Glean: Structured Extractions from Templatic Documents

Sandeep Tata
Google
tata@google.com

Navneet Potti
Google
navsan@google.com

James B. Wendt
Google
jwendt@google.com

Lauro Beltrão Costa
Google
laurocosta@google.com

Marc Najork
Google
najork@google.com

Beliz Gunel
Stanford University
bgunel@stanford.edu

ABSTRACT

Extracting structured information from templatic documents is an important problem with the potential to automate many real-world business workflows such as payment, procurement, and payroll. The core challenge is that such documents can be laid out in virtually infinitely different ways. A good solution to this problem is one that generalizes well not only to *known* templates such as invoices from a known vendor, but also to *unseen* ones.

We developed a system called Glean to tackle this problem. Given a target schema for a document type and some labeled documents of that type, Glean uses machine learning to automatically extract structured information from other documents of that type. In this paper, we describe the overall architecture of Glean, and discuss three key data management challenges : 1) managing the quality of ground truth data, 2) generating training data for the machine learning model using labeled documents, and 3) building tools that help a developer rapidly build and improve a model for a given document type. Through empirical studies on a real-world dataset, we show that these data management techniques allow us to train a model that is over 5 F1 points better than the exact same model architecture without the techniques we describe. We argue that for such information-extraction problems, designing abstractions that carefully manage the training data is at least as important as choosing a good model architecture.

PVLDB Reference Format:
Sandeep Tata, Navneet Potti, James B. Wendt, Lauro Beltrão Costa, Marc Najork, and Beliz Gunel. Glean: Structured Extractions from Templatic Documents. PVLDB, 14(6): XXX-XXX, 2021.
doi:10.14778/3447689.3447703

1 INTRODUCTION

Many documents commonly used in business workflows are generated automatically by populating fields in a template. Examples of such document types include invoices, receipts, bills, purchase orders, tax forms, insurance quotes, pay stubs, and so on. Documents of a particular type tend to contain the same key pieces of information relevant to these workflows, and processing these documents often relies on manual effort to extract such information. For example, invoice processing usually requires extraction of fields like `due_date` and `amount_due` from invoices. Documents generated from the same template, such as invoices from the same vendor, share the same layout. But there is a virtually-infinite variety of such templates possible for each document type.

The focus of our work is on *templatic*, form-like documents. Such documents tend to be *layout-heavy*: information is often presented in tables and boxes, and visual cues like spatial alignment are crucial to understanding them. This is in contrast to *text-heavy* documents such as essays, legal contracts and research papers where full sentences are used instead.

In this work we target the following challenge: Given a set of labeled examples belonging to a particular document type (say "invoices") and a *schema* – i.e. a target set of fields to extract – (say `due_date` and `amount_due`), build a system to extract the relevant information from *unseen* documents of the same type. Several recent papers [7, 10, 11, 22, 23] identify this problem as posing different challenges from traditional information extraction settings and describe techniques to build machine-learning models to automatically extract structured information from such form-like documents.

The key difficulty in solving such extraction tasks is that the same information can be laid out and described in many distinct ways. Figure 1 shows excerpts from four invoices, with a green bounding box identifying the invoice date. This field is described variously as 'Date', 'Dated', and 'Invoice Date'. In some cases the key phrase describing this field is immediately above the actual date, such as in case (a), or to the left of the date in cases (b), (c), and (d). When you consider a target schema with a dozen fields, this becomes a particularly challenging extraction problem. Keeping with the example of invoices, we may get a small set of labeled examples from vendors A, B, and C representing a few such layouts on which to train our models. We expect a model to do well on unseen documents following the same layouts as documents from A, B, C. The challenge is to do well when we receive documents from new vendors D, E, and F which are laid out differently.

We designed and built a system called Glean to address this critical business problem, and we are currently using it to build specialized document parsing endpoints for Google Cloud APIs[1]. Since it straddles the domains of natural language processing and computer vision, we developed a novel approach to solve the unique modeling challenges involved, which we discussed in our prior work [14]. In this paper, however, we focus instead on the oft-overlooked aspects

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447703

[1] https://cloud.google.com/solutions/document-ai

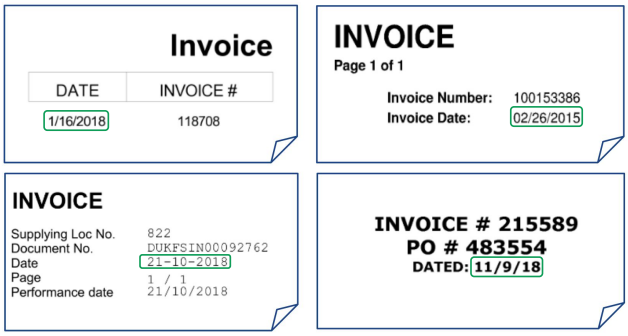

Figure 1: Excerpts from invoices from different vendors. Instances of the `invoice_date` field are highlighted in green.

of training data management and associated tooling. Our goal is to highlight the importance of careful design choices at every stage of the pipeline in contributing to the overall performance, separately from the machine learning modeling choices. In particular, we make the following key contributions.

- We present Glean, a system for information extraction, and describe the design choices we considered (Section 2).
- We summarize the core machine-learning ideas [14] in Section 3, but focus on challenges around 1) understanding and acquiring ground-truth data, 2) managing how training examples and labels are generated using the labeled data, and 3) tooling to support developers in quickly building models to solve such extraction tasks (Section 4).
- Through ablation experiments on a real-world dataset, we demonstrate that careful management of training data is *at least* as important as the modeling advances themselves, and without these techniques, extraction performance on a test set drops by more than 5 F1 points (Section 5).

2 SYSTEM DESIGN

Given a target document type (say invoices) and an associated target schema, we want to build an extraction service that takes a document image as input and returns a structured object conforming to the target schema. Glean allows a developer to build such a service without requiring any machine-learning expertise. The components of the Glean extraction system are shown in Figure 2.

2.1 Overview

As a first step, we process the input document image using an Optical Character Recognition (OCR) engine. OCR engines detect all the text on the page along with their locations, and can organize the detected text hierarchically into symbols, words and blocks along with their bounding boxes.

The second input to the system is a target schema. In the example in Figure 2, we use a toy schema for invoices that simply consists of two fields `invoice_date` and `total_amount` of type `date` and `price` respectively. Glean uses an extensible type system that also contains other basic data types like `integer`, `numeric`, `alphanumeric`, and `currency`. It also supports `address`, `phone_number`, `url`, and other common entity types.

Each type is associated with a candidate generator that identifies spans of text in the OCR output that potentially correspond to an instance of the type. We leverage an existing library of entity detectors that are used in Google's Knowledge Graph and are available through a Cloud API [2] for all the types described above. Open-source entity detection libraries can be used for common types like names, dates, currency amounts, numbers, addresses, URLs, etc. [3] Custom types including dictionaries, regular expressions, and other entity types from the Knowledge Graph (e.g. `university_name`) may be added by implementing a new C++ class. The candidate generators are designed to be high-recall – they identify *every* text span in the document that is likely to be of their type. For example, for dates, this may mean being robust to various ways of formatting dates ("15/1/2020", "15 January, 2020", "2020-01-15", etc.).

Once extraction candidates have been generated, we use a *Scorer* to assign a score for each (*field*, *candidate*) pair which estimates the likelihood that the given candidate is the right extraction value for that field. Note that multiple fields may belong to the same type (e.g. `invoice_date` and `due_date`) and may therefore share the same set of candidates. A candidate is typically represented by the text span identified by the candidate generator along with context such as text in its immediate neighborhood to provide the scoring function with additional features. We describe a machine-learned scorer in Section 3, trained using a set of labeled examples, that can generalize well to new templates.

The final component is the *Assigner*. This component takes all the scored candidates for a field and document, and assigns one of the candidates as the extraction value. The default assigner simply uses an *ArgMax* assignment strategy. However, additional business logic specific to a document type can be specified in the *Assigner* including constraints like "*invoice_date* must precede *due_date* chronologically". The output of the assigner is an assignment of a text string to each field specified in the input schema and possibly a null assignment for fields with either no candidates or only low scoring candidates. The precision of the system can be adjusted by imposing a minimum score threshold for each field.

2.2 Design Choices

We discuss several non-obvious choices that informed the design.
Using OCR: Using OCR as the first step gives us two advantages. First, it allows us to completely avoid having to deal with document format parsing (there are dozens of common office formats beyond just PDFs). Second, it enables the same stack to work with images of documents as well as native digital documents such as PDFs. Modern OCR engines have excellent accuracy on native digital documents [21], are robust to scanning noise and are starting to do well even on hand-written documents.
Why not pose this as a key-value detection problem? An alternative abstraction to solving such a problem is the "key-value detection" engine that several vendors[4, 5] have popularized. Such models simply detect a pair of neighboring strings and classify one of them as a "key" and the other a "value". This is clearly a useful building block in dealing with form-like documents. Often, there is substantial

[2] https://developers.google.com/knowledge-graph
[3] https://cloud.google.com/natural-language/docs/reference/rest/v1/Entity
[4] https://docs.aws.amazon.com/textract/
[5] https://azure.microsoft.com/en-us/services/cognitive-services/form-recognizer/

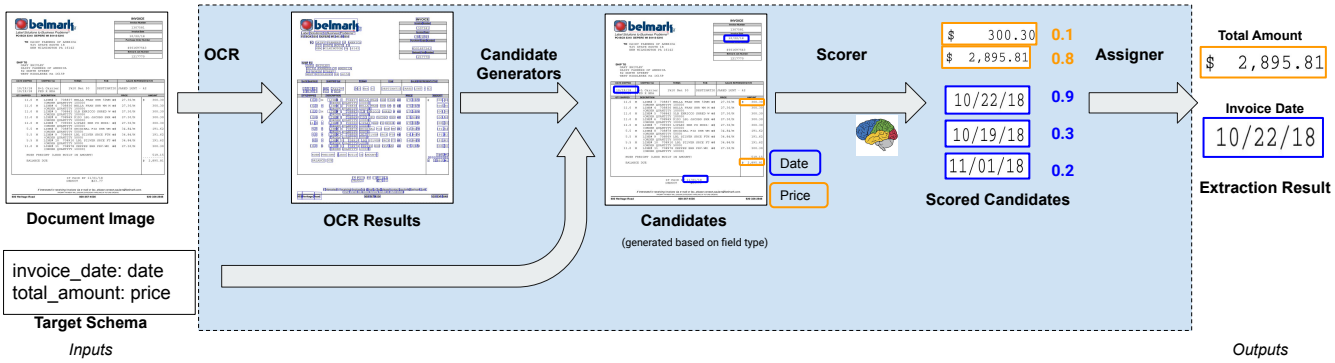

Figure 2: Glean Extraction System: The system takes a document image and a target schema as inputs, performs OCR, generates candidates, scores these candidates using a model that was previously trained on that target schema, and assigns the best candidates to the fields in the schema to produce the extraction result.

variation in the text used to represent the key from one document to another — a purchase order number may be represented variously as "P.O.", "PO Number:", "Purchase Order", "PO#" and so on. Using the output of a key-value system requires additional code to normalize and map these keys into some notion of schema managed by the application to use the output in a business workflow. Similar validation needs to be applied to the value string as well, converting strings like ":# 123" to "123" that may be detected as the value for a "P.O." key. Instead of letting each application manage the complexity of turning keys and values into structured objects that conform to a particular schema, managing this in the extraction system makes it easier to integrate into existing business workflows.

Schemas and Candidate Generators: We support a simple schema language that can specify flat schemas[6] associating each named field with a type. Fields may be marked optional or required. We associate a high-recall candidate generator with each field type, leveraging an internal library of text annotators that was developed for web-search tasks. Several open-source entity detection libraries[7] may be used to detect common types like names, dates, currency amounts, numbers, addresses, URLs, etc. that are shared across many document types. Some fields, such as "product name" in a receipt, might not lend themselves to an obvious candidate generator. For difficult fields, we resort to a fall-back candidate generator that uses "lines" detected by the OCR engine.

An attentive reader may observe that in contrast, classic sequence-tagging approaches from NLP do not require the specification of a target schema, and can be trained to label an input text sequence using a vocabulary of tags used in the training data. However, these approaches are known to require large amounts of data to work well for structured data extraction tasks [24]. For instance, an LSTM-based model that we trained on a small training corpus only recognized dates from 2019 as invoice dates since the training examples did not contain any documents from other years. Leveraging domain-agnostic candidate generators allows us to better generalize to unseen templates (e.g., by handling date formats not present in the training data) while avoiding overfitting.

[6]For simplicity, we do not discuss repeated and nested fields in this paper.
[7]NLTK: https://www.nltk.org/

2.3 Developer Workflow

The system is designed to be used by a developer who is *not* an expert in machine-learning to implement an extraction system for a particular document type (such as invoices, receipts, or bills). The developer focuses on defining a target schema, identifying high-recall candidate generators from an existing library of candidate generators, and where required, improving the coverage of candidate generators (see Section 4.4 for more details). The training and evaluation procedure itself requires no customization.

3 SCORER

We decompose the extraction task into four stages: OCR, candidate generation, scoring, and assignment, as shown in Figure 2. Our scorer model uses machine learning, where it takes as input the target field from the schema and the extraction candidate to produce a prediction score, and the model is both trained and evaluated as a binary classifier. Note that alternative formulations, such as posing this as a ranking problem [12], are also reasonable. While specific modeling choices for the scorer are not the focus of this paper, we briefly explain its architecture and how it connects to the design decisions we make. For further details of the scorer model architecture, please refer to Majumder et al. [14].

The features of each extraction candidate used in the scorer model are its neighboring words and their relative positions, as visualized in Figure 3. It is worth noting that we exclude the candidate's value from the set of features in order to avoid biasing the model towards the distribution of values seen during training, which may not be representative of the entire domain at test time.

Our scorer model learns a dense representation for each extraction candidate using a simple self-attention based architecture. Separately, it learns dense representations for each field in the target schema that capture the semantics of the fields. Based on these learned candidate and field representations, each extraction candidate is scored based on the similarity to its corresponding field embedding. The model is trained as a binary classifier using cross-entropy loss, where the target labels are obtained by comparing the candidate to the ground truth, as described in Section 4.2 below.

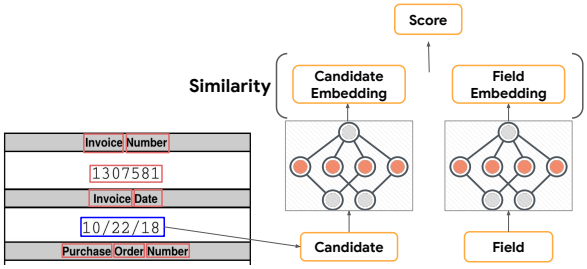

Figure 3: A candidate's score is based on the similarity between its embedding and a field embedding. A date extraction candidate "10/22/18" is shown in blue, along with its neighboring tokens, shown in orange.

Having separate stages of candidate generation and machine-learned scoring has several advantages. First, leveraging high-recall candidate generators significantly reduces the search space for the machine-learned scorer. This way, the scorer focuses on learning to understand spatial relationships and the semantics of the fields in the target schema, rather than learning to extract generic notions like numbers, dates, and addresses, or the various formats in which these may be presented in a document. Second, separately encoding the candidate and the field allows learning a field-agnostic representation of the neighborhood of a candidate. As an example, the different ways an `invoice_date` may be laid out, as in Figure 1, are not substantially different from those of `amount_due`. In other words, it is not the spatial relationships that are field-specific, it is the set of key phrases associated with it, such as "Invoice Date" and "Date of Invoicing" for `invoice_date`.

4 MANAGING DATA

This section describes design choices aimed at acquiring ground truth for a document corpus, generating training examples and labels, as well as tooling for computing and improving candidate generation.

4.1 Ground Truth

Humans provide a bounding box identifying the area containing each field of a document. We use the text detected by the OCR system within this bounding box as the text value for that field. Note that the OCR process provides the recognized symbols and a hierarchy of individual characters, words, paragraphs, and blocks. Each element in the hierarchy is associated with bounding boxes represented on the two-dimensional Cartesian plane of the document page. The ground truth for a given document for a field consists of both the bounding box and the detected text within it. Labelers are typically instructed to identify *all* instances of a field, and not just the first instance. As we describe later, this makes it easier to generate consistent training data using the different equality functions (Section 4.2) that we support.

Relying on the bounding boxes and getting the corresponding OCR text for ground truth has several advantages. It allows us to handle documents in different formats, making the labeling process simpler than having annotators providing the exact text for each field. It reduces the likelihood of typos and data-entry errors from humans that might lead to noisy training data, and is generally a quicker task than typing out precise values. However, it also injects two categories of errors from the OCR engine that affect the extraction performance: character-level accuracy, and the reading order of the words captured in the hierarchy representation.

The first category is well known: OCR may mis-recognize a character or set of characters (e.g., '4' vs 'A', or 'google' vs 'g00gle'). This is particularly common when there's very little context that might allow a language model to disambiguate a lone 'O' from a '0'. The second category of error is highly dependent on the document layout: text-heavy single column pages are less error-prone (e.g., an English text can be formed by simply sweeping the text from left to right and from top to bottom). A document with multiple columns, form fields and tables is challenging. Form-like documents often have non-trivial formatting, so errors in the reading order may make it harder for downstream models to extract longer/multi-line fields like addresses.

Our preliminary experiments showed the choice of OCR engines and its parameters have a significant impact on the overall performance. We have observed a 6.5 point difference in the F1 score depending on the choice of two internals systems available. However, improving or configuring OCR engines is out of the scope of this paper. The design choices in Glean are generally applicable irrespective of the choice of OCR engine.

4.2 Equality Functions

In order to generate training data consisting of positive and negative examples, we must compare the candidates for a field with the ground truth annotations for that field. When a candidate matches the ground truth, we label it as a positive example, and negative otherwise. The method by which we match candidates and ground truth can have a significant impact on the quality of the training data. We discuss two distinct strategies:

match-by-position compares the bounding boxes of candidates to the bounding boxes drawn by human labelers of the ground truth—in our case we consider two bounding boxes to match if their intersection-over-union is greater than a fixed threshold (say 0.5).

match-by-value performs a field type-specific semantic equality comparison between the actual value of the candidate with the value detected by OCR within the ground truth bounding box drawn by the human labeler.

Each of the above matching strategies come with benefits and drawbacks. For example, consider the match-by-position strategy. There are sometimes multiple mentions of a field in a document sharing the same value, like the `amount_due` being present both at the top and bottom of an invoice. If the human labeler did not label all such instances, this strategy would produce false negative labels. This technique may also generate false positives if, due to human error or possibly OCR misalignment, the ground truth bounding box only covers a part of the ground truth text or be unnecessarily long and cover extraneous text. These cases are largely mitigated by setting the minimum intersection-over-union threshold for a match to be less than 1.0.

The match-by-value strategy is type-specific. For example, consider a numeric field. While matching by value, "2.00" and "2" are equal. If the ground truth bounding box covered "2.00", but candidate generation only picked up "2" as a candidate, this would match by value, but not by position. Matching by value will usually be able to label all mentions of a particular value, even if the labelers missed a few instances. Therefore, this strategy has a low rate of false negatives. However, it can lead to false positives when the same value occurs multiple times in a document, but has different semantics. For example, the same date may occur both as a `delivery_date` as well as a `due_date` in an invoice. This equality function would mark both dates as positive for both fields, making it harder for the model to learn to distinguish between these fields. While match-by-value can produce noisy training data in such scenarios, it is the right equality function to use for evaluation, since the application typically cares about extracting a value, not the position or the bounding box where it occurs.

In the process of exploring different strategies, we realized that while neither one is perfect, a practical recipe is to 1) instruct the human labelers to mark all instances of a field, 2) use match-by-position to generate training data, and 3) use match-by-value to evaluate extractions. This produced the highest quality training data and led to the best extraction performance on an unseen dataset. As we demonstrate with experiments on a real-world dataset in Section 5, changing this recipe to use match-by-value for training data leads to a significant loss in extraction performance.

4.3 Label Vocabulary

Given a candidate for a particular field in a document, an equality function yields a boolean value describing if the candidate matched the ground truth. In order to manage how this information turns into labels for training examples, we consider the candidates for a given field and document, and compute a label with one of five distinct values:

- CORRECT: The ground truth was non-empty and the equality function returned true.
- INCORRECT: The equality function returned false and there was at least one candidate for that field and document for which it returned true.
- ABSENT_IN_GROUND_TRUTH: The ground truth was empty, and this is an optional field.
- FAULTY_GROUND_TRUTH: The value in the ground truth violates an integrity constraint. For instance, a field is marked required, but the ground truth for the field is empty. Or the value in the ground truth does not conform to the type constraint for the field.
- UNABLE_TO_MATCH: The ground truth value is present and not faulty. However, there was no candidate for which the equality function returned true.

This fine-grained approach to labeling the candidates allows us more control over how to generate and use the data for training a model. It should come as no surprise to any reader that real-world labeling tasks are seldom perfect. There are several papers [5] describing the challenges in getting high-quality labeled data for non-trivial tasks, especially if they require some familiarity with a narrow domain. Further, OCR errors may prevent certain documents from contributing clean labeled data. The fine-grained labels allow us to put sanity checks in place to discard data that seems problematic from the training data or, even more importantly, the validation or test data.

Recall that we model the scorer as a binary classifier by constructing examples using field and candidate pairs (Section 3). We map candidates labeled CORRECT as positives, those labeled INCORRECT or ABSENT_IN_GROUND_TRUTH as negatives, and discard the rest from the training data. The choice to discard candidates labeled FAULTY_GROUND_TRUTH is not surprising. Since the ground truth violates some integrity constraint, we're better off discarding it from our training and validation data sets, and when possible, sending that particular document back for re-labeling. A candidate may be labeled UNABLE_TO_MATCH, either because a matching function was defined too strictly (e.g. using an exact string match to compare "Dr. Alice Jones" and "Alice Jones") or the candidate generator used for that field happened to miss the actual ground truth labeled in the document (insufficient recall). In Section 5 we show using experiments on a real-world corpus that discarding such candidates results in a model with better performance on the test set.

4.4 Tooling for Candidate Coverage

High-recall candidate generators are critical to the success of the modeling approach in Glean. There are two metrics that give us a sense of how good the candidate generator for each field is:

(1) **coverage**: Among the documents that had ground truth for a given field, for what fraction were we able to generate at least one positive candidate?
(2) **fraction_correct_candidates**: Among all the candidates generated for a given field, what fraction were labeled CORRECT?

These metrics depend on the matching strategy, so we compute them for each of the matching strategies supported. The first metric, **coverage** is by far the more important – on a given corpus, it is a ceiling for the recall of our extraction system for this field. If we are unable to identify the ground truth as a candidate using one of our candidate generators configured for our field, we cannot extract the correct value. The second metric is less important, but helps us make a choice between two options with the same coverage. Intuitively, a candidate generator with a higher value of fraction_correct_candidates presents an easier problem to the scorer, asking it to identify the correct candidate out of a smaller pool. In the candidate-generation stage, the system automatically computes and outputs these two metrics for each of the matching strategies.

Consider a candidate generator for phone numbers. Many have previously described the complexity of relying on regular expressions to produce precise extractions [15, 26]. On the other hand, a high-recall regular expression can be developed relatively easily if we are willing to tolerate extraneous matches (social security numbers, zip-codes, etc.).

Field Name	Type	Cov.	Pres.	F1
amount_due	price	100%	42%	83.2%
delivery_date	date	98%	5%	78.0%
due_date	date	99%	35%	95.2%
invoice_date	date	99%	95%	96.3%
invoice_number	alphanum	99%	95%	96.9%
order_number	alphanum	88%	74%	79.3%
supplier_id	alphanum	87%	38%	79.4%
total_amount	price	99%	80%	90.8%
tax_amount	price	98%	53%	85.5%

Table 1: Target schema along with candidate coverage, fraction of docs where each field is present in the ground truth using the match-by-position strategy, and the per-field F1 scores on the test set using the Baseline model.

5 EXPERIMENTS

The experiments in this section are designed to evaluate the impact of the design choices previously outlined in Section 4. Through these experiments, we hope to convince the reader that careful management of training data is at least as important as the underlying machine-learning model. We also show that Glean models are relatively quick to train and can be served with interactive latencies.

5.1 Impact of Training Data Management

We use a dataset of ~14K documents from the payments domain. For the experiments in this paper, we used a target schema consisting of 9 fields shown in Table 1, belonging to `price`, `date` and `alphanum` field types. The third column reports the coverage number (fraction of documents with at least one positive candidate) described in Section 4.4. The fourth column reports the fraction of documents with a non-empty value for that field in the ground truth. The fifth column reports per-field end-to-end F1 scores obtained by the "Baseline" model on the test set before any ablations (described below).

In all the experiments below, we split the documents into 80%-20% training and validation splits. In each case, we train 10 models with different random initializations on the training split. We choose the model with the best validation AUC-ROC (area under the ROC curve), and evaluate it on a test set using match-by-value semantics. We use a separate test set consisting of 950 invoices where the vendors issuing these invoices were completely disjoint from those issuing the invoices in the train and validation sets. Thus, the performance reported on the test set is truly representative of how well the model does on unseen templates. We report the macro-average F1-score[8] across the 9 fields on the test set. We also report the statistical significance by computing the macro-average F1-score for each of the 10 models trained, and calculating the p-value using a t-test comparing the baseline with each row in the table. As is evident from the table, each result is statistically significant with all p-values well below 0.05.

[8]The F1 score is the harmonic mean of precision and recall. We compute an F1 score for each field, and report the arithmetic mean across the per-field F1 scores. This metric treats all fields as equally important.

Training Method	E2E F1	Delta	p-value
Baseline	87.2%	–	–
Train using match-by-value	84.8%	-2.4%	0.007
Retain UNABLE_TO_MATCH	84.9%	-2.3%	0.003
Retain overlapping candidates	81.9%	-5.3%	0.022
Disallow spaces in `alphanum`	80.2%	-7.0%	0.003
Use a simpler model architecture	85.8%	-1.4%	0.002

Table 2: End-to-end extraction performance on a test set (F1 score, larger is better) when training the same model architecture with training data generated in different ways. The last row uses a simpler model architecture for comparison.

We hold the model architecture and hyper-parameters constant across all the experiments, and only vary how the data was generated. In the first case, described as "Baseline" in Table 2, we make the following choices:

(1) We generate labels using match-by-position semantics (as opposed to match-by-value).
(2) We discard training examples which were labeled as UNABLE_TO_MATCH.
(3) While generating training examples for a given field of the schema, if a candidate overlaps a previously-generated candidate, we discard it and only retain the first candidate. This avoids candidates with nearly identical neighborhoods presenting with different labels.

Evaluation on the test set is always done using match-by-value semantics, as this is what the user of the extraction system cares about. The other rows in the table serve as an ablation study, where we reverse each of the choices described for the baseline above, and report the end-to-end F1 score on the test set.

The second row in the table reports the result of using match-by-value semantics for training. As explained in Section 4, multiple fields of the same type may *incidentally* share the same value in a document, e.g., the `delivery_date` and `due_date` in an invoice. The match-by-value equality function would treat both the corresponding candidates as positives for both fields, adding noise to the training data. Hence, the end-to-end F1 score for this approach is 2.4 points worse than the baseline.

When we are unable to find any positive candidate for a field in a document that has ground truth for that field, we label all the candidates as UNABLE_TO_MATCH. We usually discard these from the training set (not the validation and test sets) to avoid false negative labels that are artifacts of overly-strict equality functions or poor candidate generators. Instead, when we choose to retain these as negative training examples (third row in the table), we see a 2.3 F1 points drop compared to the baseline. We also find that retaining overlapping candidates, because they may have different labels while sharing the same neighborhood, results in a drop of 5.3 F1 points from the baseline.

To demonstrate the value of tooling that evaluates candidate coverage, we changed the candidate generator associated with the three `alphanum` fields – `invoice_number`, `order_number`, and `supplier_id`. The `alphanum` candidate generator uses a simple regular expression to match a string consisting of at least one digit and any letters or chosen punctuation symbols ("1234", "A1234",

Field Name	Type	Cov.	F1
Insurance Statements			
coverage_start	date	93%	91.3%
coverage_end	date	99%	97.0%
premium_amount	price	91%	90.2%
policy_id	alphanum	100%	91.0%
cancel_date	date	99%	98.4%
insurer_name	company	95%	89.0%
insurance_type	enum	100%	96.3%
property_address	address	60%	52.0%
Paystubs			
period_start_date	date	97%	88.8%
period_end_date	date	94%	87.5%
pay_date	date	97%	84.7%
gross_earnings	price	100%	93.0%
gross_earnings_ytd	price	100%	82.0%

Table 3: Schema for insurance statements and paystubs along with candidate coverage and per-field F1 scores using the Baseline model.

"A12-34", "A1/234", etc.). In the baseline, it also allows consecutive space-separated strings that match this regular expression. For this experiment, we disallowed spaces in this candidate generator to illustrate the impact that such seemingly-minor implementation choices can have on performance. As shown in the table, disallowing spaces resulted in a drop of 7 F1 points from the baseline. Much of this drop can be attributed to `supplier_id` field whose coverage dropped by 30 points and F1 by 20 points. Manual examination confirmed that these loss cases were documents where the `supplier_id` is formatted with spaces. It is worth noting that many previous papers [16, 22] have made the design choice of using a high-recall candidate generator to simplify the design of an information extraction system. However, not much work has gone into tooling to make it easy to rapidly measure and improve the recall of a candidate generator for a particular field type in the context of a document corpus. Making a bad choice here can completely hide any improvements from more sophisticated machine learning modeling for Scorer, as we show below.

Finally, to put into perspective the relative impact of careful management of training data versus using a more sophisticated modeling strategy, we trained a simpler model using the same training data as the baseline. In this simplified model architecture, rather than using a self-attention layer that was critical to the gains described in the modeling work [14], we simply combined the neighbor embeddings and the candidate position embedding using a max-pooling layer. We chose this as a comparison because this is a reasonable model that one might propose for a problem like this beyond simply using a bag-of-words. The F1 score from this model was 1.4 points worse than the baseline, as reported in the last row of Table 2. Observe that this degradation is *smaller* than the degradation in all the ablations above where we make the wrong choice with respect to generating the training data.

5.2 Other Document Types

Table 3 shows the target schema, the candidate coverage, and end-to-end extraction F1 score from the baseline model for two additional document types: home insurance statements and paystubs. We hope these high-level results illustrate how well Glean performs for diverse document types and field types.

The target schema for home insurance documents has 8 fields. The training dataset consisted of 700 documents and the test set consisted of 100 documents gathered by our partners. As is evident from the F1 scores, the model does very well on fields that have high candidate coverage. The `property_address` field has fairly low candidate coverage – our candidate generator is able to find correct `address` candidates on the document in only about 60% of the cases. As a result, the final F1 score for this field is just 52%. Manual inspection revealed that in many documents, the address was split into separate fields like: "Street Address", "City", "State", and "Zip code". The candidate generator we used for the `property_address` field was not developed to detect address strings that were split into distinct sub-fields. The only other field with end-to-end performance below 90% is the `insurer_name` field, despite a 95% coverage from the candidate generator. Examining the error cases, we found that there is often no key phrase like "Insurance Company" near the true `insurer_name` value. When there's only one company mentioned on the document, only one candidate is generated and the extraction is correct. But in the error cases we observed, multiple company names appeared on the document without such disambiguating key phrases in the neighborhood, so the model occasionally picked the wrong candidate.

The paystub document type has 5 target fields. The training set was even smaller, consisting of just 130 documents with a test set of 30 documents. While only one of the fields gets above an F1 score of 0.9, with additional training data, we expect to improve the extraction performance of all fields to above 0.9.

It is worth noting that field types like `date` and `price` which were already supported for the payment domain, we were able to use exactly the same candidate generators for these new document types as well. This highlights the value of our design choice of separating candidate generation from scoring: when adding support for a new document type, there is no need to develop a new generator to detect dates, say, in all their various formats.

5.3 Training and Serving

While the design choices in our system were not aimed at minimizing training or serving time, factoring the problem into candidate generation, scoring, and assignment makes it possible to train and serve a fairly inexpensive model. On the corpus of ~14K documents, after candidate generation, the schema in Table 1 resulted in ~1.3M examples. On a single GPU, we were able to train a model in approximately 45 minutes converging after 25 epochs.

For comparison, we also trained a BERTGrid [7] model which extracts directly from the document without generating candidates. On a GPU, training converged after 20 epochs in approximately 1090 minutes and resulted in similar to worse F1 scores across all fields. Decomposing the problem into candidate generation, scoring, and assigning allows us to use a substantially simpler model architecture compared to some of the alternatives presented in prior work. Our goal here is *not* a careful comparison of training cost – instead we make the case that our approach is at least an order of magnitude cheaper to train than alternatives that use a

	OCR	Candidate Generation	Scoring	Total
Mean	1,102	23	78	1,203
Median	1,512	21	77	1,610
95%-ile	4,100	56	133	4,289

Table 4: Serving time in milliseconds for the main stages of the Glean extraction system.

more sophisticated model architecture. This isn't necessarily an intrinsic advantage – in some settings, training 10× longer to get a 1% accuracy improvement is a perfectly reasonable trade-off since the model may only be trained and updated once every few months. The faster training time allows us to iterate faster, experiment with more modeling ideas *and* quickly produce a higher quality model for a given task.

We measured the extraction time for the validation data set using the Glean extraction system that is currently available through Google Cloud APIs. Although extraction time did not guide the design choices for scoring and candidate generation, our goal is to understand if they add a significant time to the overall extraction. Table 4 shows how long the system spent in three main stages of extraction. The OCR stage dominates the overall time, and the scoring and candidate generation stages contribute to roughly just 5% to 10% of the extraction time. The assignment stage (Figure 2) takes a negligible amount of time and is omitted from the table.

6 RELATED WORK

The Glean extraction system draws on decades of research on information extraction from text, from the web, and from templatic documents, as well as research on data management literature for information extraction tasks.

Information Extraction from Webpages Leveraging the template information to understand the visual layout within documents has been a well-known technique in the context of information extraction from webpages [2–4, 25, 27]. Although these proposed methods are relevant to extraction from templatic documents, they are not immediately applicable to our setting, as we simply do not have access to the source markup representation for the document images we would like to extract from. Similarly, Elmeleegy et al. [9] propose an unsupervised domain-agnostic technique to extract tables from HTML lists using language models and a corpus of tables. However, they do not utilize any spatial information, and assume that there is no single right answer for the table extraction problem that leads to subjective quality. Toda et al. [20] propose a probabilistic method that automatically selects segments from the input data-rich text and associates them with the appropriate fields in the form through content and style related features using a Bayesian Network. However, they do not use any spatial information, and rely on a greedy heuristic to find an approximate solution.

Data Management Data Civilizer [6] proposes an end-to-end big data management system that supports users by finding relevant data for their specific tasks; and Data Debugger [18] proposes an end-to-end data framework that enables users to identify and mitigate data-related problems in their pipelines. Our paper similarly identifies the importance of managing training data, but focuses on the data management techniques that are critical in the context of an information extraction system. The experience reported by Dagger [18] parallels what we reported on the impact of data quality on building machine-learned information extraction systems. Wu et al. [22] propose a new data model to extract information from richly formatted data for knowledge base construction– accounting for document-level relations, multi-modality, and data variety challenges which are inherent to richly formatted data. Sarkhel and Nandi [19] propose a general information extraction framework for visually rich documents where they segment a document into a bag of logical blocks and use the context boundaries of these blocks to identify the named entities within the document.

Machine Learning Katti et al. [11] propose inputting documents as 2D grids of text tokens to fully convolutional encoder-decoder networks. Denk and Reisswig [7] incorporate pretrained BERT text embeddings into that 2D grid representation. Xu et al. [23] propose integrating 2D position embeddings and image embeddings, produced with a Faster R-CNN [17] model, into the backbone structure of a BERT language model [8] and using a masked visual-language loss during pre-training. Similarly, Garncarek et al. [10] propose integrating the 2D layout information into the backbone structure of both BERT and RoBERTa [13], where they construct layout embeddings using a graph neural network using a heuristically constructed document graph. In contrast with the pre-training based approaches, our extraction system (1) requires several orders of magnitude less labeled training data (2) an order of magnitude less training and inference time, while (3) retaining the ability to tackle the harder problem of generalizing to unseen templates. In addition, Bai et al. [1] propose a neural collaborative filtering model for user-item interactions that integrate neighborhood information through an interaction network. Although their machine learning architecture is similar to ours, we construct field-candidate pairs using high-recall candidate generators instead of heuristic methods based on a provided interaction network. Finally, several ideas in our paper are complementary to the Snorkel framework by Ratner et al. [16], where they programmatically label training data using user-defined labeling functions. Similar to writing code for the labeling functions, writing code for high-recall candidate generators in the Glean extraction system is a weak form of supervision. Therefore, adopting a Snorkel-like data programming framework for building Glean models could be interesting future work.

7 SUMMARY

In this paper, we presented the problem of extracting structured data from form-like documents and a system called Glean to solve it. Glean factors the information extraction problem into 4 components – an off-the-shelf OCR engine, high-recall candidate generators, a machine-learned scorer, and an assigner. We argued that managing training data is a key challenge in building a good solution to this problem and described various design choices around tackling this. Through experiments on a large real-world dataset, we showed that these choices are at least as important as a good modeling strategy.

REFERENCES

[1] Ting Bai, Ji-Rong Wen, Jun Zhang, and Wayne Xin Zhao. 2017. A Neural Collaborative Filtering Model with Interaction-based Neighborhood. *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (2017), 1979–1982.

[2] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. Extracting Content Structure for Web Pages Based on Visual Representation. In *Proceedings of the 5th Asia-Pacific Web Conference*. 406–417.

[3] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2004. Block-based Web Search. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 456–463.

[4] Gobinda G. Chowdhury. 1999. Template Mining for Information Extraction from Digital Documents. *Library Trends* 48, 1 (1999), 182–208.

[5] Florian Daniel, Pavel Kucherbaev, Cinzia Cappiello, Boualem Benatallah, and Mohammad Allahbakhsh. 2018. Quality Control in Crowdsourcing: A Survey of Quality Attributes, Assessment Techniques, and Assurance Actions. *ACM Comput. Surv.* 51, 1, Article 7 (Jan. 2018), 40 pages.

[6] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*.

[7] Timo I. Denk and Christian Reisswig. 2019. BERTgrid: Contextualized Embedding for 2D Document Representation and Understanding. arXiv:1909.04948 [cs.CL]

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186.

[9] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. 2009. Harvesting Relational Tables from Lists on the Web. *Proceedings of the VLDB Endowment* 2 (2009), 1078–1089.

[10] Łukasz Garncarek, Rafał Powalski, Tomasz Stanisławek, Bartosz Topolski, Piotr Halama, and Filip Graliński. 2020. LAMBERT: Layout-Aware language Modeling using BERT for information extraction. arXiv:2002.08087 [cs.CL]

[11] Anoop R. Katti, Christian Reisswig, Cordula Guder, Sebastian Brarda, Steffen Bickel, Johannes Höhne, and Jean Baptiste Faddoul. 2018. Chargrid: Towards Understanding 2D Documents. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 4459–4469.

[12] Tie-Yan Liu. 2011. *Learning to Rank for Information Retrieval*. Springer-Verlag.

[13] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[14] Bodhisattwa Prasad Majumder, Navneet Potti, Sandeep Tata, James B. Wendt, Qi Zhao, and Marc Najork. 2020. Representation Learning for Information Extraction from Form-like Documents. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 6495–6504.

[15] Rachel Millner. 2008. Four regular expressions to check email addresses. https://www.wired.com/2008/08/four-regular-expressions-to-check-email-addresses/

[16] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proceedings of the VLDB Endowment* 11, 3 (Nov. 2017), 269–282.

[17] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 6 (2015), 1137–1149.

[18] El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael Stonebraker. 2020. Dagger: A Data (not code) Debugger. In *Proceedings of the 10th Conference on Innovative Data Systems Research*.

[19] Ritesh Sarkhel and Arnab Nandi. 2019. Visual Segmentation for Information Extraction from Heterogeneous Visually Rich Documents. In *Proceedings of the 2019 International Conference on Management of Data*. 247–262.

[20] Guilherme A. Toda, Eli Cortez, Altigran S. da Silva, and Edleno de Moura. 2010. A Probabilistic Approach for Automatically Filling Form-Based Web Interfaces. *Proceedings of the VLDB Endowment* 4, 3 (Dec. 2010), 151–160.

[21] Jake Walker, Yasuhisa Fujii, and Ashok Popat. 2018. A Web-Based OCR Service for Documents. In *13th IAPR International Workshop on Document Analysis Systems – Short Papers Booklet*. 21–22.

[22] Sen Wu, Luke Hsiao, Xiao Cheng, Braden Hancock, Theodoros Rekatsinas, Philip Levis, and Christopher Ré. 2018. Fonduer: Knowledge Base Construction from Richly Formatted Data. In *Proceedings of the 2018 International Conference on Management of Data*. 1301–1316.

[23] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. 2019. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. arXiv:1912.13318 [cs.CL]

[24] Zhilin Yang, Ruslan Salakhutdinov, and William W. Cohen. 2017. Transfer Learning for Sequence Tagging with Hierarchical Recurrent Networks. arXiv:1703.06345 [cs.CL]

[25] Shipeng Yu, Deng Cai, Ji-Rong Wen, and Wei-Ying Ma. 2003. Improving Pseudo-Relevance Feedback in Web Information Retrieval Using Web Page Segmentation. In *Proceedings of the 12th International World Wide Web Conference*. 11–18.

[26] Shanshan Zhang, Lihong He, Eduard Dragut, and Slobodan Vucetic. 2019. How to Invest My Time: Lessons from Human-in-the-Loop Entity Extraction. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2305–2313.

[27] Jun Zhu, Zaiqing Nie, Ji-Rong Wen, Bo Zhang, and Wei-Ying Ma. 2006. Simultaneous Record Detection and Attribute Labeling in Web Data Extraction. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 494–503.