Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification

Georgios Karagiannis[†*] Mohammed Saeed[‡*] Paolo Papotti[‡] Immanuel Trummer[†]
[†]Cornell University, USA [‡]Eurecom, France
{ gk446,lt224 }@cornell.edu, { papotti,mohammed.saeed }@eurecom.fr

ABSTRACT

Organizations spend significant amounts of time and money to manually fact check text documents summarizing data. The goal of the Scrutinizer system is to reduce verification overheads by supporting human fact checkers in translating text claims into SQL queries on an database. Scrutinizer coordinates teams of human fact checkers. It reduces verification time by proposing queries or query fragments to the users. Those proposals are based on claim text classifiers, that gradually improve during the verification of a large document. In addition, Scrutinizer uses tentative execution of query candidates to narrow down the set of alternatives. The verification process is controlled by a cost-based optimizer. It optimizes the interaction with users and prioritizes claim verifications. For the latter, it considers expected verification overheads as well as the expected claim utility as training samples for the classifiers. We evaluate the Scrutinizer system using simulations and a user study with professional fact checkers, based on actual claims and data. Our experiments consistently demonstrate significant savings in verification time, without reducing result accuracy.

PVLDB Reference Format:
G. Karagiannis, M. Saeed, P. Papotti, I. Trummer. Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification. *PVLDB*, 13(11): 2508 - 2521, 2020.
DOI: https://doi.org/10.14778/3407790.3407841

1. INTRODUCTION

Data is often disseminated in the form of text reports, summarizing the most important statistics. For authors of such documents, it is time-consuming and tedious to ensure the correctness of each single claim. Nevertheless, erroneous claims about data are not acceptable in many scenarios as each mistake can have dire consequences. Those consequences reach from retractions (in case of scientific papers [18]) to legal implications (in case of business or health reports [3]). Besides the authors, erroneous claims can have serious consequences for the target audience [4]. We present SCRUTINIZER, a system that helps teams of fact checkers to verify consistency of text and data efficiently [26].

*The first two authors contributed equally.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407841

Figure 1: Global Energy Demand history and estimates (GED), the full table has 22 rows and 70 attributes.

Index	2017	2018	...	2030	2040
PGElecDemand	22 209	22 793	...	29 349	35 526
PGINCoal	2 390	2 412	..	2 341	2 353
TFCelec	21 465	22 040	...	28 566	34 790
...	...	...	...	...	...

Our work is inspired and motivated by two real-world use cases. We describe those use cases in the following and use them as examples and benchmarks throughout the paper.

USE CASE 1. *The International Energy Agency (IEA) is a Paris-based intergovernmental organization. Every year the agency produces a report of more than 600 pages about the energy consumption and production in the world, covering historical facts and predictions both for individual countries and at the world level. We have been given access to the 2018 edition, which contains 7901 sentences with 1539 manually checked statistical claims. IEA requires each claim to be verified independently by three persons (beyond the claim text author). Hence, verification takes months of work of a team of domain experts.* SCRUTINIZER *is the first result of a collaboration aimed at reducing time and financial overheads of that verification process.*

Consider the following example claim from that scenario.

EXAMPLE 1. *The IEA database contains hundreds of relational tables with information about energy, pollution, and climate. A fragment of a table is reported in Figure 1. Consider the claim "**In 2017, global electricity demand grew by 3%**, more than any other fuel besides solar thermal, reaching 22 200 TWh.". An expert validates the claim in bold by identifying the relevant table(s) and by writing a query over such table to collect the relevant information. In the example (assuming unique values in the* `Index` *column):*

```
SELECT POWER(
 (SELECT 2017 FROM GED WHERE Index='PGElecDemand')/
 (SELECT 2016 FROM GED WHERE Index='PGElecDemand'),
 1/(2017-2016)) -1
```

Finally, the expert compares the output of the query with the claim and either validates or updates the claim.

Unlike the first, our second use case assumes verification by non-expert, anonymous crowd workers.

Use Case 2. *The spread of the Coronavirus is accompanied by a spread of misinformation. This "Infodemic" [29] is testing manual verification capacities of large social media platforms [17], thereby motivating automated methods. Some (even though not all) misclaims about the novel Coronavirus refer to numerical statistics, thereby falling within the scope of the* Scrutinizer *system. For instance, misclaims about absolute or relative case counts for specific regions and time ranges are common [45]. We therefore decided to create a public Web interface [24], based on the* Scrutinizer *system, that allows to verify statistical claims on the Coronavirus. The database used for verification consists of daily updated statistics from official sources such as the World Health Organization (WHO) and the Center of Disease Control (CDC). Our Web site, recently covered in the press [8, 35, 40], has attracted over 12,000 distinct users at the time of writing.*

In both scenarios, we verify claims given as natural language text. We verify or refute a claim by formulating a query on an associated database. Of course, the primary challenge is to translate natural language text into SQL statements. This problem has been the focus of significant prior work [1, 31, 21, 32, 47, 37, 44], inside and outside of the database community. The scenarios we consider are however specific and require a novel approach. First, prior work typically aims at automating text to query translations. Relying on purely automated translation would however not be acceptable for IEA for instance, due to the high stakes and limitations of current technology. Hence, we verify claims in a mixed-initiative approach that integrates human feedback. Second, prior work aims at translating queries instead of claims. This matters, as we can rank queries based on how close their results come to claimed values. Finally, prior work typically considers single queries. E.g., in the case of IEA, we rather deal with large documents that contain hundreds of related claims. This can be exploited to reorder claims for verification and minimize expected verification overheads. Literature on data-driven claim verification [5, 20, 23], discussed in more detail in Section 7, is more sparse at this point. In our experiments, we demonstrate that prior work cannot address the use cases we consider.

When mapping claims to queries, Scrutinizer considers a scenario-specific query search space. This search space is defined by a domain expert with SQL training and knowledge of the database schema (e.g., the head of the fact checker team in case of IEA). The same person supplies text snippets that allow Scrutinizer to formulate natural language questions about query properties. The bulk of the verification work is done by another group of fact checkers. Those can be domain experts without IT background (e.g., the fact checker team at IEA) or a crowd of anonymous users (like in our second use case). To map claims to queries, Scrutinizer introduces a scenario-specific set of classifiers. Each classifier is associated with a choice point in the query search space. This can be the choice between multiple query templates (e.g., selecting the arithmetic formula in Example 1) or filling in a placeholder (e.g., the year 2016 in Example 1). Having the result of each classification for a given claim can determine the associated query. However, if classifier confidence is below a threshold, classification results must be verified by human workers.

Our goal is to minimize overheads for human workers. Those overheads are determined by multiple factors. We expect that verifying a correct option for a query property (e.g., a proposition for an arithmetic formula) is typically faster, for workers, compared to suggesting the correct answer from scratch. We verify this intuition by a user study in Section 6. Hence, verifying claims based on high-quality classifier suggestions is cheaper than for low-confidence claims. On the other side, obtaining the correct answer for low-confidence claims and using it as training data, may improve classifier accuracy for the remaining claims. We take both factors into account when determining the order in which claims are verified. Also, we optimize the sequence of questions asked to workers about specific claims. Here, we can exploit the fact that getting answers for certain query properties implicitly prunes options for others.

Scrutinizer generates suggestions for query translations via classifiers. In principle, various types of classifiers can be used. Currently, we use pre-trained language models (in particular, Facebook's recently proposed XLM model [28]) with task-specific fine-tuning. Besides the claim text, we also exploit query evaluation results to rank query candidates. More specifically, to break draws between likely query options, we rank queries higher if their numerical evaluation results match numbers that appear in claim text. In our experiments, we demonstrate that Scrutinizer outperforms various baselines for all considered scenarios.

In summary, the original scientific contributions of this paper are the following:

- We introduce the problem of mixed-initiative data-driven fact checking and two corresponding real-world use cases (Section 2).
- We describe the Scrutinizer system (Section 3), featuring semi-automated claim to query translations (Section 4) and planning modules for optimally interacting with fact checkers (Section 5).
- We demonstrate experimentally that Scrutinizer performs well in several complementary scenarios, improving over purely manual fact checking as well as automated baselines (Section 6).

2. PROBLEM MODEL

Scrutinizer aims at mixed-initiative verification (MIV) of statistical claims from relational data. The term "mixed-initiative" refers to the fact that our approach combines feedback from human workers with automated verification.

Definition 1. *One instance of the* ***MIV Problem*** *is defined by a quadruple $\langle \mathcal{T}, \mathcal{P}, \mathcal{Q}, \mathcal{L} \rangle$. Here, $\mathcal{T}$ is the verification target. It describes the claims to verify and the database used for verification. Parameters $\mathcal{P}$ are configuration parameters. They constrain verification and contain constants used for cost-based planning. $\mathcal{Q}$ is the query template. It describes complete SQL queries to consider during verification. Finally, $\mathcal{L}$ maps template components to text snippets, used for formulating associated questions to crowd workers. A solution to MIV maps each input claim to a Boolean verification result and to an SQL query justifying that result.*

The goal is to reduce overheads for human fact checkers in MIV. Optionally, the input also contains training data mapping prior claims from the verification domain to corresponding queries. This is however not required ("cold start scenario") as training data is dynamically collected during verification. Queries can validate claims in different ways. Claims may explicitly mention numbers that can be matched against query results. We call this claim category "Explicit Claims" (some prior work is specific to that category [23]). Alternatively, claims may be verified by queries returning a Boolean value, with the semantics that the claim holds if that value is true ("Implicit Claims"). Next, we define the four components from Definition 1.

DEFINITION 2. *A **Verification Target** $\mathcal{T} = \langle C, D, s\rangle$ is a set C of claims to verify, a relational database D used for verification, and (optionally) a function $s : C \mapsto \mathcal{S}$ mapping claims to document sections. Each claim $c \in C$ is defined by a natural language description, containing the claim and relevant context. Each claim can be verified or refuted by running an associated query on database D.*

DEFINITION 3. ***Configuration Parameters** $\mathcal{P}$ include constants representing cost estimates for different types of actions performed by human checkers. The specific parameters and their semantics will be described in Section 5. Also, they include a confidence threshold ρ for verification. Human workers are only used for claims where the confidence of automated verification is below ρ. Finally, a parameter determines the number of workers asked the same question.*

DEFINITION 4. *A **Query Template** describes a space of SQL queries to consider during verification. We distinguish **Complete Templates**, describing complete SQL queries, from **Fragment Templates**, describing query parts. We define templates recursively. Any SQL query fragment, using schema elements from the target database, or an SQL keyword alone, is a (constant) template. Assuming that $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ are templates then $CAT(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ is also a template. It represents the* concatenation *of queries (or query fragments) matching templates $\mathcal{Q}_1$ to $\mathcal{Q}_n$. Similarly, $CHC(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ represents a* choice *among different templates. A query matches the choice template if it matches at least one of the templates $\mathcal{Q}_1$ to $\mathcal{Q}_n$. Finally, if $\mathcal{Q}_1$ is a complete query template describing queries that evaluate to scalar numerical results then $EXP(\mathcal{Q}_1)$ is a template as well. It matches all free arithmetic* expressions *that can be formed using queries matching $\mathcal{Q}_1$ (and constants) as operands, and logical and arithmetic operators (but no other SQL keywords such as* FROM *or* SELECT*). Templates using concatenation (CAT), choice (CHC), or free expressions (EXP) are **Composite Templates**. We call templates used as operands for concatenation, choice, or free expressions (here: $\mathcal{Q}_1$ to $\mathcal{Q}_n$) their **Components**.*

EXAMPLE 2. *We instantiated* SCRUTINIZER *in a public Web interface ("CoronaCheck") for verifying single statistical claims about the novel Coronavirus, submitted by users, using data from WHO and CDC. Data is represented as tables, containing numbers (e.g., the number of confirmed cases) for different time periods in different columns, and data for different regions in different rows. For instance, we support comparative claims about the number of confirmed cases. The corresponding query space is described by a template of the form $CAT('\texttt{SELECT}', SQ, '>', SQ)$ where SQ is a template matching lookup queries, retrieving the number of cases for specific regions and times. SQ is defined as $CAT('\texttt{SELECT}', T, '\texttt{FROM} NrCases\ \texttt{WHERE}', CP)$, where $T = CHC('Jan', 'Feb', \ldots)$ is a choice over time periods (represented as column names) and $CP = CAT(CN, '= Cname')$ models restrictions of the region (i.e., an equality predicate on the country name column). E.g., the query*

```
SELECT
 SELECT Feb FROM NrCases WHERE Cname='Germany' >
 SELECT Jan FROM NrCases WHERE Cname='Germany'
```

matches that template. Overall, we use a choice template for verification. Each component represents queries used to verify one popular type of claim, including the one described above.

Our goal is to map claims to queries. If automated verification fails, we solicit feedback from human fact checkers. The labeling function allows us to formulate questions to workers for narrowing down queries for a given claim.

DEFINITION 5. *The **Labeling Function** $\mathcal{L}$ maps query template components to text for formulating questions. It maps each choice element (CHC) to a question text, and, optionally, each choice option to a human-readable label. Also, it maps each formula element (EXP) to question text, asking workers to select an appropriate formula. No labels are needed for constant query fragments and concatenations. We call choice and expression elements also* choice points *or* query properties *as they entail claim-specific decisions.*

Next, we show how to instantiate this model for our two use cases: verifying IEA and Coronavirus claims.

EXAMPLE 3. *At IEA, claims are extracted from reports by the head of the fact checker team (potentially adapting claim text slightly to make claim context more apparent). Claims are verified from a database containing hundreds of tables containing historical data (e.g., on energy consumption and production) as well as simulation results. Claims can be verified using the query template $CAT('\texttt{SELECT}', EXP(SQ))$. Component template SQ models data lookups and is of the form $CAT('\texttt{SELECT}', YR, '\texttt{FROM}', TB, '\texttt{WHERE}', CD)$. YR denotes a specific year (generally modeled as columns by IEA), TB the table name (associated with a scenario such as global forecast of CO2 emissions), and CD=CAT('Index=',PV) a condition fixing the primary key column (generally called "Index") to a specific value. We model formulas using the year itself as numerical operand via a sub-query accessing a special identity table (mapping each year to itself). The labeling function assigns for instance YR to the question text "What is/are the corresponding year(s)?" that is shown to crowd workers to solicit feedback (see Figure 3). Also, $\mathcal{L}(EXP(SQ))$ = "Enter a formula translating the claim". For highest accuracy, IEA requires manual verification of each claim (a confidence threshold of $\rho = 100\%$). The added benefit of* SCRUTINIZER *lies in suggesting likely verification options to save checking time. As final step for each claim, workers are asked whether the associated query (with evaluation result) validates or refutes the claim.*

The query template above matches the query from Example 1. Here, the free expression uses the POWER function and

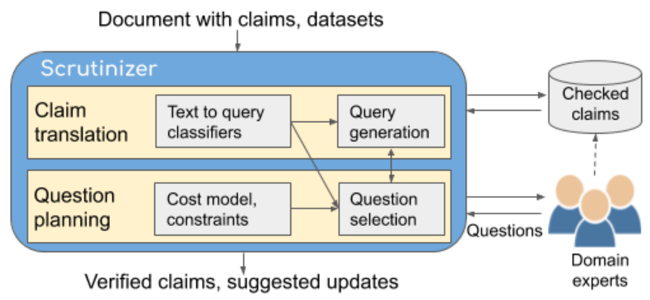


Figure 2: Architecture of SCRUTINIZER.

four instances of SQ (two lookups and two year values, internally modeled as lookups in the identity table). The current CoronaCheck version (June 2020) does not yet introduce new expressions on the fly, based on user feedback, while we consider this possibility for the IEA scenario. Hence, we use fixed formulas in the CoronaCheck templates but a free expression (EXP) element for the IEA template.

EXAMPLE 4. *The query template for CoronaCheck was described in Example 2. If classifier confidence is above a threshold of $\rho = 0.9$, users are shown a Boolean verification result. Clicking on the result reveals query and data used for verification. If confidence is below the threshold, users are shown a dialog soliciting them to select query and data themselves (by selecting from given answer options). Based on crowd feedback,* SCRUTINIZER *improves accuracy over time.*

3. SYSTEM OVERVIEW

Figure 2 shows a simplified overview of SCRUTINIZER. The system encompasses two primary components. The automated translation component leverages machine learning to identify the elements that define every claim, i.e., candidates for datasets, attributes, rows, and comparison operations. The question planning component interacts with human domain experts to verify such elements and the checking results, optimizing verification tasks for maximal benefit.

Verification comprises the following high-level steps. First, based on the query template (Definition 4), we create a set of text classifiers. We introduce classifiers for choice points in the query template as detailed in Section 4.1. Those classifiers map claim text to query properties. If available, they are trained with pairs of claims and queries from prior verification sessions. Second, as an alternative to automatic classification, we generate a set of questions on query properties about each claim. Those questions can be asked, optionally, to human fact checkers, if classification confidence is below a threshold. Section 4.2 provides details on question generation.

After those preparatory steps, an iterative verification algorithm starts. It runs until all claims have been verified with sufficiently high confidence. The corresponding pseudo-code in Algorithm 1 is simplified for readability and contains only the most important parameters (e.g., we do not explicitly refer to configuration parameters $\mathcal{P}$).

In the simplest case, based on previous training data, we are able to map each claim to a query with sufficiently high confidence. Typically, this is not possible. For instance,

Algorithm 1 Main verification algorithm.

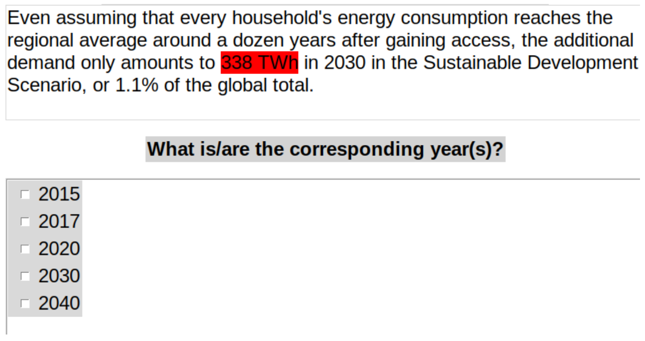
```
1: // Verify claims $C$ using models $M$
2: // and return verification results.
3: function VERIFY($C, M$)
4:     // Initialize verification result
5:     $A \leftarrow \emptyset$
6:     // While unverified claims left
7:     while $C \neq \emptyset$ do
8:         // Select next claims to verify
9:         $N \leftarrow$OPTBATCH($C, M$)
10:        // Select optimal question sequence
11:        $S \leftarrow$OPTQUESTIONS($N, M$)
12:        // Get answers from fact checkers
13:        $A \leftarrow A \cup$GETANSWERS($N, M, S$)
14:        // Retrain text classifiers
15:        $M \leftarrow$RETRAIN($N, M, A$)
16:        // Remove high-confidence claims
17:        $C \leftarrow C \setminus$VERIFIED($N, M, A$)
18:    end while
19:    // Return verification results
20:    return $A$
21: end function
```

IEA generally requires utmost precision in their reports. To achieve an accuracy close to 100%, inspection by human workers is generally required. SCRUTINIZER therefore interacts with human workers, asking them to verify classifier results or to suggest alternatives. Figure 3 shows an example screen (the lower screen half, asking workers to enter new years if needed, was cropped). We submit claims to workers in batches. Optionally, the same claims are verified by multiple workers (the number of workers can be configured). If so, we take the majority vote in case of conflicting answers. For free expressions, we use a simple normalization method (e.g., we remove spaces and capitalize symbols) before comparing answers of different workers. Depending on the scenario, we may be able to choose the order in which claims are verified. For IEA for instance, we can choose which out of hundreds of claims in a given report are verified first. We prioritize claims in a principled manner, outlined in Section 5.2, to minimize expected verification cost. We take into account the expected verification cost (which is lower if classifier suggestions are expected to be accurate) as well as utility in improving classifier precision. The latter point applies since answers from human workers can be used immediately as additional training data, thereby benefiting the verification of remaining claims.

Validating claims is easier for humans if the system suggests correct answer options for questions on query properties. As described in detail in Section 4.3, we rank likely queries (according to our classifiers) based on their query evaluation results. If presenting workers with answer options, we show them in order of likelihood to minimize expected overheads. Section 5.1 contains more details on how we select and prioritize questions asked about specific claims.

4. CLAIM TRANSLATION

We exploit three sources of information to translate claims to queries. First, we introduce classifiers that use claim text as features. Second, we request feedback from human fact checkers. Third, we use evaluation results from candidate


Even assuming that every household's energy consumption reaches the regional average around a dozen years after gaining access, the additional demand only amounts to 338 TWh in 2030 in the Sustainable Development Scenario, or 1.1% of the global total.

What is/are the corresponding year(s)?

- ☐ 2015
- ☐ 2017
- ☐ 2020
- ☐ 2030
- ☐ 2040

Figure 3: Screen soliciting workers to select relevant years for marked up claim.

Algorithm 2 Initializing ML classifiers.

```
 1: // Get classifiers for choice points in query template $\mathcal{Q}$.
 2: function InitML($\mathcal{Q}$)
 3:     // Distinguish type of query template
 4:     if $\mathcal{Q}$ is constant then
 5:         return $\emptyset$ // No classifiers needed
 6:     else if $\mathcal{Q} = CAT(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ then
 7:         // Combine classifiers of components
 8:         return $\cup_{1 \leq i \leq n}\{\text{InitML}(\mathcal{Q}_i)\}$
 9:     else if $\mathcal{Q} = CHC(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ then
10:         // Classifiers for choice and components
11:         return $\cup_{1 \leq i \leq n}\{\text{InitML}(\mathcal{Q}_i)\} \cup \{\mathcal{M}_{\mathcal{CHC}}(\mathcal{Q})\}$
12:     else if $\mathcal{Q} = EXP(\mathcal{Q}_S)$ then
13:         // Classifiers for expression and component
14:         return $\{\text{InitML}(\mathcal{Q}_S)\} \cup \{\mathcal{M}_{\mathcal{EXP}}(\mathcal{Q})\}$
15:     end if
16: end function
```

queries to rank them by likelihood. The following subsections discuss those mechanisms in more detail.

4.1 Claim Classification

For a given scenario, the query template (see Section 2) defines the search space for queries. It defines degrees of freedom in the form of choice points (i.e., one out of several options is correct) and free expressions. We introduce one classifier for each of those query properties. Those classifiers exploit claim text as well as the claim location (e.g., the claim section for IEA reports) as features. Different types of classifiers can be used. We exploit pre-trained language models that are fine-tuned by task-specific training [28].

Algorithm 2 illustrates which classifiers are created for a given query template. This is done before the actual claim verification starts. Algorithm 2 takes the query template as input, the output is a set of classifiers linked to specific choices in the query template. Function InitML decomposes input templates and collects classifiers for components via recursive invocations. The behavior of the function depends on the type of input. No classifiers are necessary for constant query templates. For concatenations, we take the union of the classifiers for the concatenation operands (the components). Similarly, we union classifiers from components for choice elements together with a new classifier (initialized with function $\mathcal{M}_{CHC}(\mathcal{Q})$) in charge of selecting one of the choice options. For free expressions, we collect classifiers for the component and introduce a new classifier to determine the expression itself (function $\mathcal{M}_{EXP}(\mathcal{Q})$). After initialization, classifiers are trained during verification, using input from human fact checkers. Optionally, they can be trained before verification starts if training data is available, e.g, in case of IEA, labels from prior text versions.

Template Type	CAT	CHC	EXP
Answer Multiplicity	$[l, u]$	$[0, u]$	$[0, \infty]$

Table 1: New bounds on answer multiplicity given components with bounds $[l, u]$.

Example 5. *Consider the query template from Example 3 (IEA). The template is defined as concatenation between a constant and a free expression. The free expression, in turn, depends on a component that represents an SQL query. Here, we introduce a classifier determining the free expression, a classifier determining the table TB within the sub-query, one for the year YR, and one for the row index PV.*

4.2 Generating Questions

Classifier accuracy is limited as automated natural language understanding is imperfect. Also, classes may be initially unknown, e.g., a claim whose associated query uses an expression that has not appeared before. To make up for the limitations of automated translation, we may request feedback from human workers. We introduce questions for each property of the query template. This means that each classifier is complemented by a question. Its answer, for a specific claim, replaces the classification result. Note that all those questions are optional. We may not ask about a query property if the confidence of the corresponding classifier is sufficiently high. Also, the answer to one question may prune possible answers for another question about the same claim. We discuss in Section 5 how to select which questions/answers to expose in which order. Here, we discuss how to generate the set of possible questions.

We omit the code for generating questions as it is similar to the one for generating classifiers (Algorithm 2) and describe the differences instead. For each initialized classifier, a corresponding question is introduced. The question uses text snippets, given via the labeling function $\mathcal{L}$ (see Section 2). For context, workers are shown the claim text along with the question. Furthermore, workers can select between multiple ranked answer options. In that context, we must decide how many answers workers can select. For instance, given a choice between constant query fragments, only one single answer can be accepted. On the other side, free expressions may contain a number of symbols that is a-priori unknown. Hence, unless the concrete expression is known, an arbitrary number of selections can be made for any answers that refer to the component template. We calculate lower (l) and upper bounds (u) on allowed answer multiplicity in a bottom-up approach with the recursive rules in Table 1. Given a question about a query template with answer multiplicity in $[l, u]$, it shows multiplicity after inserting the template as component into a composite template.

Example 6. *Consider the sub-query element (SQ) from Example 3. It must contain exactly one year, hence we have multiplicity bounds $[1, 1]$ for questions on the year as long as we consider SQ alone. However, SQ appears inside a free*

Algorithm 3 Generating candidate queries using classifiers.

```
1: // Generate most likely queries matching $\mathcal{Q}$, using
2: // $k$ most likely alternatives per choice according to
3: // ML classifiers $M$, for claim $c$.
4: function GQ($\mathcal{Q}, M, k, c$)
5:     // Distinguish type of query template
6:     if $\mathcal{Q}$ is constant then
7:         return $\mathcal{Q}$
8:     else if $\mathcal{Q} = CAT(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ then
9:         // Combine most likely queries from components
10:        return $\mathrm{GQ}(\mathcal{Q}_1, M, k, c) \times \ldots \times \mathrm{GQ}(\mathcal{Q}_n, M, k, c)$
11:    else if $\mathcal{Q} = CHC(\mathcal{Q}_1, \ldots, \mathcal{Q}_n)$ then
12:        // Most likely queries for most likely choices
13:        return $\{\mathrm{GQ}(\mathcal{Q}_S, M, k, c) | \mathcal{Q}_S \in M(\mathcal{Q}, k, c)\}$
14:    else if $\mathcal{Q} = EXP(\mathcal{Q}_S)$ then
15:        // Get most likely operands
16:        $O \leftarrow \mathrm{GQ}(\mathcal{Q}_S, M, k, c)$
17:        // Most likely expressions using operands
18:        return $\{e.substitute(O) | e \in M(\mathcal{Q}, k, c)\}$
19:    end if
20: end function
```

expression. This changes the multiplicity range to $[0, \infty]$ *according to Table 1. This means that, when asking workers which years are relevant for a claim (see Figure 3), we allow them to select arbitrarily many options.*

Beyond questions on specific query properties, manual verification of each claim ends with a question asking workers to validate the query as a whole (i.e., all properties) and to determine the final claim verification result.

4.3 Query Generation

Besides the claim text, we also use query evaluation results to rank queries. This ranking determines the order in which options are shown to workers (see Section 5). For Boolean queries, a true result typically indicates that the query verifies the claim. Hence, as a heuristic, we rank queries returning "True" above queries returning "False" (or "Null"). This assumes that accurate claims are more likely than inaccurate claims, an assumption that has been used in prior work [23] and holds for our test scenarios. Queries with numerical result typically validate a claim by calculating numbers that appear in claim text. Hence, we heuristically rank queries by the distance between query result and numbers that appear in claim text (we use the number with minimum distance if the claim contains multiple numbers).

Of course, we cannot evaluate all possible queries matching the template. Instead, we focus on queries that are likely according to classifiers and worker answers (if any). Algorithm 3 shows how we derive the scope of queries for evaluation. The pseudo-code is simplified, assuming that only classifier results but no worker answers are considered. Worker answers may prune classes further.

Algorithm 3 uses the scenario-specific query template, the trained classifiers, the current claim, and an integer parameter k as input. Parameter k determines how many of the most likely alternatives to consider for each query property. Function $M(\mathcal{Q}, k, c)$ retrieves the k most likely alternatives for the classifier associated with query template $\mathcal{Q}$, considering the text and location (section) of claim c. The algorithm composes likely queries for a template by combining likely queries for its components (obtained via recursive invocations). For instance, for concatenation, likely queries are formed via cross product between likely queries for components. For free expressions, likely queries are created in two steps. First, we retrieve a likely set of operands (Line 16) via recursive invocation. For the k most likely operands, we instantiate the k most likely expressions in all possible ways. I.e., we substitute their symbols with most likely operators, considering all possible permutations.

EXAMPLE 7. *Consider the query template from Example 3 again. Assume we obtained formula* `POWER(a/b,1/(c-d))-1` *as free expression (either by classification with sufficiently high confidence or by asking human workers). Furthermore, assume that the most likely operands for the formulas are the years 2016, 2017, and 2018, and two sub-queries retrieving energy demand data for years 2016 and 2018, respectively. We evaluate queries* $5 \cdot 4 \cdot 3 \cdot 2 = 120$ *queries, considering all possible substitutions via likely operators. As the claim text (see Example 1) contains the number 3%, we rank by the distance between query result and that number. When proposing answer options to workers, we order them according to the rank of the query they appear in.*

5. QUESTION PLANNING

Question planning consists of two tasks: determining optimal questions to verify single claims, and determining an optimal verification order between claims. We discuss the first problem in Section 5.1 and the second one in Section 5.2.

5.1 Single Claim Verification

For each claim, we generate a series of screens (Figure 3 shows an example). Each screen contains questions that are answered by a worker. Each screen is associated with one specific query property. On the upper part of each screen, workers are shown a set of answer options with regards to the current property. Those answer options are obtained from our classifiers. On the lower part of each screen, workers have the option to suggest new options, if the correct answer is not on display. The final screen for each claim asks directly for the query translating the current claim. Answers to prior questions may have allowed us to narrow down the range of possible queries. If so, the chances for confronting workers with the correct query increase.

In this scenario, our search space for question planning is the following. First, we need to decide how many screens to show. Second, we need to determine what query properties our questions should focus on. Third, we need to decide how many answer options to display on each screen. Fourth, we need to pick those answer options.

We make those decisions based on a simple cost model, representing time overhead for crowd workers for verifying the current claim. We assume that workers read screen content from top to bottom. For each answer option, a worker needs to determine whether it is correct or not. We count a per-option verification cost in our model, distinguishing cost of verifying answers about query properties, v_p, from the cost of verifying the full query (on the final screen), v_f. We choose constants such that $v_p \ll v_f$ to account for the fact that full queries are significantly longer than their fragments (which increases reading time and therefore verification cost). If none of the given options applies, crowd workers must suggest an answer themselves. We denote by s_p and s_f the cost of suggesting answers for properties and queries (again, $s_p \ll s_f$).

First, we discuss how to choose the number of screens and answer options. We denote the number of screens by *nsc* and the number of options by *nop*. Predicting the precise verification cost for specific choices of those parameters is not possible. Doing so would require knowing the right solution to each question (as it determines how many options workers will read). However, we can upper-bound verification cost in relation to the cost of verifying claims without SCRUTINIZER (the proofs for this and the following theorems can be found in the extended technical report [25]).

THEOREM 1. *Compared to the baseline, relative verification overhead of* SCRUTINIZER *is at most* $(nop \cdot v_f + nsc \cdot (v_p + s_p))/s_f$.

COROLLARY 1. *Setting* $nop = s_f/v_f$ *and* $nsc = s_f/(v_p + s_p)$ *limits verification overheads to factor three.*

We will use the aforementioned setting for most of our experiments. Having determined the number of screens and options, we still need to pick specific screens and answers. First, we discuss the selection of answer options. Note that the worst-case verification cost of a property depends only on the number of options shown (but not on the options themselves). Hence, to pick options, we consider expected verification cost instead.

We calculate expected verification cost based on our classifiers, assigning specific answer options to a probability. For a fixed property, denote by A the set of all relevant answer options. Also, denote by p_a the probability that an answer $a \in A$ is correct. We calculate expected verification cost when presenting users with an (ordered) list of answer options $\langle a_1, \ldots, a_m \rangle$ where $a_i \in A$.

THEOREM 2. *The expected verification cost for answer options* $\langle a_1, \ldots, a_m \rangle$ *is* $v_p \cdot \sum_{i=1..m}(1 - \sum_{1 \leq j < i} p_{a_i})$.

COROLLARY 2. *Selecting answer options in decreasing order of probability minimizes expected verification cost.*

We illustrate screen design by an example.

EXAMPLE 8. *Consider the query from Example 1. Our goal is to obtain feedback on the "Index" property. Our classifier ranks options in descending order of probability as follows: PGElecProd, PGElecDemand, and PGaccess. Assuming that it takes workers two times longer to look up the correct alternative themselves, compared to validating a given option, we select the first two options to display on screen (and a field to write the correct answer if not shown).*

Finally, we discuss the selection of query properties. Our goal is to select the best *nsc* properties to verify by creating corresponding screens. We define the quality of a property as follows. At any point, we consider a set of likely query translations for a claim. A good property has high pruning power with regards to the current set of candidates. This means that it allows us to discard as many incorrect candidates as possible. The following example illustrates pruning.

EXAMPLE 9. *Assume a worker selects year 2030 in Figure 3. Implicitly, this decision prunes all possible expressions that use more (or less) than one symbol (to represent the year). E.g., we prune formula* `a+1` *since it contains too few placeholders.*

How many query candidates we can prune depends on the correct property value. Depending on the answer we obtain from the fact checkers, more or less queries can be pruned. We do, of course, not know the correct answers when selecting questions. Hence, we define the expected pruning power of a set of properties as follows.

DEFINITION 6. *Given a set* Q *of query candidates, a set* S *of query properties to verify, and trained models* M *predicting a-priori probabilities for possible answers, we define the pruning power* $\mathcal{P}(S, Q, M)$ *as the expected number of queries that are excluded by obtaining answers for* S.

Next, we provide a formula for pruning power, based on simplifying assumptions. For that, we denote by a_s^i the i-th answer option for property $s \in S$ and by $E_s^i \subseteq Q$ queries that are excluded if answer option a_s^i turns out to be correct. We assume independence between pruning probabilities for different query properties. While this assumption is simplifying, it allows us to use simple optimization algorithms (we may consider extensions in the future). Also, we assume that answer options are mutually exclusive (which holds for most, even not for all, query properties).

THEOREM 3. *The pruning power* $\mathcal{P}(S, Q, M)$ *is given by* $\sum_{q \in Q}(1 - \prod_{s \in S} \sum_{i: q \notin E_s^i} \Pr(a_s^i \textit{ correct}|M))$.

Next, we discuss the question of how to find property sets maximizing the above formula. Iterating over all possible property sets is possible but expensive (exponential complexity in the number of properties). Instead, we select properties according to a simple, greedy approach. At each step, we add whichever property maximizes pruning power to the set of selected properties (when comparing properties to add, we calculate pruning power for the union between the new and previously selected properties). We stop once the number of selected properties has reached the threshold determined before. An illustrative example follows.

EXAMPLE 10. *We consider two queries and three properties. Assume property one prunes query one with probability 0.7 and query two with probability 0.3. The corresponding numbers are 0.5 and 0.9 for property two, and 0.1 and 0.95 for property three. We select screens for verifying two properties (i.e.,* $nsc = 2$*). If selecting property one, the expected number of remaining queries is* $0.3+0.7 = 1$. *It is* $0.5+0.1 = 0.6$ *for property two and* $0.9+0.05 = 0.95$ *for property three. Hence, we greedily select property two first. Assuming independence, selecting property one next leads to an expected number of* $0.5 \cdot 0.3 + 0.1 \cdot 0.7 = 0.22$ *queries remaining. Selecting property three yields* $0.5 \cdot 0.9 + 0.1 \cdot 0.05 = 0.455$ *queries. Hence, we select property one next.*

While this algorithm may seem simple, it offers surprisingly strong formal guarantees. Those guarantees are derived from the fact that pruning power is a sub-modular function [36]. We define sub-modularity below.

DEFINITION 7. *A set function* $f : S \mapsto \mathbb{R}$ *is sub-modular if, using* $\Delta_f(S, s) = f(S \cup \{s\}) - f(S)$, *it is* $\Delta_f(S_1, s) \geq \Delta_f(S_2, s)$ *for any* $S_1 \subseteq S_2$.

Intuitively, sub-modularity captures a "diminishing returns" behavior. If adding more elements to a set, the utility of new elements decreases as the set of previous elements grows. The pruning power function is sub-modular as well, according to the following theorem.

Theorem 4. *Pruning power is sub-modular.*

Next, we show that the simple greedy algorithm produces a near-optimal set of questions.

Theorem 5. *Using the greedy algorithm, we select a set of questions that achieve pruning power within factor* $1-1/e$ *of the optimum.*

Finally, we analyze time complexity (denoting by *nsc* the number of screens, by *npr* the number of properties, and by *nqu* the number of query candidates).

Theorem 6. *Finding optimal question sequences for verifying single claims is in* $O(nsc \cdot npr \cdot nqu)$.

5.2 Claim Ordering

We consider two criteria when selecting the next claims to verify. First, we consider the benefit of claim labels for training our classifiers (for automated claim to query translation). Second, we consider the expected verification cost.

The first point relates to prior work on active learning. Here, the goal is generally to select optimal training samples to increase the quality of a learned model. In our case, verified claims correspond to training samples for classifiers that translate claims to queries. We follow the popular heuristic of picking training samples with maximal uncertainty and define the training utility as follows.

Definition 8. *Let* $m \in M$ *be a model predicting specific properties of the query associated with a text claim* c. *We assume that* m *maps each claim to a probability distribution over property values. Denote by* $e(m,c)$ *the entropy of that probability distribution. We define the training utility of* c, $u(c)$ *by averaging over all models (associated with different query properties):* $u(c)=\sum_{m\in M} e(m,c)$.

The second point (verification cost) relates to the cost model discussed in the previous subsection. However, this cost model is incomplete. It neglects the cost of understanding the context in which a certain claim is placed. Intuitively, verifying multiple claims in the same section is faster than verifying claims that are far apart in the input document. Our extended cost model takes this into account. It calculates verification cost for claim batches.

Definition 9. *Denote by* C *a batch of claims for verification. For each claim* $c \in C$, *denote by* $s(c)$ *the section in which this claim is located (instead of sections, a different granularity such as paragraphs can be chosen as well). Denote by* $v(c)$ *the pure claim verification cost for* c *defined in the last subsection. Further, denote by* $r(s)$ *the cost of reading (respectively skimming) section* s. *We define the total (combined verification and skimming) cost for claim batch* C *as the sum of both verification cost over all claims and reading cost over all associated sections:* $t(C) = (\sum_{c\in C} v(c)) + (\sum_{s\in\{s(c)|c\in C\}} r(s))$.

This cost model captures the desired property that verifying claims in the same section is faster. Our approach to claim ordering is based on this model. It is not useful to determine a global claim order before verification starts. We cannot predict how the quality of classifiers (and therefore claim verification cost) will change over time. Instead, we repeatedly select claim batches that are presented to the checkers. Those claim batches are selected based on training utility and the aforementioned cost model. To select claim batches, we solve the following optimization problem.

Definition 10. *Given a set of unverified claims* C, *the goal of claim selection is to select a claim batch* $B \subseteq C$ *such that total cost of* B *remains below a threshold* t_m: $t(B) \leq t_m$. *Additionally, the minimal and maximal batch size is restricted by parameters* b_l *and* b_u: $b_l \leq |B| \leq b_u$. *Under those constraints, the goal is to maximize accumulated training utility* $\sum_{c\in B} u(c)$. *Alternatively, as a variant, we minimize the cost formula* $t(B) - w_u \cdot \sum_{c\in B} u(c)$ *where* w_u *is a weight representing the relative importance of selecting claims with high uncertainty for classifier training.*

Analyzing complexity, we find the following.

Theorem 7. *Claim selection is NP-hard.*

The fact that claim selection is NP-hard justifies the use of sophisticated solver tools. We reduce the problem to integer linear programming. This allows us to apply mature solvers for this standard problem. Next, we discuss how we transform claim selection into integer linear programming.

An integer linear program (ILP) is generally characterized by a set of integer variables, a set of linear constraints, and a (linear) objective function. The goal is to find an assignment from variables to values that minimizes the objective function, while satisfying all constraints.

We introduce binary decision variables of the form cs_i, indicating whether the i-th claim was selected ($cs_i = 1$) or not ($cs_i = 0$). Also, we introduce binary variables of the form sr_j to indicate whether section number j needs to be skimmed or not (to verify the selected claims). Next, we express the constraints of our scenario on those variables. First, we limit the number of selected claims to the range $[b_l, b_u]$ by introducing the linear constraints $b_l \leq \sum_i cs_i \leq b_u$. Next, we represent the constraint that sections of selected claims must be read. We introduce constraints of the form $sr_j \geq cs_i$ if claim i is located within section j. Furthermore, we limit accumulated verification cost of the selected claims by the constraint $(\sum_i cs_i \cdot v(c_i)) + (\sum_j sr_j \cdot r(s_j))$. Finally, we set $-\sum_i cs_i \cdot u(c_i)$ as objective function to minimize. We illustrate the transformation.

Example 11. *We select up to two out of three claims for verification (i.e.,* $b_u = 2$*) with a time limit of 4 minutes. The first two claims are located within the same section while the third one is in a separate section. We have uncertainties 0.3, 0.5, and 0.9 associated with the three claims. Each claim has an estimated verification time of one minute. We estimate one minute for skimming a section as well and set* w_u *to one. We introduce decision variables* cs_1 *to* cs_3, *representing claim selections, and* rs_1 *and* rs_2, *representing skimmed sections. Verifying claims requires skimming their sections, therefore* $cs_1 \geq sr_1$, $cs_2 \geq sr_1$, *and* $cs_3 \geq sc_2$. *Under the constraint* $\sum_{i=1..3} cs_i \leq 2$ *and* $(\sum_{i=1..3} cs_i) + sr_1 + sr_2 \leq 4$, *we minimize* $-0.3 \cdot cs_1 - 0.5 \cdot cs_2 - 0.9 \cdot cs_2$. *The optimal solution selects claims two and three.*

The time complexity for solving a linear program generally depends on the solver and the algorithm it selects to solve a specific instance. However, the number of variables and constraints often correlates with solution time.

Table 2: Ratio of supported claims.

	AggCheck	Tapas	TabFact	Scrutinizer
IEA_L	44.1%	22.7%	36.4%	100%
$C19_L$	22%	22%	38%	100%

Table 3: Verification accuracy on the small datasets.

	Tapas	TabFact	AggCheck	Scrutinizer
$C19_S$	0.64	0.76	0.4	0.80
IEA_S	0.07	0.5	0.5	0.65

Theorem 8. *The size of the ILP problem is in $O(cc \cdot sc)$ where cc is the claim count and sc the section count.*

6. EXPERIMENTS

We evaluated Scrutinizer using real data along four dimensions: (i) the accuracy and efficiency of the query generation from text w.r.t. state of the art methods, (ii) the end-to-end effectiveness of the system in real verification tasks with domain experts, (iii) the effectiveness and efficiency of question scheduling, (iv) the impact of the quality of the user feedback on the results. The code of the system is available at `https://github.com/geokaragiannis/statchecker`.

Datasets. Our experiments are based on the two use cases described in Examples 3 and 4. For the coronavirus claims, we generated 3M true claims starting from the data (we detail this process in the extended version of the paper [25]). This synthetic corpus enabled us to bootstrap the classifiers and we denote it as **$C19_L$**. For testing the system with unseen claims, we analyzed the log of more than 30K claims tested by users on the website. We found that around 60% of the claims are statistical and, among those, we have the datasets to verify 70%. From the claims that the system can check, we manually annotated 55. For the IEA claims, we obtained a document of 661 pages, containing 7901 sentences, and the corresponding corpus of manually checked claims, with check annotations for every claim from three domain experts. The annotations cover 2053 numerical claims, out of which we identified 1539 having a formula that occurs at least five times in the corpus. We denote the resulting dataset as **IEA_L**. After processing the claims, we identify 1791 relations, 830 row indexes, 87 columns, and 413 formulas. Around 50% of the values for all properties appear at most 10 times in the corpus, with the top 5% most frequent formulas appearing at least 8 times.

6.1 Statistical Claim to Query

We compare the performance of our query generator solution against three state of the art systems. The first, AggCheck [23], translates statistical claims into SQL queries for verification. The second, TabFact, exploits pre-trained language models (LMs) to encode the linearized table (taken as input) and a statement into continuous vectors for verification [6]. We fine-tuned the LM with the training examples in our experiments. The third, Tapas, is a question answering system that extends BERT's architecture to encode tables as input [16]. Its pretrained semantic parsing model takes as input a sentence and a table obtaining state-of-the-art accuracy in several datasets. We tried automatically translating claims to questions as pioneered by the ClaimBuster system [14]. The precision was however not satisfactory (e.g., we did not obtain any questions for 7 out of the 20 IEA claims considered next), upper-bounding the precision of even a perfect natural language query interface. Instead, we manually translated claims into questions.

Both natural language baselines have limitations on the input data and on the query space. However, they report top performance in semantic parsing datasets because the majority of examples in these are limited to one (small) relation and simple operations [6, 16]. These limitations are reflected in the smaller percentage of claims that the baselines can handle with their set of functions, as reported in Table 2. One of the reasons why Scrutinizer covers a much larger number of claims is its ability to combine multiple functions in the verification expression. More than 22% of the claims in **IEA_L** has a total of three or more variables and functions (42% for log claims in **$C19_L$**), we observed expressions with up to 14 operators. To enable a comparison against the baselines, we selected a subset of suitable claims from our datasets. We automatically identified explicit claims that require only one relation for the verification and with mathematical operations supported by both baselines. We denote these simpler datasets as **$C19_S$** and **IEA_S**. After the selection, these small datasets contains 0.92% and 15.6% of the claims in the original **$C19_L$** and **IEA_L**, respectively.

All baseline methods require the associated relation as input, so we fed the same information also to Scrutinizer. For Tapas, we limit the input to a sample of 11 tuples, including the one needed to verify the claim, as the system fails with the entire relation as input. For Scrutinizer and TabFact, the training set contains 3000 claims, while Tapas and AggCheck have no training step. Notice that for Scrutinizer we use the top-10 output from the classifiers without relying on the user feedback in this experiment.

Table 3 reports the results of the experiments with all systems on the test claims filtered from the large datasets to be supported by the baselines (25 test claims for **$C19_S$** and 20 for **IEA_S**). We observe that Scrutinizer outperforms all the baselines. Moreover, only AggCheck and Scrutinizer return a query to "explain" their decision, while the others are not immediately interpretable. It is also evident that Coronavirus claims are easier to handle as the number of operations, rows and attributes is smaller than for the IEA claims. For **$C19_S$**, Scrutinizer fails only for claims which requires formulas that it has not learnt yet. In terms of execution times, Scrutinizer took a total of 0.03 seconds to test all **$C19_S$** claims (Tapas 991, TabFact 23) and 0.68 for all **IEA_S** tests (Tapas 944, TabFact 18). For training, Scrutinizer took 108 (**$C19_S$**) and 164 (**IEA_S**) seconds, while TabFact took 4320 and 650, respectively.

6.2 User Study

In this experiment, we involved seven domain experts from the institution to measure the benefit of our system compared to the traditional manual workflow for verification. We trained Scrutinizer with all the annotated statistical claims and randomly selected 43 claims among the ones with the 10 formulas that cover the majority of the claims. As we only have access to the correct version of the claim, we randomly selected 25% of them to inject errors.

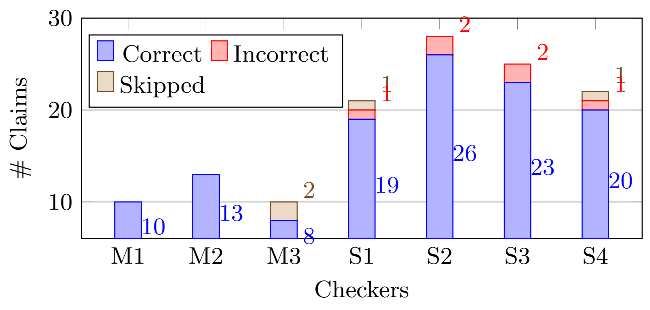


Figure 4: Number of claims verified in 20 minutes by checkers with the manual process (M1–M3) and with our system (S1–S4).

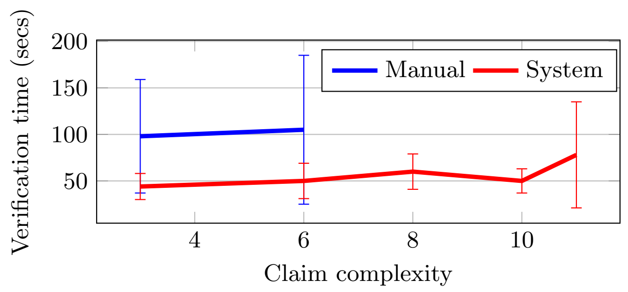


Figure 5: Average time to verify claims of increasing complexity with the Manual and System processes.

Three experts have been randomly assigned to the **Manual** process and the remaining four to the **System**-assisted process. We gave them instructions to execute the test without interruptions and without collaboration. Three claims (two correct, one incorrect) have been used for training on the new process and the remaining 40 for the study. The task given to the experts was to verify as many claims as possible in 20 minutes, given access to their traditional tools in the manual process (spreadsheets and databases) and to our system only in the second case. The order of the claims has been fixed to allow comparison among experts and the time for checking every claims has been registered.

We distinguish three cases: skipped claims, claims that have been correctly labelled, and incorrect decisions. Results for each checker are reported in Figure 4. Considering correct and incorrect checks, on average a user verifies 7 claims manually and 23 claims with SCRUTINIZER in 20 minutes. Users tend to skip a comparable amount of claims in both settings. In the System process, a few claims have been incorrectly checked. Those are all correct claims labelled as incorrect. However, by using simple majority voting over three checkers, the accuracy of the aggregate answers is 100% for both the Manual and the System groups. There was only one claim where verification time using the tool surpassed the traditional manual verification time. After investigating this case, it turned out that this was due to sequential checking. The user consulted a relation different from what we were expecting him to choose. The different relation led him to the correct answer, but such relation was used also in the previous question with the same primary key and attributes values, making this claim very fast to verify.

We also report in Figure 5 average verification time and

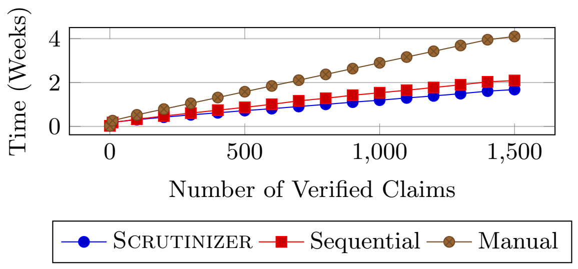


Figure 6: Accumulated verification time over verification period.

standard deviation for the two groups of checkers with claims of increasing complexity. The claim complexity is the sum of the elements in the query to verify it: number of key values, attributes, operations, constants and variables. Checkers using SCRUTINIZER take on average less than half the time to verify claims of the same complexity. We report in the plot claims for which at least two checkers have been able to process it. We are therefore not showing in the plot a checker using SCRUTINIZER who took on average 29 seconds to verify two claims of complexity 14. We remark that for one claim of complexity 6, it took 203 seconds for one of the Manual users to verify it, while for the same claim the slowest System user spent 66 seconds on the same task.

We conducted the study on a laptop (1.80GHz x 8 i7 CPU, 32 GB of memory). For any claim, testing a classifier took less than 0.2 seconds and query generation took less than half a second (0.35 seconds on average).

6.3 Simulation

In the previous subsection, we have demonstrated that SCRUTINIZER decreases verification overheads for single claim batches. Next, we study the efficiency of SCRUTINIZER when verifying entire reports. Verification time for entire reports is typically in the order of months for IEA. Hence, we cannot use another user study. Instead, we created a simulator, based on the results of our initial user study. We simulate the verification of the 2018 IEA world energy outlook report, using the original claims and original data. We assume a team of three fact checkers (which is typical for IEA). We simulate a "cold start" scenario, meaning that our classifiers have no initial training data. Instead, they use claim labels provided by simulated fact checkers. This corresponds to the worst case for our system. It represents a scenario in which the very first version of a new report is received and verified. Our model for verification time per claim is based on time measured in the user study. It takes into account reduced verification overheads once proposed query fragments are accurate. We compare three baselines. First, we consider manual verification ("Manual") which is the current default. Each claim is verified without any computational support. Second, we consider a simplified version of SCRUTINIZER. This version ("Sequential") does not optimally reorder claims, as described in Section 5.2, but verifies them sequentially (i.e., in document order) instead. We compare those two approaches against the SCRUTINIZER system. For Sequential and SCRUTINIZER, we assume that ten answer options are shown per property. For SCRUTINIZER, we use claim batches of size 100, after which we retrain classifiers and select the next claims to verify via ILP. Our simulator is

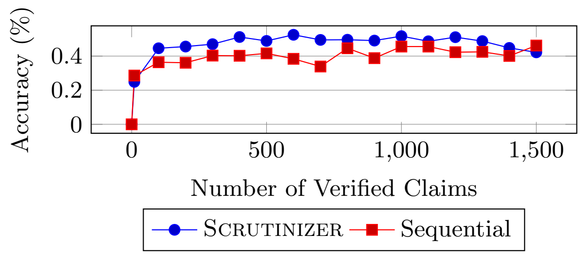


Figure 7: Evolution of SCRUTINIZER and sequential average accuracy over verification period.

implemented in Python 3, using Gurobi 9.0.1 as ILP solver. Experiments were executed on a MacBook Pro with 2.4 GHz Intel Core i5 processor and 8 GB memory.

Table 4: Summary of simulation results.

	Manual	**Sequential**	**Scrut.**
Time (Weeks)	4.1	2.1	1.7
% Savings	-	49%	59%
Avg. Accuracy	-	40%	47%
Max Accuracy	-	46%	53%
Comp. (Mins)	-	14	28

Table 4 summarizes simulation results. We report total verification time for all three fact checkers, assuming an eight hours work day and a five day week. We make the following observations. First, using SCRUTINIZER reduces verification overheads by more than factor two (circa 60%). This is consistent with the results of our user study. At the same time, it is remarkable since we consider a cold start scenario. The results show that, given a sufficiently large document to verify, the initial warmup period of the classifiers does not impact overall performance by too much. Second, we observe a positive impact due to claim ordering. While using SCRUTINIZER without that feature is still helpful, cost savings increase when claims are systematically prioritized. We performed this evaluation also with classifiers with lower accuracy and in all experiments SCRUTINIZER took less overall time than the baseline. Table 4 shows that, in the latter case, the average (and maximal) quality of classification over the entire period improves. Figure 6 shows that SCRUTINIZER and the sequential baseline are near-equivalent at the beginning of the classification process. Claim ordering pays off more and more as verification proceeds. At the same time, computational overheads are negligible for all compared systems. SCRUTINIZER spends 15 minutes in total to plan optimal question sequences, and for selecting optimal claims via ILP. The remaining 13 minutes are due to retraining classifiers.

Figure 7 analyzes classifier accuracy as a function of verification time. SCRUTINIZER has initially a lower accuracy as it selects claims that have low classifier confidence (therefore useful for learning) but are relatively cheap to verify. Soon, active learning starts paying off and the accuracy of SCRUTINIZER dominates the baseline. SCRUTINIZER postpones verifying particularly expensive claims that are associated with low classifier confidence. Once running out of other options, those claims are verified at the very end (leading to a drop in accuracy).

Figure 8 shows accuracy for SCRUTINIZER (with claim ordering) according to classifier type. The effect discussed in the last paragraph (a steep increase followed by a drop towards the end) still hold when considering classifiers separately. Further, we notice that certain properties are harder to infer from text. For instance, inferring row indices is among the hardest classification tasks. This is intuitive as the classification domain (i.e., number of rows) is typically larger than for other classifiers (e.g., columns).

Finally, in Figure 9, we analyze accuracy for the top-k labels and for different classifiers. In most cases, classifiers reach most of their potential with the first 10 entries.

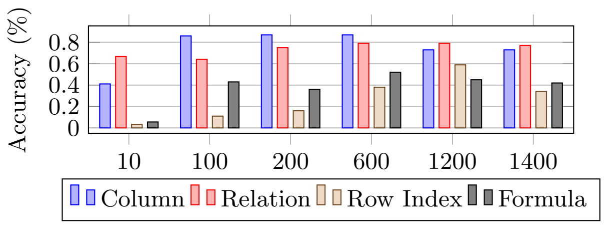


Figure 8: Evolution of classifier accuracy over verification period.

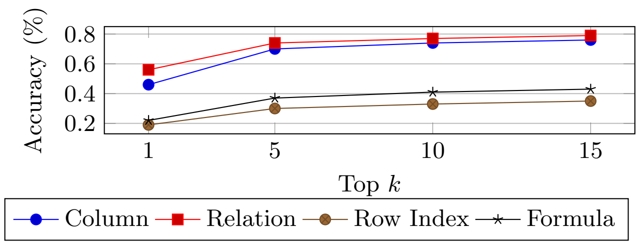


Figure 9: Top k accuracy for different SCRUTINIZER classifiers as a function of k.

6.4 Quality of User Feedback

We measure the impact of the quality of the annotations in three experiments: by injecting synthetic errors, by varying the size of the training data, and by training the models with the noisy examples gathered from the Web users of the system. We then repeat each experiment three times and report the average accuracy. We report the results for $k = 5$, as this is the default value in our experiments.

We start with the study of how the label error rate in the $\mathbf{IEA_L}$ training data affects the classifier accuracy. For each property, we randomly split the annotated claims into training and test sets (0.9/0.1 ratio). We then select a percentage of training claims at random and change their target label with one of the possible labels (minus the correct one). Figure 10 shows how accuracy varies as a function of the error rate for top-5 predictions. With an increasing number of mislabelled data points, all classifiers show decreasing accuracy. However, even with up to 30% error rate, the results are close to the case without noise. In our experiments, IEA checkers error rate is below 10% and even Web site users do not exceed the 30% rate. Finally, as expected, the accuracy gets close to random guessing for very high ratio of error injection (0.9) and classifiers with a small number of possi-

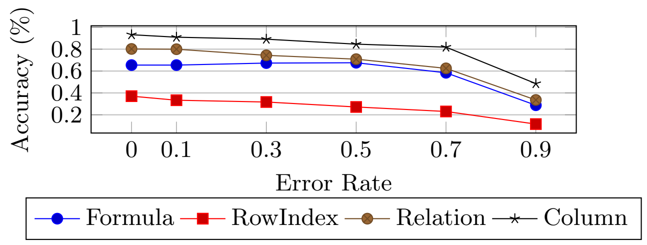


Figure 10: Classifier accuracy as a function of the percentage of erroneous training claims.

Table 5: Accuracy on $C19_L$ vs. training data quality.

	Relation	Column	Row Index
Web crowd	0.84	0.04	0.74
Generated	1	0.98	0.96

ble outputs perform better at any rate. We observe similar patterns also with top-1 results.

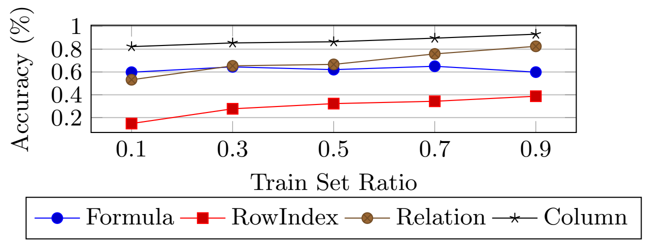


Figure 11: Accuracy versus training set size.

In a second experiment, we vary the train/test ratio by sampling increasing portions from the dataset and testing on the others. Figure 11 shows how properties with high number of classes (e.g., row_index) perform poorly when the training data is small compared to properties with a low number of classes (e.g., column).

Finally, we report on an experiment conducted with classifiers trained with the 1388 answers collected from our Web interface for the test dataset of $\mathbf{C19_L}$. Table 5 shows that, even in a very noisy setting, without majority voting to filter out incorrect answers, the classifiers for Relation and Row Index perform well, but accuracy for the Column classifier is very low. In $\mathbf{C19_L}$, the number of classes when classifying relations, rows, and columns, are all comparable. However, the semantics of claim text fragments tends to be time-sensitive for the column classifier alone (e.g., semantics of the expression “this month”). If re-training the column classifier each month (using labels collected in that month alone), accuracy varies between 0.12 to 0.49 for the months January to June. Detailed results can be found in the extended technical report [25]. While we did not collect user answers for the formula, we manually annotated 40 real examples from the log for the Formula classifier. We picked two formulas that were not in the synthetic training data, i.e., $max(a)$ and $max(a/b)$, with 20 labels each. We then incrementally added them to the training data and tested on the remaining claims. The results show that the system is able to learn the two new formulas after 12 examples per class. Also, in a small experiment described in more detail in the technical report [25], we verified that classifier confidence is significantly higher for claims translating to queries that match the current template (confidence of 0.39 versus 0.94). We plan to integrate mechanisms to automatically detect required expansions of the current template.

7. RELATED WORK

Scrutinizer targets the verification of numerical claims from raw data. Our scope differs from prior work not focused on verification (e.g., work on identifying check-worthy claims [14, 22] or on studying misinformation spread [38, 9]), from prior work verifying claims given as logical formulae (e.g., for verifying claim robustness [46]), from work on verifying claims from different types of data (e.g., Web documents [34, 42, 43] or databases of previously checked claims [41, 27, 15]), and from work focused on different claim types (e.g., claims associating entities with non-numerical properties [7, 39, 12, 11, 19]). Scrutinizer connects to work on explainable fact checking [30, 10, 2] as it verifies claims via queries whose structure can be explained to users.

Scrutinizer relates to prior efforts on data-driven analysis of statistical claims. BriQ can map one explicit claim to the input dataset and supports only 6 operations for a single user [20]. StatSearch considers a corpus of relations, but supports only SP queries (no functions) and can search (not verify) one explicit claim with a single user [5]. The closest work is the AggChecker system [23] in that it translates statistical claims into SQL queries for verification. Scrutinizer supports however a richer query model (see Section 6), it assumes verification by a crowd of fact checkers (instead of single users), and it supports verification of long documents (as opposed to short texts) by features such as question reordering for active learning. Prior work on mixed-initiative fact checking [13] does not translate claims to SQL queries.

We translate text to SQL queries, thereby connecting to prior work on natural language query interfaces (NLQI) [1, 32, 47, 31, 37, 44, 21] and learning query interfaces [21, 33] in general. Our scenario differs in multiple ways. First, most prior work assumes that domain-specific training data is either already available [47] or not required [31, 32, 1]. We propose methods for efficiently acquiring domain-specific training data and show experimentally that generic methods do not work for our use cases [6, 16]. Second, prior work [1, 32, 47, 31, 37, 44, 21, 33] typically considers translation of single queries. Here, we consider large claim collections. This motivates claim context as a feature (see Section 4) or claim selection as a planning problem (see Section 5). Finally, unlike prior work [21], we decompose query translation into sequences of simple questions (see Section 5.1) as writing entire SQL queries is beyond the capabilities of domain experts employed for verification.

8. CONCLUSION

We introduced Scrutinizer, the first system for crowdsourcing the verification of general statistical claims. Our solution effectively minimizes the amount of work needed by a group of domain experts to verify textual claims in a document. We find that professional fact checkers from IEA verify claims twice as fast using Scrutinizer, compared to their traditional workflow.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] N. Ahmadi, J. Lee, P. Papotti, and M. Saeed. Explainable fact checking with probabilistic answer set programming. In *Conference for Truth and Trust Online (TTO)*, 2019.

[3] J. S. Ash, M. Berg, and E. Coiera. Some unintended consequences of information technology in health care: the nature of patient care information system-related errors. *Journal of the American Medical Informatics Association*, 11(2):104–112, 2004.

[4] J. Brainard and P. R. Hunter. Misinformation making a disease outbreak worse : outcomes compared for influenza , monkeypox , and norovirus. *Simulation: transactions of the society for modeling, and simulation*, 96(4):365–374, 2020.

[5] T. D. Cao, I. Manolescu, and X. Tannier. Searching for truth in a database of statistics. In *WebDB*, pages 4:1–4:6, 2018.

[6] W. Chen, H. Wang, J. Chen, Y. Zhang, H. Wang, S. Li, X. Zhou, and W. Y. Wang. Tabfact: A large-scale dataset for table-based fact verification. In *ICLR*, 2020.

[7] G. L. Ciampaglia, P. Shiralkar, L. M. Rocha, J. Bollen, F. Menczer, and A. Flammini. Computational fact checking from knowledge networks. *PloS one*, 10(6):e0128193, 2015.

[8] T. Claveau. CoronaCheck : démêler le vrai du faux sur l'épidémie de covid-19. I'M Tech (`https://bit.ly/30EIk41`), 2020.

[9] E. Ferrara, O. Varol, C. A. Davis, F. Menczer, and A. Flammini. The rise of social bots. *Commun. ACM*, 59(7):96–104, 2016.

[10] M. H. Gad-Elrab, D. Stepanova, J. Urbani, and G. Weikum. Exfakt: A framework for explaining facts over knowledge graphs and text. In *WSDM*, pages 87–95, 2019.

[11] M. Gardner and T. M. Mitchell. Efficient and expressive knowledge base completion using subgraph feature extraction. In *EMNLP*, pages 1488–1498, 2015.

[12] M. Gardner, P. P. Talukdar, J. Krishnamurthy, and T. Mitchell. Incorporating vector space similarity in random walk inference over knowledge bases. In *EMNLP*, 2014.

[13] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe It or Not: Adding Belief Annotations to Databases. *Psychological science : a journal of the American Psychological Society / APS*, 16(7):17, 2009.

[14] N. Hassan, F. Arslan, C. Li, and M. Tremayne. Toward automated fact-checking: Detecting check-worthy factual claims by claimbuster. In *KDD*, 2017.

[15] N. Hassan, G. Zhang, F. Arslan, J. Caraballo, D. Jimenez, S. Gawsane, S. Hasan, M. Joseph, A. Kulkarni, A. K. Nayak, V. Sable, C. Li, and M. Tremayne. Claimbuster: The first-ever end-to-end fact-checking system. *PVLDB*, 10(12):1945–1948, 2017.

[16] J. Herzig, P. K. Nowak, T. Müller, F. Piccinno, and J. M. Eisenschlos. TAPAS: weakly supervised table parsing via pre-training. In *ACL*, 2020.

[17] J. Horwitz. Facebook's fact checkers fight surge in fake Coronavirus claims, 2020.

[18] M. Hosseini, M. Hilhorst, I. de Beaufort, and D. Fanelli. Doing the right thing: A qualitative investigation of retractions due to unintentional error. *Science and engineering ethics*, 24(1):189–206, 2018.

[19] V. Huynh and P. Papotti. Buckle: Evaluating fact checking algorithms built on knowledge bases. *PVLDB*, 12(12):1798–1801, 2019.

[20] Y. Ibrahim, M. Riedewald, G. Weikum, and D. Zeinalipour-Yazti. Bridging quantities in tables and text. In *ICDE*, pages 1010–1021, 2019.

[21] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, pages 963–973, 2017.

[22] I. Jaradat, P. Gencheva, A. Barrón-Cedeño, L. Màrquez, and P. Nakov. Claimrank: Detecting check-worthy claims in arabic and english. In *NAACL-HTL*, pages 26–30, 2018.

[23] S. Jo, I. Trummer, W. Yu, X. Wang, C. Yu, D. Liu, and N. Mehta. Verifying text summaries of relational data sets. In *SIGMOD*, pages 299–316, 2019.

[24] G. Karagiannis, M. Saeed, P. Papotti, and I. Trummer. CoronaCheck. `https://coronacheck.eurecom.fr/`, 2020.

[25] G. Karagiannis, M. Saeed, P. Papotti, and I. Trummer. Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification (Technical Report). `https://github.com/geokaragiannis/statchecker`, 2020.

[26] G. Karagiannis, M. Saeed, P. Papotti, and I. Trummer. Scrutinizer: Fact checking statistical claims (demo). *PVLDB*, 13(12), 2020.

[27] G. Karagiannis, I. Trummer, S. Jo, S. Khandelwal, X. Wang, and C. Yu. Mining an "anti-knowledge base" from wikipedia updates with applications to fact checking and beyond. *PVLDB*, 13(4):561–573, 2019.

[28] G. Lample and A. Conneau. Cross-lingual language model pretraining. *http://arxiv.org/abs/1901.07291*, 2019.

[29] Latvian Public Broadcasting. 130 countries sign up for infodemic pledge, 2020.

[30] J. Leblay. A declarative approach to data-driven fact checking. In *AAAI*, pages 147–153, 2017.

[31] F. Li and H. Jagadish. Understanding natural language queries over relational databases. *SIGMOD Record*, 45(1):6–13, 2016.

[32] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

[33] B. McCamish, V. Ghadakchi, A. Termehchy, L. Huang, and B. Touri. How do humans and data systems establish a common query language? *SIGMOD Record*, 48(1):51–58, 2019.

[34] T. Mihaylova, P. Nakov, L. Màrquez, A. Barrón-Cedeño, M. Mohtarami, G. Karadzhov, and J. R. Glass. Fact checking in community forums. In *AAAI*, pages 5309–5316, 2018.

[35] B. Milucci. Tutelare le persone dal rischio infodemia:

nasce un sito contro le fake news. Corriere della Sera (`https://bit.ly/3hmvP2S`), 2020.

[36] G. Nemhauser and L. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Mathematics of Operations Research*, 3(3):177–188, 1978.

[37] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Ozcan. ATHENA: An ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[38] W. Sherchan, S. Nepal, and C. Paris. A survey of trust in social networks. *ACM Comput. Surv.*, 45(4):47:1–47:33, 2013.

[39] B. Shi and T. Weninger. Discriminative predicate path mining for fact checking in knowledge graphs. *Knowledge-Based Systems*, 104:123–133, 2016.

[40] A. Spadaro. CoronaCheck and FakeNews. La Civiltà Cattolica (`https://bit.ly/30IfIXJ`), 2020.

[41] A. Tchechmedjiev, P. Fafalios, K. Boland, M. Gasquet, M. Zloch, B. Zapilko, S. Dietze, and K. Todorov. ClaimsKG: A knowledge graph of fact-checked claims. In *ISWC*, pages 309–324, 2019.

[42] J. Thorne and A. Vlachos. Automated fact checking: Task formulations, methods and future directions. In *COLING*, pages 3346–3359, 2018.

[43] X. Wang, C. Yu, S. Baumgartner, and F. Korn. Relevant document discovery for fact-checking articles. In *The Web Conf.*, pages 525–533, 2018.

[44] N. Weir, A. Crotty, A. Galakatos, A. Ilkhechi, S. Ramaswamy, R. Bhushan, U. Cetintemel, P. Utama, N. Geisler, B. Hättasch, S. Eger, and C. Binnig. DBPal: Weak Supervision for Learning a Natural Language Interface to Databases. pages 1–4, 2019.

[45] Wikipedia. Misinformation related to the COVID-19 pandemic, 2020.

[46] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu. Toward computational fact-checking. *PVLDB*, 7(7):589–600, 2014.

[47] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv preprint 1709.00103*, pages 1–12, 2017.