in Section 6, evaluating a carefully chosen subset of query candidates is sufficient for achieving high coverage.

C USER STUDY

Figure 12 reports the "fact-checking throughput", meaning the number of correctly verified claims per minute, grouped by user (left) and by article (right). It turns out that users are in average six times faster at verifying claims when using the AggChecker interface.

Table 10: Amazon Mechanical Turk results.

Tool	Scope	Recall	Precision	F1 Score
AggChecker	Document	56%	53%	54%
G-Sheet		0%	0%	0%
AggChecker	Paragraph	86%	96%	91%
G-Sheet		42%	95%	58%

Table 9 adopts a complementary perspective and measures the number of erroneous claims that was found (recall) and the percentage of incorrect claims among the ones marked up as incorrect (precision). We identified in total three erroneous claims in our six test case articles. As expected, users achieve highest scores when interacting with the AggChecker. Note that the result for GUI is from an additional user study where two participants were asked to fact-check the articles with erroneous claims (same ones that were used in the previous user study) using Tableau, a graphical data analytics system.

D CROWD WORKER STUDY

We conducted an additional larger user study with crowd workers, recruited on Amazon Mechanical Turk (AMT) [2]. Our goal was to show that the AggChecker can be used by typical crowd workers (whom we do not expect to have a strong IT background) and without prior training. Furthermore, we compared against yet another baseline that is commonly used by laymen to analyze data: spreadsheets. We uploaded data for all test cases into an online demo of the AggChecker as well as into a public Google Sheet document.

First, we asked 50 distinct crowd workers to verify numerical claims in a 538 newspaper article [12] via the AggChecker. We asked 50 additional workers to verify the same article using Google Sheets. We paid 25 cents per task and did not

Table 11: Check-worthy claim identification.

Method	Recall	Precision	F1 Score
Naive Bayes	11.4%	46.7%	18.0%
Max Entropy	17.9%	52.5%	26.3%
Decision Tree	1.07%	11.7%	1.96%
Heuristic-based	100.0%	23.1%	37.6%

set any worker eligibility constraints. We compare baselines in terms of recall (i.e., percentage of erroneous claims identified) and precision (i.e., ratio of actually erroneous claims among all claims flagged by crowd worker). We had only 19 respondents for the AggChecker interface and 13 respondents for the Google sheet interface over a 24 hours period. Table 10 summarizes the performance results (scope: document). While the performance of crowd workers is only slightly worse compared to the participants of our prior user study when using the AggChecker, crowd workers are unable to identify a single erroneous claim via spreadsheets.

We doubled the payment and narrowed the scope for verification down to two sentences (taken from another 538 article [16]). We deliberately selected an article with a very small data set where claims could even be verified by counting entries by hand. All 100 tasks were solved within a 24 hours period. Table 10 (scope: paragraph) shows improved results for the spreadsheet, the performance difference to the AggChecker is however enormous.

E CLAIM IDENTIFICATION

We use Mallet [26], a tool for statistical natural language processing, to train a classifier to identify check-worthy claims (more precisely, sentences containing check-worthy claims). We use our data set with annotated claims as training and test data with 10-fold cross validation. We divide all sentences into ten distinct sets balancing the number of claims in each set. Table 11 demonstrates the recall, precision and F1 score of different classifiers and our heuristic-based method. Check-worthy claims account for about 7.3% of all sentences.

F DATASET IDENTIFICATION

We show that we can use Google Dataset Search, a specialized service for finding structured data, to find the relevant data set needed to fact-check an article. If we only use the title of an article, we get a mean reciprocal rank (MRR) of 0.2547. To improve this, we use entity tagging to extract important keywords and use them in a search query. Then, we take the first five data sets from the search result and use the AggChecker to re-rank the data sets based on the verification rate (i.e., the number of claims that can be verified by a given data set). As a result, we achieve a MMR of 0.4754.

Table 12: Results on adding nearest words (best cases).

Method	F1 Score	Top-1	Top-5	Top-10
WordNet (synonyms)	47.9%	58.7%	68.4%	68.9%
word2vec (15 words)	46.2%	56.4%	66.8%	67.9%
fastText (35 words)	48.2%	53.3%	64.8%	66.1%

Table 13: Comparison of methods for matching claims to query fragments.

Method	Recall	Precision	F1 Score
Information Retrieval	70.8%	36.2%	47.9%
Sentence Similarity	77.1%	22.7%	35.1%

In addition, we test the case of having a database that can fact-check multiple articles in the same category. We verify two articles about NFL using a database containing five tables related to NFL statistics. Based on the verification rate, the AggChecker successfully identifies the correct data set and verifies claims in each article. Note that the AggChecker needs to verify every pair of an article and a data set to calculate the verification rates. This process took 27.8s to finish while fact-checking the two articles given the correct data sets took only 5.8s.

G USE OF WORD EMBEDDING

We report results using word embedding in different phases of keyword matching. First, we consider replacing synonyms from WordNet with nearest words according to the cosine similarity of word vectors. We test on two pre-trained word vectors: 1) word2vec trained on Google News data set and 2) fastText trained on Common Crawl and Wikipedia [18]. We increased the number of added words from 5 to 50 incrementing by 5. Table 12 presents the best case performance of each method. Adding synonyms from WordNet performs well on top-k coverage while fastText with 35 added words has the highest F1 score.

Next, we compare different methods for finding relevant query fragments. One method uses Lucene to compute relevance scores between pairs of claims and query fragments. The other method uses a model [36] based on word embedding and the Long Short Term Memory (LSTM) network, which computes semantic similarity scores between pairs of word sequences. We compute the similarity between the sentence containing a claim and the keywords associated with each query fragment. Table 13 shows that semantic similarity scores might not be as effective as relevance scores when matching claims to query fragments.