ABSTRACT
Spreadsheets are widely used for table manipulation and presentation. Stylistic formatting of these tables is an important property for presentation and analysis. As a result, popular spreadsheet software, such as Excel, supports automatically formatting tables based on rules. Unfortunately, writing such formatting rules can be challenging for users as it requires knowledge of the underlying rule language and data logic. We present Cornet, a system that tackles the novel problem of automatically learning such formatting rules from user-provided formatted cells. Cornet takes inspiration from advances in inductive programming and combines symbolic rule enumeration with a neural ranker to learn conditional formatting rules. To motivate and evaluate our approach, we extracted tables with over 450K unique formatting rules from a corpus of over 1.8M real worksheets. Since we are the first to introduce the task of automatically learning conditional formatting rules, we compare Cornet to a wide range of symbolic and neural baselines adapted from related domains. Our results show that Cornet accurately learns rules across varying setups. Additionally, we show that in some cases Cornet can find rules that are shorter than those written by users and can also discover rules in spreadsheets that users have manually formatted. Furthermore, we present two case studies investigating the generality of our approach by extending Cornet to related data tasks (e.g., filtering) and generalizing to conditional formatting over multiple columns.

1 INTRODUCTION
Spreadsheets are the most common table manipulation software, with around a billion monthly active users [26]. Formatting the style of cells is a fundamental and frequently used visual aid to better display, emphasize or distinguish data points in a spreadsheet. By analyzing a large public spreadsheet corpus [2, 14] we found that close to 25% of spreadsheets use some form of cell formatting. Conditional formatting (CF) is a feature that automates table formatting based on user-defined rules. It is available in all major spreadsheet manipulation tools like Microsoft Excel, Google Sheets and Apple Numbers. All these tools support predefined templates for popular rules, such as cell value is greater than a specific value. In Excel and Sheets, users can also author a custom boolean-valued formula to format cells. We find that 18% of spreadsheets in a large public spreadsheet corpus [2] use conditional formatting.

We present the Cornet (Conditional ORNamentation by Examples in Tables) system that automatically generates a formatting rule from examples of formatted cells. Cornet takes a small number of user-formatted cells as input to learn a likely formatting rule that generalizes to other cells in the column. For example, in Figure 1, after the user formats only two cells, Cornet suggests the intended rule without exposing the user to the underlying rule language.

The complexity associated with manually writing conditional formatting rules is reflected in the volume of related help forum posts on the topic. As of June 2022, more than 10,000 conditional formatting related questions were posted on the Excel tech help community alone [10]. By analyzing these posts we discovered multiple factors that contribute to the difficulty of authoring such rules manually. These factors range from fundamental logic challenges in rules to the lack of user interface support in existing platforms. We outline the most prominent factors.

First, many users are unaware of the CF feature and manually format spreadsheets, which can be highly inefficient and introduce errors. Second, even basic rule authoring requires that the user understand the syntax and logic behind conditional formatting, the predefined templates, and potentially the formula language to write more complex rules. Writing such formulas is further complicated by the absence of data type validation. For example, a user can choose numerical comparison on columns with text. This results in wrong formatting or no formatting at all. Third, when users do succeed in writing correct rules, they often write formulas that are more complex than needed to capture their intended logic.
which are given less weight during learning. To compensate for
Cornet real user tasks from public Excel spreadsheets. We use 80K of these
learned weights) and a slightly better but more costly neural ranker.

two rankers: a lightweight ranker based on manual features (and
net similarity between cells computed over the same properties that
Cornet in this approximation,
tiple rules that satisfy them. To handle potentially noisy examples
search space, some examples can be noisy and there can still be mul-
substantial cause of incorrect rules. Finally, we found that when
them and there is little information to constrain the search space.
Figure 1: Adding a CF Rule in Excel: User needs to select CF
in the “Styles” portion of the home ribbon menu and select
add rule from the drop-down menu. The image depicts: ①
the dialog box to add a new CF rule; ② the rule the user
needs to write; ③ the resulting formatted column. After the
user formats two cells, Cornet can automatically suggest
the intended CF rule for the user.

Cornet is designed to address each of these concerns. First,
Cornet can learn conditional formatting rules from as few as one
example, opening up the possibility of dynamically suggesting rules
to users. Because Cornet can learn rules for a wide range of tasks—
about 90% of our benchmarks—users can rely on Cornet to cover
a substantial amount of their conditional formatting needs. Cornet
only learns rules specific to the data type at hand, removing a
substantial cause of incorrect rules. Finally, we found that when
users write complex custom rules, Cornet can learn a shorter rule
in approximately 60% of the cases.

The key technical challenge in learning conditional formatting
rules by example is that few boolean examples (formatted or not)
result in a very weak specification: many candidate rules will satisfy
them and there is little information to constrain the search space.
To tackle this, Cornet first approximates a stronger specification
by considering other cells using semi-supervised clustering and
assigning them to formatted and unformatted groups—providing a
richer signal for pruning the search space. This clustering uses a
similarity between cells computed over the same properties that
are used to learn candidate rules.

While the approximate specification allows us to navigate the
search space, some examples can be noisy and there can still be mul-
tiple rules that satisfy them. To handle potentially noisy examples
in this approximation, Cornet uses the concept of soft examples,
which are given less weight during learning. To compensate for
the greediness of the learner, in addition to soft examples, Cornet
learns multiple candidate rules and ranks them. We propose
two rankers: a lightweight ranker based on manual features (and
learned weights) and a slightly better but more costly neural ranker.

To train and evaluate Cornet, we created a benchmark of 105K
real user tasks from public Excel spreadsheets. We use 80K of these
tasks for training and 25K for evaluation. We found that Cornet
can learn CF rules from as few as two examples and outperforms ex-
isting and custom symbolic and neural baselines that were adapted
for this task. In addition, we perform two case studies where we
explore how Cornet generalizes to other data tasks and how it
generalizes to conditional formatting over multiple columns.

This paper makes the following key contributions:

- Based on the observation that users often struggle to format
  tabular data, we introduce the novel problem of learning
  conditional formatting (CF) rules from examples.
- We propose Cornet, a system that learns CF rules from
  examples over tabular data.
- We create a dataset of 105K real formatting tasks extracted
  from public spreadsheets. We release this dataset to encour-
  age future research.1
- We evaluate Cornet extensively on 25K CF tasks, compare
  to existing and custom baselines, and show that Cornet
  outperforms both symbolic and neural baselines by 20% on
  our benchmark.

2 PROBLEM DEFINITION
Let $C = \{c_i\}_{i=1}^n$ be a column of $n$ cells with each cell $c_i \in C$
represented by a tuple $(v_i, t_i)$ of its value $v_i \in \mathcal{V}$ and its annotated
type $t_i \in \mathcal{T}$. In this paper, we consider string, number, and date as
possible types—these are available in most spreadsheet software.
We associate a format identifier $f_i \in \mathcal{N}_0$ (or simply format) with
each cell, which corresponds to a unique combination of formatting
choices made by the user. A special identifier $f_L = 0$ is reserved
for cells without any specific formatting. In this paper, we consider

A conditional formatting rule (or simply rule) is a function $r :$ $C \rightarrow \mathcal{N}_0$ that maps a cell to a formatting identifier. Given a column
$C$ and specification, a rule $r$ that satisfies the specification is one
such that $r(c_i) = f_i$ for all $c_i \in C$

EXAMPLE 1. In Figure 2, which will serve as a running example,
colored cells have $f_1$ and all other cells have $f_2$. As format identifiers,
where $f_1$ corresponds to 
[cell color: #bead4, font color: default, font size: 12, border: default].

A conditional formatting rule (or simply rule) is a function $r :$ $C \rightarrow \mathcal{N}_0$ that maps a cell to a formatting identifier. Given a column
$C$ and specification, a rule $r$ that satisfies the specification is one
such that $r(c_i) = f_i$ for all $c_i \in C$

EXAMPLE 2. Returning to Figure 2, the formatting can be described
by the following rule:

$$r_1(c) = \begin{cases} f_1 & \text{c starts with “RW” and does not end with “T”} \\ f_L & \text{otherwise} \end{cases}$$

Let $C_\ast = \{c_i \mid c_i \in C, f_i \neq f_L\}$ be the cells with formatting
applied. The goal of automatic conditional formatting by example is

to find $r$ given only a small, observed subset $C_{\text{obs}} \subset C_\ast$. Throughout
this work we will refer to the elements of $C_{\text{obs}}$ as formatted examples.
Any cell in $C \setminus C_{\text{obs}}$ is considered unlabelled, which includes all
unformatted cells.

EXAMPLE 3. In Figure 2, the user has provided two examples and
$C_{\text{obs}} = \{RW-187, RW-159\}$. The rest of the cells in the column are
unlabeled. The goal is to learn rule $r_1$ from Example 2.

1To adhere to compliance requirements, we release the data as URLs to slx files, that
can be downloaded, and scripts to generate benchmarks from the downloaded files.
In the remainder of this paper, we will consider the case where there is only one formatting identifier for simplicity. We then do not have to make assumptions about the order in which a user provides examples for different formats and the order in which the rules for different formats should be applied. Note that we can generalize the single format case to \( k \) different formatting identifiers by simply solving \( k \) different formatting by example problems, such that when learning the rule for a format identifier \( f_j \), all other format identifiers are treated as \( f_k \). This approach to multiple formats is closely aligned with popular spreadsheet software, where each format is applied using a different rule. Different rules can overlap and the order in which they are applied, as chosen by the user, determines the final color for each cell. As only 0.63% of rules in our corpus format overlapping cells, we do not consider overlapping rules and their order.

3 APPROACH

This section describes how CORNET learns formatting rules from a small number of user-provided examples. Figure 2 shows a schematic overview of CORNET’s approach. Step \( 1 \) enumerates properties of cells as predicates. Step \( 2 \) approximates the expected output using semi-supervised clustering. CORNET then iteratively generates rules that match this output in step \( 3 \), and ranks them in step \( 4 \). The following sections describe challenges and solutions for each step.

3.1 Predicate Generation

CORNET uses cell properties to reason about the target formatting. This step enumerates a set of these properties that hold for a non-empty proper subset of the cells of the given column. Each property is encoded as a predicate—a boolean-valued function that takes a cell \( c \) along with zero or more additional arguments and returns true if the property that it describes holds for the cell \( c \). To avoid type errors, all predicates are assigned a type and they only match cells of their type. Supported predicates are shown in Table 1. The predicates for CORNET have been chosen based on formatting rule operations supported by popular spreadsheet software.

For each predicate, we need to generate constant values for all additional (not \( c \)) arguments. Given a column of cells and a predicate, the goal is to initialize each additional argument to a constant value such that the predicate returns true for a non-empty proper subset of cells in the column. We do this by generating a set of constant values for each type, derived from the column values or common constants, and instantiating each predicate with combinations of constants of the appropriate types. Table 2 shows an overview of how the constant values are generated for predicates of each type.

**Table 1: Supported predicates and their arguments for each datatype (top) and also general (bottom).** The \( d \) argument in datetime predicates determines which part of the date is compared—day, month, year, or weekday. For example, \( \text{greater}(c, d) \) matches datetime cells with a date in March or later for any year.

**Table 2: Overview of constants for concretizing predicates of each type.** For example, we generate constants for text predicates from two token sources: delimiter-based splitting and prefixes.

3.2 Semi-supervised Clustering

Rather than immediately combine predicates into rules, we first predict the expected output of possible rules on the unformatted cells by clustering. There are \( 2^n \) ways to cluster a column of \( n \) cells in two clusters (formatted and unformatted) but \( 2^{2n} \) unique rules can be written with \( p \) predicates.\(^2\) In other words, many rules yield the same clustering. Clustering then allows us to leverage the relatively small search space of output configurations to find programs that generalize to similar cells. CORNET biases the predicted output towards the generated predicates by using their output to compute

\(^2\)Conjunction and disjunction can represent all boolean functions over \( p \) inputs, and there are \( 2^p \) such functions.
Initialize

1. TextStartsWith("RW")
   TextStartsWith("RS")
   TextEquals("RW-187")
   TextEndsWith("T")
   ...

2. RSRC Level
   POC
   RW-187 121
   RS-762 287
   RW-131-T 134
   RS-159 302
   RS-452-T 132
   RS-427 287
   RS-429 345
   RW-174 263
   RW-195 345
   RW-160 121
   RS-233 264

3. Predicate
   Generation
   Semi-Supervised
   Clustering
   Tree based Rule
   Enumeration
   Neural
   Ranking
   Learned
   Conditional
   Formatting Rule

4. 1 AND(TextStartsWith("RW"),
       NOT(TextEndsWith("T")))
   2 AND(TextStartsWith("RS"),
       NOT(TextEndsWith("T")))
   3 AND(TextStartsWith("RW"),
       NOT(TextEndsWith("T")))
   4 AND(TextStartsWith("RW"),
       NOT(TextEndsWith("T")))
   5 AND(TextStartsWith("RW"),
       NOT(TextEndsWith("T")))

Figure 2: CORNET architecture illustrated through the example case from Figure 1: (1) predicate generation for all cells in the table, (2) semi-supervised clustering using examples and other cells to address the challenge of unlabeled cells, (3) enumerating rules based on the clustering using multiple decision trees, (4) neural ranker to score generated rules, and (5) final learned conditional formatting rule.

Figure 3: Schematic overview of the initialize and reassign steps for clustering cells into formatted (f), unformatted (L) and unassigned (u) clusters. Only initially unassigned cells are reassigned and obtain a noisy label when this happens.

3.3 Candidate Rule Enumeration

After clustering, we have a target formatting label $\hat{f}_i$ for each $c_i$ in $C$. We now learn a rule $r$ such that $r(f(c)) = \hat{f}_i$ for all $c_i \in C_{obs}$ (hard examples) and $\sum r(f(c)) = \hat{f}_i$ for $c_i \notin C_{obs}$ is greedily maximized (soft examples). We define the space of rules and a search procedure in the following two subsections.

3.3.1 Predicates to Rules. A rule in CORNET for a column $C$ and format $f$ is a function $r_f : C \rightarrow \mathbb{B}$ that takes a cell and returns whether the cell should be formatted as $f$ or not. CORNET supports $r_f$ that can be built as a propositional formula in disjunctive normal form over predicates. In other words, every $r_f$ is of the form

$$ (p_1(c) \land p_2(c) \land \ldots \lor (p_j(c) \land p_{j+1}(c) \land \ldots \lor \ldots$$

with $p_i$ a generated predicate or its negation. Our goal is to strike a balance between expressiveness and simplicity.

3.3.2 Enumerating Rules. We greedily enumerate candidate rules by iteratively learning decision trees that predict the noisy label $\hat{f}_i$ for each $c_i$ from their predicate outputs. Each decision tree then corresponds to a rule in disjunctive normal form [3]. We identify and address three challenges: variety in rules, simplicity of rules, and coping with noisy labels. To induce variety in rules, during iteration we remove the predicate associated with the root node of the current tree from possible predicates, so any subsequent trees learned cannot use it. To induce simplicity of rules, we limit the depth of learned decision trees to $\lambda_n$ or fewer nodes. To cope with our noisy (cluster-based) labels, we 1) weigh labeled cells twice as
much as unlabeled ones to bias our tree towards perfectly classifying user-provided examples (and we perform a check after the tree is learned to ensure this property), and 2) we allow misclassifications for soft-labeled (i.e. originally unassigned) cells by allowing accuracy to fall up to \( \lambda_a \). This learning procedure is schematically shown in Figure 4.

3.3.3 Setting parameters. \( \lambda_n \) limits the complexity of generated rules. We set \( \lambda_n = 10 \) as only 17 rules in our 105K corpus had more than 10 predicates. \( \lambda_a \) filters rules with lower accuracy. We set \( \lambda_a = 0.8 \) as CF rules tend to have lower noise tolerance.

![Figure 4: Schematic overview of iterative rule learning. Steps 2 until 4 are repeated as long as the decision tree achieves the desired accuracy and there are features remaining.](image)

3.4 Candidate Rule Ranking

To choose a final rule from candidates generated by the iterative learner we rank candidates. Prior work has proposed ranking programs based on output features [27] or rule features [9]. We build on these approaches and develop a neural ranker that combines information from both types of features.

Information about the rule is captured by handpicked features: depth of rule, count of predicates used, number of arguments, mean length of arguments, number of cells and data type of column, percentage of cells that satisfy the rule, accuracy with respect to approximated labels (\( \hat{f} \)) and source of constant arguments (cell value, statistical value, popular constant, delimiter or trie).

Information about the column data is captured by turning the column into a sequence of words and using a pre-trained language model (BERT) [6] to obtain cell-level embeddings. These embeddings are augmented with information about the execution of the rule through cross-attention [21].

Both the rule and data representations previously described are concatenated and passed to a linear layer with sigmoid activation to produce a single score. This score thus combines both syntactic (rule) and semantic (data and execution) information. Figure 5 shows an overview of our ranking architecture.

![Figure 5: Ranking model: 1 model inputs - rule to be scored and the corresponding data column; 2 the column encoding pools BERT token embeddings, passes them through cross attention with the rule's execution outputs (formatted or not), and then through a linear layer; and 3 the resulting embedding is concatenated with manually-engineered rule features and fed into a final linear layer which outputs the score after applying a sigmoid activation.](image)

4 BASELINES

As we are the first to introduce the problem of learning conditional formatting rules from examples, there are no existing systems that tackle this task. We therefore adapt a variety of approaches related to this problem. Six approaches are symbolic, five of which are able to generate rules. Three neural approaches cast conditional formatting as cell classification and we consider different baseline models and cross-attention mechanisms. The following sections describe these baselines in more detail. We focus on the case where we have a single format identifier.

4.1 Symbolic

4.1.1 Decision trees. We fit a decision tree with formatted and unformatted cells as positive and negative examples, respectively. We consider two variations of encoding cells. In the first one, raw cell values are passed to the decision tree, where text columns are categorically encoded. This encoding does not allow learning rules that involve partial strings, summary statistics for numbers or date parts. In the second encoding, we therefore use the outputs of our generated predicates as features for cells. In the latter case, we perform an additional improvement by allowing the splitting criterion to use our ranker when impurity is equal across different predicates. There are then three decision tree baselines in total.

4.1.2 ILP. We cast conditional formatting as an inductive logic programming (ILP) problem over the same grammar of rules as CORNET. This requires examples (both positive and negative) and background knowledge as input and learns a program that satisfies the examples using the background knowledge. In our setting, the
background knowledge consists of the grammar and the constants extracted from the column. Again, we consider two variants by using raw cell values and by augmenting the grammar to use our generated predicates. We use popper [5], a state-of-the-art ILP tool.

**Example 5.** Consider a numerical column with values \(7, 6, 3, 4\). An excerpt of the background knowledge is

\[
\text{LessThan}(A, B) := A < B.
\]

\[
\text{const1}(7), \text{const2}(6), \text{const3}(3), \text{const4}(4).
\]

where the first line defines a predicate and the second line defines constants that the predicate can use. We define \(\text{col}(A)\) as the predicate to be learned and give \(\text{col}(3)\) and \(\text{col}(6)\) as a positive and negative example, respectively. The program produced by popper is

\[
\text{col}(A) := \text{LessThan}(A, B),
\]

\[
B := \text{const4}(4).
\]

4.1.3 Constrained Clustering. Conditional formatting can be treated as a constrained (cell) clustering problem where clusters must respect the provided formatted examples. COP-KMeans is a \(k\)-means based clustering strategy that supports linkage constraints for clusters [37]. Besides a distance function between cells and the number of clusters, it also takes must-link \(e^+\) and cannot-link \(e^-\) constraints as input. We use the size of the symmetric difference between the sets of predicates that hold for two cells to measure their distance. The formatted examples and the implicit negative examples are used to populate \(e^+\) and \(e^-\). All pairs of formatted cells and pairs of negative cells are in \(e^+\). All pairs consisting of a formatted and an implicit negative example are in \(e^-\). For example, in Figure 5, \(e^+\) contains the positive pair (RW-187, RW-159) and the negative pair (RS-762, RW-131-T). The mixed pair (RW-187, RS-762) is in \(e^-\).

4.2 Neural

To build neural baselines, we frame conditional formatting as a table/cell classification problem and pick state-of-the-art models from this domain. Two of these neural approaches are based on table embedding models and one is built on top of a language model. All neural baselines are fine-tuned on our 80K training tasks.

4.2.1 TAPAS. TAPAS [17] is a table encoding model trained for sequential question answering (SQA). We apply it to conditional formatting by using it to encode the input column and getting an embedding for each cell and applying cross-attention between the formatted cells and the rest of the column. A linear layer followed by a sigmoid activation is used to make a prediction (formatted or unformatted) for each cell. Figure 6 (a) describes the architecture.

4.2.2 TUTA. TUTA [38] is a tree based transformer model that is pre-trained on multiple table-related objectives. One of the downstream tasks it has been fine-tuned for is cell type classification (CTC). TUTA uses cell values in a table along with their position, data type and formatting information to predict the role of a cell. By considering formatings as cell types, we fine-tune it to predict the format of each cell from a partially annotated column.

4.2.3 BERT. Finally, we use an architecture similar to the TAPAS baseline, but use the BERT language model [6] to produce column embeddings. Each cell in a column is tokenized, the tokens for different cells are concatenated with a separator token in between,

![Architecture Diagram](image-url)

Figure 6: Two custom neural baseline architectures in our evaluation. We cast conditional formatting as cell classification. Green cells represent formatted examples. TAPAS directly encodes all formatted cells to a single embedding while BERT does a cell level embedding.

this sequence of tokens is embedded, and cell-level embeddings are obtained by average pooling. Tokenization and average pooling is also used to obtain individual cell embeddings for the positive examples. A cross attention layer, where the full column provides queries (Q) and formatted cells provide keys (K) and values (V), is used to combine these embeddings—a thorough discussion on attention in transformers is given in [36]. Finally, a linear layer followed by a sigmoid activation converts the cross embedding output to predictions for each cell. Figure 6 (b) shows the architecture.

5 EVALUATION

We perform experiments to answer the following questions:

Q1. Is Cornet able to quickly and correctly learn conditional formatting rules from few examples?

Q2. How do our design decisions (clustering, iterative learning and ranking) impact learning time and correctness?

Q3. How do properties of the input table (number of examples, row order and column type) impact learning?

Q4. Can Cornet learn rules that are shorter than those authored by users?

Q5. Can Cornet learn rules for spreadsheets that users formatted manually?

Additionally, we investigate two dimensions of generality – generalizing to related data tasks and generalizing to complex multi-column conditional formatting scenarios – through case studies:

CS1. We apply Cornet to two related problems: data filtering and learning the conditions for conditional formulas ([1F]).

CS2. We evaluate Cornet on the rare cases where the condition spans multiple columns (0.9% of cases in our CF rule corpus).

5.0.1 Benchmarks. To train and evaluate Cornet, we leveraged a corpus of 1.8 million publicly available Excel workbooks from the web. Among these, 236.5K workbooks contain at least one CF rule added by users. In total, we extracted 410.6K CF rules and their corresponding cell values and formatting. We deduplicate files by filename, sheets by column headers and rules by exact syntactic
match. Furthermore, we remove rules that operate on less than five cells, format the entire column or only format a single cell. After deduplicating and filtering, we retain 105K tasks where a task consists of a (formatted) column and the associated CF rule. Table 3 shows a summary of the benchmarks. We split the 105K tasks into a train set of 80K, which we use for training CORNET and baselines, and a test set of 25K tasks, which we use for evaluation.  

### Table 3: Average properties of benchmark problems divided by type. Rule depth is defined as the tree depth of the abstract syntax tree produced by parsing the rule using our grammar.

<table>
<thead>
<tr>
<th>Type</th>
<th>Rules</th>
<th># Cells</th>
<th># Formatted</th>
<th>Rule Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>13.81 K</td>
<td>107.5</td>
<td>32.1</td>
<td>2.3</td>
</tr>
<tr>
<td>Numeric</td>
<td>9.32 K</td>
<td>184.8</td>
<td>111.2</td>
<td>1.8</td>
</tr>
<tr>
<td>Date</td>
<td>1.87 K</td>
<td>73.3</td>
<td>23.5</td>
<td>1.7</td>
</tr>
<tr>
<td>Total</td>
<td>25 K</td>
<td>133.7</td>
<td>60.9</td>
<td>2.1</td>
</tr>
</tbody>
</table>

5.0.2 Evaluation Metrics. To evaluate learned rules against user-written rules, we consider three metrics: exact match, execution match and cell match. Exact match is a syntactic match between a learned rule and the user-written rule, with tolerance for differences arising from white space and alternative argument order, which do not impact execution. Execution match consists of executing two rules and comparing the formatting produced — there is an execution match if the formatting are identical. In addition to capturing the fact that different rules can produce the same formatting, execution match allows us to evaluate against baselines that directly predict formatting rather than produce rules. This distinction between exact and execution match is also made in related areas, such as natural language to code [23, 31]. We also report cell-level precision and recall over the predicted and the ground truth formatting. These are micro-averaged over all tasks. We refer to these two metrics as Cell Match.

Example 6. Because they are equivalent after removing spaces and swapping (equivalent) argument order, OR(Equals(10), Equals(20)) and OR(Equals(20), Equals(10)) are an exact match. On the other hand, TextStartsWith(“D12”) and TextContains(“D12”) are not an exact match because the rules are not equivalent. These may be an execution match on a column that only has “D12” at the start of values.

5.1 Q1. Performance

Table 4 presents an overview of our results. CORNET outperforms symbolic and neural baselines on both exact and execution match metrics. Both Popper and decision tree methods perform worse than CORNET even when provided with CORNET’s predicates. TUTA is the only neural model that is competitive with symbolic methods—likely due to being trained for cell type classification. However, TUTA does not do well at capturing syntactic patterns, and as a result does not perform as well as CORNET. CORNET is the only approach that achieves, on average, both high recall (97.8%) and high precision (92.2%). Symbolic methods achieve high precision but low recall, indicating poor generalization. Neural methods achieve high recall, but low precision, indicating too aggressive generalization.

5.1.1 Limitations of Baselines. Our symbolic baselines are limited to learning a single rule, whereas CORNET learns multiple rules and then uses more context to rank them. Neural models are heavily dependent on tokenization and mainly appear to capture semantic properties. This makes them less effective in cases that require identifying syntactic patterns, which is often the case for CF rules. In rare cases, this ability to capture the semantic meaning of text gives neural models an advantage over CORNET. For example, TUTA is able to color cells that contain High or Medium even though the single formatted example provided is High, whereas CORNET would only color High. A second advantage of neural models is that they are not bound by our conditional formatting grammar and can support some scenarios that require arbitrary Excel formulas. While CORNET does not support such cases, our analysis shows they are rare in practice (377 cases in our full corpus).

5.1.2 Execution Time. We also evaluate the time required by each system to predict formatting as a function of the number of cells in the target column. Figure 7 shows the average time taken to predict a rule as a function of the increasing number of cells in a column. We show results for CORNET, the fastest (decision tree) baseline and the symbolic (Popper) and neural baselines (TUTA) with highest execution match. Learning multiple shallow decision trees (CORNET) is faster than learning one large one. TUTA is backed by a medium-sized neural network (110M parameter) that makes inference slow in our testing environment, which has resources beyond those that a target CF user would typically have. Popper is the slowest out of these baselines as the hypothesis space quickly explodes as a result of predicate generation for different cells.

5.1.3 Memory Consumption. Table 5 shows the total amount of disk space required to store a system, and the average CPU and GPU memory used for prediction over benchmarks for CORNET and neural baselines. CORNET uses BERT for ranking, hence their
**5.2 Q2. Impact of Design Decisions**

We discuss the impact of the three main components in CORNET: semi-supervised clustering, iterative rule learning, and ranking.

### 5.2.1 Clustering

First, we carry out experiments with three different versions of our clustering approach and show the results in Table 6. First, *no clustering* removes the semi-supervised clustering step altogether. It considers user formatted cells to be positive examples and all unlabeled cells to be negative examples. Note that this ablation can still learn rules (with worse performance) because the iterative tree learning procedure in CORNET only requires satisfying the user formatted examples and tolerates noise in other examples through the accuracy threshold during learning. Second, we consider a version of clustering where there are only two clusters: one for user formatted cells (positive examples) and one for all unassigned examples. Upon termination, all cells still in the unassigned cluster are relabeled as negative examples. We label this *no negative examples* in our results table. Third, we consider a version that only has *hard negative examples* by setting the weight of labeled and unlabeled cells equal during iterative tree learning—see Section 3.3 for details.

<table>
<thead>
<tr>
<th>System</th>
<th>Disk Space</th>
<th>CPU Memory</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUTA</td>
<td>722.4</td>
<td>5.4</td>
<td>1335.2</td>
</tr>
<tr>
<td>TAPAS</td>
<td>445.6</td>
<td>2.3</td>
<td>1416.8</td>
</tr>
<tr>
<td>BERT</td>
<td>416.7</td>
<td>4.4</td>
<td>775.9</td>
</tr>
<tr>
<td>CORNET</td>
<td>419.2</td>
<td>21.3</td>
<td>804.2</td>
</tr>
<tr>
<td>CORNET Symbolic</td>
<td>2.7</td>
<td>32.4</td>
<td>0.0</td>
</tr>
</tbody>
</table>

### 5.2.2 Iterative Rule Learning

Iterative learning allows CORNET to learn multiple candidate rules and then rank them separately. However, this iterative procedure is greedy and as a result is not complete—it only considers a subset of all possible rules. To evaluate the extent to which this impacts performance, we compared our greedy approach to an iterative full search up to tree depth 5.

In Figure 8, we compare the top-1 and top-all execution match accuracy for iterative greedy search (CORNET), a single decision tree and an exhaustive search with a maximal depth of five. As expected, CORNET is slightly less expressive and loses about 3% execution match accuracy on average against the full search, but this effect reduces as more examples are given.

In Figure 9, we compare the learning time for CORNET, single decision tree and the exhaustive search strategy as a function of the depth of the rule. Our results indicate that CORNET can be up to 40x to 80x faster than an exhaustive search, despite the small decrease in execution match accuracy as shown in Figure 8.
5.3 Q3: Impact of Input Configuration

The exact input to CORNET has an effect on its performance. We thus study how different properties of this input, like the number of formatted examples, order of examples, and number of unformatted cells, affect the performance of CORNET.

First, the number of examples that a user provides influences the accuracy. Ideally, this influence diminishes after a certain number of examples. Figure 10 shows this dependency on the provided number of examples, which varies significantly across data types. For text, two examples are sufficient for more than 90% of the cases. For numbers, performance steadily improves until 15 examples are provided. We hypothesise that more examples are needed in the numeric cases because constants in numeric rules are harder to learn—examples close to the decision boundary are needed, which might only appear lower in the column. When suggesting rules to users, we can thus be more conservative in numeric columns. Note that rules for text columns are on average longer than those for numbers (2.9 predicates versus 1.6) and we can more quickly suggest rules in cases that may be harder for the user.

Second, we investigate the impact of the number of unformatted cells on performance. Less data, and thus fewer unformatted cells, might be available when deploying systems like CORNET in browsers or on mobile devices. Our aim is to estimate the minimum number of unformatted cells needed for acceptable performance. Figure 11 shows how accuracy increases with the number of unformatted cells for different numbers of formatted cells. Performance gains diminish after more than 20 unformatted cells, across settings which provide 1, 3, and 5 formatted examples.

Third, we evaluate the effect of the order in which the user provides examples. To do so, we take each formatting task and randomly shuffle the formatted (positive) rows in the column five times to create five random orderings. For each shuffled task, we apply CORNET to an increasing number of formatted examples to learn a rule. We compute three statistics from this. First, we compute an all-shuffles execution match accuracy, which is the fraction of tasks where CORNET achieves execution match in all
When comparing execution match and exact match, we find that there is a 9% difference between the all-shuffles and at-least-one-shuffle execution match accuracy, which is the fraction of tasks where CORNET achieves execution match in at least one shuffled ordering. Finally, we report an average execution match accuracy where we simply report the fraction of tasks and orderings where CORNET learns a rule with the correct execution match.

Figure 12 reports the results over these shuffling experiments. We found that there is a 9% difference between the all-shuffles and at-least-one-shuffle execution match accuracy at three formatted examples, showing that there can indeed be an effect in the ordering of formatted examples. However, the original example order—used in all other experiments for this work—roughly aligns with the average accuracy found in these shuffling experiments.

### 5.4 Q4. Simplicity of Rules

We study the extent to which CORNET can help with discoverability of this feature. We analyze manually formatted columns. From our corpus of spreadsheets, we sample 100K columns with at least 5 rules are easier to interpret, write, and maintain. This notion of length-based simplicity has been used in prior PBE systems [8].

We treat all functions, operators and arguments as individual tokens and define the length of the rule as the associated count of tokens. For example, IF(A1="Not Applicable", TRUE, FALSE) consists of tokens {IF, -, "Not Applicable", TRUE, FALSE} and thus has a length of 5. Similarly, GreaterThan(10) has a length of 2.

In Figure 13, we consider all tasks where the user wrote a custom conditional formatting formula—not a predefined template—and we compare lengths of these formulas with the rules learned by CORNET. We find that in the majority of cases (∼60%) CORNET learns shorter rules, while maintaining execution match. As more examples are given, CORNET seems to learn comparatively longer rules. This happens because tasks that need more examples to be solved are more likely to require a (longer) complex rule.

We also found that reductions in formula length can be substantial: for complex rules, where we need up to 5 examples to learn a rule, the CORNET rule can be on average up to 65% shorter than the user-written rule. Figure 14 shows the average formula length reduction as a function of the length of the original user formula. In cases where CORNET requires more examples, rules are more complex and CORNET can provide greater reductions. This suggests that CORNET can be used for rule refactoring as well.

Some concrete examples of user rules and the associated CORNET rules are shown in Table 8. When CORNET learns a shorter rule, the user has often resorted to a custom formula instead of using a built-in predicate. When the length is the same, CORNET either uses the same predicate with a different constant or a different predicate with the same constant. For different constants, due to enumeration, CORNET yields less precise numbers (10 versus 10.5). For different predicates, due to ranking, CORNET is generally more conservative and yields more specific rules (Equals versus Contains).

### 5.5 Q5. Manual (re)Formatting

Not all users are aware of CF and manually format spreadsheets. We study the extent to which CORNET can help with discoverability of this feature. We analyze manually formatted columns. From our corpus of spreadsheets, we sample 100K columns with at least 5
non-empty cells, of which at least 3 have a custom background color applied without CF. Figure 15 shows some examples.

First, we provide CORNET with all formatted cells to learn a rule. If the learned rule has fewer predicates than formatted cells, we heuristically label this as a case where the user could have written a rule. We find 93.4K such columns. This distribution is shown in Figure 16a. Our results show that 80% of these learned rules have 3 or fewer predicates, which may benefit interpretability. Next, for all the columns identified, we find the minimal number of examples the user could have given to CORNET to obtain their desired formatting. The distribution of minimum number of examples needed for these cases is shown in Figure 16b. We find that CORNET learns over 90% of the rules with 4 or fewer examples.

5.6 CS1: Filtering and Conditional Formulas

Generating predicates and learning binary classifiers from few examples can be applied to related problems. We explore two such related problems in tabular data: filtering rows of data and learning the condition for conditional formulas. Filtering maps naturally to CF, as hidden and visible cells can be treated as unformatted and formatted, respectively. Similarly, we can map the true and false branches of a conditional formula to formatted and unformatted.

We consider data filtering in both Excel (from our corpus and StackOverflow) and Python (from StackOverflow). We sourced 330 filters that users created through the spreadsheet interface from a sample of 20K workbooks, 33 FILTER formulas from StackOverflow posts, which either use the interface or a custom FILTER, and 87 Python (Pandas) filtering tasks from StackOverflow.

We sampled 250 conditional formulas from the same subset of spreadsheets used to sample filtering tasks. The conditional formulas we consider are of the form $=IF(c, a, b)$ where $a$ and $b$ do not contain nested IF operators, and where $c$ only mentions a single column. Learning conditional ($c$) in formulas is an important problem studied extensively in programming-by-example [15].

Results for both problems are shown in Figure 17. We found that the filtering tasks resulted in higher performance than learning formula conditionals. We believe this is a result of our grammar, which is well aligned with the predicates available in the filtering interface and those often used in the FILTER function. In contrast, conditional formulas may use different functions, as well as use numeric conditions more often. Extending CORNET’s grammar with such predicates could help mitigate this performance difference.
We extended Cornet learning conditions as a function of the number of examples. Model which learns the hierarchical headers and data groups in Excel can improve the demonstration of mathematical concepts. Few formal studies on CF. [28] gives detailed coverage of how this feature works in the context of Excel. [1] discusses how CF in Excel must be in the column being formatted. Note that multi-column CF rules are relatively uncommon: in our corpus of 410.6K examples, we found 3647 (0.9%) that required more than one column. Column CF rules are relatively uncommon: in our corpus of 410.6K examples, we found 3647 (0.9%) that required more than one column. For learning rules over multiple columns, the number of examples needed for high performance increases. This is expected as the number of candidates in the search space and rule complexity is exponentially higher for multi-column cases.

6 RELATED WORK

Despite the large spreadsheet userbase, there have been relatively few formal studies on CF. [28] gives detailed coverage of how this feature works in the context of Excel. [1] discusses how CF in Excel can improve the demonstration of mathematical concepts.

Recent progress in automatic table formatting includes [7] which describes CellGAN, a conditional Generative Adversarial Network model which learns the hierarchical headers and data groups in tables. Other work like [18, 24] focus on formatting cells based on table structure and cell sizes. In contrast, Cornet targets example based data formatting, and generates the associated formatting rule.

Cornet uses a programming-by-example (PBE) paradigm, which has been popularized by systems like FlashFill [15] and FlashExtract [20]. These systems, which are available in Excel, learn string transformation and data extraction programs from few input-output examples. Popper [5] is another popular inductive logic programming (ILP) system for learning programs by specifying examples.

In terms of search techniques, [25, 32] use goal-driven top-down symbolic backpropagation. This is not applicable in our setting because the boolean signal (i.e., is a cell formatted) is too-weak to derive strong-enough constraints to navigate the search space. A popular alternative in PBE is bottom-up enumeration [12, 29, 34], which is infeasible in our setting because of the large search space. The notion of re-interpretation in [16] finds outputs and programs in separate DSLs. In contrast, Cornet first hypothesizes the outputs (cell formats) and then learns the associated rule. Cornet is the first system to take an “output-first” synthesis approach motivated by the fact that our output space is smaller than program space.

Past work on using PBE systems on databases have shown great success in the domain of querying [11, 22, 25] and data understanding and cleaning [12]. Cornet builds upon these systems to solve the problem of data formatting. Past PBE work has ranked programs using program features [9, 25] or output features [22, 27]. Cornet uses both program and output features for ranking programs.

Neural approaches have previously been applied in various tasks. For example, TaBERT [39] and TAPAS [17] are popular Question Answering systems that use a neural model to encode the table and query. TUTA [38] is a weakly supervised model for cell and table type classification tasks. SpreadsheetCoder [4] is a predictive system for learning spreadsheet formulas from table context. TabNet [35] predicts cell types using a neuro-symbolic model. Unlike these systems, Cornet targets learning formatting rules.

7 CONCLUSION

In this paper, we introduced the novel problem of learning conditional formatting rules from user examples. We proposed Cornet, a system that learns such data-dependent rules from few examples. To evaluate Cornet, we created a benchmark of 105K CF tasks extracted from 1.8 million real Excel spreadsheets. To facilitate future research into this novel problem, we release our set of benchmarks.

To effectively evaluate Cornet, we compare performance to baselines by framing the problem as an ILP task, a clustering task, a cell classification task and a table fine tuning task. We also create custom neural and symbolic baselines for a more comprehensive comparison and result analysis. We study how Cornet can help with discoverability of CF and learn shorter CF rules. Finally, we present two case studies applying Cornet to other related data tasks and multi-column formatting. Cornet opens future work like predictive CF rule learning and conditional data transformations.

ACKNOWLEDGMENTS

We would like to thank Yair Helman, Almog-Ben Kandi, Sophie Gerzie, Avital Nevo, Isreala Solomon, Christian Poelitz and Yoav Hayun for their feedback on this research.