Machine Learning-Based Query Augmentation for SPARQL Endpoints

Mariano Rico[1], Rizkallah Touma[2], Anna Queralt[2] and María S. Pérez[1]
[1]*Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain*
[2]*Barcelona Supercomputing Center (BSC), Barcelona, Spain*
{*mariano.rico, mperez*}*@fi.upm.es,* {*rizk.touma, anna.queralt*}*@bsc.es*

Keywords:
Query Augmentation, Linked Data, Semantic Web, SPARQL Endpoint, Query Type, Q-Type, Triple Pattern

Abstract:
Linked Data repositories have become a popular source of publicly-available data. Users accessing this data through SPARQL endpoints usually launch several restrictive yet similar consecutive queries, either to find the information they need through trial-and-error or to query related resources. However, instead of executing each individual query separately, query augmentation aims at modifying the incoming queries to retrieve more data that is potentially relevant to subsequent requests. In this paper, we propose a novel approach to query augmentation for SPARQL endpoints based on machine learning. Our approach separates the structure of the query from its contents and measures two types of similarity, which are then used to predict the structure and contents of the augmented query. We test the approach on the real-world query logs of the Spanish and English DBpedia and show that our approach yields high-accuracy prediction. We also show that, by caching the results of the predicted augmented queries, we can retrieve data relevant to several subsequent queries at once, achieving a higher cache hit rate than previous approaches.

1 INTRODUCTION

Linked Data repositories have grown to provide a wealth of publicly-available data, with some repositories containing millions of concepts described by RDF triples (e.g. DBpedia[1], FOAF[2], GeoNames[3]). Users access the data in these repositories through public SPARQL endpoints that allow them to issue SPARQL queries, the standard query language for RDF stores. Consecutive queries received from the same client usually exhibit some patterns, such as querying identical or similar resources than previous queries.

Caching query results was first proposed to keep recently retrieved data in a memory cache for use with later queries (Dar et al., 1996; Martin et al., 2010; Yang and Wu, 2011). However, caching only works if the exact same data is accessed multiple times. In reality, it is more common to have similar consecutive queries that retrieve related resources from the repository (Bonifati et al., 2017; Mario et al., 2011). Query augmentation takes advantage of this fact, retrieving data that will potentially be used by future queries before the queries are received by the SPARQL endopint. Previous approaches to query augmentation are divided into two main categories, (1) techniques based on information found in the data source, and (2) techniques based on analysis of previous (historic) queries, as discussed in section 2.

In this paper, we present an approach to query augmentation for SPARQL endpoints based on detecting recurring patterns in historic query logs. The novelty of our approach is that we measure two independent types of similarity between queries: structural similarity and triple-pattern similarity. Using the structural similarity, we apply a machine learning algorithm to predict the structure of the next query. Afterwards, we use the triple-pattern similarity to construct aug-

[1]DBpedia: `https://wiki.dbpedia.org/`
[2]FOAF: `http://www.foaf-project.org/`
[3]GeoNames: `http://www.geonames.org/`

mented triple patterns and predict which should be combined with the predicted structure to construct the augmented query. By doing so, we construct an augmented query that takes into consideration the structure of the next query and, at the same time, retrieves data relevant to several subsequent queries.

In our approach study, we show the accuracy of our prediction algorithm using query logs of both the English and Spanish DBpedia. We also estimate the cache hit rate that can be achieved by caching the results of the predicted augmented queries, finding that our method achieves a higher hit rate than previous approaches with a smaller number of cached queries.

The rest of this paper is organized as follows: section 2 reviews the related work in the fields of 'SPARQL query analysis' and 'SPARQL Query Augmentation'. Section 3 lists some SPARQL preliminaries and introduces a running example. Section 4 describes and formalizes the proposed approach. Section 5 details our experimental study and shows the viability of our approach. Finally, section 6 concludes the paper and highlights some future work.

2 RELATED WORK

In this section, we provide an overview of the most important approaches in the two fields from which we draw our work: (1) SPARQL Query Analysis, and (2) SPARQL Query Augmentation.

2.1 SPARQL Query Analysis

The motivation to analyze the queries logged by SPARQL endpoints started with the work of Moller *et al.* (Möller et al., 2010), who promoted the creation of the USEWOD workshop[4]. They used the information in the query logs to show that, for the 4 data sets they studied, more than 90% of queries were SELECT queries.

Mario *et al.* (Mario et al., 2011) used the USEWOD 2011 dataset (7 million SPARQL queries from DBpedia and SWDF) to find the most used features and concluded that most queries are simple and include a few triple patterns and joins (Groppe et al., 2009). They also pointed that 99.7% of valid queries were SELECT queries.

Raghuveer *et al.* used the USEWOD 2012 dataset to manually collect what they called 'canonical form' of SPARQL queries in order to detect repetitive patterns in the creation of queries (Raghuveer, 2012). This might seem similar to our approach to detect query templates, but we introduce the concepts of 'inner tree' and 'surface form' and we can extract these structures automatically from any query.

The work of Bonifati *et al.* is based on the largest studied set of SPARQL query logs to date (Bonifati et al., 2017). They used over 170 million queries from 14 different sources to perform a multi-level analysis of common features in SPARQL queries. They reached similar conclusions to previous studies regarding the commonality of SELECT queries and the fact that most of these queries are simple and only contain one or two triple patterns (Bonifati et al., 2017).

Finally, Dividino and Groner classify the existing methods to measure the similarity of SPARQL queries in 4 categories: structure, content, language and result set (Dividino and Gröner, 2013). Depending on the application purposes, a combination of these 4 dimensions provides the best metric. In our approach, we perform a structural categorization of queries and combine it with content-similarity measures to match SPARQL queries in a query log.

2.2 SPARQL Query Augmentation

Query augmentation, also called *query relaxation*, aims at retrieving related information based on a user query that is potentially needed for subsequent queries. There are two main categories of query augmentation techniques: (1) techniques based on information found in the data source, and (2) techniques based on analysis of previous historic queries.

In the first category, Hurtado *et al.* suggest logical augmentations based on ontological metadata (Hurtado et al., 2008). In contrast, Hogan *et al.* propose an approach that relies on precomputed similarity tables for attribute values (Hogan et al., 2012), whereas Elbassuoni *et al.* utilize a language model derived from the knowledge base to perform query augmentation (Elbassuoni et al., 2011). Given that these techniques need data from the data source, they require at least some precomputations to be performed before they can be applied. Furthermore, they are not portable across data sources since the required information might not always be available.

In contrast, techniques that are based on historic query logs are more portable across data

[4]USEWOD Workshop: `http://usewod.org/`

sources since they do not require any specific information from the data source. Lorey *et al.* propose the first work in this category by detecting recurring patterns in past queries and creating query templates based on a bottom-up graph pattern matching algorithm (Lorey and Naumann, 2013b). The same authors extend their work by combining these templates with four different query augmentation strategies but do not reach any conclusive results on which strategy offers the best results (Lorey and Naumann, 2013a). Another approach is proposed by Zhang *et al.* who measure similarity between SPARQL queries using a Graph Edit Distance (GED) function and use similar previous queries to 'suggest' data for prefetching (Zhang et al., 2016).

Our approach belongs to the second group of query augmentation strategies, since it is based on analyzing queries received by the SPARQL endpoint. However, unlike previous approaches, we do no directly launch an augmented query but use a two-step prediction process to predict the structure of the augmented query before individually predicting which triple patterns to use. This separation allows us to take the query structure into account without performing any graph matching between each pair of SPARQL queries.

3 SPARQL PRELIMINARIES AND MOTIVATING EXAMPLE

SPARQL queries have four different query forms, namely SELECT, DESCRIBE, ASK and CONSTRUCT. Previous studies show that the most common query starts with one or more PREFIX items followed by a SELECT structure (Mario et al., 2011; Möller et al., 2010). Therefore, in our approach we only consider SPARQL queries of the SELECT form and we do not study the less common forms.

The central construct of a SPARQL SELECT query is a 'Triple Pattern'. A triple pattern is defined as $T = \langle s, p, o \rangle \in (V \cup U) \times (V \cup U) \times (V \cup U \cup L)$ where V is a set of variables, U a set of URLs and L a set of literals (Pérez et al., 2009). The three parts of a triple pattern correspond to a subject, a predicate and an object.

A set of one or more triple patterns constitute a Basic Graph Pattern (BGP). A SELECT query can contain one or more BGPs, joined with the SPARQL keywords AND, UNION or OPTIONAL. These BGPs form the query's graph pattern. Our approach takes into account the triple patterns of a query graph pattern and does not consider other features such as FILTER, LIMIT or ORDER BY.

We call a consecutive sequence of queries received by the SPARQL endpoint from the same client a 'Query Session'. As previous studies have demonstrated, queries in the same session tend to be similar to each other with only minor changes occurring between them (Dividino and Gröner, 2013; Picalausa and Vansummeren, 2011). In this paper, we define the length of a query session to be a one-hour time window.

Example. Listing 1 shows a query session consisting of four SELECT queries received by a SPARQL endpoint that will be used as a running example throughout the paper. The queries in this session look up former teams of different football players and ask for some properties of these teams. We use the line numbers in the listing to refer to the triple patterns. For instance, we refer to the triple pattern `dbr:Cristiano_Ronaldo dbo:formerTeam ?team` on line 4 as T_4.

We can see that the triple patterns of the queries in Listing 1 are quite similar to each other. For instance, T_{10} is identical to T_4 whereas T_{19} and T_{25} have a different subject but the same predicate and object. Our approach uses a supervised learning algorithm to capture the repetitive patterns of the changes occurring between the triple patterns to predict the changes that lead to the triple patterns of the augmented queries.

4 PROPOSED APPROACH

The main goal of our approach is to construct augmented queries that retrieve data relevant to subsequent queries received by a SPARQL endpoint. To do so, we first extract the structure of the queries and construct query types (Section 4.1). Second, we perform a matching of triple patterns between the queries received by the SPARQL endpoint (Section 4.2) and then construct individual augmented triple patterns using the generated matchings (Section 4.3). Afterwards, we use supervised machine learning algorithms to capture the repetitive patterns between previous queries and apply a two-step prediction process: (1) we first predict which query type should come next, and, (2) we predict which augmented triple patterns should be combined with the predicted query type to construct the augmented query (Section 4.4).

Listing 1: Example query session of SPARQL SELECT queries

```
1  Q1 : PREFIX dbr: <http://dbpedia.org/
       resource/>
2       PREFIX dbo: <http://dbpedia.org/
       ontology/>
3       SELECT * WHERE {
4         dbr:Cristiano_Ronaldo  dbo:
       formerTeam  ?team .
5       }
6
7  Q2 : PREFIX dbr: <http://dbpedia.org/
       resource/>
8       PREFIX dbo: <http://dbpedia.org/
       ontology/>
9       SELECT * WHERE {
10        dbr:Cristiano_Ronaldo  dbo:
       formerTeam  ?team .
11        OPTIONAL {
12          ?team dbo:manager ?manager .
13        }
14      }
15
16 Q3 : PREFIX dbr: <http://dbpedia.org/
       resource/>
17      PREFIX dbo: <http://dbpedia.org/
       ontology/>
18      SELECT * WHERE {
19        dbr:Iker_Casillas  dbo:
       formerTeam  ?team .
20      }
21
22 Q4 : PREFIX dbr: <http://dbpedia.org/
       resource/>
23      PREFIX dbo: <http://dbpedia.org/
       ontology/>
24      SELECT * WHERE {
25        dbr:Gerard_Pique  dbo:
       formerTeam  ?team .
26        ?team  dbo:manager  ?manager .
27      }
```

4.1 Query Types (Q-Types)

The aim of a 'Query Type', also denoted Q-Type, is to capture the syntactic structure of a given SELECT query. We compute the Q-Type of a query by generating the query's parse tree (following the SPARQL 1.1 grammar), removing the leaves of the tree and serializing the resulting tree. We denote 'surface form' to the leaves of the tree, and 'inner tree' to the rest of the tree. Therefore, we say that two queries have the same Q-Type, and hence are structurally similar, if they differ only in their 'surface form'. That is, they have the same 'inner tree' but different variable names, resources and literals in their 'surface form'.

Example. Listing 2 shows a sample SPARQL SELECT query with one triple pattern.

Listing 2: Example of a SPARQL SELECT query

```
PREFIX foaf: <http://xmlns.com/foaf
    /0.1/>
SELECT * WHERE
  {
    ?x foaf:mbox ?mbox .
  }
```

Figure 1 shows the parse tree of the SPARQL query from Listing 2. The query's surface form, which represents the text seen in the decoded query, is located in the leaf nodes of the tree. Listing 3 shows the serialization of the parse tree in Figure 1 following a top-down, left to right, visiting algorithm.

Listing 3: Serialization of the parse tree in Figure 1

```
(QUERY (PROLOGUE
  (PREFIX foaf:
    <http://xmlns.com/foaf/0.1/>))
(SELECT (SELECT_CLAUSE *)
(WHERE_CLAUSE
  (GROUP_GRAPH_PATTERN
    (TRIPLES_SAME_SUBJECT
      (SUBJECT ?x)
      (PREDICATE (PATH
        (PATH_SEQUENCE
          (PATH_ELT_OR_INVERSE
            (PATH_PRIMARY foaf:mbox))))
      (OBJECT ?mbox)))))))
```

Finally, Listing 4 is the serialization of its inner tree, that is, after eliminating the surface form of the query. Note that this serialization only contains the tokens of the SPARQL grammar.

Listing 4: Serialization of the inner tree in Figure 1

```
QUERY ( PROLOGUE ( PREFIX ( ) )
SELECT ( SELECT_CLAUSE ( )
WHERE_CLAUSE
  ( GROUP_GRAPH_PATTERN
    ( TRIPLES_SAME_SUBJECT
      ( SUBJECT ( )
      PREDICATE ( PATH
        ( PATH_SEQUENCE (
          PATH_ELT_OR_INVERSE
            ( PATH_PRIMARY ( ) ) ) )
      OBJECT ( ) ) ) ) ) ) )
```

This inner tree represents the Q-Type that allows us to group structurally-similar queries. For instance, examples of queries with the same Q-Type are Q_1 and Q_3 from the sequence of queries shown in Listing 1. We can see that both queries have the same inner structure and the differences are only present in their surface forms. On the other

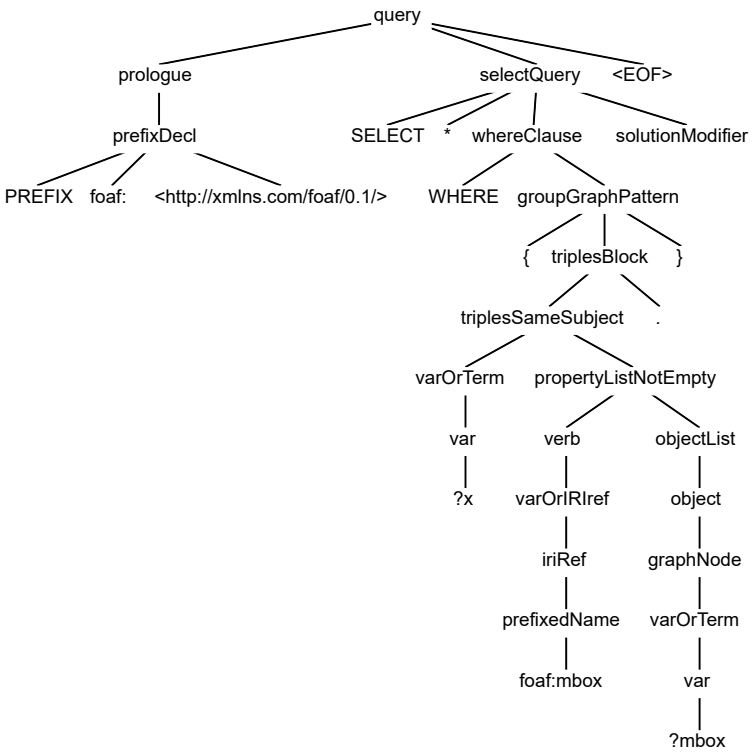


Figure 1: Parse tree of the SPARQL SELECT query in Listing 2.

hand, queries Q_2 and Q_4 have different Q-Types, since their structure is different.

As we can see, the Q-Types capture the structure of a SPARQL query, including how its triple patterns form BGPs and, if necessary, how the BGPs connect with each other using the keywords AND, UNION and OPTIONAL. This eliminates the need to do graph matching to measure the structural similarity between queries and allows to only perform simple triple pattern matching.

4.2 Triple Pattern Matching

In order to capture the changes that occur between the triple patterns of the queries received by a SPARQL endpoint, we match the most similar triple patterns together. We do so by counting the number of triple pattern parts (i.e. subjects, predicates and objects) that are different between two triple patterns. In this measure, we say that two triple pattern parts are identical, and hence their distance is 0, if they are both variables or have the same URL or literal. Otherwise, we say that their distance is 1. More formally, assuming that x_1, x_2 are either the subjects, predicates or objects of two triple patterns $T_1 = \langle s_1, p_1, o_1 \rangle$ and $T_2 = (s_2, p_2, o_2)$, we define the distance between the two parts $\Delta(x_1, x_2)$ as:

$$\Delta(x_1, x_2) = \begin{cases} 0, \ if \ (x_1 \in V \wedge x_2 \in V) \vee (x_1 = x_2) \\ 1, \ otherwise \end{cases} \tag{1}$$

We then determine the overall distance between the two triple patterns by aggregating the individual triple pattern part distances as follows:

$$\Delta(T_1, T_2) = \Delta(s_1, s_2) + \Delta(p_1, p_2) + \Delta(o_1, o_2) \tag{2}$$

This function is based on the distance function defined by Lorey *et al.* (Lorey and Naumann, 2013a). In the original definition, the authors use a Levenshtein distance to compare two URLs or literals when measuring the distance between two triple pattern parts $\Delta(x_1, x_2)$ and then use a more complex aggregation to compute $\Delta(T_1, T_2)$. We modified it in our approach since we are only interested in counting the number of different triple pattern parts between T_1 and T_2, regardless of whether they are variables, URLs or literals.

We also introduce a restriction not found in the original definition to guarantee that the

matched triple patterns are not too different from each other. We do so by limiting the distance between the matched triple patterns to $\Delta(T_1, T_2) \leq 1$, i.e. the two triple patterns are different in at most one part. If more than one triple pattern can be matched with the same distance, the one that occurs most recently in the query session is considered. If no such match can be found, we say that the triple pattern is "unmatched".

Example. Looking at the queries in Listing 1, we match their triple patterns as follows:

- Q_1: the first query in the session and there are no previous queries to do the matching.
- Q_2: the first triple pattern T_{10} is identical to T_4 while the second triple pattern T_{12} is unmatched.
- Q_3: its triple pattern T_{19} is matched to T_{10} with a change in the subject.
- Q_4: the first triple pattern T_{25} is matched to T_{19} with a change in the subject. The second triple pattern T_{26} is identical to T_{12}.

4.3 Augmented Triple Patterns

For each pair of triple patterns matched as described in Section 4.2, we construct an Augmented Triple Pattern $aug(T_1, T_2)$. If the matched triple patterns are identical, the augmented triple pattern is identical to both of them as well. Otherwise, we construct the augmented triple pattern by substituting the part that is different between them with a variable. For consistency, the same URL or literal is always replaced with the same variable. If a triple pattern is unmatched, then the corresponding augmented triple pattern is identical. Formally, we define $aug(x_1, x_2)$ as the augmented part of two triple pattern parts as follows:

$$aug(x_1, x_2) = \begin{cases} x_1 = x_2, & if\ \Delta(x_1, x_2) = 0 \\ ?var_i\ where\ ?var_i = aug(x_1, x_i) & \\ \qquad \forall x_i, & otherwise \end{cases} \tag{3}$$

We then define $aug(T_1, T_2)$ for a pair of matched triple patterns as:

$$aug(T_1, T_2) = \langle aug(s_1, s_2), \\ aug(p_1, p_2),\ aug(o_1, o_2) \rangle \tag{4}$$

The aim of augmented triple patterns is twofold. First, they capture the changes that occur between the triple patterns of queries in a session. This allows us to use them to predict the triple patterns of the augmented query based on changes in previous queries in the session. Second, they are more abstract than the original triple patterns occurring in the queries and hence they retrieve additional data that is potentially relevant for subsequent queries as well.

Example. Given the matchings between the triple patterns of the queries in Listing 1, we construct the following augmented triple patterns:

- $aug_1 = aug(T_{10}, T_4) = T_{10} = T_4$: since T_{10} and T_4 are identical.
- $aug_2 = aug(T_{19}, T_{10}) = aug(T_{25}, T_{19}) =$ `?var1 dbo:formerTeam ?team`: since T_{10}, T_{19} and T_{25} only differ from each other in the subject
- $aug_3 = aug(T_{26}, T_{12}) = T_{26} = T_{12}$: since T_{26} and T_{12} are identical.

4.4 Constructing Augmented Queries

To predict and construct an augmented query, we use the Q-Types and augmented triple patterns of previous queries in the same query session. More precisely, we use the Q-Types of previous queries to predict the Q-Type, and hence structure, of the next query in the query session. Afterwards, we predict which augmented triple patterns should be combined with the Q-Type to construct the 'surface form' of the augmented query. By doing so, we construct an augmented query that takes into account the structure of the next query and retrieve data relevant to several subsequent queries at the same time.

We formulate the prediction process as a multi-class classification problem, using one classifier to predict the Q-Type of the upcoming query and one classifier to predict *each* augmented triple pattern in that Q-Type. For the Q-Type classifier, we use as features the Q-Types of previous queries in the session. As for the augmented triple patterns, the feature vectors include one feature for each augmented triple pattern of each of the previous queries in the session, regardless of their position in the original query. The classifier is then used to predict which augmented triple pattern should come in the ith position of the predicted Q-Type.

Example. Using the queries in Listing 1, and assuming we use 2 previous queries in the classifier model, we would have the following features:

$$q\text{-}type(Q_1),\ q\text{-}type(Q_2) \rightarrow q\text{-}type(Q_3)$$
$$q\text{-}type(Q_2),\ q\text{-}type(Q_3) \rightarrow q\text{-}type(Q_4)$$

Similarly, the feature vectors of the augmented triple pattern classifiers would be the following. The first two features correspond to augmented triple patterns of Q_1, the next two features to Q_2 and so on. Note that if a query has less triple patterns than the maximum, we use the question mark '?' to indicate that this feature is missing. Classifier features for *first* triple pattern:

$$aug_1, ?, aug_1, aug_3 \rightarrow aug_2$$
$$aug_1, aug_3, aug_2, ? \rightarrow aug_2$$

Classifier features for *second* triple pattern:

$$aug_1, ?, aug_1, aug_3 \rightarrow ?$$
$$aug_1, aug_3, aug_2, ? \rightarrow aug_3$$

We then train the classifiers on historical data and when a new query arrives to the SPARQL endpoint, we compute its Q-Type and augmented triple patterns and run the information through the trained classifier to obtain the predicted augmented query. For instance, using the queries in Listing 1, let's assume that the classifiers predict that the next query, Q_5, is of type $q\text{-}type(Q_2)$ and that its augmented triple patterns are aug_2 and aug_3. Using these predictions, the surface form of the constructed augmented query would be the one shown in Listing 5. This query is then used to retrieve the data retrieved by the original next query, as well as related data potentially relevant to subsequent queries.

Listing 5: Surface form of a constructed augmented query

```
1 Q5 : PREFIX dbr: <http://dbpedia.org/
       resource/>
2      PREFIX dbo: <http://dbpedia.org/
       ontology/>
3      SELECT * WHERE {
4        ?var1  dbo:formerTeam  ?team .
5        OPTIONAL {
6          ?team dbo:manager ?manager .
7        }
8      }
```

5 APPROACH STUDY

We evaluated our approach by studying the Spanish DBpedia (esDBpedia) query logs extracted directly from the esDBpedia SPARQL endpoint and the English DBpedia (enDBpedia) logs published for the 2013 USEWOD workshops[5]. The log files contain a sequence of requests received by the respective public SPARQL endpoints and cover different periods between 2012 and 2013. We extracted the SPARQL SELECT queries from other SPARQL queries and HTTP requests for use in our experiments. Table 1 shows the most relevant facts about the extracted datasets. As we can see, the esDBpedia dataset covers more months but the enDBpedia has a more diverse dataset, both in terms of distinct SELECT queries and IPs from which the queries were made.

Table 1: Characteristics of the datasets used in our experiments. Numbers of queries and distinct queries refer to SELECT queries only.

	esDBpedia	**enDBpedia**
Total Queries	167,810	203,874
Distinct Queries	46,397	105,284
Distinct IPs	2,197	8,918
Sessions	963	619
Months Covered	12	3

We divided the logs according to the requesting IP and considered the n previous queries from the same IP in our classifiers. We experimented with different values of n to see the influence of the number of considered queries on the classifiers' results. For the esDBpedia dataset, we included the time intervals between consecutive queries as additional classifier features. We could not do the same with the enDBpedia dataset because the published logs did not include the queries' timestamps.

We also calculated the number of queries made from each IP and concluded that it seems to follow a power-law distribution, that is, a small number of IP addresses is responsible for a big number of queries. The main implication of such a generalized behavior is that the SPARQL endpoints of the Linked Data repositories could be optimized to take advantage of this 80-20 behavior. Due to space limitations, we do not include the implications of this behavior on our approach in this paper and leave it to future work.

For our classification problem, we used the J48 decision tree classifier (using Weka 3.8.1[6]) and tested the classifiers by using 10 fold cross-validation. In all of our experiments, we used as

[5] 2013 USEWOD Workshop: https://eprints.soton.ac.uk/379399/

[6] Weka: https://www.cs.waikato.ac.nz/ml/index.html

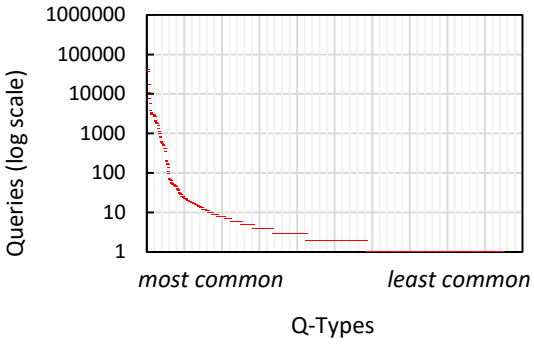
a baseline the ZeroR classifier, which predicts all instances to be of the most common class. To ensure the reproducibility of our experiments, we have made all of the training datasets and experimental results publicly available at `http://prefetch.linkeddata.es`.

5.1 Q-Type Prediction

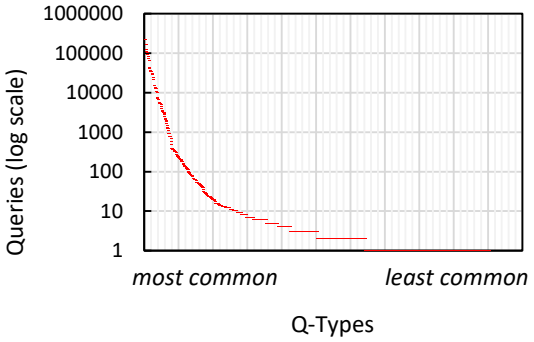
We started our study by calculating the number of generated Q-Types. We found that the queries of the esDBpedia dataset correspond to 943 Q-Types whereas in the enDBpedia logs we found 3,139 Q-Types. Figure 2 shows the distribution of queries among the computed Q-Types plotted in logarithmic scale. We can see from Figure 2 that the distribution of Q-Types is very skewed, with a large number of Q-Types corresponding to few queries and only a handful of Q-Types corresponding to the majority of queries. Given this distribution, in the rest of the experiments we only consider the most common Q-Types that cover the vast majority of the queries. More precisely, we consider 56 Q-Types that cover 98.5% of all queries in the esDBpedia dataset, whereas in the enDBpedia dataset we consider 60 Q-Types that cover 98.1% of all queries.

Using the most common Q-Types, we evaluated the classifier's precision in predicting the Q-Type of the next query when considering different numbers of previous queries, n. Figure 3 shows the classifier precision on both datasets. For esDBpedia, the classifier achieves high accuracy even when $n = 2$ and reaches a peak of 96.34% when $n = 15$. As for the enDBpedia dataset, the classifier's peak precision of 89.95% is achieved when $n = 10$. In general, the classifier achieves worse precision with the enDBpedia dataset, which indicates that the queries received by the enDBpedia SPARQL endpoint are more diverse and do not follow a predictable pattern such as with esDBpedia. Note that the baseline for this experiment is 22.09% for esDBpedia and 15.35% for enDBpedia.

We also evaluated the accuracy of the classifier with less-common Q-Types. Figure 4 shows the classifier's precision (number of correctly-classified instances divided by the total number of classified instances) and recall (number of correctly-classified instances divided by the total number of instances of the class) for each of the included Q-Types in both datasets. We chose the values of n that offer the highest overall accuracy to perform this experiment, namely with $n = 15$ for esDBpedia and $n = 10$ for enDBpedia.



(a) esDBpedia



(b) esDBpedia

Figure 2: Number of queries (in log scale) corresponding to each of the computed Q-Types. The x-axis ranks the Q-Types from most common (left) to least common (right).

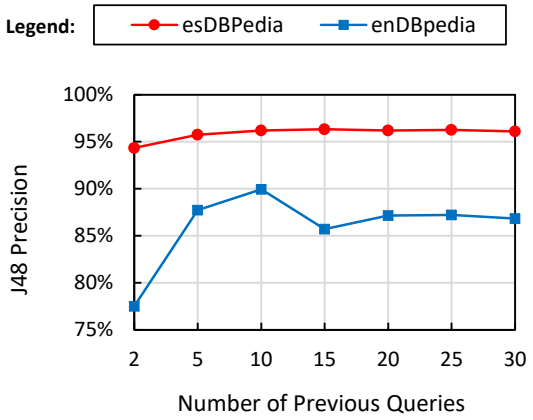
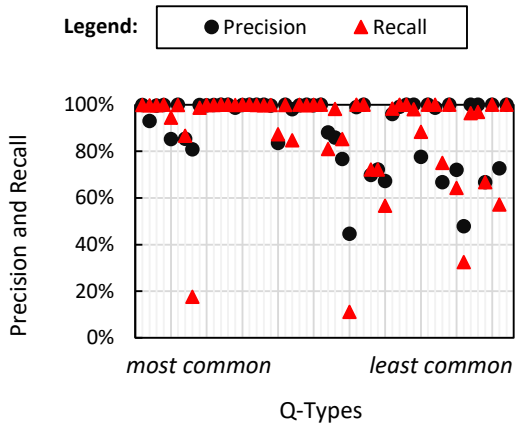
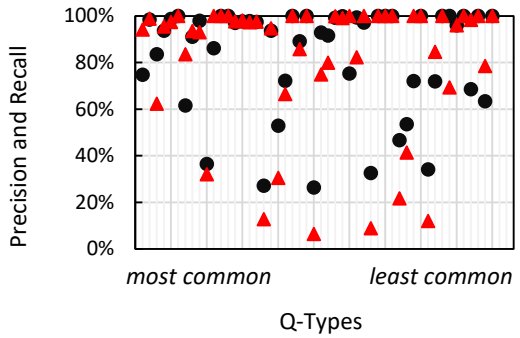


Figure 3: Precision of the Q-Type classifier.

For the esDBpedia dataset, we can see that the classifier has both precision and recall of over 80% in the majority of cases and its recall only drops

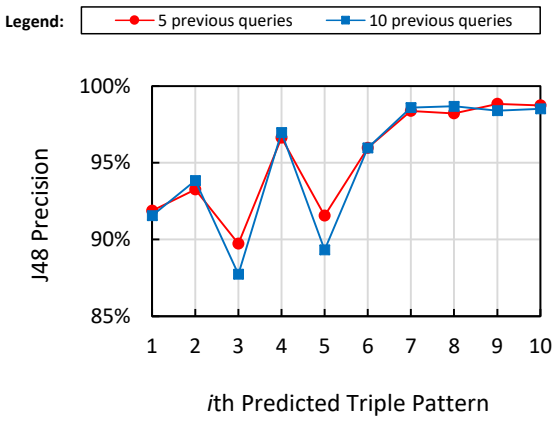


(b) esDBpedia

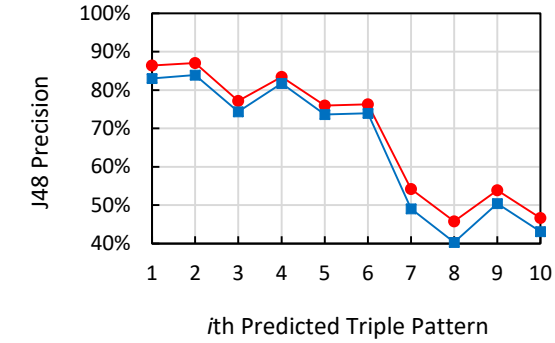


(c) enDBpedia

Figure 4: Q-Type classifier precision and recall for each of the included Q-Types. The x-axis ranks the Q-Types from most common (left) to least common (right). Each marker represents the precision (black) or recall (orange) for a Q-Type.



(b) esDBpedia



(c) enDBpedia

Figure 5: Precision of the triple patterns classifiers on the studied datasets.

below 50% for 3 of the included Q-Types. On the other hand, the classifier registers a similar drop with 8 Q-Types in the case of enDBpedia. The classifier performs badly with these types because it cannot distinguish them from other types with the used features. We argue that the solution could be to include other features in the classifier models, such as the time interval between queries in the enDBpedia.

5.2 Prediction of Augmented Triple Patterns

After evaluating the Q-Type prediction algorithm, we studied the accuracy of the classifiers in predicting the augmented triple patterns (as discussed in section 4.2) that are used with the predicted Q-Type to construct the augmented query.

Figure 5 shows the classifier's precision on both datasets, the x-axis indicates the number of augmented triple patterns in the predicted Q-Type and the two series show the results when considering 5 and 10 previous queries. A common behavior that can be observed in figure 5 in both datasets is that, unlike the Q-Type classifier, increasing n does not always increase the precision of the augmented triple-pattern classifiers. This indicates that the predicted triple patterns appear in previous queries even when $n = 5$ or $n = 10$ and any further increase only adds more unnecessary data points to the classifiers model.

It is also worth noting that the classifier results are completely different when considering queries that have more than 6 triple patterns, with the precision increasing to around 98% with esDBpedia and dropping to below 50% with enDBpedia. This can be explained as follows: 21.3% of queries in esDBpedia have more than 6 triple patterns, of which 98.2% are duplicates.

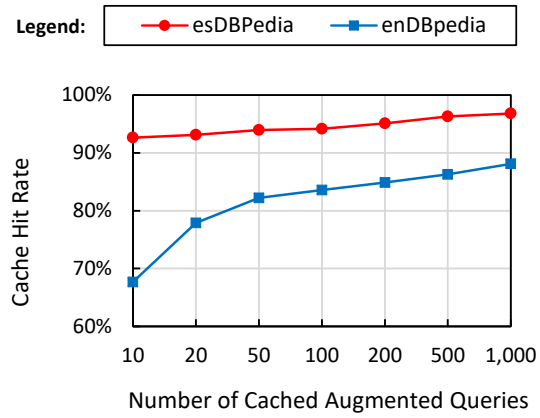


Figure 6: Cache Hit Rate based on the constructed augmented queries.

On the other hand, the percentage of queries with more than 6 triple patterns drops to only 10.8% in the enDBpedia, out of which only 33.7% are duplicates. The extremely high duplicates rate explains the high accuracy of the classifier with esDBpedia, while the small number of queries with more than 6 triple patterns in the enDBpedia dataset, coupled with the low duplication rate, is not sufficient to train a classifier model with high accuracy.

5.3 Cache Hit Rate

We performed a final experiment to estimate the 'cache hit rate' that our approach can achieve by caching the predicted augmented queries. We did so by calculating the percentage of queries for which all triple patterns occur in an augmented query previously predicted in the same session. When this happens, assuming that we cache the results of the predicted queries, we have a 'cache hit' since the cached results will also be results of the query being predicted.

Figure 6 shows the cache hit rates that can be achieved by caching different numbers of predicted augmented queries. It indicates that, for esDBpedia, we can have cached results for between 92.63% and 96.80% of future queries, depending on the number of cached queries. On the other hand, the hit rate for enDBpedia ranges between 67.70% when only caching 10 augmented queries and 88.10% when caching 1,000 augmented queries.

Compared to previous approaches, Zhang *et al.* reported an average cache hit rate of 76.65% using a dataset of enDBpedia queries of a similar size (Zhang et al., 2016) and a cache of 1,000 queries. We could not readily compare our approach to the work of Lorey *et al.* since the authors do not provide comparable measures in their evaluation (Lorey and Naumann, 2013b).

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to query augmentation in SPARQL endpoints based on measuring two independent types of similarity between SPARQL SELECT queries. We use syntactic parse trees to measure the structural similarity of SPARQL queries and create Query Types which we use to predict the structure of the next query. Independently, we measure the similarity between the queries' triple patterns, and use the similarities to construct augmented triple patterns. We then combine the two predictions to construct an augmented query that can be used to retrieve data relevant to subsequent queries in the query session.

We evaluated our approach on the SPARQL endpoint query logs of the Spanish and English DBpedia. The results show that the prediction of both Q-Types and augmented triple patterns does not require a large number of queries, only between 10 to 15, to achieve high precision. This indicates that our approach can be used in both long and short query sessions alike. In general, the classification precision is higher for the esDBpedia dataset, due to the fact that the enDBpedia logs are more diverse and contain more unique queries. For a minority of cases, namely for queries containing more than 6 triple patterns, the classifier accuracy drops for the enDBpedia due to the insufficient size of this subset of queries. However, our approach can still achieve a cache hit rate of around 85% for the enDBpedia dataset, which is considerably higher than previous augmentation approaches.

In the future, we intend to implement a full caching and prefetching system using our proposed query augmentation approach. We also plan to extend our prediction method to take into account other features of SELECT queries, such as FILTER clauses, as well as other less common forms of SPARQL queries. Finally, we want to distinguish human query sessions from sessions made by machine agents to test the effectiveness of our approach on both types and optimize it accordingly.

ACKNOWLEDGEMENTS

This work has been supported by the European Union's Horizon 2020 research and innovation program (grant H2020-MSCA-ITN-2014-642963), the Spanish Ministry of Science and Innovation (contract TIN2015-65316, project RTC-2016-4952-7 and contract TIN2016-78011-C4-4-R), the Spanish Ministry of Education, Culture and Sports (contract CAS18/00333) and the Generalitat de Catalunya (contract 2014-SGR-1051). The authors would also like to thank Toni Cortes for his feedback.

REFERENCES

Bonifati, A., Martens, W., and Timm, T. (2017). An analytical study of large sparql query logs. *Proc. VLDB Endow.*, 11(2):149–161.

Dar, S., Franklin, M. J., Jónsson, B. T., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 330–341, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Dividino, R. and Gröner, G. (2013). Which of the following SPARQL queries are similar? why? In *Proceedings of the First International Workshop on Linked Data for Information Extraction (LD4IE 2013)*, pages 1–12.

Elbassuoni, S., Ramanath, M., and Weikum, G. (2011). Query relaxation for entity-relationship search. In *Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II*, ESWC'11, pages 62–76, Berlin, Heidelberg. Springer-Verlag.

Groppe, J., Groppe, S., Ebers, S., and Linnemann, V. (2009). Efficient processing of SPARQL joins in memory by dynamically restricting triple patterns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1231–1238. ACM.

Hogan, A., Mellotte, M., Powell, G., and Stampouli, D. (2012). Towards fuzzy query-relaxation for rdf. In *Proceedings of the 9th Extended Semantic Web Conference, ESWC 2012*, pages 687–702. Springer Berlin Heidelberg.

Hurtado, C. A., Poulovassilis, A., and Wood, P. T. (2008). Query relaxation in RDF. In *Journal on Data Semantics X*, pages 31–61. Springer-Verlag.

Lorey, J. and Naumann, F. (2013a). *Caching and Prefetching Strategies for SPARQL Queries*, pages 46–65. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lorey, J. and Naumann, F. (2013b). *Detecting SPARQL Query Templates for Data Prefetching*, pages 124–139. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mario, A., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. (2011). An empirical study of real-world SPARQL queries. In *1st International Workshop on Usage Analysis and the Web of Data USEWOD 2011*.

Martin, M., Unbehauen, J., and Auer, S. (2010). Improving the performance of semantic web applications with SPARQL query caching. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part II*, ESWC'10, pages 304–318, Berlin, Heidelberg. Springer-Verlag.

Möller, K., Hausenblas, M., Cyganiak, R., and Handschuh, S. (2010). Learning from linked open data usage: patterns & metrics. In *Proceedings of the WebSci10: Extending the Frontiers of Society On-Line,*.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45.

Picalausa, F. and Vansummeren, S. (2011). What are real SPARQL queries like? In *Proceedings of the International Workshop on Semantic Web Information Management*, page 7. ACM.

Raghuveer, A. (2012). Characterizing machine agent behavior through SPARQL query mining. In *Proceedings of the International Workshop on Usage Analysis and the Web of Data*.

Yang, M. and Wu, G. (2011). Caching intermediate result of SPARQL queries. In *Proceedings of the 20th International Conference Companion on World Wide Web*, WWW '11, pages 159–160, New York, NY, USA. ACM.

Zhang, W. E., Sheng, Q. Z., Qin, Y., Yao, L., Shemshadi, A., and Taylor, K. (2016). Secf: Improving SPARQL querying performance with proactive fetching and caching. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 362–367, New York, NY, USA. ACM.