

On the synthesis of metadata tags for HTML files

Patricia Jiménez Juan C. Roldán Fernando O. Gallego Rafael Corchuelo

University of Sevilla, ETSI Informática,
Sevilla, Spain

Correspondence

Rafael Corchuelo, University of Sevilla,
ETSI Informática, Avda. Reina Mercedes,
s/n, Sevilla E-41012, Spain.
Email: corchu@us.es

Funding information

Junta de Andalucía, Grant/Award
Number: P18-RT-1060; Spanish R&D
Programme, Grant/Award Number:
TIN2013-40848-R, TIN2016-75394-R;
University of Sevilla, Grant/Award
Number: PIF Grant

Summary

RDFa, JSON-LD, Microdata, and Microformats allow to endow the data in HTML files with metadata tags that help software agents understand them. Unluckily, there are many HTML files that do not have any metadata tags, which has motivated many authors to work on proposals to synthesize them. But they have some problems: the authors either provide an overall picture of their designs without too many details on the techniques behind the scenes or focus on the techniques but do not describe the design of the software systems that support them; many of them cannot deal with data that are encoded using semistructured formats like forms, listings, or tables; and the few proposals that can work on tables can deal with horizontal listings only. In this article, we describe the design of a system that overcomes the previous limitations using a novel embedding approach that has proven to outperform four state-of-the-art techniques on a repository with randomly selected HTML files from 40 different sites. According to our experimental analysis, our proposal can achieve an F_1 score that outperforms the others by 10.14%; this difference was confirmed to be statistically significant at the standard confidence level.

KEYWORDS

embedding techniques, HTML files, metadata tags

1 | INTRODUCTION

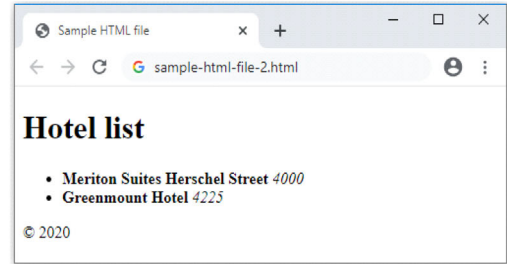
The data in an HTML file can be easily endowed with semantics by means of metadata tags. Such tags link the HTML elements that provide the data to either their corresponding entities in a knowledge graph or their corresponding types in a schema. Simply put, they pave the way for software agents that can easily sift through the Web and understand the data that it provides in human-friendly HTML files. The metadata tags are commonly encoded using RDFa, JSON-LD, Microdata, or Microformats.¹ For instance, Figure 1A shows a sample HTML file that uses RDFa: the vocab attribute in the li elements indicates that they contain data that can be structured according to a type called Hotel in the schema provided by example.org; the property attributes in the span elements indicate whether they provide the name of the hotel or the postal code. Figure 1B shows how a browser renders this HTML file. Note that the metadata tags are not shown, but software agents can read them to understand the data. Search engines are a particular kind of software agent that benefits from these metadata tags because they help them build info-boxes that feed their question-answering systems.²

As of 2013-14, metadata tags had been adopted by roughly 50% of the 10 000 most popular web sites,^{3,4} which means that there were billions of HTML files whose data could be easily understood by many software agents. Unfortunately, that does not entail that they are so common in the general Web. Recently, an analysis of the 32.04 million domains in

```

<html>
<head> ... </head>
<body>
  <h1>Hotel list</h1>
  <ul class="list">
    <li vocab="http://example.org/#" typeof="Hotel">
      <span property="name">Meriton Suites Herschel Street</span>
      <span property="postalCode">4000</span>
    </li>
    <li vocab="http://example.org/#" typeof="Hotel">
      <span property="name">Greenmount Hotel</span>
      <span property="postalCode">4225</span>
    </li>
  </ul>
  <ul class="legal">
    <li><span>&copy;</span> <span>2020</span></li>
  </ul>
</body>
</html>

```



(A) Source code with RDFa meta-data tags.

(B) Renderisation by a browser.

FIGURE 1 Sample HTML file [Colour figure can be viewed at wileyonlinelibrary.com]

the November 2019 Common Crawl has revealed that only 11.92 million domains provide metadata tags,¹ which clearly argues for a method that helps software agents deal with the documents provided by the remaining 20.12 million domains.

In the literature, there is a software-engineering proposal by Dodero et al.⁵ that can be used only when the system that produces the HTML files can be reengineered, which is not generally the case. The other proposals can be plugged into existing systems without any reengineering. Some of them target content management systems and focus on files whose content is written in natural language, namely: Adrian et al.⁶ and Ngomo et al.⁷ devised two components that intercept the HTML files output by the system and endows them with metadata tags automatically; in addition, Eldesouky et al.⁸ and Guerrero-Contreras et al.⁹ devised two text editors that help authors add metadata tags to their text. There are also some proposals that focus on data that are encoded in semistructured formats, namely: Burget¹⁰ and Efthymiou et al.¹¹ presented two proposals that attempt to link the data records in an HTML file to the entities in a knowledge graph; that is, they cannot provide metadata tags unless there is a matching in the knowledge graph. Very recently, Oulabi and Bizer¹² have presented a proposal that overcomes the previous problems since it can identify new entities that are not in the knowledge graph and map their properties onto the corresponding schema; unfortunately, it can only work on tables in which data records are arranged horizontally and the headers provide clues to find the metadata tags in the schema of the knowledge graph.

In this article, we present a system to synthesize metadata tags that does not require any knowledge graph and is not bound with a particular kind of layout. It was designed building on three components: a data processor, a learner, and a tagger. They provide several services that communicate through message queues, which facilitates assembling a microservice architecture that can be easily deployed and run on a distributed system. Its most novel component is the learner; it helps learn a tagging model using a new graph embedding technique that, contrarily to the others in the literature, can find the length of the embeddings automatically, preserves the original attributes and their classification power in the embeddings, and learns a reusable tagging model that can be applied to similar HTML files. We have experimented with our proposal and four state-of-the-art embedders on a repository with HTML files that were randomly selected from 40 different web sites; our goal was to confront our proposal with as many layouts as possible and to learn the tagging models from as few documents as possible. Our conclusion is that our proposal can learn tagging models from as few as six random documents and it can attain an F_1 score that is 10.14% better than with the other embedders. (The previous difference was confirmed to be statistically significant using hypothesis testing.) Summing up: we provide a novel and promising approach to synthesizing metadata tags for HTML files.

The rest of the article is organized as follows: Section 2 reports on the related work regarding systems that synthesize metadata tags or compute embeddings, and then discusses on them and compares them to our proposal; Section 3 provides an overall picture of our design and describes the details of its components, including their services, communication diagrams, and data models; Section 4 describes the algorithms behind the key services of our design; Section 5 reports on our experimental setting, the results of our experiments, and analyzes them using statistically sound methods; finally, Section 6 presents our conclusions and some future work. There is an appendix that reports on the catalog of attributes used to learn the tagging models.

2 | RELATED WORK

In this section, we first summarize the tagging proposals that we have found in the literature; we then report on the current approaches to graph embedding; finally, we discuss on the related work in order to put our proposal in a proper context.

2.1 | Tagging proposals

One of the earliest proposals was presented by Dodero et al.⁵ who described a methodology that can be used to reengineer a software system so that it produces HTML files with metadata tags. It is a software engineering approach that is not generally applicable, but only when a company is interested in updating their systems. Thus, we restrict our attention to proposals that can be plugged into existing systems; simply put, proposals that help synthesize metadata tags for an HTML file without reengineering the system that produces it.

The earliest such proposals focus on content management systems in which data are encoded in text that is written in natural language. Adrian et al.⁶ presented a method that consists of matching the noun phrases found in an HTML file to the values of the data properties in a knowledge graph. The complex part of their proposal is their heuristic to resolve ambiguities, which selects the matchings that result in the highest possible number of related entities in the knowledge graph. This idea lies at the heart of the more sophisticated approaches that have been published later. Ngomo et al.⁷ presented the architecture of a component that, basically, cleans the input HTML files and then runs some natural language processing tools to find words, sentences, named entities, or relations, which are fed into an ensemble of feed-forward neural networks that synthesize the metadata tags. Eldesouky et al.⁸ and Guerrero-Contreras et al.⁹ presented two proposals that are similar since they aim to produce an editor that helps content producers include metadata tags in their text without requiring any technical knowledge of the underlying technologies. Unfortunately, their articles describe the architecture and the functionality of their tools, not the details of the techniques behind the scenes.

More recent approaches focus on semistructured data that are rendered using specification sheets, record listings, or tables. Burget¹⁰ devised a proposal that decomposes the input HTML files into nonoverlapping rectangular areas. For each area, it computes its boundaries, its text, some style attributes, and its inner boxes; then each area is assigned a collection of candidate tags using DBpedia Spotlight, a regular pattern-based entity recognizer, and a classifier that was trained using attributes like fonts, colors, positions, or lexical attributes. The tags are then disambiguated using some style consistency and positioning heuristics. Another recent proposal by Efthymiou et al.¹¹ focuses on data that are encoded in tables. They presented three basic approaches to the problem and some combinations, namely: a look-up method, a word-embedding method, and an ontology-matching method. The look-up method attempts to match the data in the cells of a row to the values of the data properties of an entity in the knowledge graph; the disambiguation is addressed by finding the minimal set of cells that almost univocally identify each record. The word-embedding method first finds an embedding for the entities in the knowledge graph using random walks through their properties; it then finds an embedding for the rows of the tables by concatenating the corresponding columns; given the two previous embeddings, it is relatively easy to find which entities correspond to each row using distance functions. The ontology-matching method uses the column headers as candidate property names and then runs a third-party ontology-matching tool to find an alignment among the columns and the properties in the knowledge graph. Very recently, Oulabi and Bizer¹² presented a proposal that also focuses on tables. It first performs schema matching in an attempt to find the class that best describes a table and its columns; then, it uses a variety of scoring, grouping, selection, and fusion techniques to determine which cells can be grouped and considered a single entity; finally, it checks which of the entities in the table cannot be mapped onto any of the entities in the knowledge graph and can then be considered new.

2.2 | Embedding proposals

Generally speaking, an embedding is a function that maps multidimensional objects onto lower dimensional objects that preserve the distances among the original objects. Using embeddings has proven very useful to learn classifiers that are impossible to learn using the original objects. Typically, machine learning techniques work on data sets that consist of vectors of real numbers; there are also a few techniques that can work with vectors whose components are categorical or ordinal; but there are not many proposals that can work with interrelated data. Embedding techniques can easily

transform those interrelated data into vectors of real numbers from which a classifier can be learnt using virtually any machine-learning techniques.

We are interested in embedding techniques that work on graphs since HTML files can be naturally represented as DOM trees, which are a particular kind of graph. Note that synthesizing metadata tags for an HTML file basically amounts to finding the tags that best describe its DOM nodes. (Without any loss of generality, we can assume that nodes that have irrelevant data are tagged with a null tag). That is, graph embedding techniques may in theory help learn the corresponding classifiers. Unfortunately, we have not found any records in the literature regarding applying graph embedding techniques to synthesizing metadata tags for HTML files, which makes our research original. (Recall that Efthymiou et al.¹¹ tried an approach in which embeddings were used to find similar text, not to synthesize metadata tags.)

The proposals in the literature work on graphs of the form (N, E, A) , where N denotes a set of nodes, E denotes a set of edges between some nodes, and A is a function that endows each edge with an attribute. Wang et al.¹³ published a survey whose focus is on techniques to embed graphs in which the edge attributes are labels that endow them with semantics; Goyal and Ferrara¹⁴ presented a similar survey with a focus on techniques to embed graphs in which the edge attributes are weights that indicate how similar the connected nodes are; Cai et al.¹⁵ presented an additional survey in which the focus is on formalizing the different kinds of graphs and embedding representations in the literature. The three surveys agree in that the most common approaches to computing the embeddings rely on so-called factorization methods, structure-preservation methods, random-walk methods, and deep-learning methods, but there are also some hybrid methods; unfortunately, none of them is generally superior to the others, which means that it is necessary to perform some experimentation to find out the most appropriate for each scenario.

2.3 | Discussion

It is a bit surprising that only the proposal by Doderio et al.⁵ describes the solution from an engineering point of view. The others provide shallow descriptions of the tools that the authors have devised or focus on the techniques behind the scenes, without a clue on how they can be turned into working systems. Unfortunately, Doderio et al.'s⁵ proposal is only appropriate in a context in which the system that produces the HTML files can be reengineered to produce metadata tags, which is not generally the case. Our proposal is a system, for which we present a microservice architecture and a comprehensive description of its core component to learn metadata taggers; it does not require any reengineering of existing systems.

Adrian et al.'s⁶ proposal seems to be the seminal one since they introduced the idea of finding pieces of text in an HTML file that match an entity in a knowledge graph. The more recent proposals by Ngomo et al.,⁷ Eldesouky et al.,⁸ Burget,¹⁰ Efthymiou et al.,¹¹ or Oulabi and Bizer¹² basically differ regarding the mechanism that they use to find the matches and/or to resolve ambiguities. Unfortunately, Adrian et al.'s,⁶ Ngomo et al.'s,⁷ or Eldesouky et al.'s⁸ proposals cannot be applied in our context because they all require the data to be rendered using sentences in natural language. In our context, we have to deal with semistructured data in specification sheets, record listings, or tables, which are nongrammatical encodings. The proposal by Burget¹⁰ is the only that can work in such a context because it uses the DBpedia Spotlight component to provide an additional clue on the candidate tags that it generates, but their technique can use other clues from their regular-pattern entity recognizer and from their classifier, which assumes that the data are rendered using semistructured layouts. The problem is that it requires a knowledge graph that provides entities for every piece of data in the input HTML files. Not only is this a problem when dealing with very specific domains, but also when dealing with general domains for which the existing knowledge graphs are not complete enough. For instance, Oulabi and Bizer¹² analyzed three general domains that are well supported by the DBpedia knowledge graph; unluckily, they found 206 690 entities in their HTML files for which they could not find any matches in the knowledge graph. The proposals Efthymiou et al.¹¹ and Oulabi and Bizer's¹² can also be used in a semistructured context; unfortunately, they can only deal with tables that have headers at the top and arrange the data records horizontally. Our proposal was specifically tailored to working with semistructured data that are encoded using HTML and it does not assume that they are arranged using any particular layouts.

Graph embedding techniques are particularly interesting in this context because synthesizing metadata tags can be interpreted as a node classification problem. Graph embedding techniques allow to map the DOM nodes of a graph onto embeddings from which it is relatively easy to learn a classifier using many existing machine-learning techniques.¹⁶ We have implemented a tagger using some state-of-the-art graph embedders and many machine-learning techniques, but the results were not good at all since the best F_1 score attained in our experimental study was 0.69. We think that the problem with such embedders is manifold, namely: first, they require the user to preset the length of the embeddings, which is

problematic insofar it requires to perform grid search to find a good length that is not typically optimal; second, they attempt to preserve the distance between the DOM nodes that are connected by edges, which requires to transform the original attributes into real-valued attributes that miss their original classification power; third, but not less important: they do not learn reusable embedding models, but embedding models that can only be applied to the graphs from which they were learnt; simply put: each HTML file for which we need to synthesize metadata tags must be analyzed independently. Our proposal computes the optimum embedding length automatically, it preserves the original attributes, and it learns reusable tagging models.

Summing up: we describe our proposal and how to turn it into a working system, we address semistructured data that are not required to be encoded using a particular layout, we use a new embedding technique that does not require to preset the length of the embeddings, preserves the original attributes, and can learn reusable tagging models; furthermore, our experiments confirm that our proposal outperforms other state-of-the-art techniques and the differences were confirmed to be statistically significant.

3 | DESIGN OF OUR PROPOSAL

In this section, we describe the design of our proposal. First, we provide an overall picture and then report on its three main components. For every component, we provide a detailed description of its services, their communications, and their data models. In the sequel, we use notation $C:S$ to refer to service S as provided by component C ; in the service and communication diagrams, the message queues are identified using notation $W.M$, where W refers to the number of the workflow and M is a sequence number in that workflow.

3.1 | The overall picture

Our design revolves around three main components, namely: the `DataProcessor` component, which provides a number of utilities to transform HTML files, the `Learner` component, which is used to learn a tagging model from a set of HTML files, and the `Tagger` component, which is used to apply a tagging model to an HTML file in order to synthesize its corresponding metadata tags. Each component provides a number of services that communicate with each other by means of message queues. The services can be considered microservices because each of them provides a simple functionality within a complex workflow. This design was intended to facilitate deploying the components to a distributed system in which services can be easily scaled depending on the workload. Furthermore, distribution increases fault-tolerance, which is a key feature in real-world systems.

Regarding the technologies used to implement our design, we decided to use Spring technologies to implement the components and RabbitMQ to implement the message queues because they have proven to work well in many industrial projects. To implement most of the services, we used industrial-strength toolkits that are readily available, for example, Headless Chrome or Weka; the only exception was the service to learn tagging models and two ancillary services that rely on the new embedding technique that is presented in this article. We describe their details in the following section.

Figure 2 sketches our design. Please note that the only purpose is to provide an overall picture that helps understand the details that are described in the following subsections. Our focus here is on the three main components of our design and their interactions. There are two workflows involved, namely: the workflow that learns a tagging model from a set of HTML files and the workflow that applies a tagging model to an HTML file to synthesize its metadata tags.

The first workflow starts when the user sends a set of HTML files to the learner component. This component forwards the set to the data processor component, which computes a ground truth and returns it to the learner. The ground truth is composed of two sets, namely: a learning set from which a tagging model is learnt and a validation set that helps guide the learning process. Both sets are composed of documents with annotations; the documents are DOM-tree-based representations of the input HTML files and the annotations are mappings in which each DOM node gets an associated metadata tag that endows its contents with semantics. (We assume that there is a predefined null tag that is assigned to the DOM nodes that do not provide any data.) Then, the learner infers a tagging model, which is composed of a path and a classifier; the path is a specification of a walk through a DOM tree that selects the DOM nodes that provide the best attributes to learn the classifier from the ground truth. The component searches the space of paths and classifiers

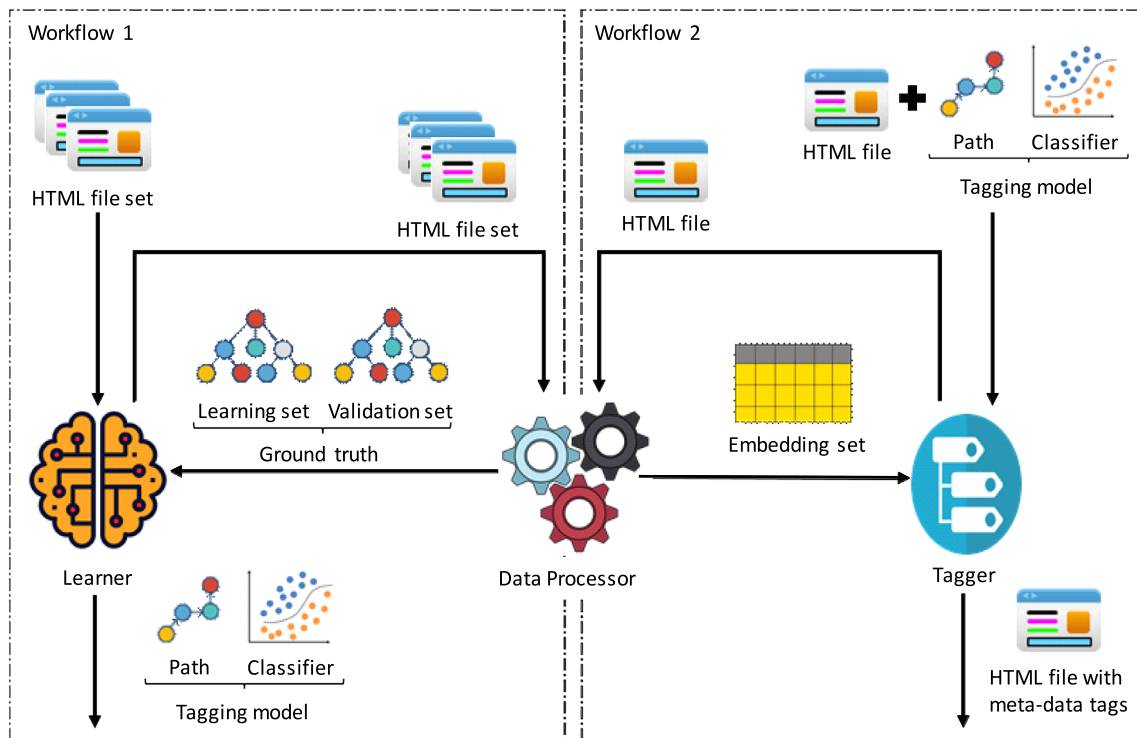


FIGURE 2 Overall picture [Colour figure can be viewed at wileyonlinelibrary.com]

incrementally; when it finishes the exploration, it assembles a tagging model from the best path and classifier that it has found and returns it to the user.

The second workflow starts when the user sends an HTML file and a tagging model to the system, which are received by the tagger component. This component sends the input HTML file to the data processor, which transforms it into an embedding set and returns it. An embedding is a vector that represents a DOM node using its attributes plus the attributes of its neighbors along the path in the tagging model. Then, the tagger applies the classifier in the tagging model to the embedding set and predicts a metadata tag for every node (including the null tag for the DOM nodes that do not provide any interesting data). The tags that are not null are then introduced in the corresponding elements of the input HTML file, which is returned to the user.

Our main innovation is regarding the learner component, which implements a new embedding technique that has proven to be superior to other state-of-the-art embedding techniques from an empirical point of view. However, the data processor component is a central part of the system since it is responsible for transforming the inputs to the system into data structures that facilitate implementing the services. This is the reason why we describe the data processor component first, then report on the learner component, and finally present the tagger component. The methods behind the key services are described in the following section.

3.2 | The DataProcessor component

The DataProcessor component provides three services that help transform the input HTML files into the data structures required by the Learner and the Tagger components, namely: the DataProcessor::Parser service, the DataProcessor::GroundTruthGenerator service, and the DataProcessor::EmbeddingSetGenerator service. Figure 3 presents its services-and-communications diagram and Figure 4 presents its data model.

The DataProcessor::Parser service transforms the input HTML files into documents. An HTML file has the text that is downloaded from a given URL, whereas a document represents the HTML file in a structured format that is amenable for further processing. The DataProcessor::Parser service can be used by the DataProcessor::GroundTruthGenerator service (1.3, 1.4) or the Tagger component (2.2, 2.3). Note that the former requires to transform a set of HTML files, whereas the latter

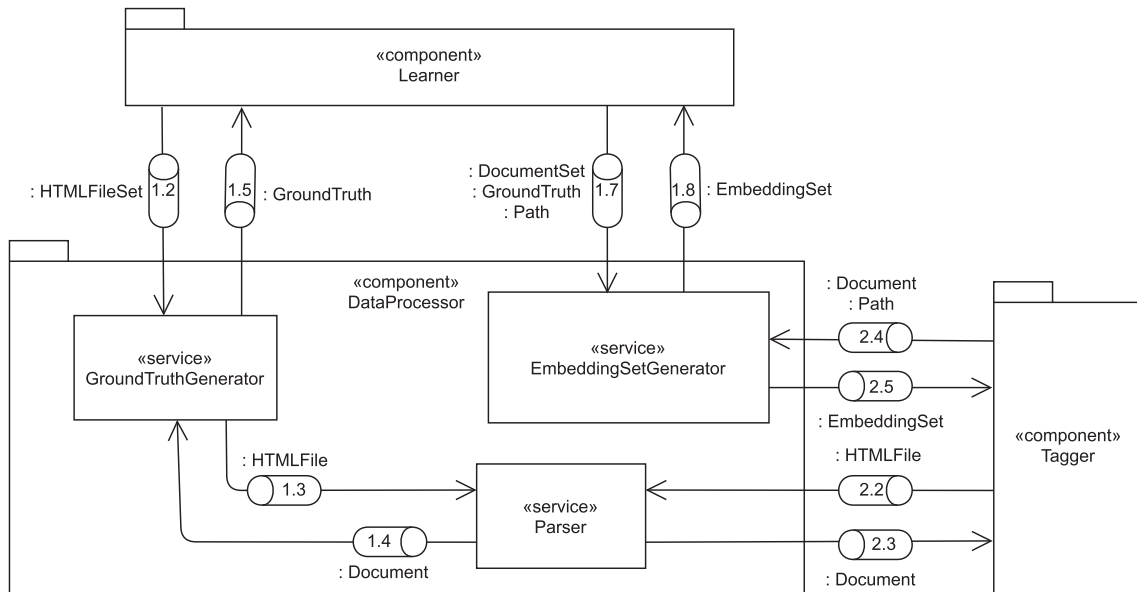


FIGURE 3 The data processor: services and communications

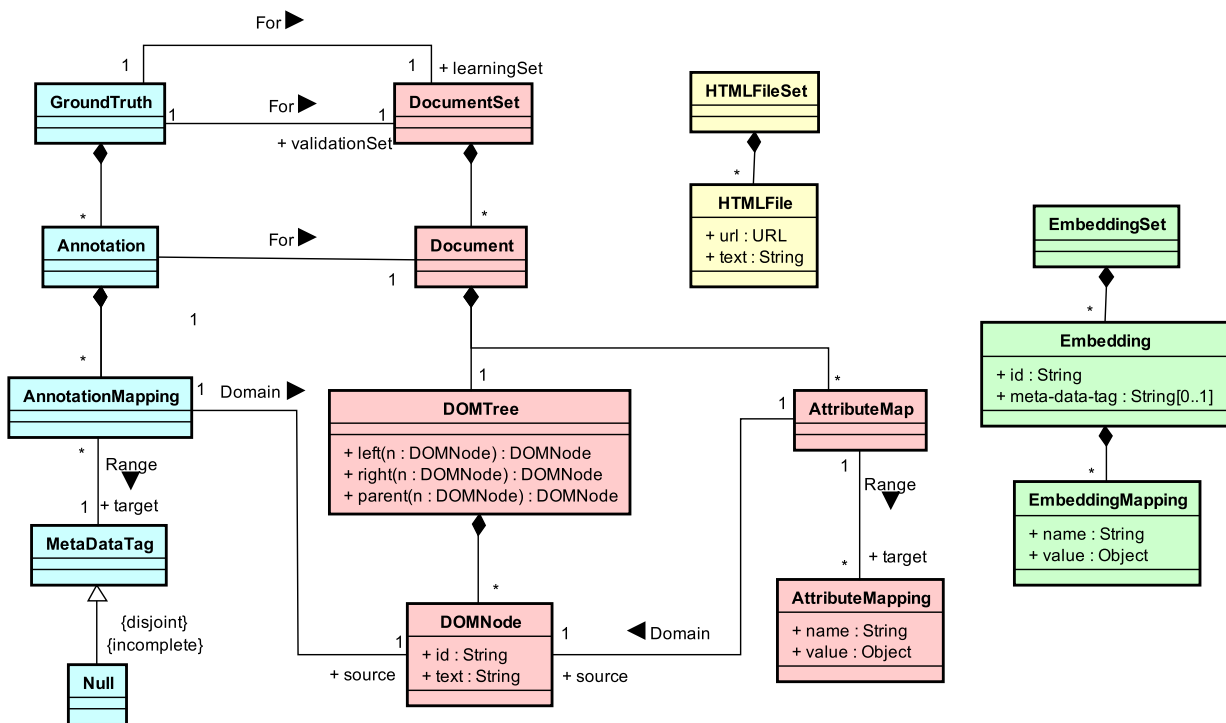


FIGURE 4 The data processor: data model [Colour figure can be viewed at wileyonlinelibrary.com]

only requires to transform one HTML file, which is the reason why we implemented two independent message queues. Parsing is a complex process that requires to tokenize the input HTML files, to analyze and correct their grammar so that it adheres to the HTML5 recommendation, and to execute the scripts so that the file can be rendered on a virtual canvas where a number of predefined attributes related to the DOM structure, the HTML, and the rendering can be computed, as well as a number of user-defined attributes. Our choice to implement the parser service was Headless Chrome, which is a solid industrial-strength headless browser. The model of the HTML files is very straightforward: note that an HTML file set is a collection of HTML files, each of which has a URL and a piece of text. The model of the documents is a bit

more involved: each document has a DOM tree and a collection of attribute maps. The DOM tree provides the collection of DOM nodes in an input HTML file plus some methods to fetch their left, right, or parent DOM nodes. The attribute map is a collection of mappings that can be interpreted as a vector that provides the values of the attributes of a particular DOM node.

The `DataProcessor::GroundTruthGenerator` service is used by the Learner component to transform the input HTML files (1.2) into ground truths (1.5). First, it transforms the input HTML files into documents by means of the `DataProcessor::Parser` service (1.3, 1.4). It then splits the set of documents into a learning set and a validation set. The learning set provides sample documents from which a tagging model must be learnt; the validation set provides sample documents that help guide the search for the best possible tagging model. Simply put: our proposal learns several tagging models from the learning set and returns the one that attains the best score on the validation set. The documents in the ground truth have annotations, which are mappings that make it explicit the metadata tag that corresponds to each DOM node (including the null tag in cases in which a DOM node does not provide any relevant data).

The `DataProcessor::EmbeddingSetGenerator` service transforms a set of documents or a single document into an embedding set that provides a vector-based representation of the DOM nodes in the documents. The embeddings are vectors with the attributes of the DOM nodes plus the attributes of their neighbors along a given path. The `DataProcessor::EmbeddingSetGenerator` service can be used by the Learner component (1.7, 1.8) and the Tagger component (2.4, 2.5). In the first case, it gets a document set, a ground truth that provides the annotations to the DOM nodes in the input documents, and a path and computes the embedding set; in the second case, it gets only a document and a path. Realize that the first case corresponds to a situation in which the Learner component is going to learn a tagging model, which implies that it has a ground truth; but the second case corresponds to a situation in which the Tagger component is used to synthesize the metadata tags of an input HTML file, which means that there is not a ground truth available. The embedding sets are typically represented as tables in which the columns refer to the attributes of the DOM nodes along a particular path and the rows correspond to the DOM nodes and their neighbors along that path. This representation is required to apply common machine-learning techniques to learn a classifier, to assess how well it performs using a performance scorer, or to apply it to a new HTML file. Note that our embedding model has two attributes called `id` and `meta-data-tag`, which provide a unique identifier to each DOM node and its corresponding metadata tag, respectively. Note that the `id` attribute is used to identify the DOM nodes during debug sessions only; note, too, that the metadata tag is optional because it must be present in the input HTML files used for learning purposes, but not in the HTML files for which the system must synthesize them. The materialization of an embedding is a collection of mappings in which each attribute in each DOM node along a path has an explicit name and an explicit value.

Example 1. Figure 5 sketches the data structures that we construct from the sample document in Figure 1. We represent it using a table in which each row corresponds to a DOM node of the HTML file; the columns report on the DOM nodes, their annotations, the DOM tree, and its attributes. Regarding the DOM nodes, we show the identifier of each DOM node and its text in order to facilitate the references; note that we do not require the identifier to adhere to a particular format, just to identify each DOM node univocally; note, too, that the text refers to the text in each node, not its children. Regarding the annotation, we use the sample metadata tags `Hotel`, `name`, and `postalCode`, plus the null metadata tag. The next column sketches the DOM tree using a visual representation that makes it clear which elements correspond to each DOM node and how they are connected. The remaining columns show a map with some sample attributes, namely: the tag of the node (`tag`), its HTML class (`class`), the co-ordinates of the upper-left corner of its bounding box (`x-pos` and `y-pos`), its depth in the DOM tree (`depth`), and an indication on whether it contains a number or not (`is-number`). Appendix A1 reports on the catalog of attributes used in our proposal.

Example 2. Figure 6 sketches two sample embedding sets. The embedding set at the top represents the embeddings where the learning process starts; note that the initial path is empty ($p = \langle \rangle$), which implies that the initial embeddings consists of the attributes of the DOM nodes only. The embedding set at the bottom corresponds to a stage in which the learning process is exploring path $p = \langle \text{Link.PARENT}, \text{Link.PARENT} \rangle$; intuitively, the path is composed of links that relate every node to some of their neighbors; in this example, every node is related to its parent and its grandparent and its embedding consists of its attributes, the attributes of its parent, and the attributes of its grandparent node. The notation p_i , where p is a path and i is an integer in range $[0..|p|]$ denotes the partial path up to the i th link. The blank cells indicate undefined values in cases in which a node does not have a parent or a grandparent node; in our implementation, we deal with such cases using a special value that is specifically generated to ensure that it can be unambiguously identified.

DOM Node		Annotation	DOM Tree	Attribute Map						
id	text	meta-data tag		tag	class	y-pos	x-pos	depth	is-number	
n ₁		null	▼	html	null	0	0	1	false	
n ₂		null	▶	head	null	0	0	2	false	
n ₃		null	▼	body	null	0	0	2	false	
n ₄	<i>Hotel list</i>	null	▼	h ₁	null	0	0	3	false	
n ₅		null	▼	ul	list	40	0	3	false	
n ₆		Hotel	▼	li	null	40	40	4	false	
n ₇	<i>Meriton..Street</i>	name	▶	span	null	41	40	5	false	
n ₈	<i>4000</i>	postalCode	▶	span	null	41	266	5	true	
n ₉		Hotel	▼	li	null	64	40	4	false	
n ₁₀	<i>Green..Hotel</i>	name	▶	span	null	65	40	5	false	
n ₁₁	<i>4225</i>	postalCode	▶	span	null	65	185	5	true	
n ₁₂		null	▼	ul	legal	104	0	3	false	
n ₁₃		null	▼	li	null	104	40	4	false	
n ₁₄	<i>&copy;</i>	null	▶	span	null	105	40	5	false	
n ₁₅	<i>2020</i>	null	▶	span	null	105	59	5	true	

FIGURE 5 Sketch of a sample document with annotations [Colour figure can be viewed at wileyonlinelibrary.com]

3.3 | The Learner component

The Learner component provides four services that help learn a tagging model from a set of input HTML files, namely: the Learner::TaggerLearner service, the Learner::ClassifierLearner service, the Learner::BaseLearner service, and the Learner::PerformanceScorer service. Figure 7 presents its services-and-communications diagram and Figure 8 presents the details of its data model.

The core is the Learner::TaggerLearner service. It implements a new technique to learn a tagging model from a set of input HTML files that have sample metadata tags. A tagging model has a path and a classifier. The path is a sequence of links that connect every DOM node in an input HTML file with some neighbors whose attributes allow to learn the best possible classifier. The links can be either Link.LEFT, Link.RIGHT, or Link.PARENT, which indicate the left sibling, the right sibling, or the parent of a DOM node, respectively. The classifiers are learnt by means of service Learner::BaseLearner, which provides an array of machine-learning techniques to learn the classifiers. We implemented it using Weka,¹⁶ which is a well-known machine-learning toolkit that integrates smoothly with Java technologies. The performance of the classifiers is measured using the Learner::PerformanceScorer service, which is also implemented using Weka.

The workflow starts when a user sends a set of HTML files to the system. The set is received by the Learner::TaggerLearner service (1.1). First, it sends the input HTML files to the DataProcessor component to create a ground truth (1.2, 1.5). Then, it starts a loop with an empty path; in each iteration, it sends the ground truth and the path to the Learner::ClassifierLearner service (1.6), which works in three steps: first, it sends the ground truth and the path to the DataProcessor component, which transforms the learning and the validation document sets into the corresponding embedding sets and returns them (1.7, 1.8). Then, the Learner::ClassifierLearner service forwards the learning embedding set to the Learner::BaseLearner service (1.9), which applies a machine-learning method to infer a classifier

id	meta-data tag	$p_i = \langle \rangle$					
		tag	class	y-pos	x-pos	depth	is-number
n ₁	null	html	null	0	0	1	false
n ₂	null	head	null	0	0	2	false
n ₃	null	body	null	0	0	2	false
n ₄	null	h ₁	null	0	0	3	false
n ₅	null	ul	list	40	0	3	false
n ₆	Hotel	li	null	40	40	4	false
n ₇	name	span	null	41	40	5	false
n ₈	postalCode	span	null	41	266	5	true
n ₉	Hotel	li	null	64	40	4	false
n ₁₀	name	span	null	65	40	5	false
n ₁₁	postalCode	span	null	65	185	5	true
n ₁₂	null	ul	legal	104	0	3	false
n ₁₃	null	li	null	104	40	4	false
n ₁₄	null	span	null	105	40	5	false
n ₁₅	null	span	null	105	59	5	true

(A) Sample embedding set for an empty path $p = \langle \rangle$.

id	meta-data tag	$p_i = \langle \rangle$						$p_{ii} = \langle \text{link.PARENT} \rangle$						$p_{iii} = \langle \text{link.PARENT}, \text{link.PARENT} \rangle$					
		tag	class	y-pos	x-pos	depth	is-number	tag	class	y-pos	x-pos	depth	is-number	tag	class	y-pos	x-pos	depth	is-number
n ₁	null	html	null	0	0	1	false												
n ₂	null	head	null	0	0	2	false	html	null	0	0	1	false						
n ₃	null	body	null	0	0	2	false	html	null	0	0	1	false						
n ₄	null	h ₁	null	0	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₅	null	ul	list	40	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₆	Hotel	li	null	40	40	4	false	ul	list	40	0	3	false	body	null	0	0	2	false
n ₇	name	span	null	41	40	5	false	li	null	40	40	4	false	ul	list	40	0	3	false
n ₈	postalCode	span	null	41	266	5	true	li	null	40	40	4	false	ul	list	40	0	3	false
n ₉	Hotel	li	null	64	40	4	false	ul	list	40	0	3	false	body	null	0	0	2	false
n ₁₀	name	span	null	65	40	5	false	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₁	postalCode	span	null	65	185	5	true	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₂	null	ul	legal	104	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₁₃	null	li	null	104	40	4	false	ul	legal	104	0	3	false	body	null	0	0	2	false
n ₁₄	null	span	null	105	40	5	false	li	null	104	40	4	false	ul	legal	104	0	3	false
n ₁₅	null	span	null	105	59	5	true	li	null	104	40	4	false	ul	legal	104	0	3	false

(B) Sample embedding set for path $p = \langle \text{Link.PARENT}, \text{Link.PARENT} \rangle$.

FIGURE 6 Sample embedding sets

and returns it (1.10). Finally, the `Learner::ClassifierLearner` service forwards the classifier and the validation embedding set to the `Learner::PerformanceScorer` service (1.11), which computes a performance score and returns it to the `Learner::ClassifierLearner` service (1.12). Our choice was to use the F_1 score, which combines precision and recall homogeneously. The `Learner::ClassifierLearner` service simply returns the classifier and the score to the `Learner::TaggerLearner` service (1.13), which must check if the previous partial path was improved, in which case it initiates a new cycle, or not, in which case it assembles the tagging model using the best path and the best classifier that it has found and returns it to the user (1.14). To extend the paths, it appends new links to the left sibling, the right sibling, and the parent of the DOM nodes.

Example 3. Figure 9 sketches a sample tagging model for our running example. In this case, the path is $p = \langle \text{Link.PARENT}, \text{Link.PARENT} \rangle$, which means that the classifier was learnt from an embedding set in which each DOM node was embedded using its attributes, the attributes of its parent node, and the attributes of its grandparent node. The classifier helps discern which of those attributes are really important regarding the embedding. In our running example, these attributes

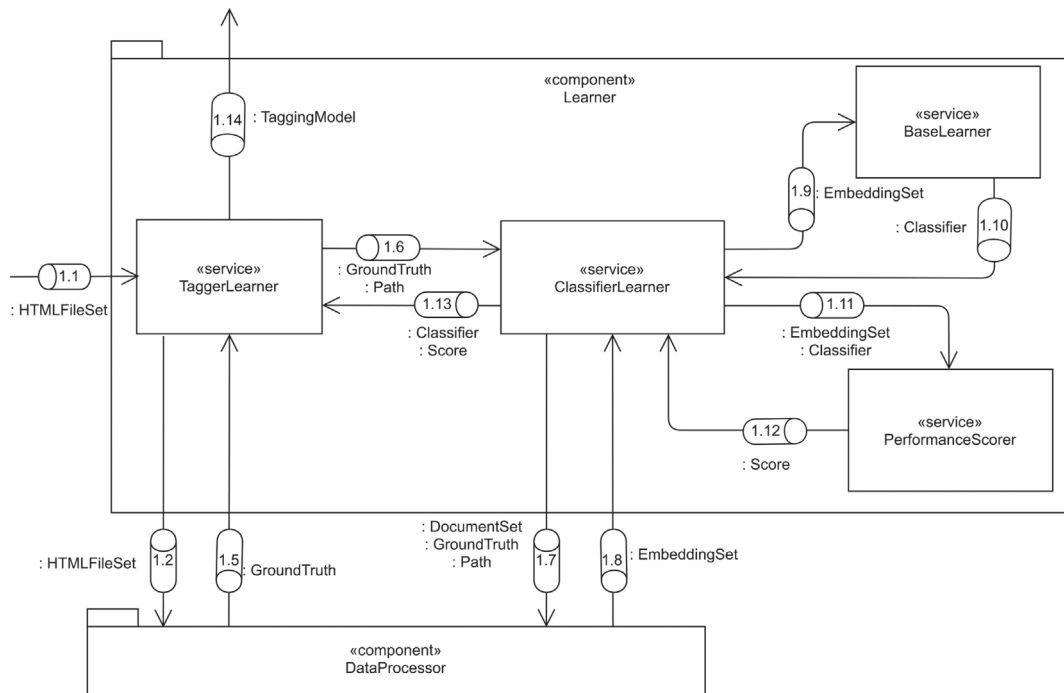


FIGURE 7 Learner: services and communications

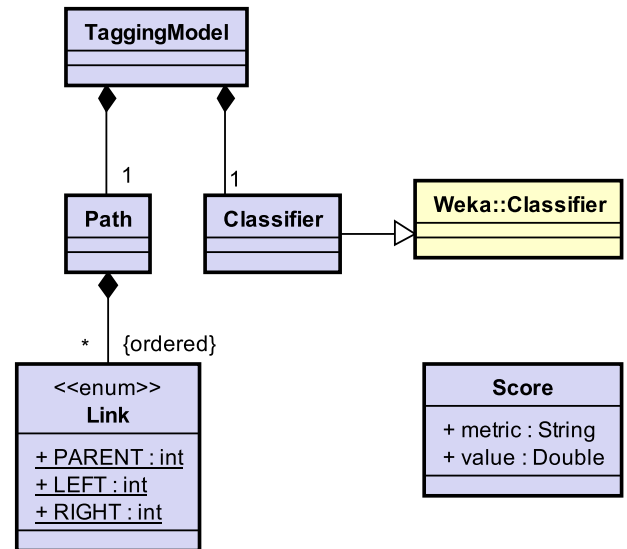
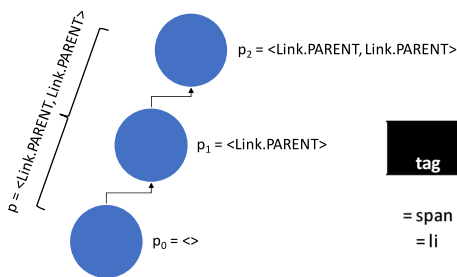


FIGURE 8 Learner: data model [Colour figure can be viewed at wileyonlinelibrary.com]



(A) Sample path.

	$p_0 = \langle \rangle$			$p_1 = \langle \text{Link.PARENT} \rangle$	$p_2 = \langle \text{Link.PARENT}, \text{Link.PARENT} \rangle$	meta-data tag
tag	y-pos	x-pos	is-number	class	class	
= span		≥ 60	true			postalCode
= li		≤ 40		= list	= list	name
		≥ 40				Hotel
						null

(B) Sample classifier.

FIGURE 9 Sample tagging model [Colour figure can be viewed at wileyonlinelibrary.com]

are the tag, the y-pos, the x-pos, and the is-number attribute of the DOM nodes, plus the classattribute of their parents and grandparents. In this example, the classifier can be easily represented using a decision table in which each row corresponds to a rule that is executed in sequence; the initial columns refer to the attributes of the DOM nodes along the path, the last column refers to the synthesized metadata tag, and the cells are of the form θv , where θ denotes an operator and v a value. For instance, the first rule indicates that a DOM node can be tagged as postalCode as long as its horizontal position is greater than or equal to 60 and it can be parsed as a number. The last rule is a catch-all rule, that is: if none of the previous rules can be applied to a DOM node, then it is tagged with a null metadata tag to indicate that it is irrelevant.

3.4 | The Tagger component

The tagger component provides two services that help apply a tagging model to an HTML file, namely: the Tagger::Synthesiser service and the Tagger::BaseSynthesiser service. Figure 10 presents its services-and-communications diagram. This component relies on the data models provided by the other components, which means that it does not require any specific-purpose data structures.

The workflow starts when the user sends an HTML file and a tagging model to the system. The request is received by the Tagger::Synthesiser service (2.1), which forwards the input HTML file to the DataProcessor::Parser service (2.2) so that it transforms it into a document (2.3). Then, the Tagger::Synthesiser service sends it together with the path in the input tagging model to the DataProcessor::EmbeddingSetGenerator service (2.4), which returns a set with the corresponding embeddings (2.5). Then, it only remains to send this set of embeddings and the classifier provided by the input tagging model to the Tagger::BaseSynthesiser (2.6). This service uses Weka¹⁶ to interpret the model and to apply it to the embedding set. It returns an embedding set to the Tagger::Synthesiser service in which each embedding is extended with an additional attribute that indicates which metadata tag describes its semantics (2.7). Finally, the Tagger::Synthesiser service rewrites the input HTML file to make it explicit the metadata tags that are not null. The result is returned to the user (2.8).

Example 4. Figure 11 sketches the results of a sample synthesis. The HTML file in the upper left corner is similar to the one that we have used in the previous examples for learning purposes, which means that the tagging model learnt previously can be applied to it. We assume that this HTML file does not have any metadata tags; the synthesis process concludes with the HTML file on the upper right corner, in which the li elements have metadata tags that indicate that their data are structured according to type Hotel in the schema provided by example.org and the span elements, but the last one, provide the values for the name and the postalCode attributes. The lower part shows the embedding set computed from the input HTML file using the path in the tagging model after the classifier in Figure 9 is applied; the only difference with respect to the initial embedding set is that the cells in column meta-data tag are set to the metadata tags predicted by the classifier. Note that we present all of the attributes of the nodes for the sake of completeness only; in our implementation, it suffices to store the attributes that are used by the classifier since this may save much storage space when dealing with many HTML files.

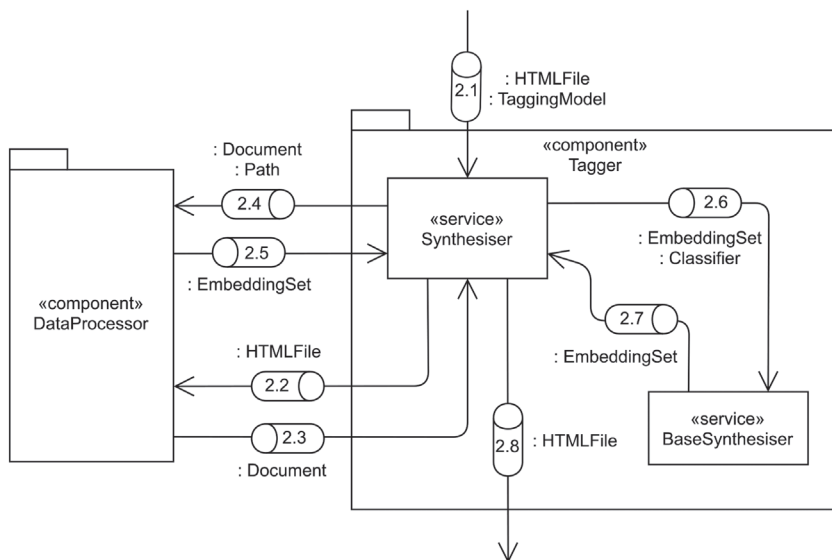
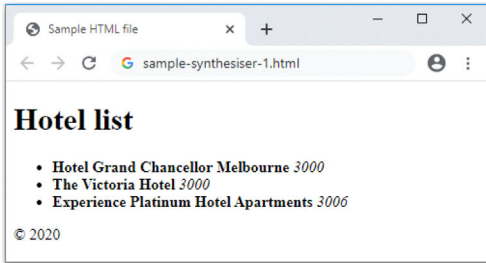


FIGURE 10 Tagger: services and communications



```

<html>
<head> ... </head>
<body>
  <h1>Hotel list</h1>
  <ul class="list">
    <li vocab="http://example.org/#" typeof="Hotel">
      <span property="name">Hotel Grand Chancellor Melbourne</span>
      <span property="postalCode">3000</span>
    </li>
    <li vocab="http://example.org/#" typeof="Hotel">
      <span property="name">The Victoria Hotel</span>
      <span property="postalCode">3000</span>
    </li>
    <li vocab="http://example.org/#" typeof="Hotel">
      <span property="name">Experience Platinum Hotel Apartments</span>
      <span property="postalCode">3006</span>
    </li>
  </ul>
  <ul class="legal">
    <li><span>&copy;</span> <span>2020</span></li>
  </ul>
</body>
</html>

```

(A) New HTML file with no meta-data tags.

(B) Meta-data tags synthesised for the new HTML file.

id	meta-data tag	$p_0=<>$						$p_1=<link.PARENT>$						$p_2=<link.PARENT, link.PARENT>$					
		tag	class	y-pos	x-pos	depth	is-number	tag	class	y-pos	x-pos	depth	is-number	tag	class	y-pos	x-pos	depth	is-number
n ₁	null	html	null	0	0	1	false												
n ₂	null	head	null	0	0	2	false	html	null	0	0	1	false						
n ₃	null	body	null	0	0	2	false	html	null	0	0	1	false						
n ₄	null	h ₁	null	0	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₅	null	ul	list	40	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₆	Hotel	li	null	40	40	4	false	ul	list	40	0	3	false	body	null	0	0	2	false
n ₇	name	span	null	41	40	5	false	li	null	40	40	4	false	ul	list	40	0	3	false
n ₈	postalCode	span	null	41	266	5	true	li	null	40	40	4	false	ul	list	40	0	3	false
n ₉	Hotel	li	null	64	40	4	false	ul	list	40	0	3	false	body	null	0	0	2	false
n ₁₀	name	span	null	65	40	5	false	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₁	postalCode	span	null	65	185	5	true	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₂	Hotel	li	null	64	40	4	false	ul	list	40	0	3	false	body	null	0	0	2	false
n ₁₃	name	span	null	65	40	5	false	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₄	postalCode	span	null	65	185	5	true	li	null	64	40	4	false	ul	list	40	0	3	false
n ₁₅	null	ul	legal	104	0	3	false	body	null	0	0	2	false	html	null	0	0	1	false
n ₁₆	null	li	null	104	40	4	false	ul	legal	104	0	3	false	body	null	0	0	2	false
n ₁₇	null	span	null	105	40	5	false	li	null	104	40	4	false	ul	legal	104	0	3	false
n ₁₈	null	span	null	105	59	5	true	li	null	104	40	4	false	ul	legal	104	0	3	false

(C) Embedding set of the sample HTML file along path $p = \langle Link.PARENT, Link.PARENT \rangle$.

FIGURE 11 Sample synthesis results [Colour figure can be viewed at wileyonlinelibrary.com]

4 | THE KEY SERVICES

In this section, we focus on describing the method that implements the `Learner::TaggerLearner` service. We also describe the methods that implement the ancillary services `Learner::ClassifierLearner` and `DataProcessor::EmbeddingSetGenerator`, which are well supported by industrial software components, but have some logic that is specific to our system.

4.1 | Learning a tagger

Figure 12 presents the method that is implemented by the `Learner::TaggerLearner` service. It gets an HTML file set F as input and returns a tagging model t as output. It works in three steps that we describe below.

```

method Learner::TaggerLearner::learn
  input F: HTMLFileSet
  output t: TaggingModel
  local p, p*: Path; c, c*: Classifier; s, s*: Score; E: Set(Path)

  – Step 1: initialisation
  G = DataProcessor::GroundTruthGenerator::generate(F)
  p = ⟨⟩
  (c, s) = Learner::ClassifierLearner::learn(G, p)
  E = ∅
  (p*, c*, s*) = (p, c, s)
  – Step 2: explore the neighbours to improve the classifier
  loop
    for l ∈ {Link.LEFT, Link.RIGHT, Link.PARENT} do
      p' = p + ⟨l⟩
      if p' is not redundant in E then
        E = E ∪ {p'}
        (c', s') = Learner::ClassifierLearner::learn(G, p')
        if s' > s* then
          (p*, c*, s*) = (p', c', s')
        end
      end
    end
    exit loop if c* = c
  end
  (p, c, s) = (p*, c*, s*)
end

– Step 3: assemble the resulting tagging model
t = new TaggingModel(path = p*, classifier = c*)
end

```

FIGURE 12 Method of the Learner::TaggerLearner service

The first step initializes the variables used by the algorithm. First, it invokes the `DataProcessor::GroundTruthGenerator` service to transform the input HTML file set `F` into a ground truth `G`. Next, it initializes variable `p` to an empty sequence, which represents the empty partial path at which the learning process starts. Immediately after, it invokes the `Learner::ClassifierLearner` service to learn a classifier and compute its performance score using ground truth `G`. The results of this service are stored in variables `c` and `s`, respectively. Finally, variable `E` is initialized to an empty set; later, this variable will be used to store the paths that have been explored in an attempt not to reexplore any redundant paths. The last statement initializes variables `p*`, `c*`, and `s*`, which store the path, the classifier, and the score computed previously. In the second step, these variables are updated to record the best alternatives found by the method.

The second step explores the neighbors of the DOM nodes in sequence. The exploration basically amounts to extending the current path `p` with links to the left sibling, the right sibling, and the parent node, as long as the extensions are not redundant. A path is redundant if it results in an embedding that uses the same DOM nodes as a path that was explored previously. (We do not provide a pseudo-code to this method since it is straightforward.) The non-redundant paths are added to set `E` and a new classifier is learnt for each of them; variables `c*`, `p*`, and `s*` are updated accordingly if the score of any of the new classifiers is greater than the best score found so far. Recall that the classifier is learnt from the learning set in the ground truth `G` and the score is computed from the validation set. Summing up: the second step implements a hill climbing strategy in which the goal is to extend the current path to the neighbor DOM node that helps learn a better classifier. The strategy stops when no extension to the current path can learn a better classifier. (Note that it is not necessary to make it explicit the case in which no extension can explore a new node because such extensions do not help learn a better classifier; that is, that case is implicitly included in the previous general stopping criterion.)

The third step is quite simple, since it assembles the tagging model that is returned by the method using the best path and the best classifier that have been found in the main loop.

FIGURE 13 Method of the Learner::ClassifierLearner service

```
method Learner::ClassifierLearner::learn
  input  G: GroundTruth; p: Path
  output c: Classifier; s: Score
  local L, V: DocumentSet; L', V': EmbeddingSet

  – Step 1: initialisation
  L = G.getLearningSet()
  V = G.getValidationSet()
  – Step 2: create embedding sets
  L' = DataProcessor::EmbeddingSetGenerator::create(L, G, p)
  V' = DataProcessor::EmbeddingSetGenerator::create(V, G, p)
  – Step 3: learn a classifier and compute its score
  c = Learner::BaseLearner::learn(L')
  s = Learner::PerformanceScorer::score(c, V')
end
```

4.2 | Learning a classifier

Figure 13 shows the method of the Learner::ClassifierLearner service. It gets a ground truth G and a path p as input and returns a classifier c and a score s as output. It works in three steps that we describe below.

The first step is an initialization step that simply retrieves the learning set L and the validation set V from the ground truth G. Recall that the former is used to actually learn the classifier, whereas the latter helps make decisions that help learn the best possible classifier from the former.

The second step invokes the DataProcessor::EmbeddingSetGenerator service to compute the embedding sets for the documents in the learning set and the validation set. Note that this method takes the learning or the validation set and the path as parameters, but it also requires the ground truth as an input parameter because it needs to fetch the annotations that correspond to their DOM nodes.

The third step uses the Learner::BaseLearner service to learn a classifier from the learning embedding set and then uses the Learner::PerformanceScorer service to compute a score on the validation embedding set. Regarding the base learner, our proposal supports any techniques that can work with multiple metadata tags and numeric, categorical, and ordinal attributes, namely: Bayes Networks, *k*-Nearest Neighbors, C4.5, Repeated Incremental Pruning to Produce Error Reduction, Naive Bayes, PART Decision Lists, Random Forests, and Sequential Minimal Optimization. Regarding the performance scorer, there are also a variety of choices,¹⁷ including the well-known F_1 score. In the following section, we report on the results of our experimental analysis, which helped us make a recommendation regarding which of the base learners performs the best.

4.3 | Creating an embedding set

Figure 14 shows the method of the DataProcessor::EmbeddingSetGenerator service. It gets a set of documents D, a ground truth G, and a path p as input. It returns an embedding set with the embeddings of the DOM nodes of the documents in D and their neighbors along path p as output. Note that the ground truth G is required so that the method has access to the annotations of the documents. The method works in two steps that we describe below.

The first step initializes the local variables of the method. It simply stores the annotation in the ground truth in variable M and then initializes the resulting embedding set E to an empty set. Note that the annotation behaves as a map; that is: given a DOM node it returns its corresponding metadata tag.

The second step computes the embeddings. It iterates over the input documents in D and their DOM nodes. In each iteration, it gets the attribute map of the document and then computes the embedding as vectors of the form $\{a_i = v_i\}_{i=1}^z$, where each a_i denotes an attribute name and v_i its corresponding value. Realize that the embedding of DOM node *n* is computed as the union of three maps, namely: the first one provides an identifier and a metadata tag to every embedding; the second one provides a vector with the predefined attributes, cf Table A1, and the user-defined attributes of the DOM

```

method DataProcessor::EmbeddingSetGenerator::create
  input D: DocumentSet; G: GroundTruth; p: Path
  output E: EmbeddingSet
  local M: Annotation; T: DOMTree

```

```

– Step 1: initialisation
M = G.getAnnotation()
E = ∅
– Step 2: computing embeddings
for d ∈ D do
  A = D.getAttributeMap()
  for n ∈ d.getNodes() do
    e = {id = n.getId(), meta-data-tag = M.get(n)} ∪ A.get(n) ∪
      ∪i=1|p| { A.get(m) | n reaches m on subpath pi }
    E = E ∪ {e}
  end
end
end
end

```

FIGURE 14 Method of the DataProcessor::EmbeddingSetGenerator service

node, cf Table A2; and the third one is the union of $A.get(m)$ for every DOM node m that is reachable from n along subpaths $p_1, p_2, \dots, p_{|p|}$, where p_i denotes the partial path up to the i th link.

5 | EXPERIMENTAL ANALYSIS

In this section, we first describe the experimental setting, then report on the performance results, and finally analyze them using statistically sound methods. We assembled a package that is available at <https://tinyurl.com/aquila-system>; it provides the repository, the HTML files, and the tools required to repeat our experimentation, including a stand-alone version of our services to learn a tagging model and to apply it.

5.1 | Experimental setting

Table 1 describes the repository of HTML files that we used to evaluate our proposal. It provides HTML files from 40 different web sites on eight different topics, namely: books, cars, doctors, events, films, jobs, players, and realty; we downloaded 30 HTML files from each web site. The categories were randomly selected from The Open Directory and the web sites were randomly selected from the 100 best-ranked web sites in each category according to Google’s search engine; the HTML files in each set were also sampled randomly from each web site. There were many cases in which we could not download enough HTML files from a web site due to a variety of problems, for example, server request limits, URLs that expire a little after they are generated, unrecoverable HTML markup errors, difficulties to reach the HTML files due to complex navigation paths that require too much user interaction, difficulties to generate the HTML on the client side due to scripts that fetch data from external resources, or difficulties to check whether two HTML files are the same or not; in such cases, we just selected another random web site and repeated the download procedure. The HTML file sets were split into learning sets with four HTML files, validation sets with two HTML files, and testing sets with 24 HTML files. Since the majority of nodes have a null tag, we balanced the resulting learning sets using Weka’s class balancer.¹⁶

We selected four state-of-the-art graph embedders as competitors, namely: ComplEx,¹⁸ HoLE,¹⁹ Node2Vec,²⁰ and TransD.²¹ We used the following procedure to transform the input HTML files into labeled graphs: both the DOM nodes and the values of their attributes were transformed into graph nodes; they were connected according to their links in the DOM tree or the values of their attributes; that is, the labels of the edges denote the corresponding link or attribute in the original DOM tree. The procedure to transform the input HTML files into weighted graphs was as follows: each DOM node was transformed into a graph node; the graph nodes were connected according to the links of their corresponding DOM

TABLE 1 Description of our data sets

Category	Schema (from Schema.org)	Site	URL	# DOM nodes	# Attr. values	# Node edges	# Attr. edges
Books	Book{name, author, isbn}	Abe Books	www.abebooks.com	30 078	14 837	28 225	20 0636
		Awesome Books	www.awesomebooks.com	20 513	10 873	53 805	411 709
		Better World Books	www.betterworldbooks.com	66 209	23 573	4342	32 003
		Many Books	www.manybooks.net	20 211	17 679	73 635	448 305
		Water Stones	www.waterstones.com	38 738	16 310	38 697	261 400
Cars	Car{color, numberOfDoors, vehicleEngine, model, mileageFromOdometer, numberOfForwardGears}	Auto Trader	www.autotrader.com	74 154	21 961	52 773	278 143
		Car Max	www.carmax.com	45 601	10 562	23 432	147 573
		Car Zone	www.carzone.ie	28 364	10 900	9782	31 024
		Classicals for Sale	www.classiccarsforsale.co.uk	59 162	20 424	10 924	6543
		Internet Auto Guide	www.internetautoguide.com	48 541	15 434	9676	52 387
Doctors	Doctor{name, address, telephone, additionalType}	Web MD	www.webmd.com	49 622	10 455	10 748	50 137
		Ame. Medical Association	www.ama-assn.org	26 304	7298	113 170	522 377
		Dentists	www.dentists.com	13 881	6452	1505	9638
		Dr. Score	www.drscore.com	14 620	7297	28 975	155 784
		Steady Health	www.steadyhealth.com	39 367	11 652	33 292	176 229
Events	Event{name, startDate, location, url}	Linked In	events.linkedin.com	11 671	12 888	7785	37 949
		All Conferences	www.allconferences.com	21 887	10 052	5 853	28 680
		M-Bendi	www.mbendi.com	8310	2361	41 547	268 066
		RD Learning	www.rdlearning.org.uk	5538	6737	129 721	530 739
		Wiki Cfp	www.wikicfp.com	11 851	8009	46 226	216 358
Films	Movie{name, director, actor, copyrightYear, duration}	Albanian Movies	www.albaniam.com	11 693	4262	13 356	49 958
		All Movie	www.allmovie.com	43 607	26 570	8237	27 394
		CIT-WF	www.citwf.com	11 682	8292	2852	15 312
		Disney Movies	www.disneymovies.com	23 681	10 744	49 643	317 042
		IMDB	www.imdb.com	80 524	44 100	81 588	391 091
Jobs	JobPosting{company, jobLocation, occupationalCategory}	Insight into Diversity	careers.insightintodiversity.com	25 558	13 307	14 101	105 239
		4Jobs	www.4jobs.com	33 399	21 271	1658	10 907
		6-Figure Jobs	www.6figurejobs.com	49 810	14 375	73 730	233 436
		Career Builder	www.careerbuilder.com	32 123	14 601	116 474	543 057
		Job of Mine	www.jobofmine.com	17 920	12 274	38 255	166 876
Players	Person{name, birthDate, height, weight, nationality}	Player Profiles	baseball.playerprofiles.com	31 170	14 372	12 184	30 830
		UEFA	en.uefa.com	7191	6189	6764	39 951
		ATP World Tour	www.atpworldtour.com	85 614	19 074	84 278	579111
		National Football League	www.nfl.com	80 333	14 561	1481	7736
		Soccer Base	www.soccerbase.com	96 013	14 465	154 822	525 299
Realty	Property{address, numberOfRooms, floorSize}	Yahoo! Real Estate	realestate.yahoo.com	36 749	13 396	22 370	145 167
		Haart	www.haart.co.uk	38 672	12 594	32 538	220 465
		Homes	www.homes.com	34 826	14 198	12 202	69 870
		Remax	www.remax.com	62 865	15 046	14 015	82 100
		Trulia	www.trulia.com	158 042	35 230	97 098	55 8793

nodes; the weights were computed by measuring the distance between the attributes of the corresponding nodes. Since the DOM nodes have real-valued, categorical, and ordinal attributes, we resorted to Foss et al.'s²² approach to compute the distances among them.

We used Weka¹⁶ to implement the base learners. In this context, we need base learners that can deal with multiclass problems (since typical HTML files require to synthesize more than two metadata tags) and numeric, categorical, or ordinal attributes (since both the predefined and the user-defined attributes are of many different types). Weka provides many different base learners that fulfill the previous requirements, namely: Bayes Networks (BN), k -Nearest Neighbors (IBK), C4.5 (J48), Repeated Incremental Pruning to Produce Error Reduction (JRIP), Naive Bayes (NB), PART Decision Lists (PART), Random Forests (RF), and Sequential Minimal Optimization (SMO).

We selected the F_1 score ($F1$) as the performance score since it balances precision (P), which measures the fraction of DOM nodes that must actually be assigned a particular metadata tag among the DOM nodes that are assigned that tag, and recall (R), which measures the fraction of DOM nodes that are assigned a particular metadata tag among the DOM nodes that must actually be assigned that tag. Weka¹⁶ provides good support to compute these scores, as well.

5.2 | Performance results

Table 2 shows the performance results that we computed from running the competitors and our proposal on our repository. (The figures are the averages across the different metadata tags in each schema.)

Regarding the competitors, they seem to help learn taggers that can attain good average recall (as high as 0.99 when combining Node2Vec and Random Forest), but poor average precision (as low as 0.10 when combining ComplEx and JRIP); in other words, the taggers learnt from their embeddings tend to return many nodes for each metadata tag and have little ability to make them apart. The best F_1 score for the competitors is 0.69, which was attained using HolE and Naive Bayes.

The empirical ranking of the competitors according to the best F_1 score they achieved is the following: HolE combined with Naive Bayes, which attained an F_1 score of 0.69, TransD combined with k -Nearest Neighbors, which attained an F_1 score of 0.57, Node2Vec combined with either C4.5, PART Decision Lists or Sequential Minimal Optimization, which achieved an F_1 score 0.29, and then ComplEx combined with k -Nearest Neighbors, which attained an F_1 score 0.29.

The results using our proposal are clearly better in every case: our precision ranges from 0.71 to 0.81 and the average is 0.78, our recall ranges from 0.67 to 0.81 and the average is 0.78, and our F_1 score ranges from 0.68 to 0.79 and the average is 0.76. The best result was attained using the Naive Bayes base learner and its net improvement is 10.14% regarding the F_1 score.

Our conclusion is that the embeddings computed by the competitors do not capture well the attributes of the original HTML files so that a good tagger can be inferred from them, whereas the embeddings computed by our proposal do. This conclusion is statistically confirmed in the following subsection.

5.3 | Analysis of the results

Table 3 shows the results of our statistical analysis. Each chart corresponds to a different base learner; the first column identifies the graph embedder used; the second column shows the empirical ranking attained.

Following the results by García and Herrera,²³ we have first used Iman-Davenport's test to find out if there are statistically significant differences in the empirical ranks. The third and the fourth columns show the corresponding statistic and P -values; note that the P -values are nearly zero in every case, which is a strong indication that the experimental results support the hypothesis that the differences in rank are statistically significant at the standard confidence level ($\alpha = .05$).

Therefore, we have to use Hommel's test to compare our proposal, which is the best-ranking one, to the others. The fifth and the sixth columns show the statistic and the P -value that result from every comparison. (We use dashes in the case of the first comparison since it does not make sense to compare our proposal to itself.) Note that the P -values are nearly zero in every case, which is a strong indication that the differences in rank between our proposal and the others are statistically significant at the standard confidence level ($\alpha = .05$).

The previous statistical results confirm our empirical conclusion: our embedding technique captures the features of the attributes of the original HTML files better than the other embedders, which results in taggers whose F_1 score can improve on its competitors as much as 10.14%.

TABLE 2 Performance results

Approach	Category	BN			IBK			J48			JRIP			NB			PART			RF			SMO		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ComplEx	Books	.14	0.80	0.24	.33	0.80	0.26	.14	0.77	0.24	.07	0.22	0.11	.14	0.80	0.24	.14	0.77	0.24	.14	0.80	0.24	.14	0.80	0.24
	Cars	.10	1.00	0.18	.10	0.99	0.18	.10	0.95	0.18	.10	0.61	0.17	.10	1.00	0.18	.10	0.95	0.18	.10	1.00	0.18	.10	1.00	0.18
	Doctors	.17	1.00	0.29	.17	0.99	0.28	.17	0.94	0.28	.11	0.39	0.17	.17	1.00	0.29	.17	0.94	0.28	.17	1.00	0.29	.17	1.00	0.29
	Events	.17	1.00	0.29	.17	0.98	0.29	.17	0.93	0.28	.05	0.24	0.09	.17	1.00	0.29	.17	0.93	0.28	.17	1.00	0.29	.17	1.00	0.29
	Films	.12	0.80	0.21	.12	0.79	0.21	.12	0.76	0.21	.05	0.26	0.09	.12	0.76	0.21	.12	0.76	0.21	.12	0.80	0.21	.12	0.75	0.21
	Jobs	.19	1.00	0.32	.19	0.99	0.32	.19	0.97	0.32	.19	0.64	0.29	.19	1.00	0.32	.19	0.97	0.31	.19	1.00	0.32	.19	1.00	0.32
	Players	.14	1.00	0.24	.14	1.00	0.24	.14	0.98	0.24	.10	0.36	0.14	.14	1.00	0.24	.13	0.98	0.24	.14	1.00	0.24	.14	1.00	0.24
	Realty	.12	0.75	0.20	.37	0.75	0.20	.37	0.74	0.21	.09	0.15	0.10	.12	0.75	0.20	.37	0.74	0.20	.12	0.75	0.20	.12	0.75	0.20
	Average	.14	0.92	0.24	.20	0.91	0.25	.17	0.88	0.24	.10	0.36	0.14	.14	0.91	0.24	.17	0.88	0.24	.14	0.92	0.24	.14	0.91	0.24
HoIE	Books	.14	0.80	0.24	.36	0.80	0.29	.26	0.80	0.32	.37	0.80	0.32	.74	0.77	0.75	.29	0.82	0.36	.36	0.80	0.29	.29	0.81	0.29
	Cars	.10	1.00	0.18	.15	1.00	0.26	.14	0.97	0.24	.14	0.85	0.24	.62	0.61	0.60	.13	0.97	0.22	.15	1.00	0.25	.12	0.93	0.21
	Doctors	.17	1.00	0.30	.18	0.81	0.30	.23	0.84	0.35	.27	0.60	0.36	.67	0.70	0.68	.22	0.84	0.35	.20	0.89	0.32	.17	0.88	0.28
	Events	.17	1.00	0.29	.19	0.99	0.32	.25	0.93	0.39	.24	0.88	0.38	.87	0.78	0.82	.25	0.93	0.39	.19	0.99	0.32	.18	0.99	0.30
	Films	.15	1.00	0.26	.18	0.99	0.31	.22	0.95	0.35	.34	0.84	0.47	.80	0.61	0.65	.23	0.96	0.37	.17	0.99	0.29	.16	0.94	0.27
	Jobs	.20	1.00	0.34	.22	0.98	0.36	.23	0.90	0.37	.39	0.82	0.51	.62	0.66	0.63	.23	0.97	0.37	.21	1.00	0.35	.21	1.00	0.35
	Players	.14	1.00	0.24	.16	1.00	0.27	.18	0.98	0.30	.15	0.84	0.25	.74	0.78	0.76	.18	0.98	0.30	.15	1.00	0.26	.15	0.97	0.26
	Realty	.09	0.60	0.16	.10	0.60	0.17	.11	0.59	0.18	.17	0.45	0.23	.63	0.59	0.60	.11	0.59	0.18	.30	0.60	0.17	.29	0.59	0.16
	Average	.15	0.92	0.25	.19	0.89	0.28	.20	0.87	0.31	.26	0.76	0.35	.71	0.69	0.69	.20	0.88	0.32	.22	0.91	0.28	.19	0.89	0.26
Node2Vec	Books	.18	1.00	0.31	.18	1.00	0.31	.19	0.96	0.31	.22	0.75	0.33	.16	0.58	0.25	.19	0.96	0.31	.18	1.00	0.31	.19	0.98	0.31
	Cars	.10	1.00	0.18	.10	1.00	0.18	.10	0.96	0.19	.12	0.94	0.22	.14	0.66	0.22	.10	0.95	0.19	.10	1.00	0.18	.10	0.97	0.19
	Doctors	.16	1.00	0.28	.16	1.00	0.28	.17	0.92	0.29	.12	0.45	0.19	.23	0.74	0.33	.16	0.93	0.28	.16	1.00	0.28	.17	0.97	0.28
	Events	.17	1.00	0.29	.17	1.00	0.29	.20	0.91	0.32	.23	0.74	0.35	.21	0.86	0.34	.20	0.87	0.32	.17	1.00	0.29	.18	0.97	0.31
	Films	.15	1.00	0.26	.15	1.00	0.26	.16	0.93	0.27	.13	0.63	0.22	.15	0.68	0.25	.16	0.93	0.27	.15	1.00	0.26	.16	0.93	0.27
	Jobs	.20	1.00	0.33	.20	1.00	0.33	.21	0.96	0.34	.24	0.85	0.38	.19	0.65	0.28	.21	0.96	0.35	.20	1.00	0.33	.21	0.97	0.34
	Players	.18	0.94	0.27	.17	0.94	0.28	.17	1.08	0.29	.19	0.81	0.30	.18	0.67	0.26	.19	0.92	0.31	.18	0.92	0.29	.17	0.95	0.30
	Realty	.17	1.00	0.29	.17	1.00	0.29	.17	0.97	0.29	.15	0.66	0.25	.19	0.77	0.30	.17	0.96	0.29	.17	1.00	0.29	.17	0.99	0.29
	Average	.16	0.99	0.27	.16	0.99	0.28	.17	0.96	0.29	.18	0.73	0.28	.18	0.70	0.28	.17	0.93	0.29	.16	0.99	0.28	.17	0.97	0.29
TransD	Books	.14	0.80	0.24	.59	0.96	0.70	.34	0.82	0.41	.32	0.82	0.36	.41	0.94	0.50	.30	0.82	0.41	.36	0.81	0.28	.57	0.92	0.65
	Cars	.10	1.00	0.18	.38	1.00	0.51	.17	0.98	0.29	.19	0.93	0.32	.13	1.00	0.23	.21	0.97	0.34	.11	1.00	0.20	.30	1.00	0.43
	Doctors	.16	1.00	0.28	.41	0.80	0.54	.31	0.84	0.44	.21	0.84	0.33	.20	0.89	0.33	0.30	0.83	0.43	.17	0.87	0.29	.35	0.80	0.48
	Events	.17	1.00	0.29	.41	1.00	0.58	.27	0.94	0.41	.23	0.96	0.37	.21	1.00	0.34	0.31	0.92	0.46	.19	1.00	0.32	.40	1.00	0.57
	Films	.15	1.00	0.26	.33	1.00	0.49	.25	0.97	0.40	.28	0.93	0.42	.20	0.99	0.33	0.27	0.97	0.42	.17	1.00	0.29	.29	1.00	0.45
	Jobs	.20	1.00	0.33	.35	0.89	0.50	.28	0.92	0.42	.29	0.88	0.44	.25	0.92	0.39	0.27	0.90	0.42	.22	1.00	0.35	.30	0.91	0.45
	Players	.14	1.00	0.25	.59	1.00	0.72	.26	0.99	0.40	.25	0.94	0.40	.22	1.00	0.35	0.25	0.99	0.40	.17	1.00	0.29	.48	1.00	0.62
	Realty	.10	0.60	0.17	.42	0.60	0.49	.18	0.59	0.28	.15	0.59	0.24	.32	0.65	0.32	0.18	0.59	0.28	.13	0.60	0.21	.37	0.60	0.46
	Average	.15	0.93	0.25	.44	0.91	0.57	.26	0.88	0.38	.24	0.86	0.36	.24	0.92	0.35	0.26	0.87	0.40	.19	0.91	0.28	.38	0.90	0.51
Ours	Books	.75	0.77	0.73	.82	0.88	0.81	.81	0.86	0.80	.85	0.87	0.80	.88	0.82	0.82	0.78	0.77	0.79	.84	0.84	0.76	.82	0.82	0.81
	Cars	.85	0.81	0.77	.75	0.79	0.80	.75	0.81	0.70	.82	0.74	0.71	.82	0.82	0.74	0.75	0.78	0.81	.81	0.73	0.77	.82	0.72	0.78
	Doctors	.81	0.85	0.84	.77	0.84	0.79	.74	0.77	0.85	.75	0.73	0.71	.89	0.79	0.84	0.86	0.73	0.78	.76	0.83	0.76	.84	0.78	0.74
	Events	.77	0.77	0.77	.82	0.79	0.84	.79	0.76	0.84	.77	0.81	0.81	.79	0.80	0.77	0.85	0.78	0.78	.78	0.75	0.80	.77	0.69	0.70
	Films	.84	0.76	0.77	.78	0.86	0.76	.81	0.83	0.83	.83	0.83	0.83	.79	0.81	0.80	0.83	0.69	0.74	.73	0.76	0.75	.45	0.43	0.45
	Jobs	.79	0.64	0.71	.73	0.67	0.68	.72	0.72	0.68	.70	0.75	0.76	.83	0.74	0.80	0.68	0.71	0.63	.73	0.75	0.78	.75	0.66	0.72
	Players	.85	0.86	0.85	.87	0.80	0.81	.79	0.81	0.78	.76	0.85	0.83	.77	0.87	0.86	0.79	0.79	0.83	.85	0.88	0.78	.62	0.70	0.69
	Realty	.72	0.79	0.78	.83	0.86	0.70	.77	0.77	0.76	.84	0.82	0.77	.74	0.76	0.71	0.75	0.81	0.75	.81	0.83	0.78	.58	0.55	0.54
	Average	.80	0.78	0.78	.80	0.81	0.77	.77	0.79	0.78	.79	0.80	0.78	.81	0.80	0.79	.79	0.76	0.76	.79	0.80	0.77	.71	0.67	0.68

TABLE 3 Analysis of results

(a) Bayes Networks (BN)						(b) k-Nearest Neighbors (IBK)					
Embedder	Ranking	Iman-Davenport		Hommel		Embedder	Ranking	Iman-Davenport		Hommel	
		Statistic	P -value	Statistic	P -value			Statistic	P -value	Statistic	P -value
Ours	1.00			—	—	Ours	1.25			—	—
HolE	3.34			6.61	$3.81E-11$	TransD	2.05			2.26	$2.37E-02$
Node2Vec	3.34	41.01	$1.86E-23$	6.61	$3.81E-11$	HolE	3.78	72.07	$1.81E-34$	7.14	$1.84E-12$
TransD	3.56			7.25	$1.27E-12$	Node2Vec	3.78			7.14	$1.84E-12$
Complex	3.76			7.81	$2.22E-14$	Complex	4.15			8.20	$9.42E-16$
(c) C4.5 (J48)						(d) Rep. Incr. Prun. Prod. Error Red. (JRIP)					
Embedder	Ranking	Iman-Davenport		Hommel		Embedder	Ranking	Iman-Davenport		Hommel	
		Statistic	P -value	Statistic	P -value			Statistic	P -value	Statistic	P -value
Ours	1.00			—	—	Ours	1.00			—	—
TransD	2.28			3.61	$3.11E-04$	TransD	2.44v			4.07	$4.79E-05$
HolE	3.54	140.39	$1.26E-50$	7.18	$1.42E-12$	HolE	3.47	109.17	$3.56E-44$	7.00	$5.11E-12$
Node2Vec	3.54			7.18	$1.42E-12$	Node2Vec	3.47			7.00	$5.11E-12$
Complex	4.65			10.32	$2.20E-24$	Complex	4.61			10.22	$6.61E-24$
(e) Naïve Bayes (NB)						(f) PART Decision Lists (PART)					
Embedder	Ranking	Iman-Davenport		Hommel		Embedder	Ranking	Iman-Davenport		Hommel	
		Statistic	P -value	Statistic	P -value			Statistic	P -value	Statistic	P -value
Ours	1.05			—	—	Ours	1.00			—	—
TransD	2.50			4.10	$4.11E-05$	TransD	2.39			3.92	$8.69E-05$
HolE	3.79	59.95	$1.40E-30$	7.74	$1.94E-14$	HolE	3.51	112.20	$7.38E-45$	7.11	$2.38E-12$
Node2Vec	3.79			7.74	$1.94E-14$	Node2Vec	3.51			7.11	$2.38E-12$
Complex	3.88			7.99	$5.38E-15$	Complex	4.59			10.15	$1.37E-23$
(g) Random Forests (RF)						(h) Sequential Minimal Optimization (SMO)					
Embedder	Ranking	Iman-Davenport		Hommel		Embedder	Ranking	Iman-Davenport		Hommel	
		Statistic	P -value	Statistic	P -value			Statistic	P -value	Statistic	P -value
Ours	1.00			—	—	Ours	1.44			—	—
TransD	2.61			4.56	$5.09E-06$	TransD	2.18			2.09	$3.70E-02$
HolE	3.69	61.53	$4.09E-31$	7.60	$5.86E-14$	HolE	3.52	46.94	$7.47E-26$	5.90	$7.08E-09$
Node2Vec	3.69			7.60	$5.86E-14$	Node2Vec	3.52			5.90	$7.08E-09$
Complex	4.01			8.52	$6.35E-17$	Complex	4.34			8.20	$9.42E-16$

6 | CONCLUSIONS

Metadata tags are an effective means to endow the data in an HTML file that was only intended to be user-friendly with semantics that help software agents understand them. Current trends suggest that the number of web sites that provide them increases year-after-year, but there are many web sites that do not provide any such tags, yet. This motivates the research regarding systems that can synthesize such metadata tags.

In this article, we present a system that synthesizes metadata tags for HTML files. Our design revolves around three components that provide a number of services that communicate by means of message queues; this design facilitates

deploying our proposal to a distributed system. The core is a service that learns a tagging model from a small set of HTML files. It differentiates from other proposals in the literature in that it does not require the data in the input HTML files to exist in a knowledge graph and can work with arbitrary layouts, not only tables. It relies on a new embedding technique that differentiates from the others in the literature in that it does not require to preset the length of the embeddings, it works on the original attributes, and it can learn reusable tagging models. Our experimental results prove that our approach can attain an F_1 score that outperforms four state-of-the-art embedders by 10.14% in the best case, which is a difference that was confirmed to be statistically significant at the standard confidence level.

In future, we are planning on doing additional research regarding the following ideas: exploring additional links (eg, the i th child or the ancestor links), changing the paths into trees (ie, analyzing the links from every node in a path, not only the last one), and exploring different strategies to compare the scores (ie, analyzing the impact of different heuristics to guide the search process). The first two ideas are intended to expand the search space, which is expected to find better taggers; the last one is intended to cut many useless search branches so that the overall process remains efficient.


ACKNOWLEDGEMENTS

This work was supported by the Spanish and the Andalusian R&D programmes through FEDER grants TIN2013-40848-R, TIN2016-75394-R, and P18-RT-1060. The work by Juan C. Roldán was also supported by a PIF grant from the University of Seville. The authors are also thankful to Dinamic Area, S.L, which provided the industrial context to develop our research and the machinery used to run our experimentation using a real-world repository.

ORCID

Patricia Jiménez  <https://orcid.org/0000-0001-5070-5904>

Juan C. Roldán  <https://orcid.org/0000-0002-9853-7230>

Fernando O. Gallego  <https://orcid.org/0000-0002-9002-0157>

Rafael Corchuelo  <https://orcid.org/0000-0003-1563-6979>

REFERENCES

1. Bizer C, Meusel R, Primpel A. *Web Data Commons: RDFa, Microdata, Embedded JSON-LD, and Microformat Data Sets. Technical Report*. Mannheim, Germany: University of Mannheim; 2019 <http://webdatacommons.org/structureddata/2019-12/stats/stats.html>.
2. Morales A, Premtoon V, Avery C, Felshin S, Katz B. Learning to answer questions from wikipedia infoboxes. *EMNLP*. 2016;1930-1935.
3. Bizer C, Eckert K, Meusel R, Mühleisen H, Schuhmacher M, Völker J. Deployment of RDFa, microdata, and microformats on the web. *The Semantic Web -- ISWC 2013 - 12th International Semantic Web Conference*. 8219. Berlin, Heidelberg: Springer; 2013:17-32.
4. Meusel R, Petrovski P, Bizer C. The WebDataCommons microdata, RDFa, and microformat dataset series. *The Semantic Web -- ISWC 2014 - 13th International Semantic Web Conference*. Cham: Springer; 2014:277-292.
5. Doderer JM, Ruiz-Rube I, Palomo-Duarte M, Vázquez-Murga J. Open linked data model revelation and access for analytical web science. *Metadata and Semantic Research - 5th International Conference, {MTSR}*. Berlin, Heidelberg: Springer; 2011:105-116.
6. Adrian B, Hees J, Herman I, Sintek M, Dengel A. Epiphany: adaptable RDFa generation linking the web of documents to the web of data. *Knowledge Engineering and Management by the Masses - 17th International Conference, {EKAW}*. Berlin, Heidelberg: Springer; 2010:178-192.
7. Ngomo A-CN, Heino N, Lyko K, Speck R, Kaltenböck M. SCMS: semantifying content management systems. *The Semantic Web - {ISWC} 2011 - 10th International Semantic Web Conference*. Berlin, Heidelberg: Springer; 2011:189-204.
8. Eldesouky B, Bakry M, Maus H, Dengel A. Seed: an end-user text composition tool for the semantic web. *The Semantic Web - {ISWC} 2016 - 15th International Semantic Web Conference*. Cham: Springer; 2016:218-233.
9. Guerrero-Contreras G, Navarro-Galindo JL, Samos J, Garrido JL. A collaborative semantic annotation system in health. *Mob Inf Syst*. 2017;2017:1-10.
10. Burget R. Information extraction from the Web by matching visual presentation patterns. *Knowledge Graphs and Language Technology - {ISWC} 2016 International Workshops: {KEKI} and NLP{&}DBPedia*. Cham: Springer; 2016:10-26.
11. Efthymiou V, Hassanzadeh O, Rodríguez-Muro M, Christophides V. Matching web tables with knowledge base entities: from entity lookups to entity embeddings. *The Semantic Web - {ISWC} 2017 - 16th International Semantic Web Conference*. Cham: Springer; 2017:260-277.
12. Oulabi Y, Bizer C. Extending cross-domain knowledge bases with long tail entities using web table data. *Advances in Database Technology - 22nd International Conference on Extending Database Technology, {EDBT} 2019*. Konstanz: Herschel, Melanie; 2019:385-396.
13. Wang Q, Mao Z, Wang B, Guo L. Knowledge graph embedding: a survey of approaches and applications. *IEEE Trans Knowl Data Eng*. 2017;29(12):2724-2743.
14. Goyal P, Ferrara E. Graph embedding techniques, applications, and performance: a survey. *Knowl-Based Syst*. 2018;151:78-94.
15. Cai H, Zheng VW, Chang KC-C. A comprehensive survey of graph embedding: problems, techniques, and applications. *IEEE Trans Knowl Data Eng*. 2018;30(9):1616-1637.

16. Frank E, Hall MA, Witten IH. *The Weka Workbench*. 4th ed. Burlington, Massachusetts: Morgan Kaufmann; 2016.
17. Ferri C, Hernández-Orallo J, Modroiu R. An experimental comparison of performance measures for classification. *Pattern Recognit Lett*. 2009;30(1):27-38.
18. Trouillon T, Welbl J, Riedel S, Gaussier É, Bouchard G. Complex embeddings for simple link prediction. *Proceedings of the 33rd International Conference on Machine Learning, {ICML} 2016*. United States: JMLR.org; 2016:2071-2080.
19. Nickel M, Rosasco L, Poggio TA. Holographic embeddings of knowledge graphs. *Proceedings of the Thirtieth {AAAI} Conference on Artificial Intelligence*. Palo Alto, CA: AAAI Press; 2016:1955-1961.
20. Grover A, Leskovec J. Node2Vec: scalable feature learning for networks. *SIGKDD*. New York, NY: Association for Computing Machinery; 2016:855-864.
21. Ji G, He S, Xu L, Liu K, Zhao J. Knowledge graph embedding via dynamic mapping matrix. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, {ACL} 2015*. Stroudsburg, PA: The Association for Computer Linguistics; 2015:687-696.
22. Foss AH, Markatoul M, Ray B. Distance metrics and clustering methods for mixed-type data. *Int Stat Rev*. 2018;87(1):80-109.
23. García S, Herrera F. An extension on eStatistical comparisons of classifiers over multiple data setse for all pair-wise comparisons. *J Mach Learn Res*. 2008;9:2677-2694.

APPENDIX A. CATALOG OF ATTRIBUTES

Our proposal relies on a number of predefined and user-defined attributes that are used to learn the tagging models. The former are computed automatically by Headless Chrome from the DOM tree, the HTML source, or the rendering; the latter were designed building on our intuition and our preliminary experimental results.

Tables A1 and A2 show the listings of predefined and user-defined attributes, respectively. There is a row per attribute and the columns report, respectively, on their category (in the case of predefined attributes), their names, how frequently they were used in our experimentation, their description, and some sample values. The catalog is larger; we have intentionally omitted the attributes that did not prove to be useful in our experimentation. Note that none of the attributes has a very high frequency, which means that none of them can individually help learn a good tagging model; it is typically the combination of many such attributes that helps learn them.

TABLE A1 Predefined attributes

Category	Name	Frequency (%)	Description	Examples
DOM	index	3.02	Index of node among its siblings	0, 1, 2
	count-children	2.64	Count of DOM children nodes	0, 1, 2
	depth	2.48	Depth in the DOM tree	0, 1, 2
	count-siblings	2.44	Count of sibling DOM nodes	0, 1, 2
	has-siblings	0.01	Has any siblings?	true, false
	is-text-node	0.01	Contains only text?	true, false
HTML	width	2.46	Width of the bounding box	50%, 25px, 4.5in
	height	2.20	Height of the bounding box	50%, 25px, 4.5in
	tag	2.20	HTML tag	p, em, table
Rendering	display	2.42	Kind of display	inline, block, flex
	border-bottom-color	2.22	Color of the bottom border	red, rgb(1, 2, 3), # 1234
	border-top-color	2.20	Color of the top border	red, rgb(1, 2, 3), # 1235
	border-left-color	2.19	Color of the left border	red, rgb(1, 2, 3), # 1236
	border-right-color	2.18	Color of the right border	red, rgb(1, 2, 3), # 1237

(Continues)

TABLE A1 (Continued)

Category	Name	Frequency (%)	Description	Examples
	vertical-align	1.94	Kind of vertical alignment	baseline, top, middle
	font-weight	1.73	Weight of the font	100, bold, normal
	font-size	1.69	Size of the font	10px, small, 50%
	line-height	1.52	Line height.	50%, 25px, 4.5in
	x-pos	1.41	X co-ordinate of the upper left bounding box corner	10px, 20px, 30px
	background-color	1.40	Color of the background	red, rgb(1, 2, 3), # 1237
	text-decoration	1.36	Kind of text decoration	overline, line-through, wavy
	y-pos	1.04	Y co-ordinate of the upper left bounding box corner	10px, 20px, 30px
	float	1.01	Kind of floating positioning	none, left, right
	white-space	0.80	Kind of white-space handling	normal, no-wrap, wrap
	background-image	0.77	Image to show in the background	none, url(\$paper.gif), url(up.png)
	margin-bottom	0.67	Margin at the bottom.	10px, 20px, 30px
	background-repeat	0.66	Kind of background image repetition	repeat, no-repeat
	clear	0.65	How the node besides floats	left, middle, none
	background-position	0.63	Position of the background image	top, 0px 0px, right 35%
	padding-left	0.58	Left padding	50%, 25px, 4.5in
	padding-top	0.56	Top padding	50%, 25px, 4.5in
	padding-right	0.54	Right padding	50%, 25px, 4.5in
	font-style	0.53	Style of the font	normal, italic, oblique
	border-left-style	0.50	Style of the left border	none, solid, dotted
	padding-bottom	0.49	Bottom padding	50%, 25px, 4.5in
	border-top-style	0.48	Style of the top border	none, solid, outset
	margin-left	0.44	Margin on the left	10px, 20px, 30px
	border-bottom-style	0.44	Style of the bottom border	none, solid, outset
	border-right-style	0.43	Style of the right border	none, solid, outset
	border-top-width	0.40	Width of the top border	50%, 25px, 4.5in
	list-style-type	0.39	Kind of list item marker	none, square, disc
	margin-right	0.36	Margin on the right	50%, 25px, 4.5in
	border-right-width	0.32	Width of the right border	50%, 25px, 4.5in
	border-left-width	0.30	Width of the left border	50%, 25px, 4.5in
	border-bottom-width	0.27	Width of the bottom border	50%, 25px, 4.5in
	text-transform	0.27	Kind of text transformation	uppercase, lowercase, capitalize
	letter-spacing	0.16	Spacing in between letters	1px, 0.01in, normal
	font-family	0.15	The family of the font	Arial, Verdana, Calibri
	text-indent	0.13	Indentation of the first line	50%, 25px, 4.5in
	text-align	0.09	Kind of horizontal alignment	left, center, right
	font-variant	0.04	Variant of the font	normal, small-caps, initial
	list-style-position	0.01	Position of the list item markers	inside, outside, initial

TABLE A2 User-defined attributes

Name	Frequency (%)	Description	Examples
count-tokens	2.95	Count of tokens	10, 20, 30
count-letters	2.84	Count of letters	10, 20, 30
count-trigrams	2.82	Count of 3-grams	10, 20, 30
count-uppercase-bigrams	2.79	Count of 2-grams whose tokens are uppercased	10, 20, 30
count-blanks	2.71	Count of blanks (whitespace, tabulators, carriage returns)	10, 20, 30
count-alphanum	2.71	Count of alpha-numeric tokens	10, 20, 30
count-bigrams	2.69	Count of 2-grams	10, 20, 30
count-uppercase-trigrams	2.61	Count of 3-grams whose tokens are uppercased	10, 20, 30
count-lowercase-trigrams	2.61	Count of 3-grams whose tokens are lowercased	10, 20, 30
count-capitals	2.61	Count of tokens that start with a capital letter	10, 20, 30
count-digits	2.25	Count of digits	10, 20, 30
count-integers	2.23	Count of tokens that represent an integer	10, 20, 30
last-token	2.20	Last token	web, \$10.95, !
count-uppercase-tokens	2.16	Count of tokens that are uppercased	10, 20, 30
first-token	2.15	First token	free, quantity, date
count-lowercase-tokens	2.10	Count of tokens that are lowercased	10, 20, 30
second-token-last-bigram	1.28	Second token in the last 2-gram	free, quantity, date
second-token-first-bigram	1.25	Second token in the first 2-gram	free, quantity, date
count-floats	1.01	Count of tokens that are float numbers	10, 20, 30
count-capitalised-bigrams	0.53	Count of capitalized 2-grams	10, 20, 30
count-capitalised-trigrams	0.50	Count of capitalized 3-grams	10, 20, 30
first-token-last-bigram	0.44	First token in the last 2-gram	free, quantity, date
first-token-first-bigram	0.41	First token in the first 2-gram	free, quantity, date
is-digit	0.03	Are all characters digits?	true, false
is-uppercase	0.03	Are all characters uppercased?	true, false
starts-with-parenthesis	0.02	Does it start with a left parenthesis?	true, false
is-url	0.02	Is it an URL?	true, false
is-money	0.02	Is it an amount of money?	true, false
count-currencies	0.02	Count of currency symbols	10, 20, 30
starts-with-capitalised-token	0.01	Is the first token capitalized?	true, false
has-bracketed-number	0.01	Is there a number in brackets?	true, false
is-number	0.01	Is it a number? (including natural, integer, and real numbers)	true, false
is-lowercase	0.01	Is it lowercased?	true, false
is-notblank	0.01	Are all character non-blank?	true, false
is-isbn	0.01	Is it an ISBN?	true, false
starts-with-punctuation	0.01	Does it start with a punctuation symbol?	true, false
starts-with-number	0.01	Does it start with a number?	true, false
starts-with-currency	0.01	Does it start with a currency symbol?	true, false
ends-with-currency	0.01	Does it end with a currency symbol?	true, false
ends-with-parenthesis	0.01	Does it end with a right parenthesis?	true, false
ends-with-punctuation	0.01	Does it end with a punctuation symbol?	true, false
is-date	0.01	Is it a date?	true, false
is-email	0.01	Is it an e-mail address?	true, false
is-blank	0.01	Does it consist solely of blanks? (Whitespace, tabulators, carriage returns)	true, false
is-alphanum	0.01	Are all tokens alphanumeric?	true, false
is-year	0.01	Is it a year?	true, false