Categorisation Methods for SoLiD Control Panel

Candidate no. 1073375

Word count: 4,985

Part B - MCompSci Computer Science

Trinity 2025

Acknowledgements

I would like to thank my parents for supporting me through everything, I could never have done any of this without your love and support.

I would also like to thank my supervisor Dr Jun Zhao for her incredible patience and support through this project, it took us a while to get here but we've done it!!

Thank you to Ned Stevenson and in turn John McManigle for their wonderful LaTeXthesis templates, this greatly simplified porting my project into Latex and saved me many headaches!

Finally, thank you to Ben Johnson for being the best friend I could ever have asked for.

Abstract

SoLiD is a protocol designed to give users greater data autonomy through its "pod" paradigm of data storage. While it achieves this on a protocol level, the prominent user interfaces for browsing a SoLiD pod all utilise a directory structure. This is unintuitive for typical social media users, harming the practical autonomy SoLiD provides them.

In this project, I explored an alternative method of presenting pod data to the user. I used existing semantic analysis and classification techniques to create a user-centric categorisation system. I tested two classification techniques, Decision Trees and Case-Based Reasoning, concluding that both techniques successfully categorised data according to these user-defined labels.

Contents

1 Introduction

In the current landscape of social media software available to the public, the main paradigm of data ownership is that each social media company owns and curates its own copy of each user's data. This means users have minimal autonomy over their data, and have difficulty updating or deleting it.

The SoLiD protocol provides an alternative where each user owns and curates their own SoLiD "pod" containing the canonical copy of their data, including their social media feeds. Social media companies would request access to this pod, running their services using this data, which would significantly improve user data autonomy. However, this depends on tools and software allowing users to find and alter the data in their pod as they wish.

The directory system built into SoLiD, while intuitive for developers, is not an accessible way to present data to the user. A method of categorising data to make it easily findable is proposed to enable the desired autonomy.

In this project I will explore a framework through which data in the SoLiD pod can be categorised appropriately. This framework begins with a semantic analysis phase, where the pod's underlying semantics are mined for useful information about the data items. This information is then passed onto a classifier, which determines a label. In this project I will investigate two classifiers:

- Decision Trees, an eager classifier which requires a training dataset to "learn" how to classify data appropriately.

- Case-Based Reasoning, a lazy classifier (requiring no training phase) that utilises a set of labels and similarity metrics, deciding on a label for each unlabelled item.

Both classifiers will be tested using a reasonably sized dataset, containing a variety of data types, to form the basis of our comparison. These algorithms

have been chosen because of their simplicity and low data requirements; SoLiD pods will not contain enough data or have sufficient processing power to train more complex models.

The overall aim of this project is to utilise these existing techniques of semantic annotation and classification within the limited context of a SoLiD pod to create a lightweight, user-centric classification system.

2 Background

2.1 SoLiD

The SoLiD (Social Linked Data) Protocol separates data and applications within the context of web applications [1]. It introduces a "pod" design paradigm, where instead of each application storing user data in a closed "silo", neither owned nor controlled by the user, their data is stored in a "pod" controlled by that user. Applications then interface with a user's pod to retrieve the necessary information.

The SoLiD protocol also includes a permissions framework that allows a granular level of control over the access rights of each item in the pod. This provides increased autonomy over the user's data, allowing users to sever an application's connection to an item in the pod at will, rather than appealing to the entity in charge of the application to remove their data and trusting that this has been done.

Another benefit of this paradigm is that it empowers the user to update or delete their data as they wish. Previously, users wanting to alter their data would be reliant on applications providing this option. Users would also have to alter this data for each application using it. This is already an arduous task, made more difficult by the presence of copies of the data made without the user's knowledge. SoLiD makes this significantly simpler, providing only one canonical source of data (the user's SoLiD pod), over which the user has complete control. This makes updating or deleting a user's data easier, providing them with more autonomy.

2.2 Knowledge Graph

A knowledge graph is a graph data structure augmented with extra semantics [2]. In it, edges are given a Uniform Resource Identifier (URI) that denotes the relation's semantics, and nodes represent objects with their own URIs. The primary format for storing knowledge graph data is the Resource Description Framework (RDF), which stores graph data as triples containing two nodes and an edge. The most

common syntax for RDF is Turtle (`.ttl` files); SoLiD uses these in its description of its underlying knowledge graphs, allowing rich semantic connections between files.

2.3 The Problem with SoLiD's Directory Structure

SoLiD gives the user more autonomy over their data in numerous ways, including its intricate sharing permissions system. However, to utilise these features, users must first locate the items in their pod that they want to alter. Currently, SoLiD organises users' data in a file structure "directory" format. For a user to find the file they want to modify, they must know how the developer decided to store it within the directory structure. This might allow a small subset of "power users" to utilise the autonomy SoLiD gives, but, for most users, searching through a directory of system files would not be a viable solution.

Furthermore, if multiple applications were used to achieve the same goal, each application might store data in different locations [3], but the user might consider all of that data to be part of the same service, and expect to find it in the same place.

2.4 The Semantic Categorisation Problem

A directory representation of files in a SoLiD pod is not an ideal way to mediate access between it and the user. Instead, I propose that data should be presented according to its utility to the user. Ideally, the relevant files and items within a SoLiD pod would be displayed according to the user's mental model of what is important, and which classification each item should have. However, files containing metadata wouldn't be considered important by the user, although the metadata itself might be. This necessitates the annotation of more "important" files with the semantics found in this metadata or elsewhere, grouping relevant information together.

Furthermore, the categories for storage should align with the user's mental model of their SoLiD pod, as there is no predefined standard for these categories.

For example, if a user employs an image-based social media service, they would expect all their posts from that social media service to be in the same "place", i.e. the same category. Going further, the user may have tried out multiple different image-based social media services, all of which were interoperable with each other despite storing posts in different directories. This user might still expect all these posts to be in the same place, but different users may consider them to be separate. Complicating things further, a user might also use a photo album-style service that simply stores images without wanting them to be grouped with their social media image posts, meaning we cannot categorise solely by file type.

All these factors result in the need for a categorisation system that adapts to the user's preferred labelling of data, utilises all available data relating to each item, and annotates each item with the appropriate data.

There is no single correct and obvious solution to these requirements. We can, however, deduce some properties we want the categorisation method to have:

- Minimal user input – users should not have to label all the data or determine the categorisation rules themselves.
- Adaptable – the solution should be easily adaptable to new categories, forms of data, or locations within the knowledge graph or directory structure.
- Robust – the solution should be able to manage uncharacteristically formatted data gracefully, in a way that still categorises as accurately as possible.
- Revisable – users should be able to assign new labels to items and have the categorisation adapt accordingly, meaning if the user disagrees with the solution's categorisation, their preferences are still considered.

3 Semantic Analysis

There are several ways to achieve the overall goal, and, for this project, I have focused on a post-processing approach, augmenting data in a user's existing SoLiD pod with semantics. This part of the program is built in React.js – most libraries available for SoLiD development are JavaScript-based, and I wanted to utilise a framework to make development easier.

Thus, the overall process involves:

A) Initial Setup and Context

We must first connect to the SoLiD pod and authenticate this connection with the user's login so we can see the full context of the SoLiD pod. **Snippet A.1** in **Appendix A** displays the authentication flow for SoLiD in the frontend, built using the solid-ui-react library from Inrupt. We then need to find the root of the directory system within this SoLiD pod - **Snippet A.2** shows the program querying the SoLiD user's WebID to find the root of their storage space, using the Comunica framework to run the required SPARQL queries.

B) Finding Items to Categorise

Next, a search of the knowledge graph for items that need categorisation is required. I used the directory structure of the pod as the basis for the search. Developers will have built their applications with this structure in mind, making this information useful in both our search and classification.

We therefore perform a breadth-first search (BFS) on the directories of the SoLiD pod to find the relevant items. These are either files that contain RDF triples, or documents like images or text files. **Snippet A.3** shows the function that queries a container in the SoLiD pod and return all the items "inside" it, updating the BFS frontier appropriately. This is used by our BFS to find all the items in the pod.

C) Grouping Item Types

Next, we must input the labels used as our classifiers' targets. This system has a JSON file in the system files containing all our labels alongside the corresponding item's URI, and a command line tool that asks for labels for items. In future, it would be beneficial to have a proper UI. This labelled data will serve as the "training data" used by our classifiers to decide the remaining items' labels.

D) Augmenting Items with Semantic Metadata

Now begins the core semantic analysis:

- The BFS returns duplicate items across types, so these items must be grouped into a single object with a list of all their types. (**Snippet A.4**)

- Each item is annotated with its BFS search path, effectively annotating it with its associated directory information. This is important as the directory information is a useful heuristic for assessing relationships between items, as the developers of a social media application may place similar documents in the same directory.

- Each content item (e.g. images or text files) is augmented with the data from its corresponding RDF `.ttl` file.

- Each `.ttl` file is assumed to describe RDF triples related to a single item, like an image, post, or document.

- The function `augmentPosts` (**Snippet A.5**) then handles the specifics of reading the `.ttl` file, using the information to annotate its target items.

- This enriches each item with structured, semantic context.

E) Handling File Format Variability

When `augmentPosts` reads turtle files to annotate the items with extra semantics, there is an inherent level of fragility - `augmentPosts` needs to know how to interpret the turtle files, which involves making assumptions about their structure. To mitigate this in future, the function should be built in a more modular manner.

In our system, `augmentPosts` checks to see if the RDF file has relations of specific types, such as `contentURL`, and annotates the appropriate items with the file's information. If no such relations exist, the turtle file itself is the item annotated with this information.

F) Final Step: Classification

Once the program has created an object for each item containing its URI and semantic annotations, the program then interacts with a FastAPI API, sending them to the classifier. The classification methods have been built in Python, taking advantage of its many ML-related libraries.

In the next chapters, I will discuss the two classification methods I have explored, both of which use these annotated objects.

4 Decision Trees

The first classification method I used was the Decision Trees (DT) method, an eager machine learning classification algorithm. It is primarily used for classification where the attributes for the items to be classified are discrete, meaning it is well suited to the discrete nature of our knowledge graph. I used an off-the-shelf solution for this implementation, in this case the decision trees implementation in the `scikit-learn` python library.

Decision Trees works in two phases – training and classification. Both phases require discrete inputs (`scikit` requires boolean or integer inputs [4]) common among all items. In the training phase, the program receives these inputs alongside a label for each item. It then creates a "decision tree" for those inputs, generating a split in the dataset (decided by one input) where each half can then be split further. The decision with the smallest entropy or largest information gain is often chosen [5], as both are metrics for the resultant split's optimality. In the classification phase, the decision tree makes the appropriate decisions based on the inputs and returns the resultant label.

This implementation requires the items being classified to be inputted and encoded in a specific way before training the decision tree. This introduces assumptions about the format of the input data, which makes the system more fragile while also losing information about the items.

These encodings work as follows:

Type Encodings

`Scikit` can only take in boolean or integer inputs for the model, meaning some preprocessing is required to convert the type annotations into boolean inputs. The program creates a list of inputs, converting each type into a boolean flag. DT can then determine whether a certain type is present by checking these flags.

Snippet B.1 collates all the types present in the training dataset, and then **Snippet B.2** converts the inputs into the required format.

Directory Encodings

It is necessary to give DT each item's directory information to help with classification, as the synthetic dataset otherwise lacks information. Fortunately, the BFS stored each item's search path, which is functionally a list of the item's directories. I created an encoding for this directory structure, similar to the one for types, in **Snippet B.2**. **Snippet B.3** collates all the directories appropriately.

Related Type Heuristic

Late into the testing of this system, a heuristic was devised to allow DT to utilise the relations between items of different types. DT struggled to utilise raw graph data as it lacked enough training data; providing a heuristic of items being related to specific types (**Snippet B.4**) was hypothesised to improve DT's accuracy. Additional relations between items would render this heuristic useless.

5 Case-Based Reasoning

The second classification method I used was Case-Based Reasoning (CBR), a lazy supervised machine learning algorithm. This method works according to four phases [6] - Retrieve, Reuse, Revise, and Retain, and involves using a bank of "cases" that have already been identified and given a designated outcome.

- Retrieve involves gathering "cases" resembling the one we would like to solve.
- Reuse involves reusing the best cases to generate a solution.
- Revise involves reviewing a given outcome to ensure it is correct.
- Retain involves keeping the newly solved case in our case bank, allowing us to process future cases optimally.

In future, Revise and Retain should be a part of whichever user interface implements this categorisation system, with the Revise phase being implemented as user-designated labelling, and the Retain phase storing the newly identified labels in with all the other labels.

The Retrieve and Reuse phases complete most of the categorisation work. It is here the most direct comparison with our other categorisation method can be made. In this part of the system, we want to retrieve the items in the SoLiD pod most similar to the item that requires categorisation and then use them to decide which category should be chosen. This is a useful method for classifying data in a SoLiD pod as it has no training phase and decides on labels by finding comparable items in the labelled dataset.

5.1 Retrieve

The input into the Case-Based Reasoner is split into two: the set of all items found by the semantic analyser (complete with all the annotations that the semantic

analyser provides), and a set of labels for some items. For a given unlabelled item, we then evaluate its similarity to all the labelled items, choosing several of the most similar items from which we would like to derive a classification.

When evaluating similarity, the system has been designed as a linear combination of different similarity metrics – different ways of checking if an item is similar to the designated one. The design is modular to allow for the inclusion of new similarity metrics, and to enable alterations to the weightings of each similarity metric within the linear combination. These metrics include:

- A type similarity metric, (**Snippet C.1**), where the metric returns the number of types common to both items

- A directory similarity metric, (**Snippet C.2**), where the metric returns the number of directories that both items are contained in.

- Directory similarity metrics with quadratic/exponential falloffs, (**Snippet C.3**), where the score returned is better for items close to the item being checked, and decreases by an exponential or quadratic factor the "further away" it is.

- Related type similarity, (**Snippet C.4**), where for items that are considered "related" to the target items, the total number of similar types they have is returned.

Each labelled item is then tagged with a score that is a linear combination of these metrics.

We then need to select several of the most similar labelled items, in this case the top eleven. This is limited enough that we don't require an excessive number of items in the pod, but large enough to gain a useful consensus.

5.2 Reuse

The aim of this phase is to create a consensus function, where, given eleven labels and their similarity scores, we choose the label most appropriate for an item. The consensus function chosen for this program was a simple majority. This functionality is shown in **Snippet C.5**.

5.3 Future Improvements

As the pod's knowledge graph becomes richer, and size of the pod's contents grows, we would likely need a more nuanced view of the pod. Fortunately, it is straightforward to add new similarity metrics to the program, including:

- Negative type similarity – a type similarity metric that removes score from items that have types that do not match the target item's types.
- Richer relation metrics – as the relations between items grow more nuanced, the semantic analyser could make note of these relations, and similarity metrics could be developed to take advantage of this extra information.

I could also create more elaborate consensus functions, such as one that chooses the label with the highest total weighting. This could be complemented by a mechanism to vary the cardinality of the consensus function. Eleven works very well for a test dataset of our size, but a pod with fewer items would benefit from a lower number, and a pod with more items could benefit from a larger number.

Finally, calibrating the weightings of each metric to optimally fit a training dataset could boost the effectiveness of the CBR if its accuracy were poor. The current solution relies on ad-hoc weightings to produce the best fit. In future it would be better to have an empirical method of determining these weightings, like an optimisation algorithm, e.g. gradient descent.

6 Evaluation of Categorisation Systems

In my motivation, I detailed that I wanted my categorisation system to be:

- Adaptable – Easily changed, accommodating new categories and new information about items in the pod.
- Robust – Able to handle uncharacteristically formatted data gracefully, and preferably categorise it in the nearest appropriate category.
- Revisable – Able to take into account user inputs to improve categorisation, and do so as simply as possible.

In addition, another quality I wanted the system to have was accuracy - the system must give my test data the correct label as often as possible.

I then faced the challenge of evaluating the categorisation systems against these criteria. The first problem I encountered was finding data to test the system with. There are few social media applications that use SoLiD [7], and the ones that do exist are very simple. I solved this problem by using synthetic test data. This synthetic test data was designed to mimic the way interoperable social media posts would be stored, using specific directories for each category of posts.

This synthetic test data has a fairly standard layout, common between their intended categories. Most categories utilise a turtle file containing RDF triples describing the post it represents, which give it a type, description, and URI. Most categories also involve multiple files being stored in specific directories. More detailed descriptions of each category are below:

Image Post

- Describes an image of varying format with a caption.
- `ImageObject` type, `contentURL`, caption, and node URI in turtle file.

- Varied both file locations in pod to mimic multiple applications storing similar data in different directories

Article Post

- Describes a post based on a text file.
- `BlogPosting` type, `contentURL`, description, and URI in turtle file.
- Text file stored in `/articles/`.
- Turtle file stored in `/posts/`.
- Straightforward to categorise.

Text Post

- Only uses a turtle file containing main text, a URI, and a `TextObject` type.
- Meant to model a more knowledge graph-based approach to building social media.
- Introduces multiple categories of turtle file, which should throw off classifiers.
- Stored in `/posts/`.

Combo Post

- Consists of a combination of an image and a text file, linked by a single turtle file.
- Turtle file contains a post URI, a caption, and a `CreativeWork` type, alongside linking to the image via image and the text file via `contentURL`.
- Image file stored in `/images/`.
- Text file stored in `/articles/`.

- Turtle file stored in `/posts/`.

PDF Post

- Consists of a PDF file placed directly into the root directory of the SoLiD pod.
- No turtle file.
- Meant to check the adaptability of the system.

Metadata Files

The other classification entered into the system is the set of turtle files being used during classification. These files are of little use to the average user but still need to be classified. It would be most fitting for files with this classification to be hidden, as they often contain information regarding the posts being classified. They are usually stored in `/posts/`.

6.1 The Tests

In my evaluation of the two classifiers, I used separate datasets for training and testing. These datasets were structured similarly, with items being stored in the same directories and each category's items having the semantics outlined above. Across the tests, I varied:

- The training dataset
- The test dataset
- The data preparation for DT
- Similarity metrics available to CBR
- The information exposed by the semantic analyser

This gave me a wider picture of the performance of our classification methods.

The training datasets are outlined in **Table 1**, with each post obeying the conventions outlined above. The training dataset consists of the item objects from the semantic analyser, alongside labels for each of those items.

Training Dataset	Contents of Training Dataset
1	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts
2	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts 6 PDF Posts

Table 1: The training datasets

The test datasets are also outlined in **Table 2**. The key difference here is the lack of labels available to the program, alongside the post data being distinct from the post data in the training dataset.

Training Dataset	Contents of Training Dataset
A	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts
B	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts 20 Typeless Article Posts (metadata doesn't contain type information) 20 Typeless Combo Posts
C	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts 20 Typeless Article Posts 20 Typeless Combo Posts 20 PDF Posts
D	50 Image Posts 50 Article Posts 50 Combo Posts 50 Text Posts 20 Typeless Article Posts 20 Typeless Combo Posts 14 PDF Posts

Table 2: The test datasets

I devised my tests as shown in **Table 3**, with the DT preparation methods and CBR similarity metrics being outlined in Chapters 4 and 5 respectively.

Test Number	Training Dataset	Test Dataset	Decision Tree Preparation	CBR Similarity Metrics	Semantic Analysis Annotations
1	1	A	Trained with Type and Directory Encodings	Type Similarity, Directory Similarity, Exponential Falloff Directory Similarity	Only annotates files if they are being described by another file due to a bug.
2	1	A	Same as 1	Same as 1	Annotates all files with the relevant information.
3	1	A	Retrained	Same as 1	Same as 2
4	1	B	Same as 3	Same as 1	Same as 2
5	1	A	Same as 3	Same metrics as 1, weightings adjusted	Same as 2
6	1	A	Same as 3	Introduced Related Type Similarity metric	Introduced relation information between related items.
7	1	A	Same as 3	Same as 6	Same as 6
8	1	A	Same as 3	Same as 6	Same as 6
9	1	A	Retrained	Same as 6	Same as 6
10	1	A	Relativity heuristic was introduced, model was retrained.	Same as 6	Same as 6

Table 3: The test plan

The motivations for each test are also outlined in **Table 4**

Test Number	Motivation
1	A test of the system straight after development had finished. Also to test performance of the system with imperfect semantic analysis, to show how the information exposed by the semantic analyser affects performance.
2	A test of the system with improved semantic analysis but no retraining, showing how each system responds to new information
3	A test with the systems retrained to take advantage of new semantic annotations
4	A test of the system's performance on items with no type information.
5	To check whether altering CBR's internal weightings improves performance in suboptimal cases.
6	To assess CBR's performance with an additional similarity metric, taking advantage of graph data exposed in the semantic analysis.
7	To assess the performance of both systems when a new category is introduced without the prerequisite labelling. A test of how gracefully they both fail.
8	To assess the adaptability of both systems by introducing a small number of labels for the new category
9	To assess the performance of systems with a new category once they have been retrained
10	To assess the performance of Decision Trees on posts with no type data, with DT utilising an extra heuristic.

Table 4: Motivations for tests

When conducting the tests, performance was primarily assessed via accuracy. This was assessed by checking how many posts in the test dataset were allocated the

correct label. A breakdown of the results is in **Table 5**, with an overall accuracy and a category-specific accuracy for each system:

	Decision Trees		Case-Based Reasoning	
Test Number	Average Accuracy	Category-specific Accuracy	Average Accuracy	Category-specific Accuracy
1	75%	Image – 100% Article – 100% Combo – 100% Text – 0%	75%	Image – 100% Article – 100% Combo – 100% Text – 0%
2	75%	Image – 100% Article – 100% Combo – 100% Text – 0%	100%	Image – 100% Article – 100% Combo – 100% Text – 100%
3	100%	Image – 100 Article – 100% Combo – 100% Text – 100%	100%	Image – 100% Article – 100% Combo – 100% Text – 100%
4	91%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo (article component) – 0% Typeless Combo (image component) – 0%	96%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo (article component) – 0% Typeless Combo (image component) – 100%
5	N/A	N/A	96%	Same as above

Table 5: The test results (Part 1)

	Decision Trees		Case-Based Reasoning	
Test Number	Average Accuracy	Category-specific Accuracy	Average Accuracy	Category-specific Accuracy
6	N/A	N/A	100%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo – 100%
7	85%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo– 0% PDF Post - 0%	92%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo– 100% PDF Post - 0%
8	85%	Same as above	100%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo – 100% PDF Post – 100%
9	92%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo – 0% PDF Post – 100%	100%	Same as above
10	100%	Image – 100% Article – 100% Combo – 100% Text – 100% Typeless Article – 100% Typeless Combo – 100% PDF Post – 100%	N/A	N/A

Table 6: The test results (Part 2)

6.2 Analysis of Test Results

My four main criteria for the categorisation methods were:

Accuracy

Both systems performed well with regards to accuracy when using data similar to the training dataset, as shown by **Test 3**. However, as shown by **Test 1**, this is dependent on the semantic analysis providing enough information. When data is introduced that Decision Trees cannot meaningfully use without extensive training, such as graph relation data, the accuracy of Decision Trees decreases when compared to Case-Based Reasoning, as shown in **Tests 4 and 6**. However, in the case of our test dataset, **Test 10** shows that even simple heuristics related to relation data can suffice to improve outcomes. CBR was still more accurate than DT in **Test 4**, despite misclassifying some data.

Adaptability

In terms of adaptability, CBR outperforms DT as it requires no data preparation or training. It can immediately utilise additional information exposed by the semantic analyser (**Test 2**), whereas DT needs an extra training phase to "learn" the new pattern (**Test 3**). When PDFs were introduced into the labelled dataset, CBR correctly classified all the remaining PDFs (**Test 8**), whereas DT had to be retrained again (**Test 9**). Finally, when neither method could classify typeless posts correctly (**Test 4**), CBR took advantage of an intuitive similarity metric (**Test 6**) whereas DT had to utilise an overly specific heuristic (**Test 10**).

Robustness

Both systems are robust enough to fulfil this requirement, with both failing gracefully when encountering data that did not fit any pattern in the training data (**Tests 4 and 7**). Both systems responded by allocating misshapen data into the category that

made the most sense within their internal logic. This resulted in CBR outperforming DT in **Test 4**, as its internal logic involves assessing similarity between items.

Revisability

The lack of any user interface for inputting labels hurts both methods, but not reclassifying the labelled data is a good baseline. When considering the ease of introducing new labels into the system, CBR once again outperforms DT, as it utilises new labels and information immediately (**Tests 2 and 8**), whereas DT requires a training phase (**Tests 3 and 9**).

7 Conclusion

Both Decision Trees and Case-Based Reasoning performed excellently according to the metrics outlined. This is an encouraging result, as it fulfils the general aim of this project – finding a way to categorise SoLiD pod data in a user-centric way, with there being two successful classifiers. Improvements can still be made, particularly regarding the revision of the data labels, as neither system has a built-in way of modifying labels. In future, it would be beneficial to build a UI that allows users to browse their categorised files and assign labels, alongside testing other classification methods under the conditions of this project.

Despite both models performing well in terms of accuracy, by all other metrics CBR is superior. As CBR doesn't require training or data preparation, it can adapt to new categories and new information about items. This is important as in a SoLiD pod these will constantly arise. The lack of a required training phase makes this method fit better into the ecosystem.

I was surprised CBR performed so well with only simple metrics and a simple consensus mechanism; I was expecting to have to introduce more advanced features. The high accuracy is probably due to the homogeneity of the synthetic test dataset, but real social media systems could be similarly homogenous. CBR performed well and is a promising avenue for future research, with many improvements identified. CBR's main weakness is that it requires similarity metrics to be created by the user, but, as shown, there can be useful results with basic metrics. It also requires enough labels to reach a consensus.

It was surprising that DT performed to such a high level of accuracy. While its adaptability was worse than CBR's, it still managed to adapt to all the test data, given enough data preparation and retraining. It also needs less user input than CBR due to it not requiring any similarity metrics, and spotting patterns in ways developers may not anticipate.

While it would be useful to test other classification methods, these two were chosen because of their low data requirements, as the classifiers can only base their labellings on the contents of a specific pod. The tests with typeless items show that both methods require thorough type data, and there might be an alternative that doesn't have this weakness.

The most prominent challenge I encountered was the lack of complex SoLiD social media data. The synthetic test data used was not an ideal solution, best reflected by our classifiers only classifying a category completely correctly or incorrectly. React.js also created a lot of challenges due the BFS requiring a large number of API calls, as this doesn't fit naturally into React's intended workflow. I have enjoyed learning how to use the SoLiD ecosystem, and have also learned a lot about knowledge graphs and the classification methods used in this project.

Overall, the project has been a success, with the utilisation of the semantic analysis and classification technologies proving to be very effective within the context of the SoLiD ecosystem.

Appendices

Appendix A

Snippet A.1

```
const handleChange = (event) => {
  setRedirectUrl(useLocation)
  setOidcIssuer(event.target.value);
};

<span>
    Log in with:
    <input
      className="oidc-issuer-input "
      type="text"
      name="oidcIssuer"
      list="providers"
      value={oidcIssuer}
      onChange={handleChange}
    />
  <datalist id="providers">
    <option value="https://solidcommunity.net/" />
    <option value="https://login.inrupt.com/" />
    <option value="https://inrupt.net/" />
  </datalist>
  </span> <br></br>
  <LoginButton
      oidcIssuer={oidcIssuer}
      redirectUrl={window.location.href}
      authOptions={authOptions}
```

```
    />
    </div>
```

Snippet A.2

```
function useStorageGetter(webID, storageSetter) {

  useEffect(() => {
    async function hookBody() {
      const { PathFactory } = require('ldflex');
      const { default: ComunicaEngine } = require('@ldflex/comunica'
    );
      const { namedNode } = require('@rdfjs/data-model');

      const context = {
        "@context": {
          "@vocab": "http://xmlns.com/foaf/0.1/",
          "friends": "knows",
          "label": "http://www.w3.org/2000/01/rdf-schema#label",
          "storage": "http://www.w3.org/ns/pim/space#storage",
          "user": webID
        }
      };
      // The query engine and its source
      const queryEngine = new ComunicaEngine(webID);
      // The object that can create new paths
      const path = new PathFactory({ context, queryEngine });
      const userNode = path.create({ subject: namedNode(webID) });
```

```
23        await userNode.storage
24          .then(newStorage => {
25            console.log("new storage: " + newStorage)
26            storageSetter(`${newStorage}`)
27          }
28          )
29      }
30      hookBody()
31      console.log("root URL obtained.")
32    }, [webID, storageSetter])
33
34
35  }
```

Snippet A.3

```
1  export default function detailsFunc(rootURL, session, frontier,
       setFrontier, nodePathArr) {
2    const { PathFactory } = require('ldflex');
3    const { default: ComunicaEngine } = require('@ldflex/comunica');
4    const { namedNode } = require('@rdfjs/data-model');
5
6    const context = {
7        "@context": {
8          "@vocab": "http://xmlns.com/foaf/0.1/",
9          "friends": "knows",
10         "label": "http://www.w3.org/2000/01/rdf-schema#label",
11         "storage": "http://www.w3.org/ns/pim/space#storage",
```

```
12          "preferredObjectPronoun": "http://www.w3.org/ns/solid/terms#
    preferredObjectPronoun"
13        }
14      };
15
16      // get an authenticated comunica session.
17      const myEngine = new QueryEngine();
18      const queryEngine = new ComunicaEngine(rootURL,{ engine:
    myEngine })
19      const path = new PathFactory({ context, queryEngine });
20      const pod = path.create({ subject: namedNode(rootURL) });
21
22      (async document => {
23        console.log("doing details search on: ")
24        console.log(rootURL)
25        const bindingsStream = await myEngine.queryBindings(await `
    SELECT ?subject ?type WHERE {
26  ?subject rdf:type ?type.
27  }`, {
28          // Set your profile as query source
29          sources: [rootURL],
30          // Pass your authenticated session
31          '@comunica/actor-http-inrupt-solid-client-authn:session':
    session,
32        })
33          const bindings = await bindingsStream.toArray();
34          let subjectsList = []
35          for(let entNum = 0; entNum < bindings.length ; entNum++) {
```

```
36          const actualEntries = bindings[entNum].entries._root.
    entries
37
38          // using objects instead of that horrific array structure
    is cleaner but more fragile.
39          // NEVER change the query bindings if you want everything
    ending in .subject or .type to break.
40
41          const entryObj = {}
42
43          for (let i = 0 ; i < actualEntries.length ; i ++) {
44          const [queryName, queryVal] = actualEntries[i]
45          //console.log(queryName + ": " + queryVal.id)
46          entryObj[queryName] = queryVal.id
47
48          }
49
50          entryObj.searchPath = nodePathArr.concat(rootURL)
51          subjectsList.push(entryObj)
52        }
53
54        function updateFrontier (frontier, listOfSubjects) {
55          // removes current node from BFS frontier
56          let newList = []
57          for (let i = 0 ; i < frontier.length ; i++) {
58            if (frontier[i].subject != rootURL) {
59              // duplicates removal:
60
```

```
61              let duplicateFound = false
62
63              for (let j = 0 ; j < newList.length ; j ++){
64                if ((newList[j].subject == frontier[i].subject &&
    newList[j].type == frontier[i].type)) {
65                  duplicateFound = true
66                }
67                }
68
69              if (!duplicateFound) {
70                  newList.push(frontier[i])
71              }
72            }
73          }
74          // now to add the new parts - this makes it BFS not DFS
75
76          for (let i = 0 ; i < listOfSubjects.length ; i++) {
77            if (listOfSubjects[i].subject != rootURL) {
78              let dupCheck2 = false
79              for (let j = 0 ; j < newList.length ; j ++){
80                if ((newList[j].subject == listOfSubjects[i].subject
     && newList[j].type == listOfSubjects[i].type)) {
81                  dupCheck2 = true
82                }
83              }
84
85              if (!dupCheck2) {
86                newList.push(listOfSubjects[i])
```

```
87                }
88              }
89            }
90            console.log("new List: ", newList)
91            return(newList)
92          }
93
94          setFrontier((frontier) => {
95            console.log("updating frontier: ")
96            console.log(frontier)
97            let freshFrontier = updateFrontier(frontier, subjectsList)
98            console.log("fresh frontier: ")
99            console.log(freshFrontier)
100           return(freshFrontier)
101         })
102     })(pod);
103
104
105 }
```

Snippet A.4

```
1 // thingsWithTypes should have the shape:
2 /*
3     object{thingURL} = {
4       types: []
5       searchPath: [] --- This should be the list from thing.
    searchPath, which SHOULD be the same for all instances
```

```
      }
*/

export default function groupWithTypes(frontier, obj) {
    for (let i = 0 ; i < frontier.length ; i++) {
        if (obj.hasOwnProperty(frontier[i].subject)) {

            obj[frontier[i].subject].types = obj[frontier[i].subject
    ].types.concat([frontier[i].type])
        }
        else {
            // beginning of list induction
            // ASSUMPTION - for a given subject, searchPath is
    constant across instances
            obj[frontier[i].subject] = {
                types: [frontier[i].type],
                searchPath: frontier[i].searchPath
            }
        }
    }
    return(obj)
}

```

Snippet A.5

```
function augmentPosts(postObj, turtleFile) {
    //console.log("Augmenting turtle post with turtle file: ")
```

```
    //console.log(turtleFile)

    let processedTurtleObj = readTurtlePostFormat(turtleFile)
    console.log("processed Turtle object: ", processedTurtleObj)


    // Combo specific augmentation:
    if(Object.hasOwn(processedTurtleObj, "https://schema.org/image")
    && Object.hasOwn(processedTurtleObj, "https://schema.org/
contentURL")) {
        // So we have ourselves a combo:
        if(Object.hasOwn(postObj, processedTurtleObj["https://schema
.org/contentURL"])) {
            // We have a contentURL
            if(Object.hasOwn(postObj, processedTurtleObj["https://
schema.org/contentURL"])) {
                // We have an image!
                for (const [key, value] of Object.entries(
processedTurtleObj)) {
                    if (key != "https://schema.org/contentURL" &&
key != "https://schema.org/image") {
                        if(key == "http://www.w3.org/1999/02/22-rdf-
syntax-ns#type" || key == "https://www.w3.org/1999/02/22-rdf-
syntax-ns#type")
                            {
                                postObj[processedTurtleObj["https://
schema.org/contentURL"]].types = postObj[processedTurtleObj["
https://schema.org/contentURL"]].types.concat(value)
```

```
21                                  postObj[processedTurtleObj["https://
    schema.org/image"]].types = postObj[processedTurtleObj["https://
    schema.org/image"]].types.concat(value)
22                              }
23                          else {
24                              postObj[processedTurtleObj["https://
    schema.org/contentURL"]][key] = value
25                              postObj[processedTurtleObj["https://
    schema.org/image"]][key] = value
26                          }
27                      }
28                  }
29
30                  // Time to relate the two objects to each other:
31                  if(typeof postObj[processedTurtleObj["https://schema
    .org/contentURL"]].relatedTo == 'undefined') {
32                      postObj[processedTurtleObj["https://schema.org/
    contentURL"]].relatedTo = []
33                  }
34                  if(typeof postObj[processedTurtleObj["https://schema
    .org/image"]].relatedTo == 'undefined') {
35                      postObj[processedTurtleObj["https://schema.org/
    image"]].relatedTo = []
36                  }
37
38                  postObj[processedTurtleObj["https://schema.org/
    contentURL"]].relatedTo = postObj[processedTurtleObj["https://
    schema.org/contentURL"]].relatedTo.concat(processedTurtleObj["
```

```
https://schema.org/image"])
39              postObj[processedTurtleObj["https://schema.org/image
"]].relatedTo = postObj[processedTurtleObj["https://schema.org/
image"]].relatedTo.concat(processedTurtleObj["https://schema.org/
contentURL"])
40          }
41      }
42  }
43
44
45
46  else if (Object.hasOwn(processedTurtleObj, "https://schema.org/
contentURL")) {
47      if(Object.hasOwn(postObj, processedTurtleObj["https://schema
.org/contentURL"])) {
48          //postObj[processedTurtleObj["https://schema.org/
contentURL"]]
49
50          for (const [key, value] of Object.entries(
processedTurtleObj)) {
51              if (key != "https://schema.org/contentURL") {
52                  if(key == "http://www.w3.org/1999/02/22-rdf-
syntax-ns#type" || key == "https://www.w3.org/1999/02/22-rdf-
syntax-ns#type")
53                      {
54                          postObj[processedTurtleObj["https://
schema.org/contentURL"]].types = postObj[processedTurtleObj["
https://schema.org/contentURL"]].types.concat(value)
```

```
                    }
                else {
                    postObj[processedTurtleObj["https://schema.org/contentURL"]][key] = value
                }
            }
        }

    }
}

else {
    // oh no I have no way of getting the URL of the turtle file being read.
    // IT'S IN THE TURTLE FILE INFO!!!
    if (Object.hasOwn(turtleFile, "internal_resourceInfo")) {
        if (Object.hasOwn(turtleFile.internal_resourceInfo, "sourceIri")) {

            // If there is no content URL or image, annotate the ttl file with the info in it.
            for (const [key, value] of Object.entries(processedTurtleObj)) {
                if (key != "https://schema.org/contentURL") {
                    if(key == "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" || key == "https://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                        {
```

```
76                                  postObj[turtleFile.
    internal_resourceInfo.sourceIri].types = postObj[turtleFile.
    internal_resourceInfo.sourceIri].types.concat(value)
77                              }
78                          else {
79                              postObj[turtleFile.internal_resourceInfo
    .sourceIri][key] = value
80                          }
81                      }
82                  }
83
84
85              }
86          }
87      }
88
89
90  }
```

Appendix B

Snippet B.1

```
def typeCollate(valsDict):
    collatedTypes = []
    for key, value in valsDict.items():
        valTypes = value["types"]
        for theType in valTypes:
            if not (theType in collatedTypes):
                collatedTypes.append(theType)
    print("collated types: ")
    print(collatedTypes)
    return(collatedTypes)
```

Snippet B.2

```
def getDataDict(item, collatedTypes, collatedSearchPaths,
    collatedRelatedTypes, valsDict):
    # This will need to change as we add more data
    preparedItemDict = dict()
    for key1, value1 in item.items():
        if (key1 == "types"):
            for theType in collatedTypes:
                if theType in value1:
                    preparedItemDict[theType] = 1
                else:
                    preparedItemDict[theType] = -1
        elif (key1 == "searchPath"):
            for thePath in collatedSearchPaths:
                if thePath in value1:
                    preparedItemDict[thePath] = 1
                else:
                    preparedItemDict[thePath] = -1
    if "relatedTo" in item:
```

```
        for theRelType in collatedRelatedTypes:
            for relItem in value1:
                if theRelType in valsDict[relItem]["types"]:
                    preparedItemDict[theRelType] = 1
                else:
                    if preparedItemDict[theRelType] != 1:
                        preparedItemDict[theRelType] = -1
    else:
        for theRelType in collatedRelatedTypes:
            preparedItemDict[theRelType] = -1
    return(preparedItemDict)
```

Snippet B.3

```
def searchPathCollate(valsDict):
    collatedSearchPaths = []
    for key, value in valsDict.items():
        valTypes = value["searchPath"]
        for thePath in valTypes:
            if not (thePath in collatedSearchPaths):
                collatedSearchPaths.append(thePath)
    print("collated search paths: ")
    print(collatedSearchPaths)
    return(collatedSearchPaths)
```

Snippet B.4

```
def relTypeCollate(valsDict):
    collatedTypes = []
    for key, value in valsDict.items():
        if "relatedTo" in value:
            relVals = value["relatedTo"]
            for relatedItem in relVals:
                if relatedItem in valsDict:
                    valTypes = valsDict[relatedItem]["types"]
```

```
                    for theType in valTypes:
                        if not (theType in collatedTypes):
                            collatedTypes.append(theType)
    #print("collated types: ")
    #print(collatedTypes)
    return(collatedTypes)
```

Appendix C

Snippet C.1

```
# ---------- TYPE SIMILARITY METRIC ----------
    collatedTypes = []
    for key, value in labelledDict.items():
        valTypes = value["types"]
        for theType in valTypes:
            if not (theType in collatedTypes):
                collatedTypes.append(theType)
    # Let's just go for number of similar types:
    def typeMeasure(item, optionalWeightings):
        commonTypeCount = 0
        for theType in item["types"]:
            if theType in mainValue["types"]:
                commonTypeCount += 1
        return(commonTypeCount)
    typeMetric = Metric("Common Type Metric", typeMeasure)
    metricDict = setMetric(metricDict, typeMetric, 2, None)

# -------------------------------------------
```

Snippet C.2

```
# --------- LINEAR LOCATION METRIC -----------
    def linearFileMeasure(item, optionalWeightings):
        commonLocationCount = 0
        for location in item["searchPath"]:
            if location in mainValue["searchPath"]:
                commonLocationCount += 1
        return(commonLocationCount)
    linearFileMetric = Metric("Linear File Metric",
   linearFileMeasure)
    metricDict = setMetric(metricDict, linearFileMetric, 3, None)
```

```
# ------------------------------------------
```

Snippet C.3

```
# --------- FALLOFF LOCATION METRIC ----------
    def falloffFileMeasure(item, optionalWeightings):
        commonLocationCount = 0
        for location in item["searchPath"]:
            if location in mainValue["searchPath"]:
                commonLocationCount += 1
        # Exponent Amount is the value unrelated to the common locations in the formula, e.g. used for quadratic falloff
        exponentAmount = optionalWeightings["exponentAmount"]
        # Exponent Factor used to include common loc count in the formula, e.g. used for exponential falloff
        exponentFactor = optionalWeightings["exponentFactor"]
        # Base Amount is the constant amount multiplied by in the base of the exponent
        baseAmount = optionalWeightings["baseAmount"]
        # Base Factor is the common loc count included in the base of the exponent
        baseFactor = optionalWeightings["baseFactor"]

        #baselineDepth is the amount you should get if you have a complete match for most things, i.e. the average depth of an item
        baselineDepth = optionalWeightings["baselineDepth"]

        itemVal = (baseAmount + baseFactor * (commonLocationCount))**(exponentAmount + exponentFactor* (commonLocationCount))
        baselineVal = (baseAmount + baseFactor * (baselineDepth))**(exponentAmount + exponentFactor* (baselineDepth))
        return(itemVal/baselineVal)
    falloffFileMetric = Metric("Falloff File Metric",
```

```
    falloffFileMeasure)
23      quadraticFalloffDict = {
24          "exponentAmount": 2,
25          "exponentFactor": 0,
26          "baseAmount": 0,
27          "baseFactor": 1,
28          "baselineDepth": 3
29      }
30      #metricDict = setMetric(metricDict, falloffFileMetric, 2,
    quadraticFalloffDict)
31      exponentialFalloffDict = {
32          "exponentAmount": 0,
33          "exponentFactor": 1,
34          "baseAmount": 2,
35          "baseFactor": 0,
36          "baselineDepth": 3
37      }
38      metricDict = setMetric(metricDict, falloffFileMetric, 2,
    exponentialFalloffDict)
```

Snippet C.4

```
1  # ------ Relation Type Similarity Metric -------
2      def relationTypeMeasure(item, optionalWeightings):
3          #mainValue is the val we are comparing to
4          similarTypes = 0
5          if "relatedTo" in mainValue.keys() and "relatedTo" in item
       .keys():
6              mainRelatesList = mainValue["relatedTo"]
7              mainRelatedTypes = []
8              for relUrl in mainRelatesList:
9                  if "types" in valsDict[relUrl]:
10                     mainRelatedTypes += valsDict[relUrl]["types"]
11             itemRelatesList = item["relatedTo"]
```

```
            itemRelatedTypes = []
            for relUrl in itemRelatesList:
                if "types" in valsDict[relUrl]:
                    itemRelatedTypes += valsDict[relUrl]["types"]
            for relatedType in itemRelatedTypes:
                if relatedType in mainRelatedTypes:
                    similarTypes += 1
                else:
                    similarTypes -=1
        # adding negativity to non-similar types
        elif "relatedTo" in item.keys():
            itemRelatesList = item["relatedTo"]
            itemRelatedTypes = []
            for relUrl in itemRelatesList:
                if "types" in valsDict[relUrl]:
                    itemRelatedTypes += valsDict[relUrl]["types"]
            for relatedType in itemRelatedTypes:
                #by definition mainValue not related to anything
                similarTypes -=1

        return(similarTypes)

    relationTypeMetric = Metric("Relation Type Similarity Metric",
    relationTypeMeasure)
    metricDict = setMetric(metricDict, relationTypeMetric, 4, None
    )
```

References

[1] A. V. Sambra, E. Mansour, S. Hawke, *et al.*, "Solid: A Platform for Decentralized Social Applications Based on Linked Data,"

[2] D. Fensel, U. Şimşek, K. Angele, *et al.*, "Introduction: What Is a Knowledge Graph?" In *Knowledge Graphs: Methodology, Tools and Selected Use Cases*, Cham: Springer International Publishing, 2020, pp. 1–10.

[3] R. Dedecker, W. Slabbinck, J. Wright, P. Hochstenbach, P. Colpaert, and R. Verborgh, "What's in a pod? – a knowledge graph interpretation for the Solid ecosystem," in *Proceedings of the 6th Workshop on Storing, Querying and Benchmarking Knowledge Graphs*, M. Saleem and A.-C. Ngonga Ngomo, Eds., ser. CEUR Workshop Proceedings, vol. 3279, Oct. 2022, pp. 81–96. [Online]. Available: `https://solidlabresearch.github.io/WhatsInAPod/`.

[4] *1.10. Decision Trees*, https://scikit-learn.org/stable/modules/tree.html.

[5] L. Rokach and O. Maimon, "Decision Trees," in *Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, Eds., Boston, MA: Springer US, 2005, pp. 165–192.

[6] A. Aamodt and E. Plaza, "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[7] *Solid applications*, https://solidproject.org/apps.