Learning URI Selection Criteria to Improve the Crawling of Linked Open Data

Hai Huang(✉) and Fabien Gandon

Inria, Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
{Hai.Huang,Fabien.Gandon}@inria.fr

Abstract. As the Web of Linked Open Data is growing the problem of crawling that cloud becomes increasingly important. Unlike normal Web crawlers, a Linked Data crawler performs a selection to focus on collecting linked RDF (including RDFa) data on the Web. From the perspectives of throughput and coverage, given a newly discovered and targeted URI, the key issue of Linked Data crawlers is to decide whether this URI is likely to dereference into an RDF data source and therefore it is worth downloading the representation it points to. Current solutions adopt heuristic rules to filter irrelevant URIs. Unfortunately, when the heuristics are too restrictive this hampers the coverage of crawling. In this paper, we propose and compare approaches to learn strategies for crawling Linked Data on the Web by predicting whether a newly discovered URI will lead to an RDF data source or not. We detail the features used in predicting the relevance and the methods we evaluated including a promising adaptation of FTRL-proximal online learning algorithm. We compare several options through extensive experiments including existing crawlers as baseline methods to evaluate their efficacy.

Keywords: Linked Data · Crawling strategy · Machine learning · Online prediction

1 Introduction

Linked Data extends the principles of the World Wide Web from linking documents to that of linking pieces of data to weave a Web of Data. This relies on the well-known linked data principles [1,10] including the use of HTTP URIs that can be dereferenced and the provision of useful and linked description upon access so that we can discover more things.

Recently, a large amount of data are being made available as linked data in various domains such as health, publication, agriculture, music, etc., and the Web of Linked Data is growing exponentially [8]. In order to harvest this enormous data repository, crawling techniques for Linked Data are becoming increasingly important. Different from conventional crawlers, crawling for Linked Data is performed selectively to collect structured data connected by RDF links.

P. Hitzler et al. (Eds.): ESWC 2019, LNCS 11503, pp. 194–208, 2019.
https://doi.org/10.1007/978-3-030-21348-0_13

The target of interest makes it distinct from focused crawlers which select the collection of Web pages to crawl based on their relevance to a specific topic.

The design objective of Linked Data crawlers is to fetch linked RDF data in different formats – RDF/XML, N3, RDFa, JSON-LD, etc. – as much as possible within a reasonable time while minimizing the download of irrelevant URIs – i.e. leading to resources without RDF content. Therefore our challenge is to identify as soon as possible the URIs referencing RDF sources without downloading them. The research question here is: *can we learn efficient URI selection criteria to identify sources of Linked Open Data?*

To solve this problem, in this paper we propose and compare methods on real data to predict whether a newly discovered URI will lead to RDF data source or not. We extract information from the targeting and referring URI and from the context (RDF data graph) where URIs and their links were discovered in order to produce features fed to learning algorithms and in particular to an FTRL-proximal online method employed to build the prediction model. FTRL-proximal is a time efficient and space efficient online learning algorithm, which can handle significantly larger data sets. It is also effective at producing sparse models which is important when the feature space is huge. FTRL-proximal outperforms the other online learning algorithms in terms of accuracy and sparsity.

The contributions of this work include: (1) we identify the features to predict whether a target URI will lead to some RDF data or not; (2) we adapt the FTRL-proximal algorithm to our task and build an online prediction model; and (3) we implement a Linked Data crawler with the online prediction model and evaluate its performance.

The paper is organized as follows. Section 2 introduces related work, followed by preliminary knowledge in Sect. 3. Section 4 describes feature extraction and online prediction model. In Sect. 5 we present the implementation of the proposed crawler. The experimental setup and results are described in Sect. 6. We conclude our work in Sect. 7.

2 Related Work

Semantic Web/Linked Data Crawlers. Semantic web crawlers differ from traditional web crawlers in only two aspects: the format of the source (RDF format) it is traversing, and the means to link RDF across data sources. There exist some work [6,12,13] in the field of Semantic Web/Linked Data crawling. The two main representative crawlers for Linked Data are LDSpider [13] and SWSE crawler [12]. They crawl the Web of Linked Data by traversing RDF links between data sources and follow the Linked Data principles [1,10]. They offer different crawling strategies such as breadth-first and load-balancing traversal.

In order to reduce the amount of HTTP lookups and downloading wasted on URIs referencing non-RDF resources, these previous works apply heuristic rules to identify relevant URIs [12,13]. The URIs with common file extensions (e.g., html/htm, jpg, pdf, etc.) and those without appropriate HTTP Content-Type Header (such as application/rdf+xml) are classified as non-RDF content URIs.

The content of these URIs would not be retrieved by these crawlers. Although this heuristic-based method is efficient, it impairs the recall of the crawling. This method makes the assumption that data publishers have provided correct HTTP Content-Type Header but this is not always the case. It can happen that the server does not provide the desired content type. Moreover, it may happen that the server returns an incorrect content type. For example, Freebase does not provide any content negotiation or HTML resource representation [9], and only text/plain is returned as content type. In [11], it is reported that 17% of RDF/XML documents are returned with a content-type other than application/rdf+xml. As a result, a huge volume of RDF data is missed by these methods.

Focused Crawlers on the Web. Traditional focused crawlers on the Web aim to crawl a subset of Web pages based on their relevance to a specific topic [4,5]. These methods build a relevancy model typically encoded as a classifier to evaluate the web documents for topical relevancy. Our work here is different in the sense that we do not filter on the topics but on the type of content. Meusel et al. [17] proposed a focused crawling approach to fetch microdata embedded in HTML pages. Umbrich et al. [18] built a crawler focused on gathering files of a particular media type and based on heuristics.

Online Prediction. Our problem is also related to the classic online binary prediction in which data becomes available in a sequential order and the learner must make real-time decisions and continuously improve performance with the sequential arrival of data. Some methods have been developed such as Stochastic Gradient Descent (SGD) [3], RDA [20], FOBOS [7] and FTRL-Proximal [15,16]. Among them, FTRL-Proximal which is developed by Google has been proven to work well on the massive online learning problem of predicting ad click-through rates (CTR). We adapt the FTRL-Proximal to our prediction task in this work.

3 Preliminary Knowledge

In this section, we introduce some basic definitions and notations used throughout the paper. The Linked Data crawler targets a kind of structured data represented in RDF format on the Web. The Resource Description Framework (RDF) provides a structured means of publishing information describing entities and their relationships in RDF triples. The main concepts of RDF and Linked Data we need here are:

Definition 1 ***(RDF Triple).*** *A triple $t = (s, p, o) \in (U \cup B) \times U \times (U \cup B) \times (U \times B \times L)$ is called an RDF triple where U denotes the set of URI, B the set of blank nodes and L the set of literals. In such a triple, s is called subject, p predicate, and o object.*

Definition 2 ***(RDF Graph).*** *An RDF graph $G = (V, E)$ is a set of RDF triples such that V is the node set and E is the edge set of G.*

Definition 3 ***(HTTP Dereferencing).*** *The act of retrieving a representation of a resource identified by a URI is known as dereferencing that URI. We define HTTP dereferencing as the function* $deref : U \rightarrow R$ *which maps a given URI to the representation of a resource returned by performing the HTTP lookup operations upon that URI and following redirections when needed.*

4 Prediction Model for Crawling Criteria

In this section, we present the prediction task, the feature sets extracted for the task of prediction and then describe the prediction model based on FTRL-proximal online learning algorithm.

4.1 Task Description

Since the task of Linked Data crawler is to fetch RDF data on the Web we are interested in a kind of URIs that we call RDF-relevant URIs:

Definition 4 ***(RDF-Relevant).*** *Given a URI* u*, we consider that* u *is RDF-relevant if the representation obtained by dereferencing* u *contains RDF data. Otherwise,* u *is called non RDF-relevant. We note* U^R *the set of RDF relevant URIs and* U^I *the set of non RDF-relevant URIs with* $U = U^R \dot\cup U^I$*.*

For the URIs that have certain file extensions such as *.rdf/owl or HTTP Content Types Headers such as application/rdf+xml, text/turtle, etc., it is trivial to know the RDF-relevance of them. In this work, we focus on a knid of URIs called hard URIs whose RDF-relevance *cannot* be known by these heuristics.

Definition 5 ***(Hard URI).*** *We call* u *a hard URI if the RDF relevance of* u *cannot be known straightforwardly by its file extension or HTTP Content-Type Header.*

For example, URI u with HTTP Content-Type header text/html is a hard URI since RDFa data could be embedded in u. As reported in [9], the URIs with HTTP Content-Type Header text/plain may contain RDF data so they are hard URIs too.

Then, for our prediction task, we consider four types of URIs involved in the prediction we want to make: the target URI, the referring URI, direct property URIs and sibling URIs.

Definition 6 ***(Target URI).*** *We call target URI and note* u_t *the URI for which we want to predict if the* $deref(u_t)$ *will lead to a representation that contains RDF data i.e. if* u_t *is RDF-Relevant.*

Definition 7 ***(Context RDF Graph).*** *Given an RDF relevant URI* u_r *containing the RDF graph* G_t^r*, we define* $G_t^r = (V, E)$ *as the context RDF graph of URI* u_t *if* u_t *appears in* G_t^r *as a node, i.e.,* $u_t \in V$*.*

Definition 8 ***(Referring URI).*** *Given a target URI u_t, we call "referring URI" and note u_r the RDF-relevant URI that was dereferenced into a representation $deref(u_r)$ containing the context RDF graph G_t^r in which we discovered the target URI u_t.*

Definition 9 ***(Sibling Set).*** *The sibling set of u_t in the context RDF Graph G_t^r denoted by Sib_{u_t} is the set of all other URIs that have common subject-property or property-object pair with u_t in G_t^r. Sib_{u_t} is formally defined as:*

$$\begin{aligned} Sib_{u_t} = \{s|\exists p, o \in G_t^r, s.t.(u_t, p, o) \in G_t^r \wedge (s, p, o) \in G_t^r\} \\ \vee \{o|\exists s, p \in G_t^r, s.t.(s, p, u_t) \in G_t^r \wedge (s, p, o) \in G_t^r\} \end{aligned} \tag{1}$$

Definition 10 ***(Direct property Set).*** *The direct property set PS_t is the set of properties that connect u_t in the context graph G_t^r:*

$$PS_t = \{p|\exists s, o \in G_t^c, s.t.(u_t, p, o) \in G_t^r \vee (s, p, u_t) \in G_t^r\} \tag{2}$$

Prediction Task. Using these definitions we can now define our prediction task. Suppose that a hard URI u_t is discovered from the context graph G_t^r obtained by dereferencing a referring URI u_r. We want to predict if u_t is RDF-relevant based on some features extracted from u_t, u_r, PS_t and Sib_{u_t}. Our task is to learn the mapping $Relevant : U \rightarrow \{0, 1\}$ with $Relevant(u)$ equals 1 if u is RDF-relevant ($Relevant|_{U^R} \mapsto 1$) and equals 0 if u is not RDF-relevant ($Relevant|_{U^I} \mapsto 0$)

4.2 Feature Extraction

We distinguish between two kinds of features that can be exploited for the prediction intrinsic and extrinsic.

Definition 11 ***(intrinsic URI features).*** *The intrinsic features of a URI u are features obtained by an extraction $F_{int} : U \rightarrow F$ that relies exclusively on the URI identifier itself.*

An example of an intrinsic feature is the protocol used e.g. `http`.

Definition 12 ***(extrinsic URI features).*** *The extrinsic features of a URI u are features obtained by performing some network call on the URI $F_{ext} : U \rightarrow F$.*

The URI generic syntax consists of a hierarchical sequence of several components[1]. An example of URI is shown in Fig. 1.

The intrinsic features $F_{int}(u)$ consider that the different components of a URI u are informative and contain helpful information for prediction. The components include: scheme, authority, host, path, query, fragment information. We generate the intrinsic features of u based on these components.

For example, the features $F_{int}(u)$ when u equals the URI shown in Fig. 1 are described as follows:

[1] RFC 3986, section 3(2005).

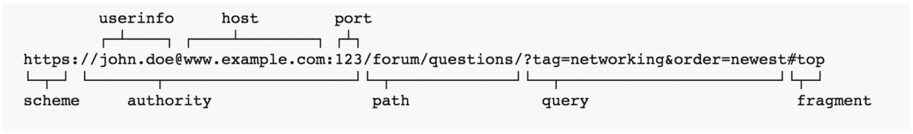
userinfo
host
port
https://john.doe@www.example.com:123/forum/questions/?tag=networking&order=newest#top
scheme
authority
path
query
fragment

Fig. 1. Example of a URI with component parts.

– feature_u_scheme=‘https’
– feature_u_authority=‘john.doe@www.example.com:123’
– feature_u_host=‘www.example.com’
– feature_u_userinfo=‘john.doe’
– feature_u_path=‘forum/questions’
– feature_u_query=‘?tag=networking&order=newest’
– feature_u_fragment=‘top’

We further distinguish two kinds of extrinsic features:

Definition 13 ***(URI header features).*** *These extrinsic features of a URI u are obtained by an* `HTTP HEAD` *call $F_{head} : U \rightarrow F$ and do not require to download the representation of the resource identified by u.*

Definition 14 ***(URI representation features).*** *These extrinsic features of a URI u are obtained by an* `HTTP GET` *call $F_{get} : U \rightarrow F$ and characterize the content obtained after a complete download of the representation of the resource identified by u.*

We distinguish F_{head} and F_{get} because they have different costs in terms of network access time. The URI header feature we will consider in this paper is the content type e.g. feature_u_t_contentType=‘text/html’.

URI representation features come from the content of the referring URI u_r, namely the context graph G_t^r of u_t which include the direct properties and siblings information of u_t.

Definition 15 ***(URI similarity features).*** *The similarity between two URIs u_s and u_t denoted by $simValue(u_s, u_t)$ is defined using the Levenshtein distance [14] between the two strings representing these URIs. In order to reduce the feature space, a threshold τ is set for the similarity value and if the similarity value is larger than τ, we discretize the similarity as $simValue(u_t, u_s) = high$ and otherwise $simValue(u_t, u_s) = low$.*

Definition 16 ***(RDF relevance features).*** *The boolean characteristic of being RDF-relevant for a URI u is noted $F_{RDFrel}(u)$ and returns true if u is RDF-revelant and false otherwise.*

With the types of features explained above we can now define four atomic feature sets based on the sources the features come from to explore and evaluate when training and predicting the RDF-relevance of a target URI u_t:

- $F_{+t} = F_{int}(u_t) + F_{head}(u_t)$ is a feature set considering the intrinsic and header features of the target URI u_t.
- $F_{+r} = F_{int}(u_r)$ is a feature set considering the intrinsic features of the referring URI u_r.
- $F_{+p} = \bigcup_{p \in PS_t} F_{int}(p)$ is a feature set including the intrinsic features of each direct property of the target URI u_t.
- $F_{+x} = \bigcup\limits_{u_s \in Sib_{u_t}} F_x(u_s)$ is a feature set including feature crosses that combine the intrinsic, header, similarity and relevance features of the sibling URIs of u_t. This supports predictive abilities beyond what those features can provide individually and can be interpreted as using a logical conjunction 'AND' to combine these features $F_x(u_s) = F_{int}(u_s) \times F_{head}(u_s) \times F_{sim}(u_s, u_t) \times F_{RDFrel}(u_s)$

With these definitions we can now consider and evaluate any combination of feature sets. We note F_{+a} the feature set with a being a combination of the atomic feature sets as defined above. For instance an experiment using the feature set F_{+t+r} will only consider as inputs the intrinsic and header features of the target URI u_t and the intrinsic feature of referring URI u_r. In Sect. 6.1, the predictive abilities of different combinations of the feature sets are examined.

4.3 Feature Hashing

During the process of crawling, the crawler will encounter URIs belonging to millions of domains, and the number of properties could be tens of thousands. Obviously, the potential feature space will be huge. Thus, we use feature hashing [19] technique to map high-dimensional features into binary vectors. Feature hashing uses a random sparse projection matrix $A : \mathbb{R}^n \rightarrow \mathbb{R}^m$(where $n \gg m$) in order to reduce the dimension of the data from n to m while approximately preserving the Euclidean norm. In this work, hash function MurmurHash3[2] is adopted to map the feature vectors into binary vectors. By hashing the features we can gain significant advantages in memory usage since we can bound the size of generated binary vectors and we do not need a pre-built dictionary.

4.4 Online Prediction

We now present our prediction method. In this paper we assume that data becomes available in a sequential order during the crawling process. The predictor makes a prediction at each step and can also update itself: it operates in an online style. Compared to batch methods, online methods are more appropriate for the task of crawling since the predictor has to work in a dynamic learning environment. FTRL-Proximal [15,16] developed by Google has been proven to work well on the massive online learning problems. FTRL-proximal outperforms the other online learning algorithms in terms of accuracy and sparsity, and has

[2] https://github.com/aappleby/smhasher/wiki/MurmurHash3.

been widely used in industry, e.g., recommender systems and advertisement systems. We adopt this learning algorithm in our work.

We use $\mathbf{x}_t$ to denote the feature vector of URI u_t and $y_t \in \{0, 1\}$ the true class label of u_t. Given a sequence of URIs $u_1, u_2, \cdots, u_r, \cdots, u_t$, the process of online prediction based on FTRL-Proximal algorithm is shown in Algorithm 1. We adapted the original algorithm to make it output binary values by setting a decision threshold τ. If the predicted probability p_t is greater than the decision threshold $\tau \in [0, 1]$, it outputs prediction $\hat{y}_t = 1$; otherwise $\hat{y}_t = 0$.

Algorithm 1. Online prediction with the FTRL-Proximal algorithm

```
   Input: URIs $u_1, u_2, \cdots, u_T$
   Result: $\hat{y}_1, \cdots, \hat{y}_T$
1  for $t = 1$ to $T$ do do
2      get feature vector $\mathbf{x}_t$ of $u_t$;
3      probability $p_t$= $sigmoid(\mathbf{x}_t \cdot \mathbf{w}_t)$;
4      if $p_t > \tau$ then
5          output $\hat{y}_t$=1;
6      else
7          output $\hat{y}_t$=0;
8      end
9      observe real label $y_t \in \{0, 1\}$;
10     update $\mathbf{w}_{t+1}$ by equation (3);
11 end
```

At round $t + 1$, the FTRL-Proximal algorithm uses the update formula (3):

$$\mathbf{w}_{t+1} = argmin_{\mathbf{w}}(\sum_{s=1}^{t} \mathbf{w} \cdot \mathbf{g}_s + \frac{1}{2}\sum_{s=1}^{t} \sigma_s \parallel \mathbf{w} - \mathbf{w}_s \parallel_2^2 + \lambda_1 \parallel \mathbf{w} \parallel_1). \tag{3}$$

In Eq. (3), $\mathbf{w}_{t+1}$ is the target model parameters to be updated in each round. In the first item of equation (3), $\mathbf{g}_s$ is the gradient of loss function for training instance s. The second item of equation (3) is a smoothing term which aims to speed up convergence and improve accuracy, and σ_s is a non-increasing learning rate defined as $\sum_{s=1}^{t} \sigma_s = \sqrt{t}$. The third item of equation (3) is a convex regularization term, namely L1-norm which is used to prevent over-fitting and induce sparsity.

Subsampling. Not all URIs are considered as training instances since we are interested in hard URIs. We exclude from training URIs with the extensions such as *.rdf/owl. Inversely, URIs with the file extension *.html/htm are included in the training set since they may contain RDFa data.

The true class label y_t is required to update the predictor online. In our scenario, to observe the true class label of a URI we have to download it and check whether it contains RDF data or not. We cannot afford to download all URIs because of the network overhead, and our target is to build a prediction model that avoids downloading unnecessary URIs. There, an appropriate subsampling strategy is needed. We found that the positive URIs are rare (much less than 50%) and relatively more valuable. For each round, if URI u_t is predicted positive namely $\hat{y}_t = 1$, we retrieve the content of u_t and observe the real class label y_t. Then the predictor can be updated by new training instance $(y_t, \mathbf{x_t})$. For those URIs predicted negative, we only select a fraction $\epsilon \in]0, 1]$ of them to download and observe their true class label. Here ϵ is a balance between online prediction precision (which requires as many URIs as possible for online training) and downloading overhead (which requires as few non-RDF relevant URIs downloaded as possible). To deal with the bias of this subsampled data, we assign an importance weight $\frac{1}{\epsilon}$ to these examples. In our experiment (Sect. 6.2), we set ϵ as the ratio of the number of positive URIs to the number of negative URIs in the training set.

5 Implementation of Crawler

We now detail the prototype we tested for the Linked Data crawler and explain the Algorithm 2 it implements.

Initialization. As shown in Algorithm 2, the proposed crawler starts from a list of seed URIs and operates an breadth-first crawling since this often leads to a more diverse dataset instead of traversing deep paths within some given sites. The maximum crawl depth d_max is set for crawling. The $Frontier$ data structure is initialized by the seed URI list S.

Politeness. At the beginning of each round, a naive crawler obtains a URI from $Frontier$ to retrieve. However, it would lead to the problem that the crawler issues too many consecutive HTTP requests to a server and is considered “impolite” by the server. Thus, we group the URIs in $Frontier$ into different sets $pld_{0..n}$ based on their Pay Level Domains(PLDs). URIs are polled from PLD sets in a round-robin fashion, which means in a round each set pld_i has one chance to select a URI to retrieve (Line 6). We also set a minimum time delay min_delay for each round. If the minimum crawl time of a round is less than min_delay, the crawler will sleep until the condition of minimum time delay is satisfied (Line 36–Line 39).

Prediction. This is the core of crawling (Line 8–Line 29). Once a URI u_t with feature vector $\mathbf{x_t}$ is polled, the predictor predicts the class label $\hat{y}_t$ of u_t, $\hat{y}_t \in \{0, 1\}$. If $\hat{y}_t = 1$, we retrieve the content of u_t and get the real class label y_t, and the predictor can be updated by the new training example $(y_t, \mathbf{x_t})$. If the prediction is correct (u_t is RDF relevant), the RDF graph G_t of u_t is written to local storage and the child URIs in G_t with their feature vector are added to $Frontier$ for future rounds (Line 9– Line 17). However, as discussed in Sect. 4.4, naively training on this subsampled data would lead to significantly biased predictions. To deal with this bias (Line 19–Line 28), the crawler downloads a fraction ϵ of URIs that are predicted negative (Line 19–Line 21). For the case of false negative (Line 22–Line 26), the RDF graph is written to local storage and the child URIs with their feature vector are added in $Frontier$. The predictor is updated by the example $(y_t, \mathbf{x_t})$ with importance weight $\frac{1}{\epsilon}$ (Line 27).

Algorithm 2. Crawling on Linked Data

```
Data: A seed list of URIs $S$, maximum crawl depth $d\_max$, minimum time
      delay $min\_delay$
Result: A collection of RDF triples
1  initialize $Frontier{=}S$, $pld_{0..n} = \emptyset$;
2  while $depth < d\_max$ do
3      add URIs in $Frontier$ to $pld_{0..n}$;
4      $startTime$=current_time();
5      foreach $pld_i$ do
6          get uri $u_t$ from $pld_i$;
7          if $u_t = dref(u_t)$ then
8              $\hat{y_t} \in \{0,1\} = predict(\mathbf{x_t}, \mathbf{w})$;
9              if $\hat{y_t}$=1 then
10                 download the content of $u_t$;
11                 observe class label $y_t \in \{0,1\}$;
12                 if $y_t = 1$ then
13                     write RDF graph $G_t$ contained in $u_t$ to the local storage;
14                     generate feature vectors for URIs in $G_t$;
15                     add URIs with their feature vectors in Frontier;
16                 end
17                 update the predictor by the new example ($y_t$,$\mathbf{x_t}$) ;
18             else
19                 if random[0,1] $< \epsilon$ then
20                     download the content of $u_t$;
21                     observe class label $y_t \in \{0,1\}$;
22                     if $y_t = 1$ then
23                         write RDF graph $G_t$ contained in $u_t$ to the local storage;
24                         generate feature vectors for URIs in $G_t$;
25                         add URIs with their feature vectors in Frontier;
26                     end
27                     update the predictor by the new example ($y_t$,$\mathbf{x_t}$) with
                       important weight $\frac{1}{\epsilon}$;
28                 end
29             end
30         else
31             if $dref(u_t)$ is unseen then
32                 add uri $dref(u_t)$ in $Frontier$ ;
33             end
34         end
35     end
36     $timeSpan$=current_time()- $startTime$;
37     if $timeSpan < min\_delay$ then
38         wait($min\_delay - timeSpan$);
39     end
40 end
```

Reducing HTTP Lookup. To generate the feature vector of a URI u_t, the crawler has to send HTTP header requests to get the content type of u_t. There exists a large number of redundant HTTP lookups during crawl. To overcome this issue, we build a bloom filter [2] for each kind of MIME types. Bloom filter is a space-efficient probabilistic data structure which is able to fit a billion of URIs in main memory. Once the content type of a URI is known by HTTP lookup, the URI is added to the corresponding bloom filter. For a newly discovered URI u, we submit u to each bloom filter. If a bloom filter reports positive, it indicates that u has the corresponding content type[3]. If no bloom filter reports positive, we have to get the content type of u by sending HTTP request and then store u in the corresponding bloom filter based on its content type.

6 Evaluation

In this section, we firstly evaluate the predictive ability of different combinations of atomic feature sets introduced in Sect. 4.2 and then compare the performance of the proposed crawler with some baseline methods including offline methods. Lastly, we report on experiments on the processing time to evaluate the efficiency of the proposed method.

6.1 Feature Set Evaluation

In this experiment, we evaluate the predictive ability of different combinations of feature sets introduced in Sect. 4.2 by several offline/batch classifiers. We firstly introduce the dataset, metrics and offline classifiers used in the experiment and then discuss the results of the experiment.

Dataset. The dataset used in the experiment is generated by operating a Breadth-First Search (BFS) crawl. The crawl starts a set of 50 seed URIs which are RDF relevant and randomly selected from 26 hosts. During the crawl, we only keep the hard URIs whose RDF relevance cannot be known straightforwardly. Finally we generate a dataset with 103K URIs. The dataset includes 9,825 different hosts. For each URI, we generate features and the class label. To check RDF relevance for each URI, we use the library *Any23*[4].

Static Classifiers and Metrics. The static classifiers including SVM, KNN and Naive Bayesian are used to examine the performance of different combinations of feature sets. We use accuracy and F-measure as metrics to measure the performance, which are defined as:

$$accuracy = \frac{\#correct\ \ predictions}{\#predictions}$$

[3] Bloom filter may report false positive results (but not false negatives) with a low chance. Thus it is possible that a URI has a wrong content type feature.

[4] https://any23.apache.org/.

$$F - measure = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Evaluation of Combinations of Feature Sets. The aim of the experiment is to explore the predictive performance of the combinations of feature sets. As described in Sect. 4.2, the feature sets include F_{+t} derived from the target URI u_t, F_{+r} derived from the referring URI u_r, F_{+p} derived from the direct properties of u_t and F_{+x} derived from the siblings of u_t.

In Table 1 we report the results for the 4-element combination of feature sets ($F_{+t+r+p+x}$), all 3-element combinations and the 2-element combinations (F_{+t+x}, F_{r+p}, F_{+p+x}) which have the best performance in their class.

Table 1. Performance of the combinations of feature sets

Combination of feature sets	KNN		Naive Bayes		SVM	
	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy
$F_{+t+r+p+x}$	0.6951	0.7407	0.7154	0.7462	0.7944	0.7722
F_{+t+p+x}	0.6261	0.6832	0.7094	0.7413	0.7801	0.7643
F_{+t+r+x}	0.6773	0.7121	0.7111	0.7448	0.7829	0.7650
F_{+t+r+p}	0.7592	0.7731	0.7660	0.7701	**0.8216**	**0.7902**
F_{+r+p+x}	0.6015	0.7010	0.6328	0.7075	0.6839	0.7074
F_{+t+x}	0.5582	0.6912	0.6012	0.6172	0.6828	0.6277
F_{r+p}	0.3953	0.5810	0.4874	0.6097	0.6790	0.6424
F_{+p+x}	0.4392	0.5739	0.6086	0.6238	0.6689	0.6269

Generally speaking, from Table 1 we can see the feature sets are helpful to the prediction task. Among all combinations, the 3-element combination F_{+t+r+p} outperforms the other combinations with F-measure 0.8216 and accuracy 0.7902. The 4-element combination $F_{+t+r+p+x}$ as the second best combination scores F-measure 0.7944 and accuracy 0.7722. We found that augmenting sibling features to F_{+t+r+p} is not helpful to improve the performance in the cases of three classifiers. We also found that the performance of F_{+r+p+x} which is derived by excluding feature set F_{+t} from $F_{+t+r+p+x}$ decreases a lot (the worst in all 3-element combinations) compared to the performance of $F_{+t+r+p+x}$. It indicates that the features from target URI u_t are important.

Although the batch classifiers performs well in the experiment, it does not mean they are suited for the task nor that they work well too in an online scenario. We show the performance of crawlers with offline classifiers and the proposed crawler with online classifier in the next section.

6.2 Online Versus Offline

In this experiment, we evaluate the performance of the proposed online prediction method against several baseline methods.

Metrics. The aim of the Linked Data crawler is to maximize the number of RDF-relevant URIs collected while minimizing the number of irrelevant URIs downloaded during the crawl. For our proposed method, the crawler has to download a fraction ϵ of URIs even though they are predicted negative. To better evaluate the performance of our approach, we use a *percentage* measure that equals the ratio of retrieved RDF-relevant URIs to the total number of URIs[5] crawled:

$$percentage = \frac{\#Retrieved\ RDF\ relevant\ URIs}{\#All\ retrieved\ URIs}$$

Methods. We implemented the proposed crawler denoted by **LDCOC** (Linked Data Crawler with Online Classifier). The decision threshold τ in Algorithm 1 is set to 0.5 and the parameter ϵ of LDCOC is set to 0.17 according to the ratio of the number of positive URIs to the number of negative URIs in the training set used in Sect. 6.1. As baselines, we also implemented three crawlers with offline classifiers including SVM, KNN and Naive Bayes to select URIs. The classifiers are pre-trained with two training sets (with size 20K and 40K). The BFS crawler is another baseline method to be compared to. As suggested in Sect. 6.1, we use the feature set F_{+t+r+p} for the experiment.

Table 2. Percentage of retrieved RDF relevant URIs by different crawlers

Crawler	Percentage
BFS	0.302
crawler_NB (20K)	0.341
crawler_NB (40K)	0.345
crawler_SVM (20K)	0.402
crawler_SVM (40K)	0.413
crawler_KNN (20K)	0.331
crawler_KNN (40K)	0.324
LDCOC ($\tau = 0.5, \epsilon = 0.17$)	**0.655**

[5] We only consider hard URIs.

Results. Table 2 shows the percentage of retrieved RDF relevant URIs by different crawlers after crawling 300K URIs. The results show that the crawlers with offline classifiers perform slightly better than BFS crawler. Considering that the Linked Data Web is a dynamic environment, the crawlers with offline classifiers pre-trained by a small size training set would not improve the performance a lot. This is the reason why we developed the crawler with an online classifier. The results show that our proposed crawler LDCOC outperforms crawlers based on static classifiers.

6.3 Processing Time of Per Selection

The processing time of selecting a URI is important since it affects the throughput of the crawling. The time to select one URIs mainly includes two parts: (1) feature generation; (2) prediction and predictor updating. Table 3 shows the average processing time per selection. LDCOC performs better than the other three crawlers with respect to processing time. Different from LDCOC crawlers with offline classifiers do not have to update during the crawl and only the prediction time is counted. LDCOC is based on FTRL-proximal algorithm which has been proven to work efficiently on the massive online learning problem of predicting ad click-through rates. The online efficiency of LDCOC can be guaranteed.

Table 3. Avg. processing time per selection

Crawler	Avg. processing time
crawler_NB	52.98 ms
crawler_SVM	66.62 ms
crawler_KNN	70.22 ms
LDCOC	**49.78** ms

7 Conclusion

We have presented a solution to learn URI selection criteria in order to improve the crawling of Linked Open Data by predicting their RDF-relevance. The prediction component is able to predict whether a newly discovered URI contains RDF content or not by extracting features from several sources and building a prediction model based on FTRL-proximal online learning algorithm. The experimental results demonstrate that the coverage of the crawl is improved compared to baseline methods. Currently, this work focuses on crawling linked RDF (RDFa) data. Our method can now be generalized to crawl other kinds of Linked Data such as JSON-LD, Microdata, etc. For future work, we are investigating more features such as the subgraphs induced by URIs and additional techniques such as graph embedding to further improve the predictions.

Acknowledgement. This work is supported by the ANSWER project PIA FSN2 N°P159564-2661789/DOS0060094 between Inria and Qwant.

References

1. Berners-Lee, T.: Linked data - design issues (2006). https://www.w3.org/DesignIssues/LinkedData.html
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
3. Burer, S., Monteiro, R.D.C.: A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. Math. Program. **95**(2), 329–357 (2003)
4. Chakrabarti, S., van den Berg, M., Dom, B.: Focused crawling: a new approach to topic-specific web resource discovery. Comput. Netw. **31**(11–16), 1623–1640 (1999)
5. Diligenti, M., Coetzee, F., Lawrence, S., Giles, C.L., Gori, M.: Focused crawling using context graphs. In: VLDB, pp. 527–534 (2000)
6. Dodds, L.: Slug: A Semantic Web Crawler (2006)
7. Duchi, J.C., Singer, Y.: Efficient learning using forward-backward splitting. In: NIPS, pp. 495–503 (2009)
8. Ermilov, I., Lehmann, J., Martin, M., Auer, S.: LODStats: the data web census dataset. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 38–46. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_5
9. Färber, M., Bartscherer, F., Menne, C., Rettinger, A.: Linked data quality of dbpedia, freebase, opencyc, wikidata, and YAGO. Seman. Web **9**(1), 77–129 (2018)
10. Heath, T., Bizer, C.: Linked Data: Evolving the Web Into a Global Data Space, vol. 1. Morgan & Claypool Publishers, San Rafael (2011)
11. Hogan, A., Harth, A., Passant, A., Decker, S., Polleres, A.: Weaving the pedantic web. In: LDOW (2010)
12. Hogan, A., Harth, A., Umbrich, J., Kinsella, S., Polleres, A., Decker, S.: Searching and browsing linked data with SWSE: the semantic web search engine. J. Web Sem. **9**(4), 365–401 (2011)
13. Isele, R., Umbrich, J., Bizer, C., Harth, A.: LDspider: an open-source crawling framework for the web of linked data. In: Proceedings of the ISWC 2010 Posters & Demonstrations Track (2010)
14. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. **6**, 707–710 (1966)
15. McMahan, H.B.: Follow-the-regularized-leader and mirror descent: equivalence theorems and L1 regularization. In: AISTATS, pp. 525–533 (2011)
16. McMahan, H.B., et al.: Ad click prediction: a view from the trenches. In: SIGKDD, pp. 1222–1230 (2013)
17. Meusel, R., Mika, P., Blanco, R.: Focused crawling for structured data. In: CIKM, pp. 1039–1048 (2014)
18. Umbrich, J., Harth, A., Hogan, A., Decker, S.: Four heuristics to guide structured content crawling. In: ICWE, pp. 196–202 (2008)
19. Weinberger, K.Q., Dasgupta, A., Langford, J., Smola, A.J., Attenberg, J.: Feature hashing for large scale multitask learning. In: ICML, pp. 1113–1120 (2009)
20. Xiao, L.: Dual averaging method for regularized stochastic learning and online optimization. In: NIPS, pp. 2116–2124 (2009)