Improving content discovery through combining linked data and data mining techniques

Ross Fenning

April 25, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

Media companies produce ever larger numbers of articles, videos, podcasts, games, etc. – commonly collectively known as "content". A successful content-producing website not only has to develop systems to aid producing and publishing that content, but there are also demands to engineer effective mechanisms to aid consumers in finding that content.

Approaches used in industry include providing a text-based search, hierarchical categorisation (and thus navigation thereof) and even more tailored recommended content based on past behaviour or content enjoyed by friends (or sometimes simply other consumers who share your preferences).

1.1 Problems

There are several technical and conceptual problems with building effective content discovery mechanisms, including:

- Large organisations can have content across multiple content management systems, in differing formats and data models. Organisations face a large-scale enterprise integration problem simply trying to gain a holistic view of all their content.
- Many content items are in fairly opaque formats, e.g. video content may be stored as audio-visual binary data with minimal metadata to display on a containing web page. Video content producers may not be motivated to provide data attributes that might ultimately be most useful in determining if a user will enjoy the video.
- Content is being published continuously, which means any search or discovery system needs to keep up with content as it is published and process it into the appropriate data structures. Any machine learning previously performed on the data set may need to be re-run.

1.2 Hypothesis

The following hypotheses are proposed for gaining new insights about an organisation's diverse corpus of content:

- Research and software tools around the concept of *Linked Data* can aid us in rapidly acquiring a broad view (perhaps at the expense of depth) of an organisation's content whilst also providing a platform for simple enrichment of that content's metadata.
- We can establish at least a naïve mapping of an RDF graph representing a content item to an attribute set suitable for data mining. With such a mapping, we can explore applying machine learning – particularly unsupervised learning – across an organisation's whole content corpus.
- Linked Data and Semantic Web *ontologies* and models available can provide data enrichment beyond attributes and keywords explicitly avaiable within content data or metadata.
- We can adapt established machine learning approaches such as clustering for data published continuously in real time.
- Many content-producers currently enrich their web pages with small amounts of semantic metadata to provide better presentation of that content as it is shared on social media. This enables simple collection of a full breadth of content with significantly less effort than direct integration with content management systems.

CHAPTER

TWO

BACKGROUND

This chapter discusses some of the existing research and technologies around machine learning, RDF and combining them. It also covers some of the advantages of using linked data and RDF in an enterprise setting and what tools and approaches are well-defined enough that a corporation could build on top of them rapidly.

Data mining activities such as machine learning rely on structuring data as *feature sets*[2] – a set or vector of properties or attributes that describe a single entity. The process of *feature extraction* generates such feature sets from raw data and is a necessary early phase for many machine learning activities.

The rest of this chapter will show:

1. that extracting feature sets from RDF[1] graphs can be done elegantly and follows naturally from some previous work in this area; and
2. that the RDF graph is a suitable and even desirable data model for content metadata in terms of acquiring, enriching and even transforming that data ahead of feature extraction.

2.1 Data Mining

TODO

2.2 RDF and Feature Extraction

The RDF graph is a powerful model for metadata based on representing knowledge as a set of subject-predicate-object *triples*. The query language, SPARQL, gives us a way to query the RDF graph structure using a declarative pattern and return a set of all variable bindings that satisfy that pattern.

For example, the SPARQL query in Listings 2.1 queries an RDF graph that contains contact information and returns the names and email address of all "Person" entities therein.

[1] http://www.w3.org/TR/PR-rdf-syntax/

Notably, Kiefer, Bernstein and Locher[7] proposed a novel approach called SPARQL-ML – an extension to the SPARQL[13] query language with new keywords to facilitate both generating and applying models. This means that the system capable of parsing and running standard queries must also run machine learning algorithms.

Their work involved developing an extension to the SPARQL query engine for *Apache Jena*[2] that integrates with systems such as *Weka*[3]. A more suitable software application for enterprise use might focus solely on converting RDF graphs into a neutral data structure that can plug into arbitrary data mining algorithms.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
  ?person foaf:mbox ?email.
}
```

Listing 2.1 : Example SPARQL query for people's names and email addresses

If we consider an RDF graph, g, to be expressed as a set of triples:

$$(s, p, o) \in g$$

this query could then be expressed as function $f : G \to (S \times S)$ where G is the set of all possible RDF graphs and S is a set of all possible strings. This allows the result of the SPARQL query to be expressed as a set of all SELECT variable bindings that satisfy the WHERE clause:

$$q(g, n, e) = \exists p.(p, type, Person) \in g \ \wedge (p, name, n) \in g \wedge (p, mbox, e) \in g$$

$$g \in G \models f(g) = \{(n, e) \subseteq (S \times S) \mid q(g, n, e)\}$$

This could be generalised to express a given feature set as vector $(a_1, a_2, ..., a_n)$:

$$g \in G \models (a_1, a_2, ..., a_n) \in f(g)$$

and in the case where all $a_k \in f(g)$ are literal (e.g. string or numeric) values, we can thus consider a given SPARQL query to be specific function capable of feature extraction from any RDF graph into sets of categorical or numeric features.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?topic
WHERE {
  ?article rdf:about ?topic .
}
```

Listing 2.2 : SPARQL query to determine what

[2]https://jena.apache.org/
[3]http://www.cs.waikato.ac.nz/ml/weka/

This might allow a query that extracts a country's population, GDP, etc. provide feature extraction for learning patterns in economics, for example. However, this is limited to features derived from single-valued predicates with literal-valued ranges. It is not clear how to formulate a query that expresses whether or not a content item is about a given topic.

In the RDF model, it would be more appropriate to use a query like that in Listings 2.2 where for a given ?*article* identified by URI, we can get a list of URIs identifying concepts which the article mentions. Such a query might be expressed as function $f' : G \to \mathcal{P}(U)$ where U is set of all URIs such that:

$$g \in G \models f'(g, uri) = \{t \mid (uri, about, t) \in g\}$$

An approach of generating attributes for a given resource was proposed by Paulheim and Fürnkranz[9]. They defined specific SPARQL queries and provided case study evidence for the effectiveness of each strategy.

Their work focused on starting with relational-style data (e.g. from a relational database) and using *Linked Open Data* to identify entities within literal values in those relations and generated attributes from SPARQL queries over those entities.

For a large content-producer, there is a more general problem where many content items do not have a relational representation and the content source is a body of text or even a raw HTML page. However, the feature generation from Paulheim and F urnkranz proves to be a promising strategy given we can acquire an RDF graph model for content items in the first place.

2.3 RDF in the enterprise

2.3.1 Data Fragmentation in Organisations

2.3.2 Enterprise Integration

2.3.3 Organisational Difficulties with Enterprise Integration

2.3.4 Linked Enterprise Data

CHAPTER

THREE

SYSTEM DESIGN

In this chapter, a system is inductively derived and concretely design to make use of multiple strategies for:

1. gathering (meta)data about all of an organisations content items;
2. extracting metadata not explicitly modelled in source content management systems;
3. further enriching that metadata with information not explicitly present in the content item itself; and
4. applying machine learning to that content metadata to gain new insights about that content.

Initially, a business context is described to produce a design for a system that could be a applied within a media or content-producing organisation. This context will guide all design decisions.

3.1 Context

TODO

3.2 Use Cases

3.3 Technical Architecture

The use cases depicted and described in Section 3.2 naturally lead to an application with two distinct interfaces: notification of new content to consider for data mining and retrieval of information about content clusters created to date.

Figure 3.2 shows a high-level view of a "Content Miner" software application that provides both interfaces. A set of "notifier" applications can be created to connect different content production and management systems to the data mining system such that it is notified when new content is created. The notification need only be an *IRI* that uniquely identifies that content item.

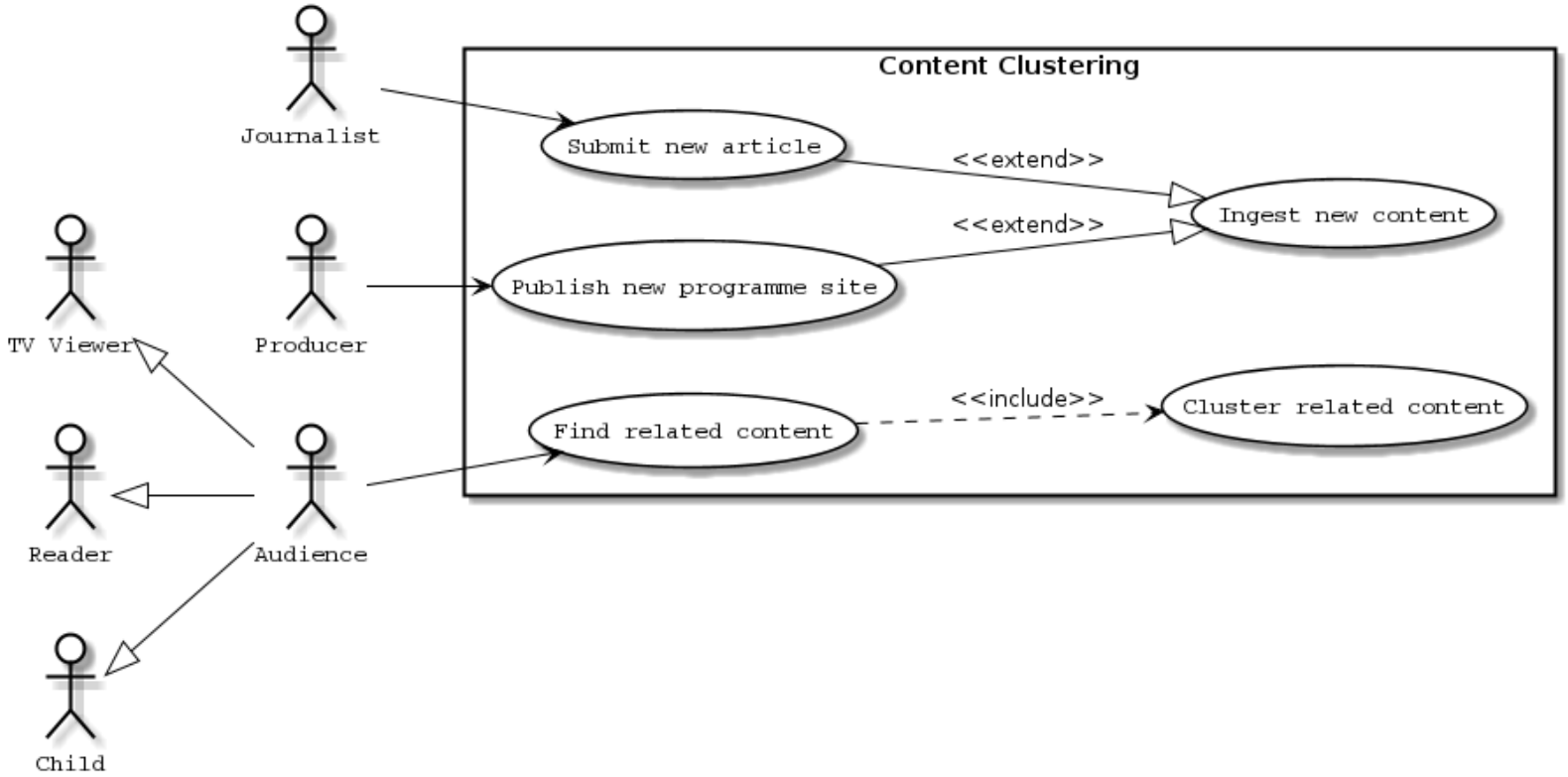

Figure 3.1: Use case diagram for content clustering system

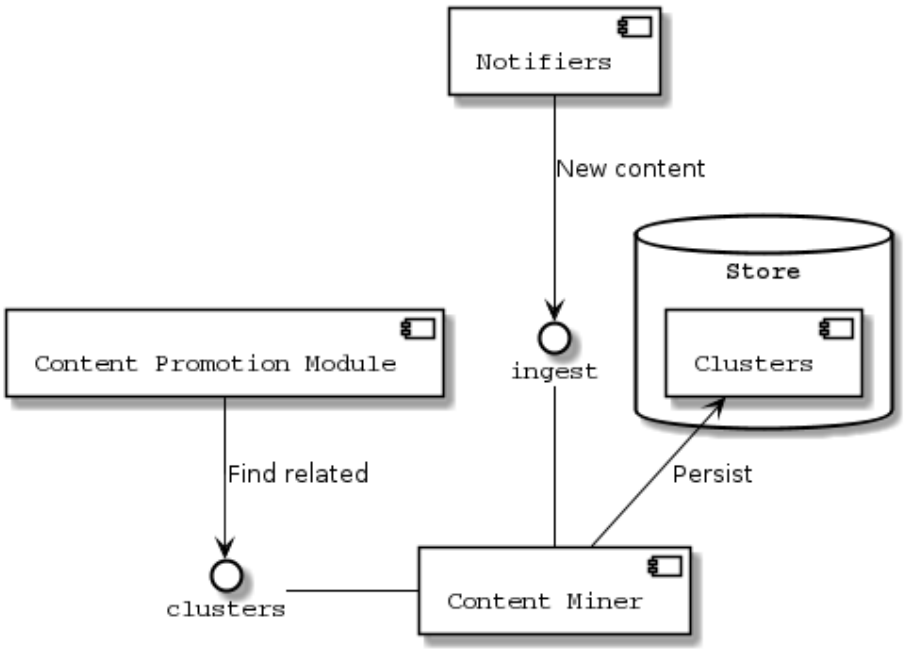

Figure 3.2: High-level component diagram with interfaces for each use case

A *Content Promotion Module* is also depicted as an example application of the use case where website visitors or audience members then use the data mining system to find content related to a page they are currently viewing. The focus of this research is to provide enough visualisation in order to evaluate the quality of content clusters generated, but it is an important design principle to consider use cases at all layers of the system.

3.4 Data Pipeline

A core subsystem in the overall system is a conceptual data pipeline whose input is a URI or IRI identifying a content item published on an organisation's website and the output is feature sets ready for applying machine learning.

In this section, a theoretical pipeline is inductively defined in steps such that an application of this pipeline would choose to implement some subset of all potential pipeline stages as appropriate for the relevant problem domain.

In Chapter 4, aa system is engineered that implements as many of these pipeline stages as possible such that a running instance of the application can configure which components to use and which not to use. Then in Chapter 6, an evalution of the system is given while it is running each component in isolation to demonstrate which of the theoretically-defined processes in this chapter appears to be most effective in generating feature sets specifically for clustering web content.

3.4.1 Definitions

This system requires some initial definition of some data structures in use:

IRI
: The input to the system is a character string conformant to the IRI syntax defined in RFC 3987[1]. This allows more generality offered by URIs[2] but is trivially made compatible with systems that use URIs through the conversion algorithm defined in section 3.2 of RFC 3987. Note that the public URL by which the public can read or otherwise consume the content is a valid identifier, but we are not restricted to that.

Feature Set
: The final output of this data pipeline is a data structure analogous to a relation or tuple per IRI fed into the system. Every IRI should have a literal value against all possible columns or fields. For binary fields, (e.g. the presence of absence of a concept tag), a more pragmatic structure might be a list of tags positively associated with the IRI rather than explicitly assigning $false$ to all tags to which the content item does not pertain. This is analogous to a spare matrix when dealing with a large number of dimensions.

Named RDF Graph
: The structure used throughout most of the data pipeline is that of an RDF graph. This is used for all the benefits outlined in Section 2.3 such as ease of transformation and combining of data sets. Named graphs are used such that all data acquired are keyed back to the IRI of the content item being processed. This also allows all graphs to be combined in a *triplestore* if needed to allow SPARQL queries across the combined data for all content items. This can be modelled as a data structure in many programming languages, but where a serialisation is used (e.g. examples shown here or to send the data between components), the JSON-LD[14] syntax will be used.

3.4.2 Identity Graph

```
{
  "@id": "http://example.com/entity/1",
  "@graph": []
}
```

Listing 3.1 : Identity graph for a content item in JSON-LD syntax

With the knowledge only of a content item's IRI, we are arguably only able to produce an empty named RDF graph. Such a graph for an example IRI `http://example.com/entity/` is illustrated in JSON-LD syntax in Listings 3.1.

The most naïve feature set we can generate from such an RDF graph is clearly a singleton relation `("http://example.com/entity/")` where a single IRI field has the value `"http://example.com/entity/"`. It is also clear that a set of one-dimension feature vectors with unique values in each is not suitable for

[1] http://tools.ietf.org/html/rfc3987
[2] http://tools.ietf.org/html/rfc3986

any form of machine learning activity. This does, however, illustrate a baseline for a working software application that is – at least in the syntactic sense – transforming IRI inputs to feature sets outputs. Such a *null* feature generator is depicted in Figure 3.3.

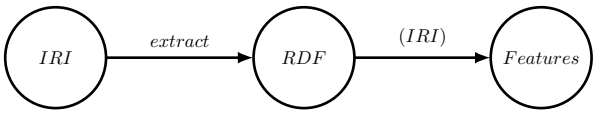

Figure 3.3: Null feature generator

Note that Figure 3.3 shows all three data structures involved despite having no functional use. We can also see top-level definitions of the process where we first *extract* semantic information in RDF from a content item indentified by IRI and then *generate* features therefrom. More useful models can now be inductively defined by adding atomic subcomponents that may each add value to the overall transformation.

There are three clear axes along which we can improve this pipeline: *extract* more RDF data knowing only an item's IRI, expand or *enrich* an existing RDF graph and then improve how we *generate* features for data mining. In the first instance, we can consider the former and add a single pipeline stage for expanding the RDF graph.

3.4.3 RDF Extraction

Tim Berners-Lee outlined four rules[1] for Linked Data, rule number three of which states "When someone looks up a URI, provide useful information, using the standards". If we assume that many pages have embedded some semantic web or RDF data, then a simple extraction strategy would be to deference the content item's IRI via an HTTP GET and pass the content to a parser capable of extracting RDFa, microformats, etc.

Many tools such as the RDFLib[3] provide functionality for taking a URL and returning an RDF graph of all data found when fetching the resource it represents, so this is arguably an ideal first choice in attempting to learn something about a content item from its IRI.

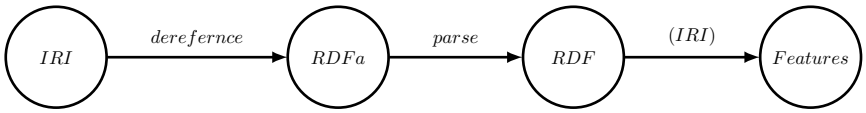

Figure 3.4: Semantic web data extraction

Figure 3.4 depicts the pipeline with a simple dereference step added. Note that the feature set generated is still the singleton relation with only the IRI value. Thus the next step should be to add a step that improves the feature set generation.

[3]https://github.com/RDFLib/rdflib

3.4.4 Feature Set Generation

Paulheim and Fürnkranz[9] described a number of SPARQL queries for generating feature sets from RDF data, which could inspire a simple query such as that shown in Listings 3.2. This query generates a boolean `true` value for any properties that match and implies `false` for those that do not.

```
SELECT ?p ?v
WHERE { ?iri ?p ?v . }
```

Listing 3.2 : Generates field `content_?p_?v` with value `true`

As also noted by Paulheim and Fürnkranz, this overlooks the *open world assumption*[12]. However, application of clustering algorithms on binary data can employ asymmetric distance metrics such as Jaccard similarity coeffcient[15], which notably avoids deriving similarity from negative values. That is, two content items lacking a particular property will contribute no information about their (dis)similarity. Thus we safely avoid inadvertently grouping together one item that genuinely lacks the property with another that indeed has the property, but we lack the positive assertion thereof in the data extracted.

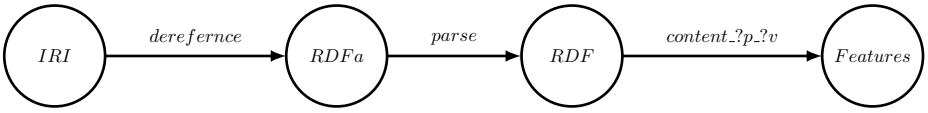

Figure 3.5: Semantic web content extraction with basic SPARQL feature generation

The basic pipeline in Figure 3.4 can thus be augmented with this basic feature extraction to produce the pipeline depicated in Figure 3.5.

3.4.5 RDF Enrichment

The third and final direction in which this data pipeline can be improved is in terms of data enrichment. A simple strategy here is to repeat the deferencing used in Section 3.4.3, but for each IRI found as the object of a triple in which the initial IRI is the subject. Formally:

$$g \in G \models \exists p, o.(IRI, p, o) \in g \rightarrow g' = deref(o)$$

That RDF graphs can be modelled as mathematical sets as in Section 2.2 means we can express a graph enriched this way as a *union* of the initial graph with each graph returned from all dereferencing:

$$g \in G \models g' = g \cup \bigcup \{deref(o) \mid (IRI, p, o) \in g\}$$

Figure **??** shows the data pipeline with this additional enrichment stage. Note that now we have potential for some stages being executed in parallel.

So far in this section, we have inductively built up a data pipeline from a "null" base working with only identity graphs to a simple pipeline capable of *extracting* an RDF graph, *enriching* it and then *generating* features from it.

What needs to be proven through experimentation is *which* of these provides the information required for effective data mining. As part of this experimentation, we can now look at further techniques and approaches to try.

3.4.6 Improving Extraction

In order to gain a larger set of RDF data at the start of the pipeline, we can derive further ways to get information about a content item given only its IRI as input.

Rizzo and Troncy[11] defined a framework called NERD capable of combining multiple entity extraction systems to provide a unified way of identifying – and disambiguating – named entities within a given body of text. With such a system, we can create a second, parallel RDF extraction strategy that creates a graph of triples in the form:

$$(IRI, rdf{:}about, Entity)$$

where *Entity* is an IRI representing a concept or entity believed to be found in the content's textual content. A data pipeline complementing the RDFa-based extraction is depicted in Figure 3.7. Note the ability to apply a simple set union to the result of each extraction as with the enrichment.

Another strategy can be to infer a relationship between two content items where one contents an HTML link to another. It is not always possible to derive precise semantics of such a link (unless the publisher has kindly provided a `rel` attribute), but a weak relationship such as:

$$(IRI_1, ex : related, IRI_2)$$

might prove – through experiementation – to be useful enough for data mining insights.

The fourth and final extraction strategy explored is acquiring metadata from bespoke Content Management Systems and other internal APIs. This is generally the only option used in enterprises settings, as discussed in Section 2.3. This is likely to be the richest source of information where an enterprise has typically preferred bespoke integrations against non-hypermedia interfaces, so experimentation should help quantity or qualify the value such a direct integration adds (perhaps to consider it in combination with the cost of bepsoke, repeated integration projects).

The assertion explored here is that such bespoke integrations can *complement* cheaper work such as extracting RDFa with pre-built tools (and thus be developed one-by-one after the initial release of an application such as this data pipeline). Note that repeated custom integration projects means that each data source requires a different application be developed (as opposed to the reuse of a single RDFa parser or HTML link scraper). This also means we are not necessarily comparing like-for-like if we introduce only one at a time. It also makes it difficult to evaluate data mining of a diverse content corpus if an integration against an API provides additional metadata for only, say, 10% of that corpus.

These challenges aside, it is clear that bespoke integrations have a clear place in this data pipeline being applied in a real enterprise setting. Now that we have a complete set of theoretical stages for *extraction*, the remaining improves lie now in the *enrichment* and *feature generation* stages.

3.4.7 Improving Enrichment

In addition to enriching through dereferencing linked entities, it is proposed to explore the following options:

- inferring relationships to hypernyms as defined by Wordnet[8];
- inferring facts based using rules derirved from expert domain knowledge;
- using RDFS and OWL to generate new triples with well-established Ontology rules.

In the first approach, we can consider a relationship rule such as:

$$(IRI, ex : related, ex : Dog)$$

and *infer* the fact:

$$(IRI, ex : related, ex : Animal)$$

and produce an enriched graph containing all additional facts inferred in this way.

When dealing with proper nouns and named entities, inferring facts based on domain knowledge may be more appropriate. For instance, the rule in n3 syntax:

```
{
  ?article ex:takesPlaceIn ?city .
  ?city a ex:City .
  ?city ex:capitalOf ?country .
} -> { ?iri ex:takesPlaceIn ?country }
```

might be useful ot help cluster articles that take place in the same country – even if the countries are not always explicitly mentioned therein. Such an inference requires knowledge about cities and countries to write and domain experts for different types of content might be able to offer more nuanced rules.

An example for BBC content might be to infer that all articles written under the *Newsround* brand is suitable for children or that programmes that have broadcast times during the day are also suitable for children.

The third and final proposed improvement makes use of standard tools to find *closures* using, e.g. RDFS, ontology rules. With this approach, we can infer that entities that have a given type or class also have their superclasses and supertypes. This gives us similar inference to hypernyms, but with knowledge present in well-established ontologies.

An obvious example might where two content items have been identified as related to the same concept – so they would be candidates for clustering together – but when RDF data are extracted, it is found that two different URIs have been used for each:

$$(IRI_1, <http://dbpedia.org/property/related>, ex1 : entity)(IRI_2, <http://dbpedia.org/property/related>,$$

In the RDF graph for IRI_2, say, we might find the source had provided an `owl:sameAs` assertion such as:

$$(ex2 : anotherEntity, owl : sameAs, ex1 : entity)$$

This is possible in the case where the second item's data source uses its own set of identifiers for entities, but has chosen to provide equivalences to a more standard set of identifiers (e.g. DBpedia). With this information, our data pipeline can infer:

$$(IRI_1, <http://dbpedia.org/property/related>, ex1 : entity)(IRI_2, <http://dbpedia.org/property/related>,$$

and the feature generation stage might provide the common features `dbprop_related_ex1_entity=true` for both content items.

3.4.8 Improving Feature Generation

The feature generation outlined so far relies solely on boolean values indicating whether or not a given content item is related by some property to some entity. This recreates the concept of *tagging* where a given object is either associated or not associated with a series of *tags*.

One of the advantages of the RDF graph model is that we are not constrained necessarily to properties and attributes directly applicable to the entity. We could imagine adding a level of indirection to the query in Listings **??** to create the query in Listings **??**.

```
SELECT ?p1 ?p2 ?v
WHERE {
  ?iri ?p1 ?o .
  ?o ?p2 ?v .
}
```

Listing 3.3 : Generates field `content_?p1_?p2_?v` with value `true`

With the this query, features of the form `content_?p1_?p2_?v` can be generated. An example of this might be where a television programme content item has information about the actors that appeared therein, e.g. `ex:hasActor`, and furthermore we have information about those actors such as where they were born, e.g. `ex:bornIn`. With the path created by following both of these predicates, it is possible to create features for a television programme such as `content_ex:hasActor_ex:bornIn_Edinburgh` and we can potentially find similarity between programmes where the actors were born in the same city.

Perhaps a third step in the predicate path followed can give us even more useful features. The example above could be expanded to `content_ex:hasActor_ex:bornIn_ex:cityIn_Scotlan` to allow the more general ability to cluster programmes with Scottish actors, for instance.

Appropriate experimentation should show whether more value is gained by adding these additional levels of indirection.

3.4.9 Maximal Data Pipeline

In this section, a data pipeline was inductively built up from a base, identify pipeline with suggestions for potential improvements in different stages. An

application of all the ideas discussed so far might look like that depicated in Figure 3.8.

The goal of this work is to evaluate how the quality of unsupervised clustering changes with respect to enabling different combinations of the maximal data pipeline. In the next section, some of the technical architecture of this pipeline is outlined and the next chapter will cover how the overall system is then implemented.

3.5 Pipeline Architecture

TODO

3.6 Architecting for Experimentation

TODO

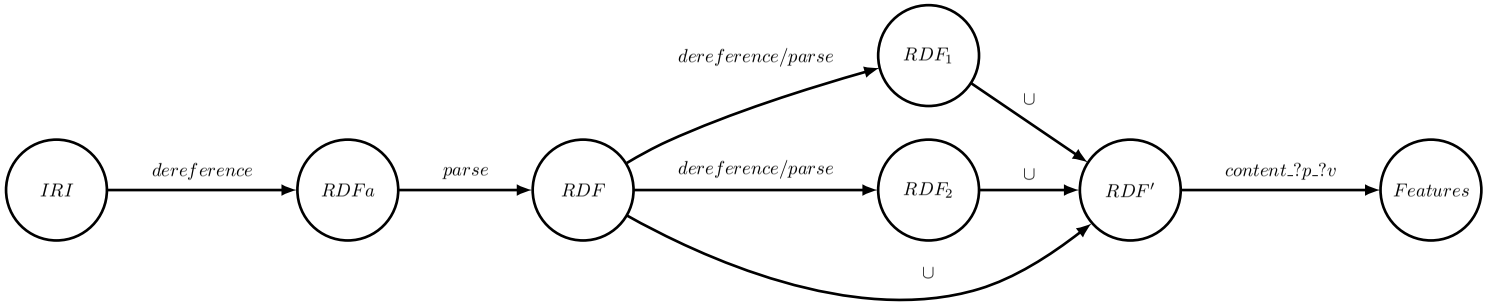

Figure 3.6: Semantic web content miner with addtional dereferencing of linked entities

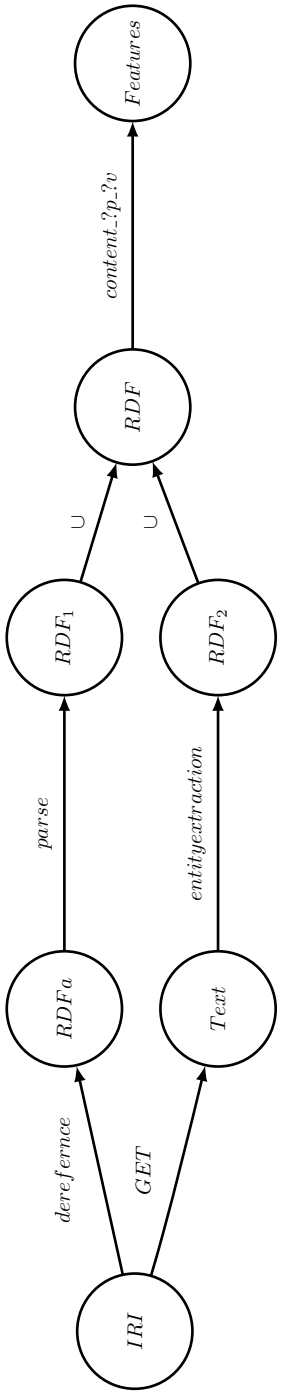

Figure 3.7: Named entity extraction in addition to semantic web extraction

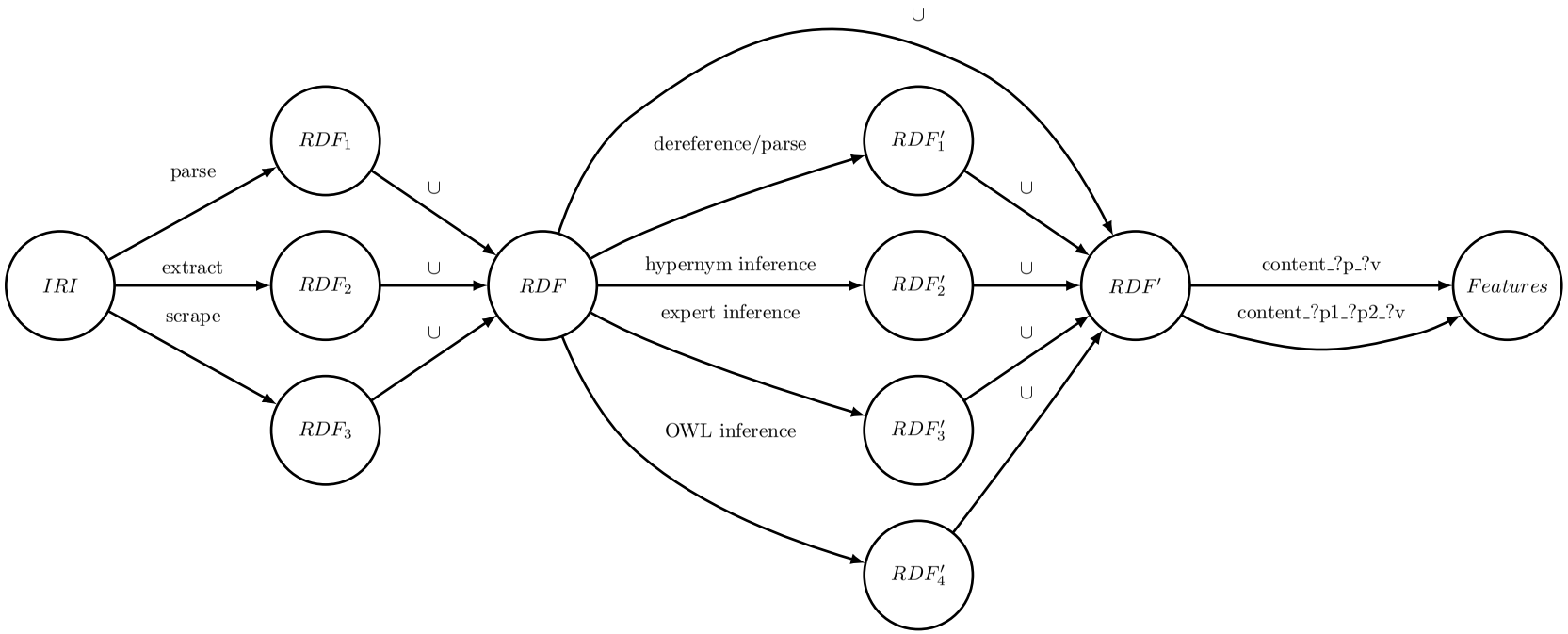

Figure 3.8: Maximal Data Pipeline

CHAPTER

FOUR

IMPLEMENTATION

In this chapter, some of the details of the system implementation are described. Initially, we discuss some of the implementation strategy employed due the experimental nature of the system and the desire to run multiple, competing approaches side-by-side in an efficient way.

The remaining sections walk through the implementation of each part of the data pipeline from upstream to feature sets and the chapter concludes with implementation of the clustering and generation of clusters for evaluation.

4.1 Software Architecture

All functionality was implemented as part of a single Python module named *distillery* that is run via the Unix command line, with different subcommands for each feature. Each command was designed around the Unix Philosophy[10] of doing one job per command and that they were composable via Unix pipes. This allows pipelines to be composed:

```
distillery extract <iris.txt | distillery enrich | distillery generate
```

that resemble the entire theoretical data pipeline described in section 3.4 without the need for middleware such as message queues. More typical enterprise architectures around message oriented middleware or event-driven architecture might need to be employed to scale this system to millions of documents, but the simple approach above is sufficient for tens of thousands of documents.

Even this simple approach would scale very well with suitably low network latency. The largest bottleneck observed was the necessity to do at least one HTTP request per item ingested and then enrich via an HTTP request per object of a triple where the IRI is a subject. For some content items, this could easily be over 100 HTTP requests at which point parallel CPU cores or asynchronous I/O programming does not overcome the demands on network performance.

Throughput is a particular challenge when data processing systems are first launched. The day-to-day volume of item creations and updates may be low enough for even a low-scale application, but the performance demands increase significantly when we wish to bootstrap a backlog of all content items historically

published – of which could easily be tens of millions for organisations such as the BBC.

For this, there is an appeal to designing such a system on a cloud computing platform such that it can be scaled up for the initial import of existing data and then back down for ongoing updates when that import completes. Such design is out of scope for this paper, but some discussion of limitations of the existing system in section **??** highlights where this system may be architected differently to be more suitable for such a cloud-based deployment.

The higher-level compositional architecture of the individual data processing stages can, of course, be altered independently of the design of those stages. The following sections will describe in greater detail those composable modules.

4.2 Avoiding Redundancy Across Multiple Experiments

The design of the data pipeline in chapter 3 would suggest an implementation with each stage implemented as a function converting a graph into a new, richer graph.

That is, whilst the different strategies illustrated in the maximal data pipline in Figure 3.8 can be run in parallel, the extraction stage itself simply converts the identity graph described in section 3.4.2 to a single graph (after having taken the union of all the strategies).

In the experiment that is the subject of this paper, the intent was to compare each of the extraction strategies individually, but also all possible (union) combinations thereof. This presents at least two potential implementation strategies:

1. Build the whole data pipeline and machine learning system such that it is configurable which strategies are invoked and run an instance per configurable combination; or

2. implement the software to produce all possible outcomes along the pipeline.

The former approach is stronger if the aim is to build the system closer to how it would be implemented in an enterprise: only one set of clusters is needed and even a production, enterprise system would be configurable if maintainers wished to enable or disable features as desired.

The latter strategy is arguably better for experimentation as every IRI fed into the source end of the pipeline is processed at the same time for every extraction and enrichment approach. This gives better assurance that the same input is used to evaluate each competing system.

Other operational reasons include being able to run the system on a single machine if necessary and that there is no repeated work: extraction by one means can be used in isolation, but also feed into a union with another technique without having to recalculate that graph. That is, if we have graphs A and B generated once each, then we can output A, B and $A \cup B$ for comparison. A parallel copy of each system would be calculating A and B twice each.

The limitation here is a production-ready implementation of the system would have be built again from the ground-up, perhaps reusing functions and library code from the experimental version, as the application itself will be

structured around this idea of multiple outputs for each input. This is likely to be the case for any experimental system and is arguably in line with Brooks' classic assertion to "Plan to Throw One Away"[3].

```
def all_extracted_graphs(iri):

    extraction_strategies = [
        dereference, extract_entities, find_links
    ]

    # This creates all 3 RDF Graphs only once
    graphs = [s(iri) for s in extraction_strategies]

    # Empty set removed from powerset
    for idx, gs in enumerate(powerset(graphs) - {{},}):
        # union defined elsewhere: list(graph) -> graph
        graph = reduce(union, gs)
        # Loop counter idx tells us from which technique
        # combination each graph comes
        yield idx, graph
```

Listing 4.1 : Python function that generates all possible RDF Graphs

An illustrative Python function is shown in Listings 4.1 where each of the three extraction techniques is invoked only once and a powerset function is used to generate all possible unions of those graphs and thus yield all possible usages or non-usages of each strategy to generate RDF graphs. The details of the three extraction functions are covered in section **??**.

4.3 Obtaining IRIs

It is a substantial undertaking to design, build and test a production-quality enterprise system that processing all content produced by a media organisation. An experimental system needs to focus on quick results and therefore is optimised to work only over a small sample of that content.

As introduced in section **??**, all that content is also likely to be distributed across multiple databases, content management systems and other stores. Whilst it is hypothesised in this research that semantics and linked data provide a good way to extract data about the total breadth of all content without any bespoke integration against internal systems, there still remains the enterprise integration problem of *discovery* of those content items.

Essentially, the data pipeline outlined in section 3.4 notably requires that known IRIs of content items are fed into it, but makes no statement about how those IRIs are acquired. This is a deliberate design decision to decouple the concepts of discovery from this extraction/enrichment workflow. Thus item may easily be re-ingested several times if it is updated (or simply periodically) and a modular approach like this defends against upstream systems changing and having to rebuild the triggers that notify when content is created or updated.

What follows it that the enterprise integration task has been reduced (we do not need to write bespoke code to extract basic data about every content item)

but not wholly eliminated (some custom adaptors need to be created to notify the pipeline of creation or update events in each respective data store).

However creation of any such notification adaptors is out of scope for this research altogether, so a heuristic approach was employed where ten different, themed content aggregation pages on the BBC website[1] were polled for any new content promoted by editorial teams. This provided a way to find new items shortly after they are published and also ensured the data sample ultimately used focused on content that was deemed noteworthy or interesting in the last few months.

This is not an effective method for obtaining a large number of items quickly as it is entirely dependent on the rate these aggregate pages are updated. Running slowly over a long period yielded approximately ten thousand individual IRIs with little effort nor need for large amounts of integration work.

4.4 Extracting RDF Graphs

Listings 4.1 hinted at three individual functions capable of obtaining an RDF graph for a given IRI, all based on the extraction strategies outlined in section 3.4.3.

4.4.1 Dereferencing

Listings 4.2 shows how trivial a dereference function can be if we use the comprehensive RDFLib library for Python. The library can handle content type negotiation[6] to ensure it can happily dereference and parse any response that contains semantics, particularly RDFa (including embedded Turtle) and microdata within an HTML page.

```
from rdflib import Graph

def dereference(iri):
    g = Graph(identifier=iri)
    g.parse(iri)
    return g
```

Listing 4.2 : Python function that generates all possible RDF Graphs

4.4.2 Entity Extraction

DBPedia Spotlight[4] was chosen to perform the entity extraction. A hosted version is available as a free service and it has been shown to be effective in many cases. A true comparative study of the merits of alternative entity extractors is out of scope for this paper.

```
import requests
from rdflib import Graph, URIRef, Namespace
```

[1]Including, among others, the BBC Homepage (http://www.bbc.co.uk), BBC Arts (http://www.bbc.co.uk/arts) and BBC Science (http://www.bbc.co.uk/science)

```
FOAF = Namespace('http://xmlns.com/foaf/0.1/')

def extract_entities(iri):
    g = Graph(identifier=iri)

    text = requests.get(iri).text

    spotlight_response = requests.post(
        'http://spotlight.sztaki.hu:2222/rest/annotate',
        data={
          'text': text, 'confidence': 0.8, 'support': 20
        },
        headers={'Accept': 'application/json'},
        timeout=600)

    if spotlight_response.ok:
        r_json = spotlight_response.json()
        if 'Resources' in r_json:
            annotations = {
                resource['@URI']
                for resource in r_json['Resources']
                }
            if annotations:
                for entity in annotations:
                    g.add((
                        URIRef(iri),
                        FOAF.topic,
                        URIRef(entity)))
    return g
```

Listing 4.3 : Python function that uses DBPedia Spotlight to extract entities from web pages

Listings 4.3 shows a basic implementation of a function that fetches a page and then feeds the content into a request to DBPedia Spotlight. All entities thus found are then fed in as objects to a series of `foaf:topic` triples.

There is much that can be tweaked in this implementation and the final version used in the experiment relied on a ad hoc whitelist of XPATH queries known to narrow down the content to the main article content on BBC pages. This is because the implementation shown in listings 4.3 does nothing to prevent entities being extracted from page chrome such as navigation links and other cross-site concerns.

In a given organisation, such an approach might well be sufficient to extract main article content from a controlled set of pages. In a production system, it may be more appropriate to consider readability systems such as that from Arc90.

4.4.3 Hyperlink Relationships

In listings 4.3, we see a simple Python function that uses the Beautiful Soup library[2] to search across all links in a document and infer a "related" property on the assumption that pages like to other pages that are in some way related.

```
import requests
from rdflib import Graph, URIRef, Namespace

DBPROP = Namespace('http://dbpedia.org/property/')

def find_links(iri):
    g = Graph(identifier=iri)

    text = requests.get(iri).text
    doc = BeautifulSoup(text)

    # For all <a> tags that have an href attribute...
    for a_tag in doc.find_all('a', attrs={
        'href': re.compile('.+')
    }):
        # ... handle href values being relative links ...
        other_iri = urljoin(iri, a_tag.attrs['href']).strip()

        # ... infer triple if IRI is to another BBC page
        if bbc_url.match(url):
            g.add((
                URIRef(iri),
                URIRef(DBPROP.related),
                URIRef(other_iri),
            ))

    return g
```

Listing 4.4 : Python function generates triples based on links to other pages

This falls foul of similar issues to entity extraction as described in section 4.4.2 in that site-wide navigation links and other supporting page elements may link to very general pages (e.g. a persistent link to "home" on every page).

With appropriate feature selection, this may have no impact, but for this experiment a simple heuristic was employed to narrow down to the "interesting" part of BBC pages.

Another feature in listings 4.4 is a restriction to consider only links to other BBC pages. There may be a positive or negative effect in addtionally noting external pages linked to from content pages, but that is left to future research to consider.

[2]https://www.crummy.com/software/BeautifulSoup/

4.5 Enriching and not Enriching

The combinatorial consequence of the usages and non-usages of the three extraction techniques described so far is that the experimental system developed is already comparing seven (2^n less the case where no extraction is employed).

In order to maintain focus in the experiment, only one of the enrichment techniques described in chapter **??** was implemented and its usage and non-usage merely doubled the number of systems for comparison to fourteen.

This proved sufficient to give indicative results as to whether there is value added through any enrichment, but future research could certainly compare the merits of different enrichment approaches.

```
import rfc3987

def get_objects(graph):
    return {str(row.o) for row in graph.query('''
            SELECT DISTINCT ?o
            WHERE {
                ?iri ?p ?o .
            }
            ''', initBindings={'iri': graph.identifier})}

def enrich(graph):
    new_graph = copy(graph)

    for o in get_objects(graph):
        if rfc3987.match(o, rule='IRI'):
            new_graph.parse(iri2uri(o))

    return new_graph
```

Listing 4.5 : Python function that enriches a graph via deferencing objects

Listings 4.5 illustrates a simple function to deference all objects of triples where the current IRI is a subject. Note the convention to set the graph's own identifier to that of the content item of interest.

Using SPARQL in this way opens up possibilities to perform multiple queries to choose IRIs to dereference. For example, we may wish to include semantics for objects reachable by following two predicates on the RDF graph and consider much more indirect data. Evaluating the utility of this is left for future work.

4.6 Feature Set Generation

The feature generation strategy employed was dervied from a distilled portion of the approach introduced by Paulheim and Fürnkranz[9].

Listings 4.6 shows an illustrative, recursive function that "walks" the RDF graph starting with the content item's IRI as the initial subject.

```
from rdflib Literal, URIRef

def generate(g, subject=None, depth=3, features=None, prefix=''):
```

```
    if depth <= 0:
        return
    if not features:
        features = {}
    if not subject:
        subject = g.identifier
    for p, o in g.predicate_objects(subject=subject):
        if type(o) == URIRef:
            new_prefix = prefix + clean(p) + '_'
            features[new_prefix + clean(o)] = True
            generate(g, depth=(depth - 1), features=features, prefix=new_prefix
        elif type(o) == Literal:
            features[clean(p)] = str(o)

    return features
```

Listing 4.6 : Python function that generates feature sets from RDF graphs

The `clean` function simply provides some cosmetic cleaning up of predicate IRIs to prefer CURIEs as they are more compact. A typical example feature might then be `foaf:topic_dbpedia:Albert_Einstein: true` where the leaf object is an entity or `ogp:title: "Gandhi: Reckless teenager to father of India"` if the leaf object is a literal.

4.7 Feature Selection

A simple selection heuristic was employed, again inspired by the work of Paulheim and Fürnkranz[9], to go some way to reduce potential noise in the subsequent machine learning phase.

In this experiment, all features that have the same value or entirely unique values were removed and values that occurred twice or more for a given feature had to make up over 5% of possible values for that feature. That is, if a feature had over 95% of its values being unique or all the same, then it was rejected.

4.8 Clustering Implementation

Hierarchical, agglomerative clustering was implemented by implementation of a `cluster` function that was merge the two "nearest" clusters on each invocation. This assumes that all content items not yet in a cluster are implicitly in a singleton cluster and the ability to merge one at a time allowed for control of when to cease merging.

The closeness of two potential, candidate clusters was calculated via a max linkage on the basis that it ensures that any two content items within a cluster to be considered related. This avoids the so-called *chaining phenomenum* where single linkage may create clusters where two unrelated items are members because they are each related to an item of chain of related items therebetween.[5]

The distance between content items was defined by the Jaccard distance due to its suitability for binary and categorical data.

4.9 Results Generation Strategy

With all of the components described so far in place, an experiment was run processing nearly 10,000 content items from the BBC website with each going into one of fourteen combination of extraction and enrichment approaches.

For each approach, hiearchical clustering was applied repeatedly until the next merge would produce a cluster with cohesion below 0.5. This is reasonably arbitrary as each technique will have different concepts of cohesion, but it provided a strong basis on which to evaluate each approach on whether its cohesion meaningfully predicts whether a human would agree that cluster contains related items.

CHAPTER

FIVE

RESULTS AND ANALYSIS

5.1 Comparison of Clusters Produced

5.2 Strengths and Weaknesses of Different Approaches

5.2.1 Embedded Semantics Extraction

5.2.2 Entity Extraction

5.2.3 Hyperlink Relationships

5.2.4 Enrichment by Dereference

5.3 Recommendations for Production Use

CHAPTER

SIX

EVALUATION

6.1 Qualitative Survey

6.1.1 Survey Design

6.1.2 Survey Results

6.1.3 Further Observations and Discussion

6.2 Design Limitations

6.3 Implementation Difficulties

CHAPTER

SEVEN

CONCLUSIONS AND FUTURE WORK

7.1 Challenges and Opportunitues for Semantics in the Enterprise

7.2 Suitability for Semantics and Data Mining in Media Organisations

7.3 Suggestions for Initial Implementations in the Enterprise

7.4 Future Research

7.4.1 Better Enrichment

7.4.2 Patterns for Reducing Data Noise

7.4.3 Deeper Study of Machine Learning on Semantics

BIBLIOGRAPHY

[1] Tim Berners-Lee. Linked data-design issues (2006). *URL http://www. w3. org/DesignIssues/LinkedData. html*, 2011.

[2] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[3] Frederick P Brooks Jr. The mythical man-month (anniversary ed.). 1995.

[4] Joachim Daiber, Max Jakob, Chris Hokamp, and Pablo N. Mendes. Improving efficiency and accuracy in multilingual entity extraction. In *Proceedings of the 9th International Conference on Semantic Systems (I-Semantics)*, 2013.

[5] Brian S. Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. *Hierarchical Clustering*, pages 71–110. John Wiley & Sons, Ltd, 2011.

[6] Roy Fielding and Julian Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. 2014.

[7] Christoph Kiefer, Abraham Bernstein, and André Locher. *Adding data mining support to SPARQL via statistical relational learning methods*. Springer, 2008.

[8] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[9] Heiko Paulheim and Johannes Fümkranz. Unsupervised generation of data mining features from linked open data. In *Proceedings of the 2nd international conference on web intelligence, mining and semantics*, page 31. ACM, 2012.

[10] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[11] Giuseppe Rizzo and Raphaël Troncy. Nerd: a framework for unifying named entity recognition and disambiguation extraction tools. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 73–76. Association for Computational Linguistics, 2012.

[12] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach, 2010.*

[13] Toby Segaran, Colin Evans, and Jamie Taylor. *Programming the semantic web.* " O'Reilly Media, Inc.", 2009.

[14] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. Json-ld 1.0. *W3C Recommendation (January 16, 2014)*, 2014.

[15] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2005.