Identifying Argument Overlaps in Data Interchanges

Connor Ness

Submitted in partial fulfilment of the requirements of
Edinburgh Napier University
for the Degree of
MSc Computing

School of Computing

2021

Authorship Declaration

I, Connor Ness, confirm that this dissertation and the work presented in it and my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of any larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understood the penalties associated with Academic Misconduct.
6. I also confirm that I have obtained **informed consent** from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Type name: Connor Ness
Date: 18th August 2021

Matriculation No: 40272321

General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name under *one* of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

Connor Ness

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

Argument data interfaces are methods of displaying the components of an argument, these interfaces area also used to populate databases as a method to view and export argumentation data for analysis and are used in the generation of online corpora of argumentation. The databases which hold these corpora are useful to argumentation research and are filled by users creating annotated argument maps, the culmination of these maps which are of the same domain create a corpus which can be used for analysis of argumentation. However, in argumentation the expression of the same claim in multiple forms can occur across differing arguments. This could be defined as an overlap of a claim as the same idea is being presented, but merely takes on a different form. From research, the identification of these overlaps has not been implemented in any tool relating to popular data interfaces such as AML or AIF. The detection of these overlaps would be the beginning of the process to merge multiple argument maps into one large map that would contribute to a larger argument as a whole - allowing representation of each facet of an argument in a deep way. Through extraction of several files of an argument data interface, argument nodes can be processed through string comparison algorithms to identify and extract these overlaps. Implementation of the extraction of nodes from an interface, and the comparison of each through multiple string comparison methods would serve as a strong start to the begin developing the ability to merge argument maps.

Contents

List of Tables

List of Figures

Acknowledgements

I thank Dr Simon Wells for his invaluable advice and supervision of this project - his suggestions and correspondence greatly assisted in the shaping and research that define this dissertation.

I would like to thank my family for their support during my years of study and their understanding and help with the stress and time constraints which come with student life.

My cat Bedivere contributed to my sanity during long writing and implementation sessions by providing company and meaningfully one-sided conversations.

Finally I thank my loving partner Jade for her support both in emotional and understanding of time constraints during this project, whose presence helped calm and focus me during this time.

Chapter 1

Introduction

There is a lack of research towards the merging of argument maps in online corpora, this dissertation aims to implement a method with the aim of being used to identify claims within argument maps which are similar in meaning. To implement this method, the creation of a corpus with simple to extract and process data will be required. For this reason the Simple Argument Description Format (SADFace) was selected due to its easy to understand principles and singular implementation allowing the extraction process to be made simple. Several different argument data interchanges other than SADFace exist, the most popular in argument mining being the Argument Markup Language (AML) and the Argument Interchange Format (AIF). Both have their advantages but lack a general standardised software implementation

SADFace hold's argument claims within "argument atom nodes" these nodes are to be extracted into a list to be compared to each other. As SADFace's implementation is JSON based, Python's standard library *json* can be used to extract the relevant variables from the SADFace file simply. Once extracted, the compiled argument nodes are compared through several string comparison algorithms with the intention of each building a list of argument pairs determined by if the "acceptable distance" between argument strings is within a reasonable amount to be identified as an overlap. The implementation will be designed to take in any set of SADFace files for comparison.

The SADFace files to be processed are pre-analysed by hand to identify overlaps, this list will then be used to evaluate the effectiveness of the comparison algorithms.

The string comparison algorithms implemented take in two passed strings and provide a measurable value as an output, therefore each potential combination of argument pairs are created and processed through the algorithms this way. With an increase in files compared comes a multiplicative increase

in nodes contained dependant on the size of the argument being processed. The algorithms implemented each generate a list of overlaps as identified by the "acceptable distance" value ascribed to each process. Once all algorithms have generated a list, these lists are cross compared to create a new list of overlaps as detected as an occurrence across all algorithms.

Once the list of overlaps has been created by each function, these lists are compared to the hand identified list of overlaps in order to determine how many detected overlaps were detected correctly and incorrectly, this output will help refine what value each algorithm should utilize as an "acceptable distance" to flag an argument pair as overlapping in meaning.

An output is then generated to display the list of overlaps from each algorithm and the process of reoccurring overlaps from all algorithms. This output will be simple in nature as a text file but future development is recommended to implement a form of interactable user interface to assist in the process of analysing arguments for potential overlaps, this should help the process of merging argument maps at potential locations of overlaps to form a singular large map of arguments to provide each facet of an argument a depth of discussion.

The project involves several facets and communities of research and all are discussed due to the "untouched ground" nature of the project, discussions for corpus creation as relevant to the argument mining community as well as databases which hold corpora in specific argument data interchanges are made to give perspective into the value of annotated argument corpora and the differences in how online corpora are represented. The specifics of different argument data interchanges are discussed along with the advantages and disadvantages of each are mentioned to provide context for why the implementation was created with specifically SADFace in mind. Python was also utilised as the language for this implementation, several reasons such as development time and a community of natural language processing engineers utilising the language contributed to the languages selection.

The initial design of the project and the constraints to be considered are discussed to give an overview of the implementations functionality. The changes made between design and implementation are minor due to the proper segmentation of each function as based on the segmentation of each facet which contributes to the relevant literature. Each functions implementation are detailed to inform the specifics of processes as individual components, this involves reading in SADFace files and extraction of arguments, comparing every argument to each other through explained implemented string algo-

rithms and finally the output generation. Issues faced in development are listed for the intention of displaying evidence of project growth.

An evaluation of the implementation with considerations to the aims detailed is completed and recommendations for useful changes and implementations that would advance and bring further usage of the implementation are detailed - however as described everything recommended was outwith scope of this implementation.

The project created is viewable on the public github repository at: `https://github.com/ConnorNess/SADFace-Overlap-Detection`. The code is also posted within the appendices (see 7.1).

1.1 Aim and Objectives

- To create SADFace files from the same domain using a pre-existing format
- Attempt the creation of a conversion tool from the chosen format to SADFace
- Hand select all identifiable overlaps within the SADFace files
- To implement a python script which will output identifiable overlapping nodes in SADFace
- To use at least two string comparison algorithms
- Evaluate the output generated by string comparators against hand identified overlaps
- The created script must be designed well enough to be understood by another developer for the purpose of improvements
- Discuss what improvements can be made and what ones are believed to have the greatest effect if implemented

1.2 Scope

The corpus created should contain at least 50 arguments, this number is somewhat arbitrarily chosen but the arguments within should be memorable

and unambiguous in meaning to better identify overlaps by hand. The domain chosen is not relevant but it is necessary for all arguments to be of the same domain or overlaps will not be found.

The implementation will be focused on the extraction of SADFace files, other data interchanges will not be considered unless extra development time is possible within the proposed work plan (see **??**).

The functions will compare every single possible argument pair, this is inclusive of arguments from the same map. Potentially a feature could be implemented to select if same-map arguments should be compared or not however this is not seen as necessary.

At least two string comparison algorithms should be implemented. They will be basic in implementation as more advanced methods would take a large amount of development time to implement, suggestions for advanced algorithms will be made as a concluding word.

1.3 Research Questions

Primarily focused on the implementation, questions are made to how the work will be undergone and selections made for the implementation.

- Which string comparison algorithms to select and the benefits to each one specifically, alongside what metrics to be used.
- The use of the deliverable in the argument mining community.
- How the deliverable can be implemented as a feature in larger software, or how it may be used as a standalone process.

1.4 Thesis Structure

This thesis discusses the work done as the development process has functioned.

- Literature Review
  - Discussion of relevant literature pertaining to each facet of the work undergone.

- Design Overview
  - Detailed planning of the components necessary for a functioning implementation and a definition of the project scope.
- Implementation Discussion
  - Discussion of how components were implemented and how they function, and how relevant work was undertaken.
- Evaluation
  - Evaluation of the outputs of implemented algorithms and a discussion of how well each feature was implemented.
- Conclusion
  - Finishing discussion of how the deliverable was implemented and how it met the aims described as well as future work to be done on the deliverable.
- Appendices
- References

Chapter 2

Literature Review

2.1 Introduction

Natural Language Processing (NLP) is a linguistic based approach utilising artificial intelligence and machine learning to extract, process, analyse, and generate annotated written text – this could be from a book, online discussion, or transcript video. The functionality and utilisation of Natural Language Processing aims to convert natural language into data structures and code comprehendible by a computational structure. NLP has had major relevancy to modern implementations such as the rise in virtual assistants like Cortana for Windows 10, these assistants require the ability to take in natural, non-programmatically formatted inputs of language to then process the desired output and then relay this output to the user again in a naturally spoken way. NLP is a subfield of general speech and language processing which aims to model language on a rule-based system. (Lippi and Torroni, 2015)

Difficulty in NLP comes from the nature of natural language being vague and non-conforming to all participants of a language. Several factors such as double meaning words, tone, the relation between those participating in conversation, even body language and misunderstandings in a conversation will affect the flow of a conversation which can be incredibly difficult to determine from a simple transcript of speech. The detection of these facets of language can be both essential to comprehension of a dialogue, and simultaneously unknown to a viewer of this dialogue. For example, if two radio hosts were discussing a topic and one was becoming visibly agitated by the conversation but not in tone, a listener would not be able to fully comprehend the emotion displayed by the speaker which could be important to understanding the meaning behind an argument or where the speaker is drawing

their conclusions from. In a scenario such as this, a computational structure would have to be able to notice discrepancies such as tone difference, adjustments in speech, note when misunderstandings are made in order to exactly meet the needs of the user, these factors are not always able to be noted by any participant other than the speaker at times. In an online discussion, text cannot sufficiently relate linguistic tools such as sarcasm effectively and for a speaker to indicate a technique like this is being used can undermine the flow of an argument however the risk comes that the opposition will not notice this tool. Misunderstandings such as this can be difficult in certain scenarios for those who use the natural language to detect which translates to an algorithm having further difficulty in comprehension of a conversation. (Beysolow II, 2018)

Essentially the depth of communication between people contributes to the difficulty of implementations in NLP, language cognition aims to understand how the cognitive functions of the brain interpret and process language. Linguistics is the scientific study of language in use several topics within linguistics contribute to understanding of natural language such as semantics – the meaning of words and their relationship to each other to form phrases.

Arguments are a form of critical thinking within natural language where the participants conduct a dialogue with the goal to convince an opposition dialogue to reach the same conclusion when the contributing parties may have conflicting viewpoints on a topic. Argument mining aims to implement solutions to automatically identify, process, and analyse an argument within natural language. Mining requires the correct identification of what is "important" to an argument and what is not within the context of the discussion, the ability to remove non-important dialogue or to extract the backbone of an argument without losing any depth, meaning, or relevancy in the process. (Mochales and Moens, 2011)

An issue presented by the processes of argument mining is the ability to present this automatic output in an analysable fashion. Argument markup languages aim to solve this issue by creating tools which assist in the visualisation and construction of processed arguments. Argument maps and diagrams are methods of this visualisation where statements are linked together by how each statement supports or is in opposition to another statement.

The Argument Markup Language (AML) is a simplistic tree format saved as an XML file which allows for the transmission of an argument between packages, AML formatted arguments populate the database AraucariaDB which is tied to the software tool for argument analysis – Araucaria. Araucaria assists argument diagramming and reconstruction which ties nodes (which contain the actual text of an argument) to schemes (which represent

how nodes relate to each other). However, the Argument Markup Language being in XML format can make it generally difficult to parse information from without the use of specialised software like Araucaria as mentioned which majorly aids the creation of argument maps. (Rowe and Reed, 2008)

Argument Interchange Format (AIF) was designed to help alleviate the shortcomings of the Argument Markup Language due to AML being designed for use in specialist software. Designed with the aim of facilitating the development of multiple-agent systems where individuals are capable of argument-based interactions and to facilitate data sharing with tools for argumentation manipulation and visualization. AIF has two node types – information and scheme, which are used to represent information within an argument (Rahwan and Reed, 2009). Schemes can further contain multiple subtypes – Inference or supporting statements, Conflict or opposition to a statement, and Preference. Preference schemes relate to when an argument is "preferred" or viewed as stronger within an argument for example an argument from an expert will be preferred when compared to an argument from an uninformed participant. The "edges" or links between nodes are inferred within AIF from which nodes are present and is not input by an analyst. The Argument Interchange Format's depth provides useful high-level representation in arguments, but this flexibility of this depth means users must understand the complexity that comes with AIF in order to work with the format. (Modgil and McGinnis, 2008)

SADFace, the Simple Argument Description Format. A JSON based document format designed to be simple and capable of being adapted into pre-existing systems. SADFace being of a simplistic JSON format means it will not require specialised software in order to interact with and is laid out to represent an understandable model of argumentation structure.

The JavaScript Object Notation format (JSON) is a subset of JavaScript but is not itself a programming language but instead a data interchange format. JSON is designed only to carry information to be used in the communication between multiple processes requiring formatted data exchange. JSON is essentially a text representation structured as either a list of name/value pairs or a list of values where sets of pairs are separated by curled braces " " and individual values are named for example "text", divided by a colon ":" and then followed by the value "generic sentence here." Within sets, arrays or lists can be declared by use of square brackets "[ ]" for example "["sentence 1", "sentence 2"]. (Smith, 2015)

SADFace represents claims in an argument as argument atoms, notated as "atoms" within the document, an argument will consist of multiple of these atoms linked to form a flow of statements, these atoms have unique identi-

fiers. In a structured argument a concluding atom will need to be supported by a premise atom. This support is displayed through "schemes" which is used to represent argumentative reasoning such as "support" or "conflict" as atoms will relate to each other in an argument, separate atoms may be in contention of a conclusion or may assist in the reasoning behind a conclusion. Atoms and schemes are connected by edges with a direction to assert the relation dynamic between nodes. These edges are uniquely identifiable and follow a source atom to a target which are retrieved by use of the atom's unique identifier. Similar to AIF, atoms cannot be linked together directly and must go via a scheme node as separate statements must have a known relationship for the link to be meaningful in an argument.

Python is utilised in a large amount of natural language processing topics due to its functionality in machine learning and numerous libraries contributing to NLP such as the Natural Language ToolKit (NLTK) have made it the de facto language for natural language processes. As a scripting language with powerful standard libraries and an extensive and growing number of APIs (Application Programming Interfaces) python is extremely useful as an environment to immediately start development of a solution or to test a new algorithm by use of functions which can also be attributed to its ease of text processing and creation of dictionaries or lists. Python's json library can be used to easily populate a dictionary (or dict) with simple commands such as 'json.load(file)' which helps in this project as after loading a json file, we can simply use 'file.get('nodes')' to retrieve all our SADFace nodes. (Sarkar, 2016)

2.2 Corpus

The assembly of a corpus can be in part contributed by the inclusion of already constructed corpora however these corpora must have some form of relation through either domain or discussion in order to form a cohesive and useful corpus. Visser et al. (2018) aims to blend corpora together through the use of inter-textual correspondence to connect corpora based on their domain. Essentially if two or more separate corpora discuss the same topic, they can potentially be connected together to create a larger corpus with more depth. Several corpus databases are in active use, quite popularly is AIFdb, a database for the Argument Interchange Format composes thousands of argument maps with a multiplicative number of individual nodes within these maps.

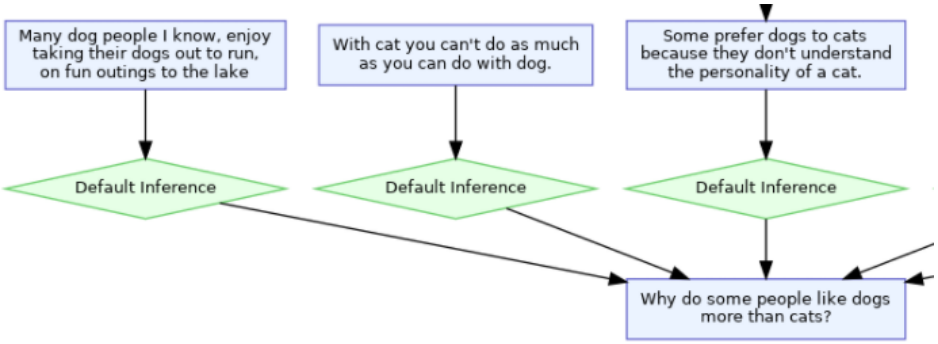

Figure 2.1: AIF 12126 from AIFdb

Automatic formatting of argumentative writing has increased massively in recent times particularly in scientific publications (Bornmann and Mutz, 2015), these publications use argumentative language in abundance which could potentially provide a great number of corpora for use in analytical tools. The increase in corpus could accompany overlapping and redundancy within argument maps within the same domain. Annotated corpus based in arguments within scientific publications are already being created (Lauscher et al., 2018) which provide robust, human created analytical corpora of researched topics.

Corpora outwith argument mining related formatting such as the *args.me* corpus provide a constantly updating platform with strict formatting to be used by the average person to effectively communicate a stance on a topic by either pro or con. Users submit arguments as a start to a discussion or potentially in reference to an individual point effectively generating a non-visualised map of linked arguments pertaining to a singular topic. (Ajjour et al., 2019) analyse this corpus and output classifications for the relation of arguments. Should a stance be clearly pro or con in language and another stance is related to this it can be said to have a *Same-Side* classification, the clarity in one stance can be used as a comparison to the ambiguity of another. *Stance* classification identifies whether an argument is for or against a topic. *Argument Relation* classification identifies the relationship between two arguments in form of support, attack, or neither. While these classifications are not directly relevant to an argument markup corpus, the *Argument Relation* classification is similarly used in markups such as AIF and SADFace. Nodes in these markups cannot be directly linked and must pass through an identifier for how these nodes relate (Figure 2.1) (Support and Conflict being common within SADFace).

As mentioned by Reed (2006), corpora should be large in size in order to robustly conclude results from research - testings using small corpora are more indicative of the need for further large scale testing however analysed corpus provide evidence of initial utility of a tool. Large corpora are susceptible to flawed argument structures as a dataset grows past hand selected schema, Reed (2006) notes that the corpora used in their study contained a large number of normative arguments (arguments that conclude with what *should* happen usually based on personal beliefs. Arguments from implication where a conclusion is drawn from related facts making them more reliable are also noted as being not uncommon, but less common than normative arguments. Differing argument types can be *generally* more trustworthy than others which is usually directly due to the amount of supporting evidence used within said argument. The more evidence within arguments within corpora can make the corpus more reliable for testing purposes for certain acts such as identification of value of arguments within a scientific or professional setting, however this is not typically reflected in "real world" arguments which may contain very limiting amounts of evidence.

Visser et al. (2018) also discuss how the merging of related corpora can add enhanced depth in terms of density of arguments and the addition of further related arguments not originally mentioned within a base corpus. The inclusion of related corpora can also identify the most discussed point of an argument or identify which evidence is more commonly called upon within arguments of the same domain - if a singular argument is used more than others, this may be indicative of how one side of an argument views the importance of this information. The research done aimed to investigate how both an argument and the outside discussion of an argument (in their case a live TV debate and a web forum's response to the debate) can be merged in order to generate this larger corpus while indicating popularity and divisiveness of certain arguments and claims.

2.3 Argument Mining

Argument mining by definition is the automatic process of identification, extraction, and processing of argumentation in natural language. By developing computational structures to comprehend argumentation structure, arguments can be retrieved and processed to succinctly extract what side of a topic is being taken, the points made and evidence pointed to by an individual, and even the reasons that led to this point being formed.

Lawrence and Reed (2020) discuss various components of argument mining. Lawrence separates argument analysis into several components. *Text Segmentation* is the process of extraction of text fragments that structure the argument made. *Argument/Non-Argument Classification* in which text segments are classified by which segments are inclusive of the argument being analysed and which aren't - a text segment can use argumentative language but may not contribute to the specific argument being analysed. *Simple Structure* notes the link between classified arguments by various relations such as convergent arguments, where separate premises arrive towards the same conclusion, and linked arguments, where separate premises support the arrival of a conclusion. *Simple Structure* may also identify the relations such as support or conflict between statements. *Refined Structure* as implied, refines *Simple Structure* further potentially through argumentation analysis tools to identify the argumentation scheme in relation to others.

Argument mining differs from opinion mining in which computational structures analyse emotional sentiments in a topic which do not follow a typical argumentation structure. Opinion mining however does not simply conclude a positive or negative stance but may, for example, determine specific semantic relations between an opinion statement and the linguistic methods used within to identify the role of certain key words to relay a stance in natural text - see Kim and Hovy (2006).

Lippi and Torroni (2015) push for the increased use of machine learning implementations of argument mining approaches. Lippi notes how argument mining methods typically approach natural text and dividing processing tasks in order to output an annotated argument markup. A machine learning approach to argument mining would require a large number of annotated arguments from many corpora, this presents an issue as while many corpora are available and in an increasing volume, they are not all created by the same group of experts and do not follow a guideline of annotations or extraction meaning differences will be present within these datasets comparatively. Machine learning has also been of great benefit to the development of artificial intelligence through deep learning approaches which could be applied to the same end in the field of argument mining.

2.4 Argument Data Interchanges

The Argument Markup Language (AML) defines both texts and the relationship between these tests through tags describing argument components. AML implements an eXtensible Markup Language (XML) type of format-

ting to generate a tree like structure which allows AML files to be translated through style sheets into, for example, HTML files (Reed and Rowe, 2001). AML is tailored for use in the Araucaria software package created by Dundee University available at http://araucaria.computing.dundee.ac.uk/ (At time of writing, Araucaria and its database were inaccessible). Araucaria allowed users to diagram and output argument text creating and defining relationships between text nodes. Diagrams created in Araucaria are outputted as either JPEG image files for documentation or as previously mentioned, XML files. AML and Araucaria were also contributors towards AraucariaDB, the first online available argument corpus which assisted the growth of freely available argumentation corpora. Unfortunately AML's formatting makes it challenging to parse and requires the use of an XML tool-chain to interact with which, as stated by Wells (2020a), is becoming less relevant to modern software.

The Argument Interchange Format (AIF) was developed to counter the shortcomings of AML, namely that AML was designed for use within particular software as opposed to being used as a component within a larger system of argument tools making AML difficult to separate from Araucaria. AML also being designed for users to create argument diagrams rather than for automatic processing such as a lack of semantic model (Rahwan and Reed, 2009). AIF was designed for the aims of assisting the development of multi-agent systems and the facilitation of data interchange for argument visualization and editing software. AIF defines node types by information *i-nodes* which contain claims and premises in an argument or scheme *s-nodes* which define the relations between *i-nodes*. Specifically AIF defines these relations as either inference (node supports the claim of another), conflict (node opposes claim of another), and preference (node is *preferred* to another i.e. expert opinion compared to non-expert opinion).

As opposed to AML's rather rigid structure AIF receives development with the aim of extending its functionality such as Reed et al. (2008) who aimed to enhance AIF's functionality to accompany the representation of argument protocols (how a dialogue will proceed) and history (how a dialogue actually proceeded). The work is evidence of AIF's ability and interest in improvement as a system as it moved towards popular use.

AIF uses defined syntax and semantics to effectively represent an argument providing a flexible cohesive system of representation. However AIF requires RDF and XML tool-chains to function and has somewhat complex terms in its definition and use. (Chesnevar et al., 2006) discusses the semantics of AIF such as relational links between nodes and individual node attributes. Nodes may contain attributes relevant to the representation like

a title or creator name and interestingly polarity - much like pro and con like used in arg.me. Nodes are linked by *Edges* which are simply pointers which are required between related nodes, these edges are not able to be linked between two *i-nodes* as there must be an explanation for the relation between two claims but are able to be linked between two *s-nodes* as these can imply a sort of meta-reasoning for how i-nodes may be in conflict or support towards a larger claim.

2.5 SADFace - The Simple Argument Description Format

A JSON based argument description format, SADFace aims to make the use of argument based data more simplistic from a software development perspective. SADFace works as a main facet which integrates the tools within OAPL - the Open Argumentation PLatform, a platform of argument software libraries and UIs (Wells, 2020b). SADFace follows AIF concepts such as nodes, schemes, and edges and the requirement of nodes to pass through schemes to be linked. Due to being of JSON format, SADFace does not require any specialist software to parse.

An example of a SADFace JSON file is shown:

```
{
    "edges": [
        {
            "id": "2229_e1",
            "source_id": "2229_n_s1",
            "target_id": "2229_n_a1"
        },
        {
            "id": "2229_e2",
            "source_id": "2229_n_a2",
            "target_id": "2229_n_s1"
        },
        {
            "id": "2229_e3",
            "source_id": "2229_n_a3",
            "target_id": "2229_n_s1"
        },
        {
            "id": "2229_e4",
```

```
            "source_id": "2229_n_a4",
            "target_id": "2229_n_s1"
        }
    ],
    "metadata": {
        "core": {
            "analyst_email": "connor.ness@outlook.com",
            "analyst_name": "Connor Ness",
            "created": "2021-06-26T14:05:20",
            "description": " ",
            "edited": "2021-06-26T14:05:20",
            "id": "2021-06-26-14-05-20-cat2229",
            "notes": " ",
            "title": "cat2229",
            "version": "0.1"
        }
    },
    "nodes": [
        {
            "id": "2229_n_a1",
            "metadata": {},
            "sources": [],
            "text": "Cats make good pets",
            "type": "atom"
        },
        {
            "id": "2229_n_s1",
            "metadata": {},
            "name": "support",
            "type": "scheme"
        },
        {
            "id": "2229_n_a2",
            "metadata": {},
            "sources": [],
            "text": "they are affectionate",
            "type": "atom"
        },
        {
            "id": "2229_n_a3",
            "metadata": {},
```

```
            "sources": [],
            "text": "They are clean",
            "type": "atom"
        },
        {
            "id": "2229_n_a4",
            "metadata": {},
            "sources": [],
            "text": "they are entertaining",
            "type": "atom"
        }
    ]
}
```

To explain the file shown. Edges like in AIF link nodes together from source to target to indicate flow of an argument. Authors of files may fill in relevant data to a SADFace file in the indicated "metadata" section. Finally nodes are like in AIF also, statements and claims captured and represented, these nodes are uniquely identifiable for use by edges and have a type. Atom type nodes, or argument atoms contain claims within an argument - in general an argument will comprise of several linked atoms. SADFace is open source and freely available in a Git repository (https://github.com/siwells/SADFace).

SADFace's use over AIF can mainly draw from AIF's lack of implementation in the form of a software library which relates to the lack of use by developers adding enhanced functionality outside of theoretical extensions which can have constraints such as limiting the number of additional representation software (Reed et al., 2008). SADFace was developed within AIF principles but with a base which can permit both comprehension and extensible functionality (Wells, 2020a).

2.6 Argument Databases

Bex et al. (2013) discuss *The Argument Web*, a large web of connected arguments from online postings constructed in a structured view. The current online climate has several facets which limit constructive discussion such as one-sided base popularity rankings (see Reddit's upvote/downvote or Twitter's like) rather than more deep interactions such as agreement or disagreement, platforms' isolation from each other also contributes to an unconnected discussion. The proposed *argument web* would link argument data together by support and conflict with a topic and the claims that provide evidence for each side of a topic. *The argument web* is based on the AIF ontology

due to its separation of information and reasoning which provide a useful interactivity when linking an argumentative conversation.

AIFdb as mentioned is a web platform for hosting argument maps and corpora within the Argument Interchange Format and aims to simplify the process of corpora creation and editing in order to collect and share corpora for free use (Lawrence et al., 2015). Specifically Lawrence mentions AIFdb Corpora which extended the initial functionality of AIFdb by providing users with a platform to create these corpora from argument maps within the database, search for named listed corpora, and to export a large map of the corpora as a visualisation method.

To construct a SADFace corpus, several argument maps could be collected from pre-existing corpora from the same domain. This could be done through various databases however two major databases of AIFdb and AraucariaDB compile Argument Interchange Format and Argument Markup Language respectively - if one has more advantages to the other it could be used to generate a corpus more effectively through self made conversion tools or hand done conversions reducing the chance of invalid/incorrect SADFace files being created.

The features contained within AIFdb such as search interfaces for individual nodes and maps (Figure 2.2) but also mentions the downside of no way to group argument maps together by relation. AIFdb allows users to easily view argument maps through an internal interface which displays any selected map and allows output to several file types (SVG, DOT, JSON, RDF) (Lawrence and Reed, 2014).

Challenges within argument mining come from a lack large datasets of annotated argument maps, these argument databases aim to alleviate the shortcomings presented by a lack of data. Corpora created for the purpose of annotating specific fields are also being created like Green (2014) who endeavours to construct a corpus of scientific papers in the biomedical field of research, and Walker et al. (2014) who annotate arguments leading to either successful or unsuccessful outcomes within judicial settings specifically in relation to vaccination compensation - this can draw from both scientific and not based reasoned arguments.

Wachsmuth et al. (2017) aimed to develop a framework search engine specifically for argumentation which became the previously discussed *args.me*. *args* aimed to create a user process in several stages:

- A user will query a specific field
- *args* retrieves arguments and ranks them by score (specifically using

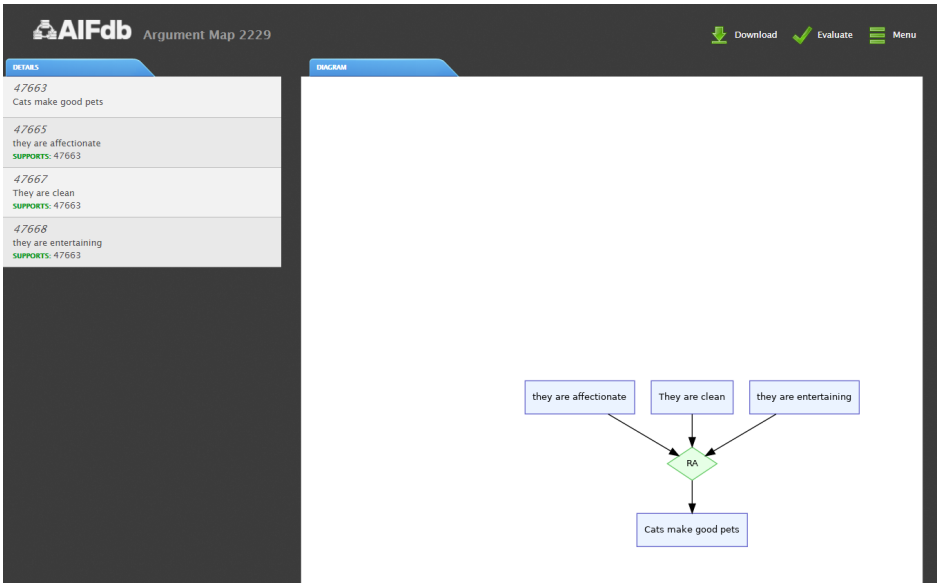

Figure 2.2: Argview within AIFdb

the PageRank method - see Page et al. (1999))

- Arguments are then presented to the user by pro or con

As opposed to AIFdb which searches corpora by words contained, *args* searches for arguments themselves meaning neither have direct competition to the other and are rather complimentary to the argument search process.

To construct a small SADFace corpus for this project, one database was to be chosen as this would allow an increased ease of conversion from one data interchange to another. Opposing the use of Araucaria is the official site - http://araucaria.computing.dundee.ac.uk/ - does not load within reasonable time, AIFdb can take quite notable periods of loading times but does eventually properly load and allow use of the interface where Araucaria has not provided consistent access. AIF being close in design to SADFace pointed to AIFdb being used as a basis from which maps would be converted into SADFace formatting which is why AIFdb was ultimately selected as a selection ground.

Similar to AIF's relation to AIFdb, SADFace is used in the datastore ArgDB which assists in the creation and management of SADFace datasets. ArgDB can be instanced for reasons such as security or testing in order to store datasets separately should they require (Wells, 2020a)

2.7 String Comparison

A major component of argument comparison methods involves string comparisons - where two strings of text are compared through an algorithm to receive an applicable output. Sun et al. (2015) discusses several methodologies for string similarity evaluations. Sun researches differing string similarity metrics which calculate the syntactic similarity of two inputs, these metrics can be divided into *Character-based Similarity* and *Token-based Similarity*.

Character based metrics compare similarity dependant on the sequencing of characters across the strings.

Wei (2004) discusses *edit distance*, a method initially developed by Levenshtein. Levenshtein implementations are powerful due to efficiency and the low use of computational power but Wei believes the lack of contextual analysis for words in determining what actions of edit distance calculation are employed. For context, Levenshtein distance measures how many "edits" must be performed in order to transform one string to another through methods of insertion, deletion, and replacement. Wei based the work on Markov Random Field (MRF) theory, a set of properties which may define a random field of positions which is representative of dependencies - an estimate of the most desirable configuration of MRF is calculated by assigning "cliques" which affect how desirable a connection between two fields are, this is known as MRF-MAP (Most a Posteriori). Dynamic programming is utilised within Markov Edit Distance (MED) to find the minimum distance via assigning "cliques" to the properties of editing within Levenshtein to generate two effective MEDs, one which handles situations where an input is simply a shuffle of the other, and one where generalised situations requiring substitution, deletion, and replacement and using the MEDs to calculate the minimum number of actions required.

Token based metrics first break apart strings into tokens by various metrics such as words (or specific keywords), phrasings, or punctuation then compare the tokens.

Through use of a large text corpus, *Word2Vec* employs a neural network based approach. Word2vec aims to better represent word data by categorising word tokens by assigning a numeric vector to them - the more related the word the closer the vectors are. Ayyadevara (2018) discusses word2vec through explanation of its functionality while the depth that the functionality adds to the method. The ability to calculate the relatability of differing words that do not directly equal in meaning, but infer the same expression, can be a powerful tool in string comparison.

Word2vec has been extended to sentence2vec which implies the same idea but instead of learning from word representations, the algorithm learns from whole sentences and then even further doc2vec which learns from entire documents. These extensions can assist in capturing deeper relations between entire phrasings or the implication derived from a statement as a whole.

Sun et al. (2015) later discusses hybrid metrics which evaluate tokens by character such as *Soft TF-IDF with Lin and Levenshtein (SLL)* which evaluates tokens by the Lin metric to return a threshold then Levenshtein within this threshold, showing how while many techniques in both character and token based evaluations exist - they are not necessarily exclusive. Sun compares the performance of these metrics by measuring precision - the number of correct outcomes for the positive class, recall - the number of correct predictions for the positive class out of all possible positive predictions, and F-measure - a comparative ratio of precision and recall. This is in line with OAEI's evaluation criteria (see Chen and Zhang (2018)). Sun concludes that hybrid metrics do not improve performance by very much if at all but also require more computational processing time and resources, also that certain techniques that may use WordNet databases (popular in various token based metrics) also require more computational processing time.

WordNet evaluations are popular within token based metrics, Budanitsky and Hirst (2006) evaluate five different lexical distance determinants which use WordNet as a core function. Budanitsky first must define how to compare these outcomes and how an algorithm differs from another in output based on the conditions they are processing. Evaluation was completed through three methods:

- An examination of mathematical properties which are desirable, based on work by Wei (1993) and Lin et al. (1998)
- Compare to human judgement, difficulty with this evaluation comes from a lack of large datasets on the topic
- Evaluate performance measures by segmentation and exclusion of individual systems used by a method

Methods such as Lin's Universal Similarity Measure (Lin, 1997) and Resnik's Information-based Approach Resnik (1995) are ran through these evaluations to which the author concludes natural language processing methodologies would benefit from generalised implementations of word *relatedness* are more beneficial but less available and simplistic than word similarity.

Kysela (2018) compares string similarity algorithms specifically for the purpose of harmonising differing names given to locations (Points of Interest (POI) as refered to by Kysela) on geosocial networks. Kysela utilises Jaro-Winkler, Levenshtein (plus Damerau), Jaccard, and Cosine similarity algorithms. Kysela noted that the output of Levenshtein and Damerau-Levenshtein differ from the others used as the former output an integer distance and the later output a ratio, therefore the formulas used were redesigned to output a ratio as this would allow the algorithms to become more comparable on a larger scale. Kysela concludes that Cosine similarity best identifies outcomes in the situation described - where a desired outcome is already identified, but is reliant on an appropriately selected threshold to achieve the greatest number of correct matches while minimising false outcomes.

Another string comparison evaluation was undergone by Cohen et al. (2003), who evaluated comparison methods in the context of matching names and records. Cohen tested Soft TFI-DF, Levenshtein (plus Winkler), and Jaro (plus Winkler) and through these evaluations notes Monge-Elkan has the best outcome of character based metrics, but Soft TFI-DF has the best outcomes of both token based, and hybrid based metrics which coincides with Sun et al. (2015) who stated hybrid not being more effective than character or token measures. For this context, Cohen concluded string distances to be not useful to in comparisons to entities with complex structures (unlike AIF or SADFace having cohesive and understood implementations), although contributes this to a lack of datasets formed into records which equates to less robust testing methods as comparisons cannot easily be made through various dataset testings.

A lecture by Znamenskij (2015b) (which discusses examples similar to their paper Znamenskij (2015a)) notes several shortcomings within string comparison methods such as Damerau-Levenshtein which utilises a longest common subsequence (LCS) selection method which lacks the ability that common substring methods have to detect large character block relocations which can reduce the number of edits that must be made. Utilising number of common substring (NCS) methods can be useful in situations of restructured sentence comparisons as they can cut down several dozen edits to one or two - which would far more accurately depict the similarity calculated output.

2.8 Python for Natural Language Processing

A large amount of NLP projects at least in part involve Python for several reasons. Python's power comes from its ability to reduce development time in part due to the languages numerous standard libraries which extend the language by adding iterators, collection tools, generators, and so forth (Sarkar, 2016). Several external libraries like the popular Natural Language Toolkit (NLTK) developed for use within NLP also add to the weight Python can contribute to a project in this field. A paper by Wagner (2010) discusses the book Natural Language Processing with Python (Bird et al., 2009), Wagner notes how the book covers both knowledge of general Python use alongside NLP techniques - this allows the book to be more applicable to a larger community of natural language programmers that may lack experience in Python which would help draw more work in NLP towards the direction of the inclusion of Python. Bird et al. (2009) breaks down the NLP process by chapter discussing how Python's ontology can be effectively used to classify and structure analysed and annotated text inputs, such as part-of-speech taggers which attaches a part of speech tag (noun, adj, verb, etc.) to each word in an input sentence - words are tokenized for this processes in NLTK.

Similarly "Python Natural Language Processing" (Thanaki, 2017) discusses how Python can be used to comprehend the meaning behind a sentence by analysis techniques such as morphological analysis (morphology being the study of linguistic structures and formations), alongisde syntactic and semantic analysis. Thanaki (2017) discusses the importance of text preprocessing which typically involves various methods to remove fuzziness from text such as stemming where a word is reduced to its stem or "root" and lemmatization in which words are reduced in consideration of the canonical structure of the word, its "base". The advantages of either are that stemming is an algorithmic process making it faster, where lemmatization utilises a word corpus to identify the base of the word which makes it more accurate.

As mentioned, one of Python's greatest assets in natural language programming is the Natural Language Toolkit (NLTK). Papers by the creators Steven Bird (Bird, 2006) and Edward Loper (Loper and Bird, 2002) have been published which directly discuss the purpose and advantages of the library. Bird discusses the need for NLTK to assist in reducing the pipeline for effective language analysis, how Perl has advantages in string processing and Prolog for parsing text, a tool which could be used within Python would cut the effective three languages down to one for projects in NLP. Python was selected by Bird due to the ease of learning the language and Python's strength in a shorter development cycle and ability to be consistently updated along

with the numerous graphics libraries which can be used to create interfaces for user experiences, in addition Python's syntax allows an ease of creating documentation and comprehension of code. Python is also applicable to *Object Oriented Programming* as stated by Loper and Bird (2002), which has various strengths over others such as procedural or functional programming languages due to modularity and flexibility. Loper and Bird designed NLTK around simplicity, ease of use, and consistency which allowed the library to grow into a reliable and powerful tool. In the initial paper (Loper and Bird, 2002), Loper and Bird conclude how NLTK can be utilised by both computer scientist and linguists and was designed for this purpose by considerations such as Python's selection for language.

Later Bird (2006) discusses the developments NLTK received in years post-delivery, Bird mentions how the structures within NLTK became more complex which required an introduction of architecture for tokens to unify data types - however this was considered over complicated for programmers forcing them to use "awkward code structures" (Hearst, 2005). Hearst mentions how the alterations made resulted in several drawbacks in the context of the course they ran. NLTK's releases did not correlate with the documentation released resulting in inconsistencies such as lacking functionality that was stated as present. These criticisms refined the development of NLTK to become more simplified as in line with the original intention. This reveals the worth of product delivery for limited specific uses with the intention of further development as a tool in NLP can be enhanced to be more applicable and become a strong component of the NLP development pipeline.

Libraries outside of Python are also used however, the Stanford CoreNLP Toolkit is a Java developed technology library. Manning et al. (2014) present CoreNLP and discuss its process of annotation pipelines. Like NLTK, CoreNLP has been designed for ease of access and comprehensibility taking inspiration from Bird et al. (2009) concluding that the approach of accessibility directly benefits later development. CoreNLP also received an update to better simplify its system architecture. CoreNLP has various advantages to NLTK but requires more resources due to several features such as a probabilistic parser as compared to NLTK requiring a specified grammar parser, being Java based also slows down development and testing time which is undesireable for this project.

Python is also able to parse JSON quite well Droettboom et al. (2015) discusses JSON in the context of specific languages. Python has a JSON module in its standard library and maps the data types from JSON to Python. Data types with differences are such as Python utilising "dicts" or dictionaries over

"objects", each uses these slightly differently such as Python dictionaries being able to use any hashable input as a key whereas JSON requires keys to be of string type.

2.9 Summary

From the discussed literature we can draw several conclusions which can provide insight to valuable choices which can be made for development in this project, literature was reviewed which would cover each facet of implementation from what language to use, why SADFace has been selected, and how string comparators have been evaluated in the past - this can also lead us to discuss future development past this project for improvements and algorithms which could enhance the deliverable.

For Corpora, a larger dataset is preferable but not at the cost of mismatching topic/domain, research done by groups such as Visser et al. (2018) can help with the creation of large interlinked corpora with matching commonalities but currently an available implementation with a large accessible dataset with a usable interface is not available. Online corpora communities such as *arg.me* (Ajjour et al., 2019) are beneficial to the creation of growing, evolving argument based corpus but would require a conversion to SADFace tool to be used in this project effectively. Testing done on smaller corpora are more indicative of the need for deeper research to draw robust conclusions rather than conclusive work themselves (Reed, 2006) but initialisation of a project is a necessary first step to begin this research which brings the implication that for this project, several hundred files may be unnecessary for initial construction and time in the development phase could be better spent. Large corpora also come with difficulty of flawed or non-reasoning based arguments which can skew data for the context of argument research.

This project is undertaken to contribute to the argument mining community but is not exclusive to this field. An explanation of the field and its components was completed in order to provide sufficient context for the reason for implementation and how the deliverable may be utilised within a specific field. Several advancements in argument mining are being made with pushes towards a machine learning based approach (Lippi and Torroni, 2015). Lawrence and Reed (2020) provide an explanation for the components of argument mining in general theory with descriptions for how each section of analysis plays into the overall process. Opinion mining is also mentioned, which this deliverable is also applicable to use in as a discussion of opinions

may still be mapped onto an interface - the differences between are relevant to the specific fields but not to the process of extraction and comparisons which we aim to accomplish.

Argument Data Interchanges are a format used within argument mining, the most popular being AML and AIF. AML however, is quite rigid and clearly implemented for the specific use within Araucaria making it difficult to work with externally (Wells, 2020a). AIF is quite usable having a dedicated database from which argument maps can be exported into a number of different file types - but the lack of defined type and complex ontology can make comprehension of each faculty within AIF difficult to grasp fully (Chesnevar et al., 2006). AIF has benefits over AML due to receiving further enhancements externally to the ontology (Reed et al., 2008), an adapting interchange can be more useful to a testing environment than a rigid one. SADFace is a data interchange defined within JSON format constraints in line with the rules of AIF making it simplistic to being working with (Wells, 2020a), the main downside to SADFace being the lack of a large dataset of documents.

SADFace is used in this project but to counteract the issue of an available dataset, one will be created. This could be done from scratch or we could use available argument databases such as Araucaria and AIFdb. Previous discussions of the advantages of each data interchange show AIF to be more applicable towards SADFace. We discuss databases such as *arg.me* for the purpose of explaining the process of developing an argument database and how this can affect the inputs received (Wachsmuth et al., 2017), *args* requires users to state pro or con but this could be changed to a scale or outright removed - the data held within a database is affected by its constraints. A database can also be used differently by those who utilise it, *args* searches by topic and argumentation whereas AIFdb searches by words contained within a corpus - this could potentially effect the results of a dataset acquired from multiple sources therefore the best choice may be to utilise a singular database when creating a SADFace corpus.

The comparison of arguments within SADFace files will be done through string comparison algorithms, therefore literature was selected both for multiple methods which are evaluated, and literature discussing individual methods in depth to provide context for advantages to certain methods. Various types of comparison exist - namely character and token based, hybrid as well but as noted by Sun et al. (2015) these do not return more favourable results by enough of a margin to equate to their increased computational usage, this is also evident by results from Cohen et al. (2003). Comparisons such as Markov Edit Distance (Wei, 2004) and the algorithms used by Budanitsky

and Hirst (2006) are powerful and would contribute well to the project - but implementation of more advanced algorithms falls outside of scope therefore the more preferred implementation would be using simplistic but popular algorithms. More simplistic algorithms tend to be of character based metrics rather than token based, as used by Znamenskij (2015b), an NCS (Number of Common Substrings) implementation could generate more accurate overlaps by providing accurate detection of rephrased sentences. Token based metrics could utilise more powerful algorithms but may require too many external libraries or computational power to be used in a sufficient manner, therefore they will be considered for post-implementation should there be opportunities for implementation later on.

Choices for language used are between Python and Java. A number of factors affect this choice such as development time (Loper and Bird, 2002), strength of text analysis (Sarkar, 2016), and usability of the languages respective NLP toolkit - while this may not affect this deliverable, this is worth consideration for longevity of the project. Python is selected, primarily due to development time for this thesis, but also the wide range of literature on NLTK implying an increased use of which in research which further implies more external documentation to be available and an active community to refer to work of should this library be used within the deliverable lifecycle at all.

Chapter 3

Design

3.1 Introduction

During the literature review, no papers were found directly working to identify overlaps by individual nodes in a map. Therefore there was no known process to derive the development process from, however literature relating to the specific individual components of the project are taken into consideration.

3.2 Corpus

This project requires multiple processes in order to meet the aims as described. A corpus must be created to be tested and analysed, from literature - AIFdb was selected as the source for this corpus. Several large corpora are available such as argument maps from "Moral Maze" BBC Episodes which specifically make arguments over topics from morality, to arguments over vaccination. While these corpora deal with important societal issues and will contain deep and meaningful arguments - they may also contain arguments which are difficult to classify as overlapping or not. Simplistic arguments were taken from generic maps such as the discussion of pets - arguments pertaining to simplistic topics are more suited as we must be able to concretely identify if an argument is similar enough to be described as containing overlapping ideas or not. AIFdb's search feature was used to find maps containing references such as "pet" "cat" or "dog", these maps were then exported to JSON format and edited to become readable - no text was altered outside of formatting. Shown is an example of AIF JSON:

```
{"nodes":[
    {"nodeID":"47663","text":"Cats make good pets",
        "type":"I","timestamp":"2015-02-05 09:01:50"},
    {"nodeID":"47665","text":"they are affectionate",
        "type":"I","timestamp":"2015-02-05 10:22:38"},
    {"nodeID":"47666","text":"RA",
        "type":"RA","timestamp":"2015-02-05 10:22:38"},
    {"nodeID":"47667","text":"They are clean",
        "type":"I","timestamp":"2015-02-05 10:22:38"},
    {"nodeID":"47668","text":"they are entertaining",
        "type":"I","timestamp":"2015-02-05 10:22:38"}],
"edges":[
    {"edgeID":"64930","fromID":"47665","toID":"47666",
        "formEdgeID":null},
    {"edgeID":"64931","fromID":"47666","toID":"47663",
        "formEdgeID":null},
    {"edgeID":"64932","fromID":"47667","toID":"47666",
        "formEdgeID":null},
    {"edgeID":"64933","fromID":"47668","toID":"47666",
        "formEdgeID":null}],
"locutions":[]}
```


As previously discussed, nodes contain the actual data from an argument claim, or display the relation between claims. This distinction is identified by the "type" variable within individual nodes.

- *i-nodes* contain information - the claims of an argument.
- *RA-nodes* or Inference signify a relation of support between claims
- *CA-nodes* or Conflict denote a relation of disagreement between claims
- *PA-nodes* or Preference identify a preferred claim to another - typically evident of an argument from an expert compared to one of a non expert in the scenario of conflicting claims.

Edges connect nodes to each other, therefore they require a source and target ID for each node, this also displays the flow of an argument from how a claim leads to another.

Figure 3.1 displays the previously shown JSON AIF map as is shown within AIFdb. As seen from the map and JSON, the nodes "they are affectionate", "they are clean", and "they are entertaining" all connect to an

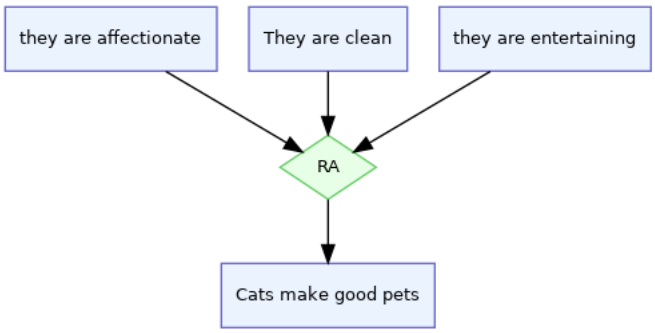

Figure 3.1: Argument Map 2229 as displayed within AIFdb

Figure 3.2: Argument Map 12527 as displayed within AIFdb

RA-node type as they all contribute to support of the claim "cats make good pets" which the RA-node is connected to.

The AIF corpus created contains arguments similar to map 2229, in the case where arguments are comprised of nodes multiplicative several times over (argument map 12527 contains around 12 times more nodes), these arguments will be broken up into smaller components where possible. Map 12527 is very large but comprises of singular arguments that contribute to the same claim - almost like a merge of different arguments, this would be relatively obvious where to separate the map.

3.2 shows map 12527, as seen contains more than 30 arguments non inclusive of the scheme nodes which connect the claims. 3.3 displays a segmentation of this map, which concludes in a point that contributes to the final conclusion but contains enough information to be considered an argument of its own.

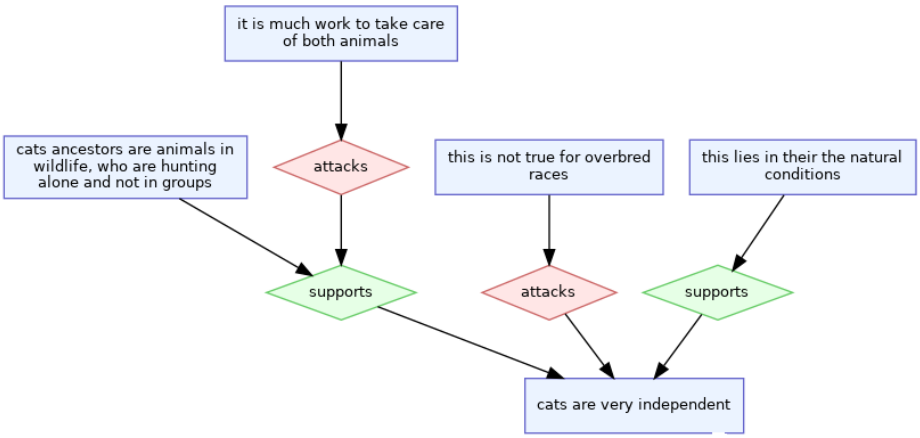

Figure 3.3: Part of argument Map 12527, broken into an individual argument map

3.3 AIF-to-SADFace

Once compiled, the AIF corpus will be converted into SADFace formatting, an implementation of an automatic converter by script could assist the corpus creation process. This script would be non-essential to the project as a whole but would provide future work an ease of conversion so time is set aside to develop a script to automatically convert identified files - a limited amount of development time will be spent due to the non-essential nature so to provide adequate time to the actual implementation.

A pseudocode mock-up of this functionality was created to illustrate how this feature could be implemented:

```
DEFINE AIF filepaths
DEFINE SADFace output path

FOR EACH in AIFpath
      open AIF as READ
      open SADFace as WRITE

      AIF SPLIT by comma (,)

      nodes ARRAY
      edges ARRAY

      FOR EACH line
```

```
        STRING thisSADnode NULL
        IF line CONTAINS "nodeID"
            STRING id = STRING AFTER '"nodeID":'
        IF line CONTAINS "text"
            STRING text = STRING AFTER '"text":'
        IF line CONTAINS "type"
            IF STRING AFTER "type" = "I"
                STRING type = "atom"
            IF STRING AFTER "type" = "RA"
                STRING type = "support"
            IF STRING AFTER "type" = "CA"
                STRING type = "conflict"
            IF STRING AFTER "type" = "PA"
                STRING type = "conflict"

        thisSADnode =   "id": + id,
                        "metadata": {},
                        "sources": [],
                        "text" + text,
                        "type" = type

        nodes APPEND thisSADnode

        STRING thisSADedge NULL
        IF line CONTAINS "edgeID"
            STRING id = STRING AFTER '"edgeID":'
        IF line CONTAINS "fromID"
            STRING source = STRING AFTER '"fromID":'
        IF line CONTAINS "toID"
            STRING target = STRING AFTER '"toID":'

        thisSADedge =   "id" + id
                        "source_id" + source
                        "target_id" + target

        edges APPEND thisSADedge

    thisSADFace =
    {
        edges,
```

```
    "metadata": {
    "core": {
        "analyst_email": "",
        "analyst_name": "",
        "created": "datetime",
        "description": " ",
        "edited": "",
        "id": "datetime + filename",
        "notes": " ",
        "title": "filename",
        "version": "0.1"
    },

    nodes
}
```

The aim of this script will be to capture all the data from nodes and edges from within AIF, output them in the accepted format of SADFace and also fill in the relevant metadata. The script does not have to be completely accurate as the intention is to speed up the process of conversion, if the developer needs to check over the files - as long as the overall time is less than doing by hand - the implementation will serve its purpose.

To ensure data integrity, files will be saved in each state.

- Initial AIFdb output - an un-formatted JSON file
- Readable AIF file - indentations and spacings between elements
- SADFace conversion script output
- Cleaned/Hand converted SADFace files

Files are held within their own directory, primarily this is only relevant to the final SADFace files which will be used within the implementation to pull all claims but will serve as evidence of the conversion process required to form the corpus.

3.4 Hand-Detected Overlaps

After the SADFace corpus has been created, files will be manually reviewed in order to produce a list of overlapping claim statements and these will be

contained within a readable file for use within the implementation. An example of a claim that could be said to be overlapping is contained within AIF map 2229 and 12099. Within 2229 - the claim "They are clean", is present alongside 12099 containing "They're extremely clean". Both statements make claims towards cats being clean animals therefore they are said to overlap in the expressed idea. The overlaps will be represented by their ID rather than output of the text.

After conversion, there will be at least 14 SADFace files containing around a total of 80 atom nodes for analyses. Argument maps were intentionally selected with the intention of finding overlaps to provide the aim for at least a quarter of the arguments to have an overlap pair, more is preferable but for implementation a low amount of ambiguity for what defines an overlap is necessary as we will evaluate the algorithms used based on the number of correctly detected against incorrectly detected overlaps.

3.5 Constraints

In order to keep the implementation lightweight and easily installable and usable, the implementation will aim to only use the Python standard library where possible. If a non-standard library is used then it must have sufficient reasoning.

From the standard library, the expected packages to be used are as follows:

- os - used for filepath variables
- json - needed to interact with JSON files (both decode and encodes)
- itertools - a powerful tool for creating efficient loops and iterators
- math - as implied, mathematical functionality such as rounding

Outside of libraries, the intention within development is to make use of Python's Object Orientation to create functions, this will be useful for development and testing as certain features can simply not have their function called in order to test specific implementations. This will also be beneficial to long term development, such as later implementations of new string comparison algorithms.

3.6 Implementation Requirements

In order to meet the aims of the project, at least two string comparison algorithms will be implemented in order to compare the outputs of each and

can be used to identify an increased likelihood of overlap - if both algorithms detect and overlap, this may be more trustworthy than if only one does.

The implementation must:

- Be able to read in, process, and extract nodes from the directed SADFace files - failing this would mean all other functionality is impossible.
- Process each atom node and compare to each other node in the array.
- Process these nodes through the string comparison algorithms
- Identify and store overlaps as flagged by these algorithms
- Compare the hand overlaps to the detected overlaps to identify how many overlaps were successfully, and how many were unsuccessfully, detected
- Output the detected overlaps and their accuracy to the user

The implementation should:

- Be able to read in identified hand overlaps for automatic comparison
- Compare detected overlaps and output duplicates noted - these will then also be compared to the hand overlaps

3.7 Evaluation Metrics

In order to evaluate string comparison algorithm accuracy, we will need to store both hand detected overlaps and algorithm detected overlaps within lists. These lists will hold overlaps in the same way:

```
overlaps = [
[overlap1a, overlap1b],
[overlap2a, overlap2b],
[overlap3a, overlap3b]
]
```

Each entry in a list signals a detected overlap - overlap1a and overlap1b are two nodes which are marked as overlapping. As designed, the implementation will have a list for hand overlaps, each algorithm's output of overlaps, and potentially a list which compiles duplicate detected overlaps from all algorithms used.

```
algorithm1 = [
[overlap1a, overlap1b],
[overlap2a, overlap2b],
[overlap3a, overlap3b]
]

algorithm2 = [
[overlap4a, overlap4b],
[overlap2a, overlap2b],
[overlap5a, overlap5b]
]

both_algorithms = [
[overlap2a, overlap2b]
]
```

After processing, each of these lists would be compared to a hand overlap list much in the same way an algorithm comparison list would be created. This would be used to identify what was correctly detected and what wasn't.

```
detected_algorithms = [
[overlap1a, overlap1b],
[overlap2a, overlap2b],
[overlap3a, overlap3b]
]

hand_detected = [
[overlap1a, overlap1b]
[overlap4a, overlap4b]
]

correct_detection = [overlap1a, overlap1b]
incorrect_detection = [ [overlap2a, overlap2b],
        [overlap3a, overlap3b] ]
```

The number of correct and incorrect detections would also be tracked by an integer in order to measure effectiveness, algorithms would need to be given an acceptable range in which to accept an overlap as true, this will be derived from testing and comparing how many correct and incorrect outputs there are and what can be considered an acceptable fault level.

3.8 String Comparison Algorithms

Due to time constraints, and the nature of the project not having direct comparisons, simplistic algorithms will be selected for implementation over more advanced ones. This will lead to likely only character-based metric algorithms as opposed to token-based. This has several benefits like development time and can keep the project within library constraints - several token-based metrics utilise a Wordnet or external library to tokenize sentences and words. Unfortunately this would mean the algorithms are not as accurate however this would be outside of the project scope and therefore has been identified as a "stretch goal" to be implemented if constraints allow.

The algorithms selected will be utilised by passing extracted arguments in pairs. A loop which compares each argument extracted will be implemented - this has been designed with pseudocode:

```
ARRAY test1list
ARRAY test2list

FOR token1, token2 IN arguments
    test1 = algorithm1(token1, token2)
    IF test1 < acceptable_distance
        [token1, token2] APPEND to test1list

    test2 = algorithm2(token1, token2)
    IF test2 < acceptable_distance
        [token1, token2] APPEND to test2list
```

3.9 Implementation Layout

As illustrated by figure 3.4, the created process will have several functions in order to meet the aims of the project. Functions for reading in SADFace files and compiling the argument atom nodes into an arguments list, and to read in hand detected overlaps into a list will be used - it should be noted in a final deliverable version of this project the "hand overlap" related functions simply need to be removed or not called. The arguments list will then be looped to compare each possible combination of two arguments and these two will be processed through each string algorithm. The output lists built through string algorithms will then be compared to the hand overlaps list, this will then indicate the number of correctly and incorrectly identified overlaps by each algorithm, if possible functionality to compare the outputs of algorithms will also be completed and evaluated in the same way. An

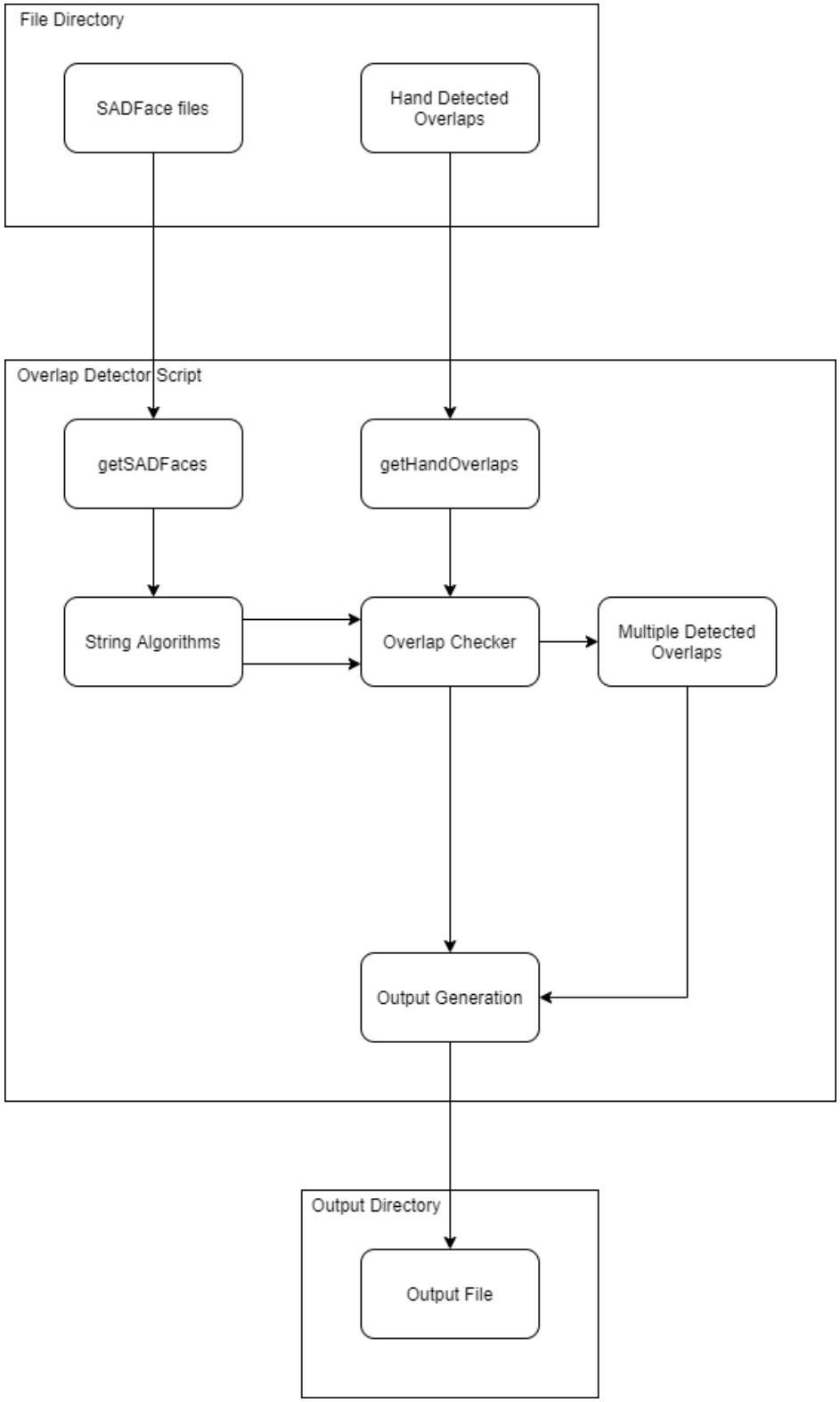

Figure 3.4: Designed layout of the project functionality

output file (likely a text file output) will then be created which lists each outputs of each algorithm and the detection numbers related.

3.10 Summary

The implementation process will include several facets:

- Creation of an AIF corpus
- Implementation of an AIF-to-SADFace converter script
- Creation of a SADFace corpus
- Creation of a list of hand detected overlaps by ID
- Desired implementation will utilise a limited number of non-standard Python libraries
- The output of string comparison algorithms will be evaluated by comparing the output lists to the hand overlap list
- The implemented algorithms will be simplistic iterations of character-based string comparisons so the project remains within scope

The implementation will also follow the described design of figure 3.4, this will keep in line with object oriented development and allow an ease of testing functionality in development.

Chapter 4

Implementation

4.1 AIF-to-SADFace

Based on the pseudocode previously designed, an attempt and the implementation of an automatic AIF-to-SADFace converter was created. Ultimately the script was unable to be fully implemented within the given time-frame for development, as further development would risk leaving less time for the actual implementation (See *Code - AIF-to-SADF.py* for the current non-functional iteration of the script). As the script was non functional, selected argument maps were converted from AIF to SADFace by hand, this was done by copying the data over manually from AIF into the relevant sections within an empty SADFace layout.

```
{
    "edges": [
        {
            "id": "",
            "source_id": "",
            "target_id": ""
        },
        {
            "id": "",
            "source_id": "",
            "target_id": ""
        }
    ],
    "metadata": {
        "core": {
            "analyst_email": "connor.ness@outlook.com",
```

```
            "analyst_name": "Connor Ness",
            "created": "2021-06-20T13:56:22",
            "description": "",
            "edited": "2021-06-20T13:56:22",
            "id": "2021-06-20-13-56-22-b45e",
            "notes": "",
            "title": "",
            "version": "0.1"
        }
    },
    "nodes": [
        {
            "id": "",
            "metadata": {},
            "sources": [],
            "text": "",
            "type": "atom"
        },
        {
            "id": "",
            "metadata": {},
            "sources": [],
            "name": "support",
            "type": "scheme"
        }
    ]
}
```


This made conversion simple as from an AIF file, the "nodes" section simply had to have the ID and text placed within the relevant fields within SADFace "nodes", type must be defined as well. Any *i-node* in an AIF would be an "atom" type in SADFace, any *RA, CA, or PA -node* would become a "scheme" type in SADFace with the "name" variable being either support for RA, or Conflict for CA/PA nodes. Edges were simplistic to translate, the edgeIDs were inserted into the "id" under "edges" in SADFace, "fromID" became "source_id" and "toID" became "target_id".

The process of building a SADFace corpus was more time consuming than if a readily available converter was available so a smaller corpus was created in the time frame than if the script was functional.

4.2 Implementation Layout

The implementation stayed as close to the initial design of functionality as possible, minor changes were made and a small amount of code used outwith functions was used, primarily to loop through the argument comparisons and to decide if the function called returned a value which could be considered as an overlap.

Figure 4.1 provides a function diagram detailing variables passed to each function and illustrating the files retrieved by functions getSADFaces and getHandOverlaps.

4.3 Reading in SADFace and Hand Overlaps

The script begins by first retrieving the hand detected overlaps. This is done by locating file "overlaps.txt" by use of the *os* library. The overlaps.txt file contains a line for each overlap, by the SADFace "id" of each atom node. This list is representative of the overlaps in claims within the SADFace files as detected by the author

```
2229_n_a1 2235_n_a1
2229_n_a1 12099_n_a2
2229_n_a1 12527_n_a1
2229_n_a1 self_2_n_a1
2235_n_a1 12099_n_a2
2235_n_a1 12527_n_a1
2235_n_a1 self_2_n_a1
12099_n_a2 12527_n_a1
12099_n_a2 self_2_n_a1
12527_n_a1 self_2_n_a1
2229_n_a2 2235_n_a2
2229_n_a2 12099_n_a10
2229_n_a2 self_2_n_a4
2235_n_a2 12099_n_a10
2235_n_a2 self_2_n_a4
12099_n_a10 self_2_n_a4
2229_n_a3 12099_n_a9
2229_n_a3 self_2_n_a3
12099_n_a9 self_2_n_a3
2229_n_a4 12099_n_a11
12099_n_a4 12527_1_n_a1
12099_n_a4 12527_n_a2
```

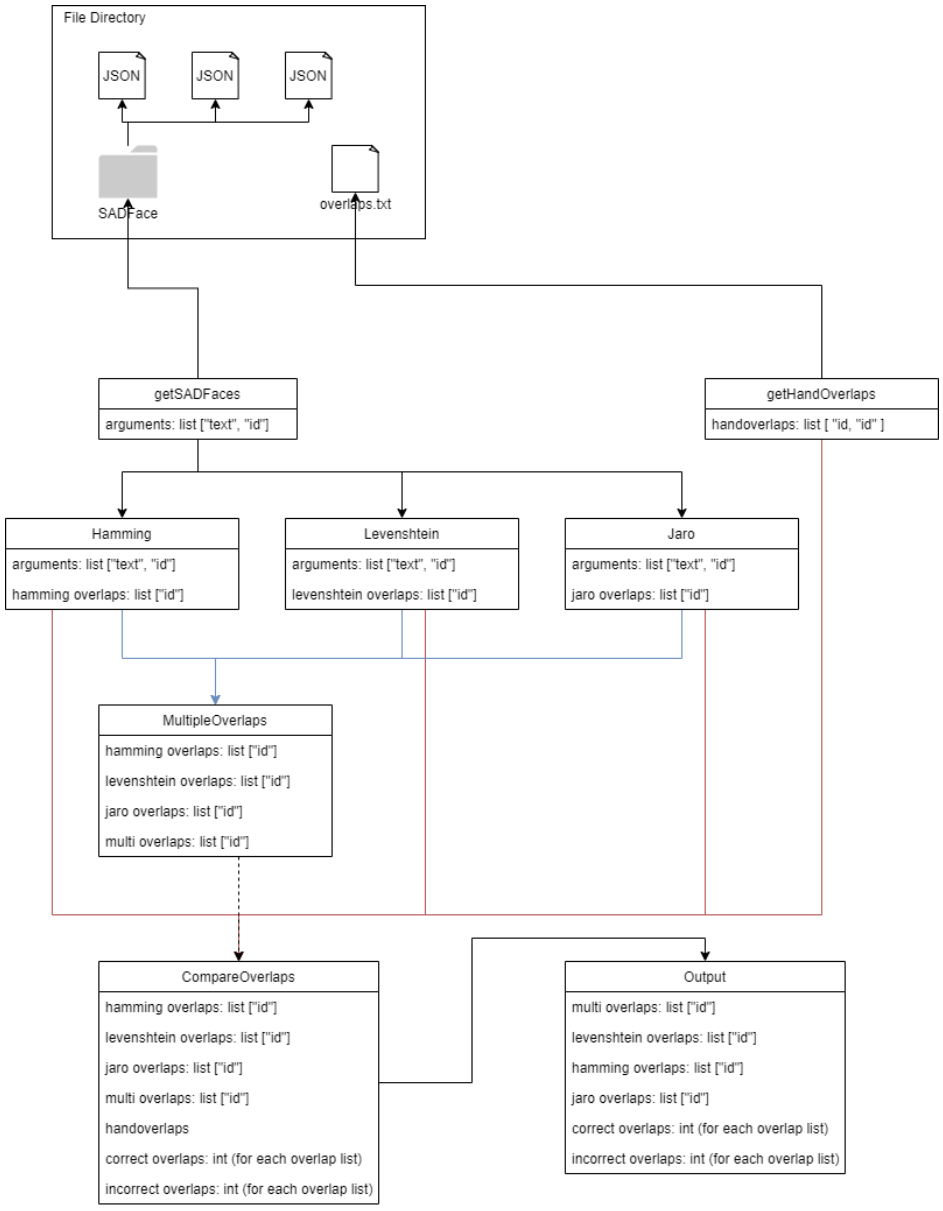

Figure 4.1: Function overview of OverlapDetection.py

```
12099_n_a4 self_2_n_a5
12527_1_n_a1 12527_n_a2
12527_1_n_a1 self_2_n_a5
12527_n_a2 self_2_n_a5
12527_n_a7 12527_4_n_a1
12527_n_a7 self_2_n_a6
12527_4_n_a1 self_2_n_a6
12527_n_a8 self_2_n_a2
12527_n_a8 12527_4_n_a2
self_2_n_a2 12527_4_n_a2
12527_n_a4 12527_2_n_a1
12527_n_a4 self_2_n_a7
12527_2_n_a1 self_2_n_a7
```

This list is read in to a python list "handoverlaps" which is passed to getHandOverlaps. "handoverlaps" is structured as:

```
[
[2229_n_a1,      2235_n_a1],
[2229_n_a1, 12099_n_a2],
[2229_n_a1, 12527_n_a1],
etc...
]
```

This is done to ensure correct comparisons later on - if the list was structured otherwise the list would need each element to be segmented by *.split()* in each loop.

```
def getHandOverlaps(handoverlaps):
    handoverlapsfilepath = os.path.dirname(os.path.realpath
        (__file__)) + '\\Pets\\overlaps.txt'
    with open(handoverlapsfilepath) as handoverlapsfile:
        handoverlaps.extend([line.split() for line
                in handoverlapsfile])
    handoverlapsfile.close()
```

handoverlapspath is a string variable which uses *os.path.dirname(os.path.realpath(_file_))* to locate the directory that the script is currently running from, this is done to ensure the script can be run from different machines. *+ '\\Pets \\overlaps.txt'* designates the location of the overlaps file, which is held within the corpus main folder. The file is then opened using the filepath specified and then each line is read in as a seperate entry into the "handoverlaps" list.

The nodes from SADFace files are then retrieved using getSADFaces:

```
def getSADFaces(arguments):
    domainpath = (os.path.dirname(os.path.realpath(__file__))
        + "\\Pets\\SADFace\\Hand Done\\")
    SADFaces = os.listdir(domainpath)

    for each in SADFaces:
            with open(domainpath + each) as SADFace:

                data = json.load(SADFace)
                nodes = data.get('nodes')
                count = len(nodes)

                for i in range(count):
                    thisarg = []
                    try:
                        thisarg.append(data['nodes'][i]['text'])
                        try:
                            thisarg.append(data['nodes'][i]['id'])
                            arguments.append(thisarg)
                        except KeyError:
                            print("no id")
                    except KeyError:
                        pass
```

Similar to getHandOverlaps, *os* is used to retrieve the file locations to fill "SADFaces" which is a list of all files within the directory by *SADFaces = os.listdir(domainpath)* - it should be noted if a non SADFace file is within the directory an error will occur. getSADFaces also receives the list "arguments", this will hold each nodes ID and text data. The script loops through each file and decodes the JSON by *data = json.load(SADFace)*, nodes are then retrieved by *nodes = data.get('nodes')*, then the number of entries within "nodes" is calculated by *count = len(nodes)*.

Using this count will allow the retrieval of information within each node, the for loop builds an individual argument to be appended to "arguments" list. try excepts are used when attempting to retrieve both "text" and "id" to allow the script to continue if the current node is not an atom, but instead a scheme which would not contain a "text" variable. This is why "text" is retrieved first, if no "text" variable is present then the node is not an atom so there is no need to retrieve "id" so the next node should be checked instead.

data[nodes][i][VARIABLE] is used to access the indicated variable "text" or "id" in each node which is automatically appended to the "thisarg" list.

Figure 4.2 illustrates the path taken by functions getSADFaces and getHandOverlaps within OverlapDetection.py.

4.4 AIF-Compatibility

While developed around SADFace and other data interchange formats compatibility being out of scope, additional development time allowed specifically AIFdb's JSON export from AIF maps to be applicable to the tool. This was able to be included due to the similar principles used by SADFace based upon AIF. Both SADFace and AIFdb JSON use "nodes" as the object so *thisarg.append(data['nodes'][i]['text'])* did not need to be change. The main issue with compatibility is the node types, in SADFace - scheme nodes contain no 'text' variable so these nodes are automatically skipped over however in AIFdb JSON scheme nodes do contain a 'text' variable, the difference is noted by the 'type' variable which will indicate *i-nodes* for example by 'type': 'I'. To accept both SADFace atoms and AIF's *i-nodes* only, the implementation of 'getSADFaces' was changed to 'getArguments' and a change was added:

```
try:
        thisarg.append(data['nodes'][i]['id'])
        arguments.append(thisarg)
except KeyError:
        try:
                if(data['nodes'][i]['type'] == "I"):
                        thisarg.append(data['nodes'][i]['nodeID'])
                        arguments.append(thisarg)
        except KeyError:
                print("no id")
```

By including a nested try/except and an if check for only 'I' type nodes the function is now able to collect both SADFace atom nodes and AIFdb JSON nodes. This addition being out of scope is not an integral component to the project as a whole but was implemented due to the increased compatibility that allowing AIFdb corpora to be analysed brings.

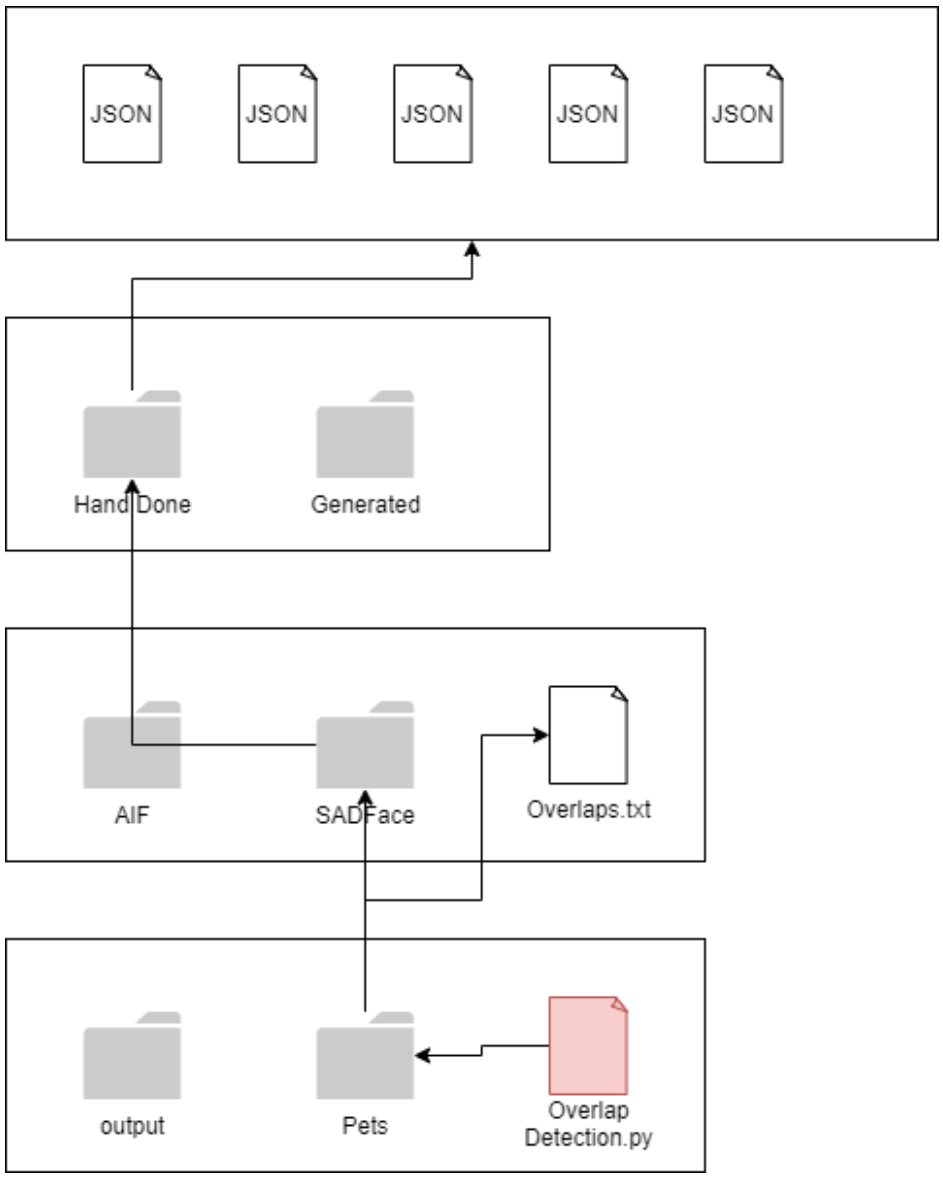

Figure 4.2: OverlapDetection.py accessing SADFace

4.5 String Comparison Algorithms

The algorithms selected for implementation were Hamming distance, Levenshtein edit distance, and Jaro distance. Hamming and Levenshtein return an integer value representative of the number of edits to change a string where Jaro returns a similarity ratio between 0 and 1; 0 being no similarity, 1 being complete similarity.

The *itertools* library is used to loop through the "arguments" list to ensure all elements are compared only once to avoid duplicates. This is done using the *combinations* method, which takes in an iterable (in this case being "arguments") and r which is representative of the length of tuples to be combined (this being 2 as only 2 strings are compared each time).

```
for a, b in itertools.combinations(arguments, 2):
```

From this loop, "a" and "b" are representative of the individual argument elements being compared. Index 0 of a,b contains the "text" variable from a SADFace node, and index 1 contains the "id" variable of the same node. Within this loop a,b are used as variables within each algorithm which are represented by "token1" and "token2" within each algorithm respectively.

4.5.1 Hamming Distance

Hamming distance calculates the number of different characters at equal locations between two strings. The hamming distance between "abc12" and "abd123" is 2 as the strings differ in character 3, and string 2 is one character longer.

Prior to calling the hamming function, the shortest text string between a,b is determined, this is to pass the variables by longest, shortest to the algorithm as the shortest must be used to compare to the longest or otherwise the algorithm returns an error as characters compared are out of bounds. This is done simply by:

```
if len(a[0]) < len(b[0]):
        hamdist = hamming(b[0], a[0])
    else:
        hamdist = hamming(a[0], b[0])
```

Once called, hamming loops through each character for the length of the shortest string, checking for differences in each character at each position. After the loop has finished, if the strings are not of equal length, the remainder of the length of the longest string is added to the distance.

```
def hamming(token1, token2):
```

```
    distance = 0
    for char in range (len(token2)):
        if token1[char] != token2[char]:
            distance += 1
    distance += (len(token1) - len(token2))
    return distance
```

The returned distance is then checked to be low enough to be considered as a potential "overlap" in claims, this will be discussed later as the process to determine acceptable distance was acquired through testing.

4.5.2 Levenshtein Distance

The Levenshtein distance compares the number of edits that must be performed to convert string 1 into string 2, this is done via insertion, deletion, and replacement to characters in a string. The Levenshtein distance between "puck" and "truck" is 2, "p" is substituted for "t" and "r" is inserted into character position 2.

```
def levenshtein(token1, token2):
    distances = np.zeros((len(token1) + 1,
        len(token2) + 1), dtype = int)

    for t1char in range(len(token1) + 1):
        distances[t1char][0] = t1char

    for t2char in range(len(token2) + 1):
        distances[0][t2char] = t2char

    a = 0
    b = 0
    c = 0

    for t1char in range(1, len(token1) + 1):
        for t2char in range(1, len(token2) + 1):
            if (token1[t1char-1] == token2[t2char-1]):
                distances[t1char][t2char] =
                        distances[t1char - 1][t2char - 1]
            else:
                a = distances[t1char][t2char - 1]
                b = distances[t1char - 1][t2char]
                c = distances[t1char - 1][t2char - 1]
```

```
                if (a <= b and a <= c):
                    distances[t1char][t2char] = a + 1
                elif (b <= a and b <= c):
                    distances[t1char][t2char] = b + 1
                else:
                    distances[t1char][t2char] = c + 1

    return (distances[len(token1)][len(token2)])
```

The implemented algorithm makes use of the external library "NumPy", which was utilised outwith the aim of minimising external libraries due to both popularity in usage and its strengths in allowing the creation of dimensional arrays. In the shown algorithm, NumPy is represented as "np" and is used to create a distances matrix. The matrix is created by using the lengths of each token to define the number of rows (token1) and columns (token2) by the characters of the tokens passed in *distances = np.zeros((len(token1) + 1, len(token2) + 1), dtype = int)*. The characters are then filled into their respective position by the for loops *for t1char in range(len(token1) + 1): distances[t1char][0] = t1char*.

The a, b, c variables are used later to compare the characters relevant to the current character comparison, they will be used to find the smallest number of edits needed at each character comparison.

The matrix is then given integers for each row and column which relates to each character in each token. Two for loops are used to loop through each character in each token *for t1char in range(1, len(token1) + 1): for t2char in range(1, len(token2) + 1):*. The statement *if (token1[t1char-1] == token2[t2char-1]):* checks if the characters are equal and therefor not in need of alteration. If the characters are not equal, a,b, and c are set to the values corresponding to the number of edits required, the smallest number of edits is then selected and entered into the current matrix position.

The matrix for "puck" and "truck" is shown:

0	t	r	u	c	k
p	1	2	3	4	5
u	2	2	2	3	4
c	3	3	3	2	3
k	4	4	4	3	2

Several websites and forums were used to assist the creation of this algorithm, primarily the first link shown:
`https://blog.paperspace.com/implementing-levenshtein-distance-word-autocomplete-`

#:~:text=The%20Levenshtein%20distance%20is%20a,transform%20one%20word%20into%20another.
https://gist.github.com/tommy/14631
https://www.cuelogic.com/blog/the-levenshtein-algorithm
https://www.datacamp.com/community/tutorials/fuzzy-string-python

4.5.3 Jaro Distance

Jaro distance compares two strings and outputs a ratio between 0 and 1 which represents the similarity between the strings - 0 being no similarity, 1 being completely equal. The Jaro formula is:

```
Similarity = 1/3 * (m/s1 + m/s2 + (m–t)/m
```

Where *m* is the number of matching characters across both strings, *s1* and *s2* are the each strings respective length, and *t* is the number of transpositions that must be performed on the strings to make them equal.

The Jaro distance between "puck" and "truck" is 0.78333.... as the inputted formula would be:

```
Similarity = 1/3 * (3/4 + 3/5 + (3−1)/3 = 0.78333....
```

```
def jaro(token1, token2):
    maxdist = floor(max(len(token1), len(token2)) /2) −1
    match = 0
    chars_t1 = [0] * len(token1)
    chars_t2 = [0] * len(token2)

    if (token1 == token2):
        return 1

    for t1char in range(len(token1)):
        for t2char in range (max(0, t1char−maxdist),
                min(len(token2), t1char + maxdist+1)):

            if(token1[t1char] == token2[t2char] and
                chars_t2[t2char] == 0):
                chars_t1[t1char] = 1
                chars_t2[t2char] = 1
                match += 1
                break
```

```
if(match == 0):
    return 0
transpositions = 0
point = 0

for t1char in range (len(token1)):
    if(chars_t1[t1char]):
        while(chars_t2[point] == 0):
            point += 1
        if(token1[t1char] != token2[point]):
            point += 1
            transpositions += 1
transpositions = floor(transpositions/2)
jaroratio = (match/ len(token1) + match / len(token2)
    + (match - transpositions + 1) / match)/ 3.0

return(jaroratio)
```

The function can be broken into segments. Where the first nested for loop is done to calculate *m*, *for t1char in range(len(token1)): for t2char in range (max(0, t1char-maxdist), min(len(token2), t1char + maxdist+1)):*. The second for loop calculates the *t*, *for t1char in range (len(token1)):*. *if (token1 == token2): return 1* and *if(match == 0): return 0* are used for the cases of complete similarity and no similarity.

The matches for loop checks for matching characters, and determines if a character has a match by assigning a 1 to the character. This prevents the same character being used for a match more than once.

To calculate transpositions, the standard library *math* is utilised, specifically the *floor* method which returns a rounded down number. This is required as when comparing the two strings within the transposition for loop. The number of transpositions calculated is based on the number of matching characters out of place however if two characters are to be swapped this would be output as two transpositions, where this is actually only one transposition so the number of transpositions is half what is calculated, hence *transpositions = floor(transpositions/2)*. *floor* is used to round down in the scenario an odd number of transpositions is returned as half a transposition is not possible.

Several websites and forums were used to assist the creation of this algorithm, primarily the first link shown:
`https://www.geeksforgeeks.org/jaro-and-jaro-winkler-similarity/`

https://www.kaggle.com/alvations/jaro-winkler
https://rosettacode.org/wiki/Jaro_similarity

4.6 Multiple Overlap Checking

To provide a greater accuracy to the results of each algorithm, once all algorithms have output their respective lists of overlaps, these lists are compared for duplicates. This is done simply by use of *itertools*, specifically the *product* method which returns a cartesian product of all comparable variables. *overlapsA, overelapsB,* and *overlapsC* is representative of Levenshtein, Hamming, and Jaro.

```
def multipleOverlaps(multi_overlaps, overlapsA, overlapsB,
    overlapsC):

    overlaps = itertools.product(overlapsA, overlapsB,
        overlapsC)

    for a, b, c in overlaps:
        if a == b == c:
            multi_overlaps.append(a)
```

By breaking up each overlap product into *a,b,c* they are then easily compared by $a == b == c$, in the scenario all overlaps are equal this means the overlap was detected by each separate algorithm so it is appended to a seperate list "multi_overlaps". This is the list that can be considered a culmination of all algorithm outputs and will provide the most accurate results.

The issue with the function in its current state is that it assumes all algorithms are equal in identifying overlaps, if an algorithm is more accurate than the others - it is still weighed equally against other algorithm outputs.

4.7 Comparing Overlaps

Once all algorithms and the multiple occurrence check has output the list, the list of each function is checked against the list "handoverlaps". Each entry in a detected overlap list is checked if an equal is contained within "handoverlaps". To ensure that formatting does not cause any misses, each index in an overlap is checked against each index in the equivalent "handoverlap" and when both are equal a one is added to the match integer. If both are equal then match will equal 2 and therefor be a correctly detected

overlap and is added to the "correctoverlap" list, otherwise the overlap is added to "incorrectoverlap" - both lists also have a count to represent how many correct and incorrect hits each algorithm has.

```
def compareOverlaps(handoverlaps, overlaps, correctoverlap,
        correctcount, incorrectoverlap, incorrectcount):

    for overlap in overlaps:
        for handoverlap in handoverlaps:

            match = 0

            if overlap[0] == handoverlap[0] or overlap[0]
                == handoverlap[1]:

                match += 1
            if overlap[1] == handoverlap[0] or overlap[1]
                == handoverlap[1]:

                match += 1
            if overlap[0] == overlap[1]:
                match = 0

            if match == 2:
                break


        if match == 2:
            correctoverlap.append(overlap)
            correctcount += 1
        else:
            incorrectoverlap.append(overlap)
            incorrectcount += 1

    return correctcount, incorrectcount
```

4.8 Acceptable Overlap Distance

As previously mentioned, the algorithms return a numerical value describing the distance between strings. In order to determine what value is best repre-

sentative of what can be considered an overlap, this value was changed and tested through compareOverlaps. It should be noted that the decided upon acceptable overlap took into consideration two factors:

- Overlaps are to be compared against each other, so can be slightly more lenient than a standalone output
- The tool is not intended to work autonomously, likely usage in the current implementation is intended to flag potential overlaps, rather than deal with merging them within the tool

The number of maximum correctly detected overlaps is 35, therefore the consideration for each acceptable distance is to be as close to 35, but with a lower number of false matches

4.8.1 Levenshtein

For Levenshtein, the variable *levdist* contained the distance between the compared strings.

```
levdist = levenshtein(a[0],b[0])
    if (levdist <= 10):
        thislev.append(a[1])
        thislev.append(b[1])
        levoverlaps.append(thislev)
```

By changing the compared value within the if statement and exporting the output, comparisons between each value can be made as number of matches against number of false matches

The changes made and their respective output are shown by table 4.8.1 and are graphed on figure 4.3. The selected acceptable overlap was 10 for Levenshtein, this is due to the overlaps detected by lower ranges to be almost exclusively extremely similar statements, but above this range to be too dissimilar, the same method of selection which takes note of the previously discussed reason is applied to Hamming and Jaro.

4.8.2 Hamming

For Hamming, the altered variable is compared to *hamdist*. As previously mentioned, first the strings are compared by length prior to function execution.

```
if len(a[0]) < len(b[0]):
        hamdist = hamming(b[0], a[0])
```

Acceptable Distance	Matches	False Matches
1	5	3
2	5	3
3	5	6
4	5	9
5	5	10
6	5	14
7	5	17
8	7	26
9	7	38
10	10	61
15	14	227

Table 4.1: Levenshtein results

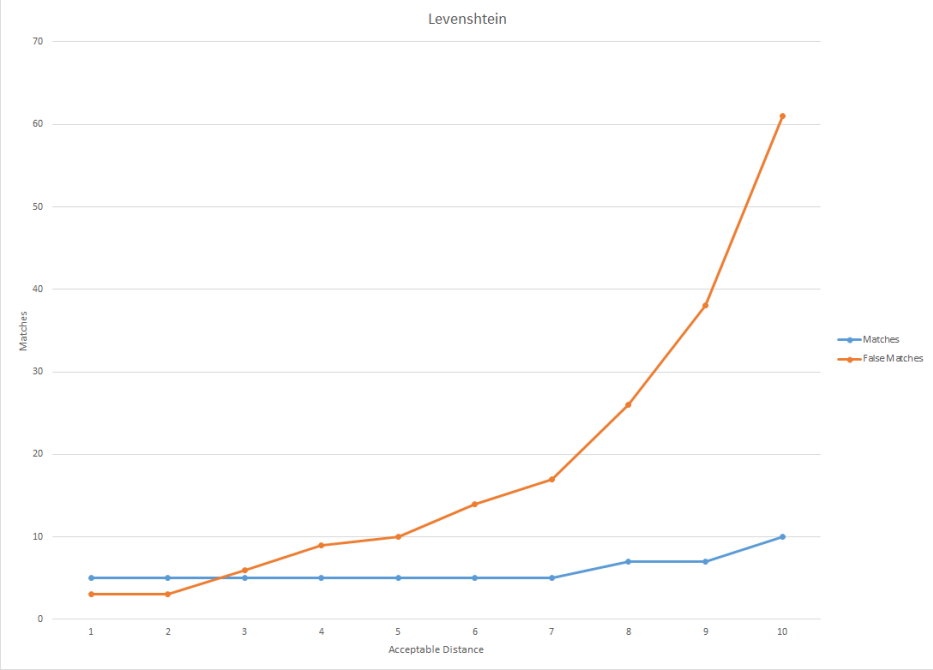

Figure 4.3: Line graph of Levenshtein acceptable distances

Acceptable Distance	Matches	False Matches
1	5	3
2	5	3
3	5	6
4	5	8
5	5	9
6	5	13
7	5	13
8	5	18
9	6	31
10	6	46
11	6	58
12	6	61
13	7	86
14	7	107
15	7	131
16	9	171
17	10	194
18	12	245
19	13	289
20	13	345

Table 4.2: Hamming results

```
else:
    hamdist = hamming(a[0], b[0])
if(hamdist <= 17):
    thisham.append(a[1])
    thisham.append(b[1])
    hamoverlaps.append(thisham)
```

The changes made and their respective output are shown by table **??** and are graphed on figure 4.4. The selected acceptable overlap was 17 for Hamming. Hamming's results are mostly similar for quite a large number of distances, this is likely due to hamming being quite naive as a comparison tool.

4.8.3 Jaro

For Jaro, the altered variable is compared to *jaroratio*. This is the only non integer value returned so the compared value differs here.

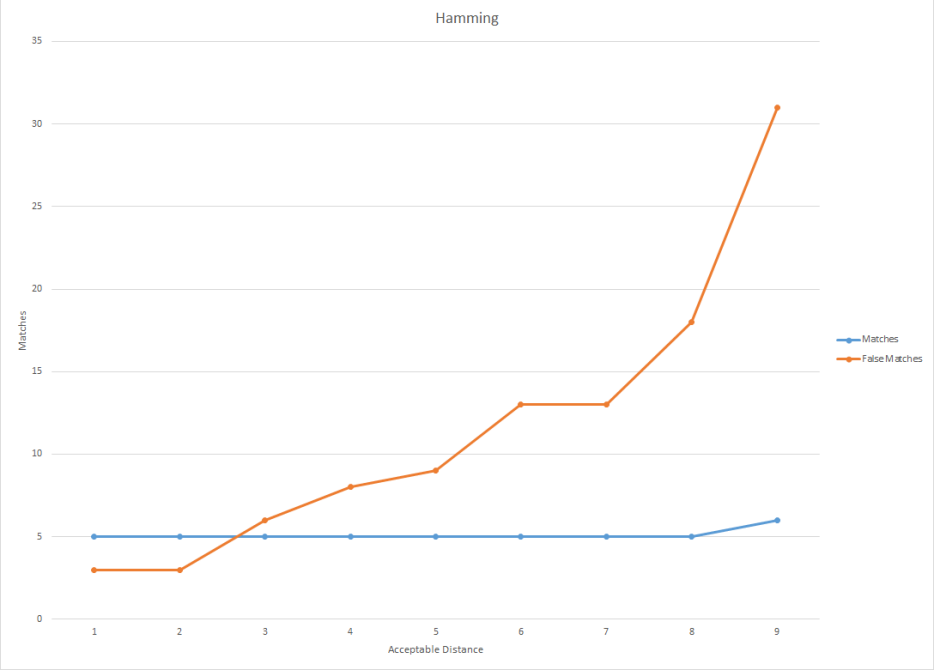

Figure 4.4: Line graph of Hamming acceptable distances

```
jaroratio = jaro(a[0], b[0])
    if (jaroratio >= 0.75):
        thisjaro.append(a[1])
        thisjaro.append(b[1])
        jarooverlaps.append(thisjaro)
```

The changes made and their respective output are shown by table **??** and are graphed on figure 4.5. The selected acceptable overlap was 0.7 for Jaro. Jaro's results are interesting as when as low as 0.5, the algorithm almost detects all 35 overlaps but is however exponential in the number of false matches, this is why 0.7 was selected as quickly after this ratio, the number of false matches increases at a rate far above the number of correct matches.

4.9 Output

4.9.1 Test Output

Once all functions have generated a list of overlaps alongside an integer of correct and incorrect matches, the script then executes output()

```
def output(multi_overlaps, multitrue, multitruecount,
```

Acceptable Distance	Matches	False Matches
0.95	5	3
0.9	5	3
0.85	5	3
0.8	5	4
0.75	6	8
0.7	9	61
0.65	12	400
0.6	17	1262
0.55	30	2506
0.5	33	3244

Table 4.3: Jaro results

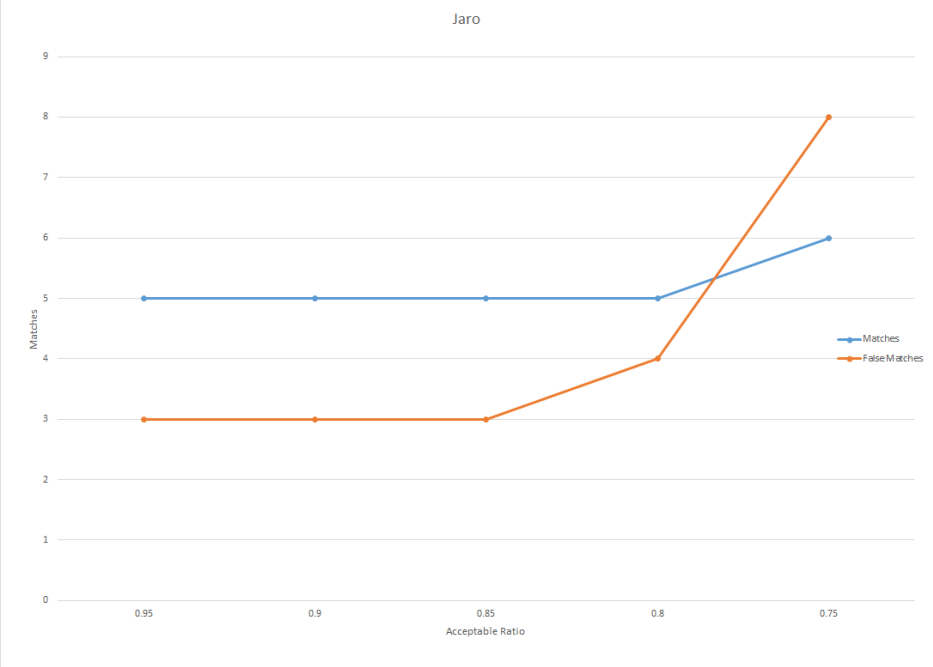

Figure 4.5: Line graph of Jaro acceptable distances.py

```
    multifalse, multifalsecount,
    levoverlaps, levtrue, levtruecount,
levfalse, levfalsecount,
hamoverlaps, hamtrue, hamtruecount,
hamfalse, hamfalsecount,
jarooverlaps, jarotrue, jarotruecount,
jarofalse, jarofalsecount):

currenttime = datetime.datetime.now()
filename = str(currenttime.day) + str(currenttime.hour)
    + str(currenttime.minute) + str(currenttime.second) + '.txt'

filepath = os.path.dirname(os.path.realpath(__file__))
    + '\\output'
if not os.path.exists(filepath):
    os.makedirs(filepath)

line_overlaps = 'Detected by Levenshtein, Hamming, and Jaro:'
with open(os.path.join(filepath, filename), 'w') as f:
    f.write('Detected by Levenshtein, Hamming, and Jaro:\n')
    for overlap in multi_overlaps:
        f.write(str(overlap) + '\n')
    f.write('\nMultiple: \n' + str(multitruecount) +
    ' Detected correctly\n' + str(multifalsecount) +
    ' Detected incorrectly\n')

    f.write('\nDetected by Levenshtein:\n')
    for overlap in levoverlaps:
        f.write(str(overlap) + '\n')
    f.write('\nLevenshtein: \n' + str(levtruecount) +
    ' Detected correctly\n' + str(levfalsecount) +
    ' Detected incorrectly\n')

    f.write('\nDetected by Hamming:\n')
    for overlap in hamoverlaps:
        f.write(str(overlap) + '\n')
    f.write('\nHamming: \n' + str(hamtruecount) +
    ' Detected correctly\n' + str(hamfalsecount) +
    ' Detected incorrectly\n')

    f.write('\nDetected by Jaro:\n')
```

```
for overlap in jarooverlaps:
    f.write(str(overlap) + '\n')
f.write('\nJaro: \n' + str(jarotruecount) +
' Detected correctly\n' + str(jarofalsecount) +
' Detected incorrectly\n')
```

From this function, a txt file named by the current calendar day, hour, minute, and second is created. This file is then filled by the multiple occurrence overlaps, how many are correctly identified and how many aren't, and then the same for each of the three algorithms individually. The function once again uses *os* to determine the filepath directory and makes use of *if not os.path.exists(filepath): os.makedirs(filepath)* to create a directory "output" if it does not already exist. An example of this output is shown, the example has removed several of the overlaps stated for readability - "..." represents removed overlaps:

```
Detected by Levenshtein, Hamming, and Jaro:
['12125_n_a3', '2229_n_a3']
['12527_n_a1', '12527_dog_n_a1']
['12527_n_a2', '12527_1_n_a1']
...

Multiple:
5 Detected correctly
22 Detected incorrectly

Detected by Levenshtein:
['12099_n_a9', '2229_n_a3']
['12099_n_a9', 'self_1_n_a4']
['12125_n_a3', '2229_n_a3']
...

Levenshtein:
10 Detected correctly
61 Detected incorrectly

Detected by Hamming:
['12099_n_a5', '12099_n_a9']
['12099_n_a5', '12125_n_a3']
['12099_n_a5', '2229_n_a2']
...
```

```
Hamming:
10 Detected correctly
194 Detected incorrectly

Detected by Jaro:
['12099_n_a2', '12527_n_a5']
['12099_n_a2', '12527_3_n_a1']
['12099_n_a5', 'self_2_n_a7']
...

Jaro:
9 Detected correctly
61 Detected incorrectly
```

This output was altered once a user output was created. No longer being used to determine overlaps the output now breaks apart the detection lists into correctly and incorrectly detected so these can be viewed easier:

```
Detected by Levenshtein, Hamming, and Jaro:

Correct Detection: 6
['12099_n_a9', '2229_n_a3']
['12527_n_a2', '12527_1_n_a1']
...
Incorrect Detection: 32
['12099_n_a9', 'self_1_n_a4']
['12125_n_a3', '2229_n_a3']
...
```

```
Detected by Levenshtein:

Correct Detection: 11
['12099_n_a9', '2229_n_a3']
['12527_n_a2', '12527_1_n_a1']
...
Incorrect Detection: 74
['12099_n_a9', 'self_1_n_a4']
['12125_n_a3', '2229_n_a3']
...
```

etc....

This could be altered if needed later - perhaps into a developer oriented JSON output with two JSON files for each algorithm, true and false as calculated by a respective hand overlap created for the corpus. This would be recommended as the alterations to "acceptable distance" are in consideration to the dataset built upon rather than as a generalised whole. The change was implemented after creation of the "user" output as this allowed the test output to be more in depth with the structures without sacrificing user readability.

4.9.2 User Output

After implementation, the output method was revised. The test output was a text file containing only ID's of overlapping nodes as calculated by each algorithm, this was useful in refining the algorithm results to increase accuracy but wasn't very usable as the output of a tool. A simple JSON output comprised of overlaps as detected by the "multi_overlap" occurence checker was created, the JSON output would follow the structure of:

```
{
overlap_id: "unique ID"
node_id_1: ""
node_text_1: ""
node_id_2: ""
node_text_2: ""
}
```

To create this output, a new function was created, *build_json*.

```
def build_json(overlaps, arguments):
    jsonoverlaps = []

    for thisoverlap in overlaps:

        overlap_id = str(uuid.uuid4())
        overlaps = []
        for thisnode in thisoverlap:
            for each in arguments:
                if thisnode == each[1]:
                    overlaps.append(each)
```

```
        overlap = {
            "overlap_id": overlap_id,
            "node_id_1": overlaps[0][0],
            "node_text_1": overlaps[0][1],
            "node_id_2": overlaps[1][0],
            "node_text_2": overlaps[1][1]
        }
        jsonoverlaps.append(overlap)

    filepath = os.path.dirname(os.path.realpath(__file__))
        + '\\output'
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    currenttime = datetime.datetime.now()
    filename = str(currenttime.day) + str(currenttime.hour)
        + str(currenttime.minute)
        + str(currenttime.second) + '.json'

    with open(os.path.join(filepath, filename), 'w') as f:
        json.dump(jsonoverlaps, f)
```

The function builds a string "overlap" using several variables. *overlap_id* is generated using the Python library "UUID", specifically "UUID4" which generates a unique ID - this is assigned to be an identifier for each specific overlap. Each overlap's node ID and contained text is then also used to create this string. The string is then appended to a created JSON file which is named and output using the same methods as the test output. The implementation was created to be a cross-platform approach to the generation of a usable output from an input of files.

An example of this output is shown:

```
[{
        "overlap_id": "103b4eae-db7b-4a1b-913d-59f420c5c06a",
        "node_id_1": "They're extremely clean",
        "node_text_1": "12099_n_a9",
        "node_id_2": "They are not clean",
        "node_text_2": "self_1_n_a4"
}, {
        "overlap_id": "13940ede-ba44-42ea-a9a9-f222f191608c",
        "node_id_1": "they eat less",
        "node_text_1": "12125_n_a3",
        "node_id_2": "They are clean",
```

```
        "node_text_2": "2229_n_a3"
}]
```

4.10 String Cleaning

After implementation, an oversight in string comparisons was detected. Algorithms would compare punctuation and would note a difference between upper and lower case which would generate a larger distance between strings that were almost identical. For example "This was good." compared to "this was good" through Levenshtein and Hamming would generate a distance of 2 despite the sentences being functionally similar.

By using the Python standard 'string' library punctuation can be removed easily:

```
a_no_punctuation = a[0].translate(str.maketrans('', '', string.punctua
```

Removal of upper case is even simpler due to not requiring a library:

```
a_clean = a_no_punctuation.lower()
```

By using these methods argument text is quickly cleaned to make comparisons more accurate.

4.10.1 Acceptable Distance Revision

By this change, the "acceptable distance" variable would need to be revised to determine accuracy. This was also to be done for the Jaro algorithm as minor ratio changes increased the number of matches by a large margin. The selected acceptable distance will once again take consideration into the occurrence checker, having more false matches than true matches is not necessarily a flaw as the intention of the tool would be to permit manual review of each overlap by human to determine if an overlap is true or not. Minor changes to the files validated meant the number of overlaps potentially to be found correct were 38

The retested algorithms had various changes as output, becoming slightly more accurate (Figure 7.1): Specifics on how each algorithm were changed are displayed by Tables 4.10.1, 4.10.1, and 4.10.1. Based on this the selected "acceptable distances" were altered for each algorithm, it was also taken into consideration algorithm depth. As Levenshtein and Hamming are more simplistic than Jaro, they are permitted a larger error margin. This consideration allows more overlaps to be detected but they will not be output unless Jaro has also identified these overlaps. For the comparisons, there are

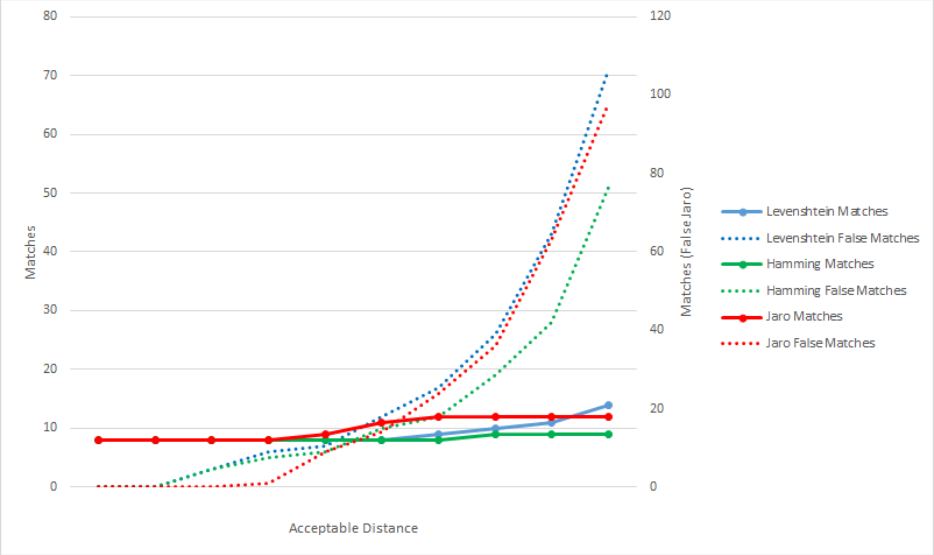

Figure 4.6: All algorithms acceptable distance changes after cleaning strings

a total of 38 correct overlaps to be detected out of 3486 total combinations of each node input, this is around 91 times in scale. This scale is taken into consideration as many more false overlaps exist in comparison. This is also in part due to checking nodes in the same map rather than exclusively checking against other map nodes.

The selected acceptable distance for each algorithm is intended to have no more than around 10 false overlaps to 1 correct for Levenshtein and Hamming, and around 5 false overlaps to 1 correct for Jaro, these numbers are *somewhat* arbitrate as they are specifically chosen using the dataset created, but with consideration to the tool being used outwith. Based on the results:

- Levenshtein Acceptable Distance = 12
- Hamming Acceptable Distance = 13
- Jaro Acceptable Ratio = 0.71

4.11 Additional Algorithms

Several string comparison algorithms were implemented after the change to string cleaning, these were outwith the original scope of the project and were developed with remaining time after completion.

Acceptable Distance	Matches	False Matches
1	8	0
2	8	0
3	8	3
4	8	6
5	8	7
6	8	12
7	9	17
8	10	26
9	11	43
10	14	71
11	14	95
12	14	110
13	14	142
14	15	192
15	17	239

Table 4.4: Levenshtein results

Acceptable Distance	Matches	False Matches
1	8	0
2	8	0
3	8	3
4	8	5
5	8	6
6	8	10
7	9	12
8	10	19
9	11	28
10	14	51
11	14	59
12	14	79
13	14	91
14	15	112
15	17	134

Table 4.5: Hamming results

Acceptable Distance	Matches	False Matches
0.95	8	0
0.9	8	0
0.85	8	0
0.8	8	1
0.75	9	9
0.74	11	14
0.73	12	24
0.72	12	36
0.71	12	63
0.7	12	98
0.69	12	153
0.68	12	220
0.67	14	309
0.66	15	406
0.65	15	508

Table 4.6: Jaro results

4.11.1 Jaro-Winkler

An advancement on the Jaro algorithm which utilises the previously output Jaro ratio. Jaro-Winkler uses the similarity and adds this to a prefix length (multiplied by the scaling factor 0.1. The prefix is then multiplied by 1 minus the Jaro ratio. This allows additional accuracy in the ratio by taking a prefix based on the number of similar characters at the start of the string which creates the potential return of a higher ratio when compared to Jaro.

```
def jaro_winkler(s1, s2, jaro_ratio):
    prefix = 0

    for i in range(min(len(s1), len(s2))):
        if (s1[i] == s2[i]) :
            prefix += 1
        else :
            break

    prefix = min(4, prefix)
    winkler = jaro_ratio
    winkler += 0.1 * prefix * (1 - winkler)

    return(winkler)
```

Acceptable Distance	Matches	False Matches
0.95	8	0
0.9	8	0
0.85	8	6
0.8	11	88
0.75	15	320
0.74	21	367
0.73	21	408
0.72	21	428
0.71	21	468
0.7	21	524
0.69	21	574
0.68	21	649
0.67	21	734
0.66	22	832
0.65	22	922

Table 4.7: Jaro-Winkler results

Table 4.11.1 shows the accepted ratio as an overlap changes has on detection of matches with how many are correct and how many are not, this is also illustrated by figure 4.7. This output shows that compared to Jaro, Jaro-Winkler detects slightly less overlaps in total but has a less exponential increase in false matches - even if the large multiplicative increase is still present.

4.11.2 Jaccard

The Jaccard algorithm, known as Jaccard Index or Jaccard Similarity Coefficient measures similarity against diversity in sets. This can be applied to strings by converting a string into a list of characters ("a dog" becoming ["a", " ", "d", "o", "g"]) and comparing using this list. The comparison takes the intersection of a list (where each list coincides) as an integer and divides this by the union of the lists, (length of list one plus length of list two then subtracting the intersection).

```
def jaccard(token1, token2):
    token1chars = []
    for char in token1:
        token1chars.append(char)

    token2chars = []
```

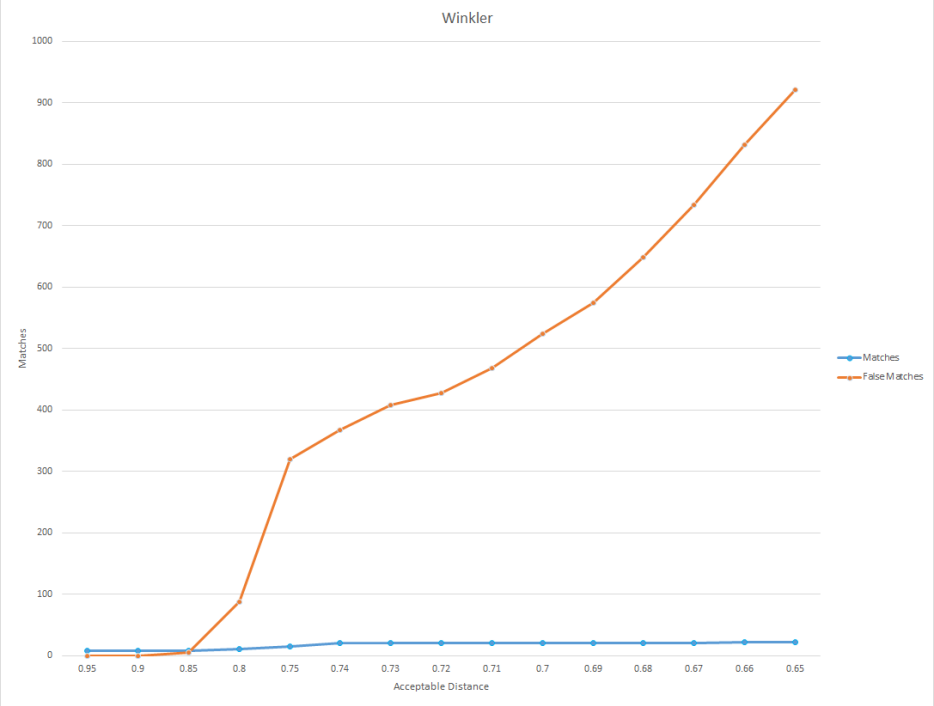

Figure 4.7: Jaro-Winkler Acceptable Distance Changes

```
for char in token2:
    token2chars.append(char)

intersection = len(list
    (set(token1chars).intersection(token2chars)))
union = (len(set(token1chars))
    + len(set(token2chars))) - intersection
jaccard_similarity = (float(intersection) / union)
return(jaccard_similarity)
```

Table 4.11.2 and figure **??** show the number of correct and incorrect overlap detections by the algorithm through changing the acceptable distance. The output shows a similar trend to Jaro-Winkler but detects slightly less overlaps, however for each correct identification - more incorrect identifications are made at a rate compared to Jaro-Winkler.

Acceptable Distance	Matches	False Matches
0.95	8	1
0.9	8	21
0.85	8	51
0.8	10	123
0.75	10	289
0.74	10	289
0.73	10	340
0.72	10	389
0.71	13	430
0.7	13	498
0.69	13	543
0.68	14	635
0.67	14	635
0.66	16	741
0.65	16	827

Table 4.8: Jaccard results

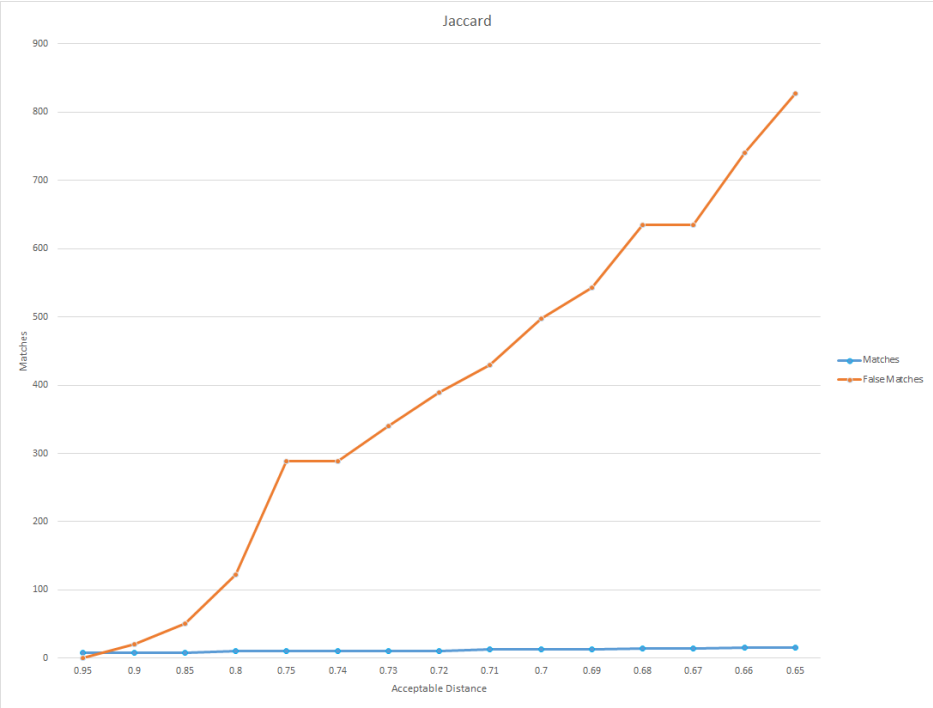

Figure 4.8: Jaccard Acceptable Distance Changes

4.12 Multiple-Occurrence Check

With the addition of more algorithms, a change was made to the multiple occurrence checker to allow more identified overlaps without sacrificing accuracy. Rather than checking all 5 algorithms and only counting each one, the algorithms were separated by similarity of results. This led to the construction of a list of Hamming v Levenshtein duplicates, and Jaro-Winkler v Jaccard duplicates. These lists are then combined and have any duplicates removed. This list is compared to the output of Jaro and any duplicates within either list are counted as to be used as the final output.

```
def multipleOverlaps(multi_overlaps, levoverlaps, hamoverlaps,
    jarooverlaps, winkleroverlaps, jaccardoverlaps):

    dup_multi = []

    editoverlaps = itertools.product(levoverlaps, hamoverlaps)
    #Levenshtein and Hamming
    for a, b in editoverlaps:
        if a == b:
            dup_multi.append(a)

    ratiooverlaps = itertools.product(winkleroverlaps,
        jaccardoverlaps)
    #Winkler, and Jaccard
    for a, b in ratiooverlaps:
        if a == b:
            dup_multi.append(a)

    edit_and_ratio = [] #Everything but jaro

    #Get rid of duplicates
    for each in dup_multi:
        if each not in edit_and_ratio:
            edit_and_ratio.append(each)

    final_check =
        itertools.product(jarooverlaps, edit_and_ratio)
    #Finally, Jaro as our safe guard
    for a, b in final_check:
        if a == b:
```

```
multi_overlaps.append(a)
```

This change allowed additional leniency for algorithms without sacrificing accuracy.

4.13 User Version

A user implementation is also available of the public repository. While functionally similar this version removes any comparisons to "hand overlaps" along with the text file output - outputting only the JSON construction. Functions and variables removed from this version are:

- getHandOverlaps()
- Each algorithm's true/false list of overlaps and counts for each
- compareOverlaps()
- output()

The user iteration functions exactly the same but merely only compares input JSON. The differences between each version are viewable in the appendices (7.1 and 7.2)

4.14 Issues faced

Several issues during development were faced but all known issues were corrected in the development time frame, most were logic errors or difficulty with libraries that were new to the developer (such as json). Several are listed:

- Issues retrieving SADFace nodes, initial attempts involved using $x = data.get(nodes)$, $y = x.get(text)$
- How overlaps were represented, initially if an overlap could be said to include more than just 2 nodes this was represented in a single line [string1] [string2] [string3] rather than the current implementation of [s1][s2], [s1][s3], [s2][s3]. While this uses less space, the file was more difficult to read in and compare overlaps too - this also required more explanation of use where the current implementation is simple and easy to understand.

- The Jaro implementation occasionally returned values above 1, this was due to both an error in the final calculation and the transposition rounding
- Levenshtein initially did not function correctly in a previous version that did not utilise NumPy
- Hamming initially crashed the script and returned an error as an oversight when strings were of different length was not considered
- Comparing overlaps required rewriting after identifying that in the scenario an overlap [x][y] was compared to a hand overlap of [y][x] would return an incorrect overlap
- As previously mentioned, the AIF-to-SADF.py does not function, this was due to non completion of the script within the allocated development time for said script. This however is not considered a failure of the project as the implementation laid outside of the scope and was merely attempted to help construct a corpus.
- AIF compatibility - This was exclusive to AIFdb's JSON export rather than a general AIF compatibility. However a generalized implementation would not be possible due to there existing no standard for how AIF principles are represented in differing file types such as JSON. Issues were presented by attempting to add dual-compatibility, initially the type of interchange being processed had to be indicated by a variable.

An issue not faced was time constraints, while a deeper implementation of algorithms was desired for the project - this was identified early on as a "stretch goal" which was potentially outside of scope. The desired features implemented will be discussed during the conclusions chapter under "Future Work". The project was successfully implemented to the desired level within the development time allocated.

4.15 Summary

The design specification was central to the implementation and was not altered, this meant development was productive within the time-frame allocated and that re-designs to describe the changed state of the design were not needed to be done. All issues that were noticed during development were successfully solved which resulted in the implementation meeting the aims

as described, more algorithms were desired to be implemented but this was already known to be potentially out of scope.

The corpus was successfully created as intended with low-level argumentation from AIFdb converted into SADFace JSON, this allowed all nodes to be extracted correctly for processing.

The nodes are read into a list which is then looped through each possible combination once to compare the text of each node via the string algorithms Hamming, Levenshtein, and Jaro. Compatibility to reading AIFdb JSON exports was also implemented at a later date.

An output text file is then created which lists all overlap IDs and how many were correct in identification, next to how many were not. This output contains a list for each algorithm alongside the compared overlap list which is generated if an occurrence of an overlap is present in all three algorithm overlap lists. Post implementation added the JSON output which would ascribe an ID to each overlap along with displaying each node in the overlap's ID and Text variables - this would allow a low level cross-platform output to be processed easier rather than needing to search for each node ID for the text manually.

Chapter 5

Evaluation

5.1 Analysis of Implemented Features

The AIF-to-SADFace script was non functional, this was an aim of the project but is more in line with being a "side goal" rather than primary due to not being integral to functionality of the project, the conversion tool would however be useful to creating a larger corpus quicker but also usable by other projects using SADFace as a central component.

Minor changes were made between design and implementation as illustrated by figures 3.4 and 4.1. Initially the creation of a multiple overlap occurrence was to be generated during the checking of hand overlaps - this was changed to be created prior to detection for both efficiency and modality. In design the "output generation" was somewhat vague so this was changed to better display the process flow of each component relevant to the output.

SADFaces are read in correctly and only pull text from atom nodes, this function had issues early into development but this was solved appropriately by looping through the length of "nodes" from the supplied JSON and skipping over nodes that do not contain a "text" variable which is exclusive to atom nodes.

The string comparison algorithms selected were implemented correctly with all issues solved as mentioned previously. A combination of the algorithm outputs based on acceptable distance is shown:

As shown, Hamming and Levenshtein are very similar in output results but this is by Hamming having a larger value assigned to "acceptable overlap" than Levenshtein- from this we can identify Levenshtein to be more accurate

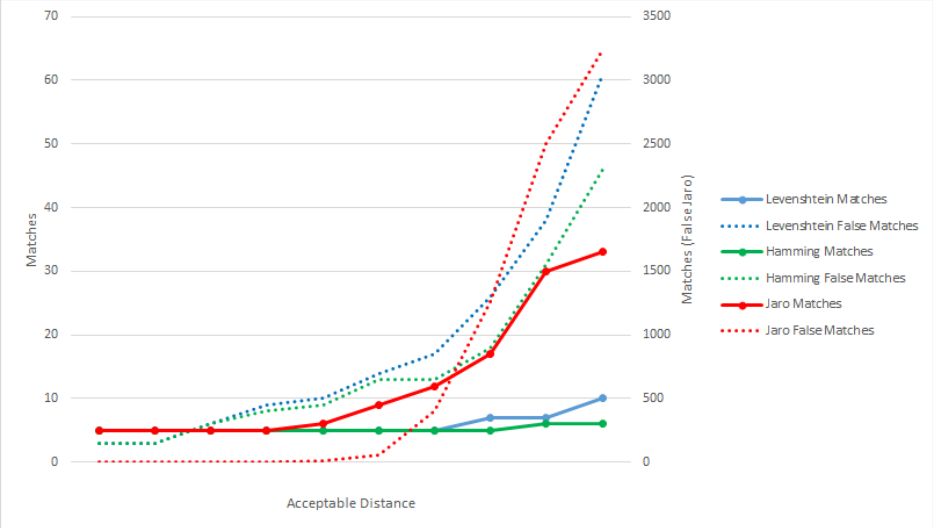

Figure 5.1: Acceptable distance changes output

than Hamming which is in line with the complexity of implementation. Jaro is more accurate at detecting overlaps but at the cost of a significantly higher number of false matches when given leniency in "acceptable overlap".

The introduction of multiple overlap checking allows the output to have a greater degree of accuracy in results, if all algorithms determine an overlap to be present this can be trusted more as correct than if only one does. This was initially unplanned and was added late in design but the inclusion of the function greatly increases reliability of the output.

The output generated is a simple text file listing each list of detected overlaps and an integer representation of correct and incorrectly identified overlaps, this functions as a representation of the work done but is not as usable as an implementation like a user interface - further depth could be added such as exporting the texts relevant to each ID but this would create extremely large text files. This issue was solved by the creation of the JSON output, the JSON holds each overlap with a unique ID along with each node in the overlaps ID and contained text. This solution grants the tool enhanced use by generating a usable output.

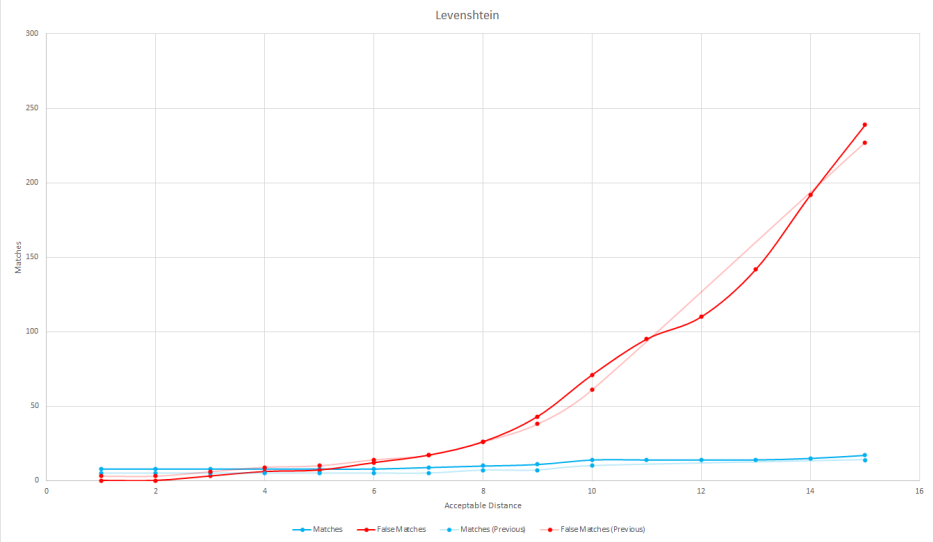

Figure 5.2: Levenshtein - Unclean (Previous) against Cleaned strings

5.2 String Cleaning

The later change to include string cleaning assisted in overall accuracy and required re-tuning of the "acceptable distance" variable key to each algorithm identifying an overlap. The previous measurements of each output based on changing the variable also gave an indication as to the range of values to be focused on (more specifically with Jaro due to the ratio nature). Each graph shown has the previous outputs transparent underneath the values recorded when the strings have been cleansed of punctuation and made lower case.

As shown by tables 4.8.1 and 4.10.1 by the changes made to the strings input, the number matches identified by both correct and incorrect increase slightly for comparison - previously with an "acceptable distance" of 15, 14 matches and 227 false matches were identified. At the same distance once cleaned, 17 matches and 239 false matches are identified showing that by making the slight alteration more matches are able to be discovered within a smaller edit distance. (See figure 5.2 for a graph visualisation).

Tables4.8.2 and 4.10.1 show the output from changes made to Hamming along with the graph representation figure 5.3. As expected the change in output for Hamming coincides with Levenshtein output changes. Hamming is now able to detect more overlaps both correct and incorrect, but still detects less than Levenshtein due to its simplicity.

Tables 4.8.3 and 4.10.1 and figure 5.4, Jaro identifies slightly more over-

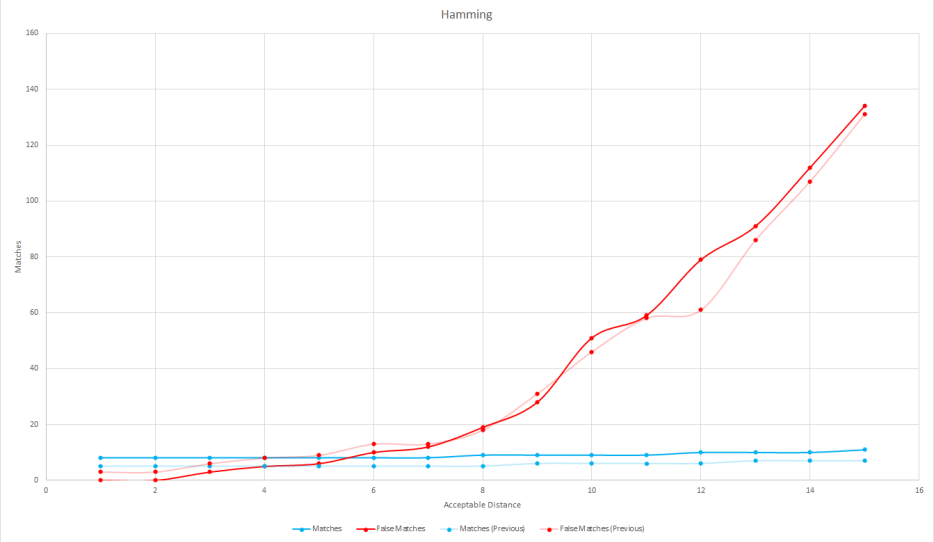

Figure 5.3: Hamming - Unclean (Previous) against Cleaned strings

laps with around the same increase in false matches by each ratio change. Table 4.10.1 put more focus onto measuring between 0.75-0.65 as a ratio as the previous outputs under these ratios were deemed unusable due to the severity of identified false matches.

The changes output by Levenshtein and Hamming were expected, as each measure individual character differences by removing several of these while maintaining the actual content of the string - only actual differences in letter characters are detected which would allow more matches to form with less edit distance. Jaro's output was expected to be slightly more noticeably better but instead has a more gradual, but still improved, detection rate increase.

5.3 Processing Time

Using the standard library *time* in Python, the process time for functions to execute was calculated to display which provides the greatest amount of computational load. This was done as during development processing time began increasing, this was remedied for development by only checking a small number of SADFace files. The expectation, since the solution decreased time, was that the *itertools.combinations* loop to compare arguments took the most amount of time - this would be due to an exponential increase in comparisons with each compared file as every individual node is compared against every other one.

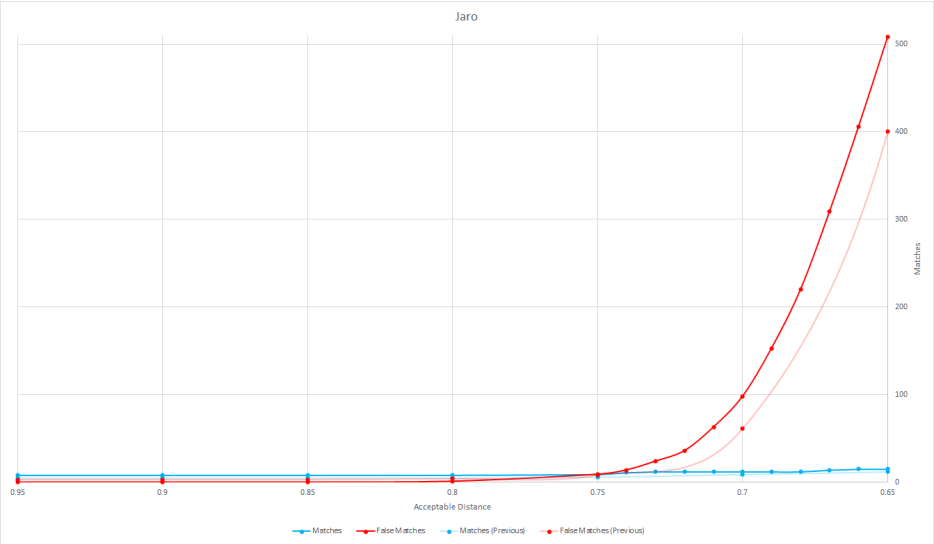

Figure 5.4: Jaro - Unclean (Previous) against Cleaned strings

Function	Time (Seconds)
getHandOverlaps	0.00095
getArguments	0.003989
Combinations Loop	12.87581
Compare Overlaps (Avg.)	0.00224
Multiple Overlaps	0.07568
Output	0.00196
JSON_build	0.00299

Table 5.1: Time for functions to process

Table 5.3 displays that the expectation of the combinations loop taking the most amount of processing time to be true by quite a large factor, every other function takes less than a tenth of a second where the loop takes almost 13, this would also increase with every comparison algorithm added. Ways to minimise the comparison time on the user side would be to minimise the number of files compared to - however a total of 15 seconds to process around 80 arguments is seen as acceptable in the current state. On developer side, currently every single argument is compared - this is inclusive of arguments on the same map. The reason for this inclusion was that AIF contains a common "rephrase" node which does as said, rephrase the same statement - this is technically unnecessary and the detection of such on large maps could be useful.

The results achieved were ran on a desktop containing an i7-7700K CPU @ 4.2GHz and 16GB of RAM, this is non standard for most desktops which would process this so it should be considered that these processing times would be increased in the majority of all structures executing the script.

Chapter 6

Conclusions

The implementation is available at the public git repository: `https://github.com/ConnorNess/Argument-Overlap-Detection`

6.1 Summary of Work Done

Literature relevant to each component of the project was discussed which allowed the project to be designed appropriately in line with several facets. The corpus was created which contained around 80 arguments total all pertaining to the discussion of pets, this was chosen for simplicity of overlap analysis - the aim was to minimise the ambiguity of detected overlaps to enable accuracy in development for the implemented algorithms. A larger corpus would have given more reliable and conclusive results however this would come with the risk of ambiguity in the results alongside reduced development time.

The project is designed to be used within the argument mining community for the intention of contributing to corpora merging to create larger maps of deeper discussions, this is shown through the corpus input being in an Argument Data Interchange format - SADFace. The project was developed around SADFace due to AIF lacking a "set" software implementation like SADFace being singularly of JSON implementation. AIF can be exported in the JSON format through AIFdb however AIF also has certain inclusions which could contribute to ambiguity in the corpus such as *PA-nodes* (preference nodes) which give one claim more weight in a map compared to another, for these reasons SADFace was utilised and AIF maps were converted to SADFace by hand using both the exportable JSON and the argument map viewer within AIFdb's interface. A script to automatically convert AIF-to-SADFace was attempted to be implemented but was not functional within the allocated time to development. AIFdb was used exclusively to create the

corpus, this was to ensure cohesion in individual arguments in the corpus creation however alternate argument databases could have been used such as *args.me*. This however would have required more time to create the corpus as the arguments in such databases would need to be manually translated from natural text into SADFace.

String comparison evaluations were undergone which included a discussion of literature which dealt with string algorithm analysis, along with the design of the project which required implementation of these algorithms to keep within project scope so the deliverable would be complete in time. For this reason simple character-based metrics were used - specifically Hamming distance, Levenshtein edit distance, and Jaro distance. These were all successfully implemented and tested to determine an "acceptable distance" to determine if an overlap is present, this was done by comparing the overlaps to manually detected overlaps and evaluating how many were correctly and incorrectly identified. This "acceptable distance" was then considered in reference to a multiple overlap checker which would validate if an overlap occurred in all implemented algorithm outputs.

Python was utilised for the development of this project due to its popularity in natural language processing and access to the Natural Language ToolKit which is identified as potentially important to future development as features contained in the toolkit are useful to the creation of deeper algorithms and access to a Wordnet. Java was considered for the same reasons (Stanford CoreNLP rather than NLTK) however Python's advantages for development time was another consideration as this would allow extra time for solving issues with the implementation.

The implementation was developed with near replication of the initial design overview, only minor changes were made which were mostly inclusive of a consideration for how the desired implementation of how multiple occurrence overlap checker would fit into the process along with how the "arguments" list loop would be utilised via *itertools*. Issues faced during development were mentioned but all known issues were successfully solved in the allocated development time frame. All algorithms successfully output a list of detected overlaps by ID, these lists were then compared to create another multiple occurrence list where an overlap detected by all implemented algorithms were output - this would add accuracy to the results and this list can be considered the "conclusive" output of overlaps. Pseudocode was created for each several important non-string algorithm functions and were implemented using this design and were only changed to meet the syntax of Python or if errors in the design would result in a false output.

For the string algorithms - Hamming was implemented based on descrip-

tion due to its simplicity, Jaro and Levenshtein were implemented using several resources each which are both cited under the implementation section discussing each.

By evaluating the outputs based on changing the "acceptable distance" variable for each algorithm (see figure 5.1) it can be seen that Jaro has a greater detection rate for correct overlaps than Hamming or Levenshtein but will detect an exponential number of false matches with only minor changes to the acceptable distance variable.

The output generated by the script details each algorithm's detected overlaps by ID along with an integer representing how many are correctly or incorrectly identified, this value was used to determine what "acceptable distance" to settle on for implementation - however by design this value is easily changed should the selected value not be as conclusive for a different corpus.

6.2 Main Conclusions

The implementation met almost all desired aims of the project, only failing to meet the implemented AIF-to-SADFace conversion tool - this however is seen as less important to the project as a whole than other aims.

The implemented algorithms function as intended output as described, however the output is merely the ID's of an overlap pair rather than the text itself. While this was the described scope of identification, perhaps this could be advanced further to allow selection of an overlap to view the actual text itself, a UI of sorts could be created for this purpose - this was completely out of scope but a mock up of a potential UI was created to illustrate how this would provide a more advanced use of the script (Figure 6.1.

The script could also be altered to only accept the relevant files, currently if a directory contains anything other than the applicable files it will fail to process them acceptably and will quit out of the process.

Overall the implementation is understandable and correctly commented with each function and variables appropriately named to their process making comprehension of the script by other developers simple which will allow enhancements planned to be implemented by others effectively.

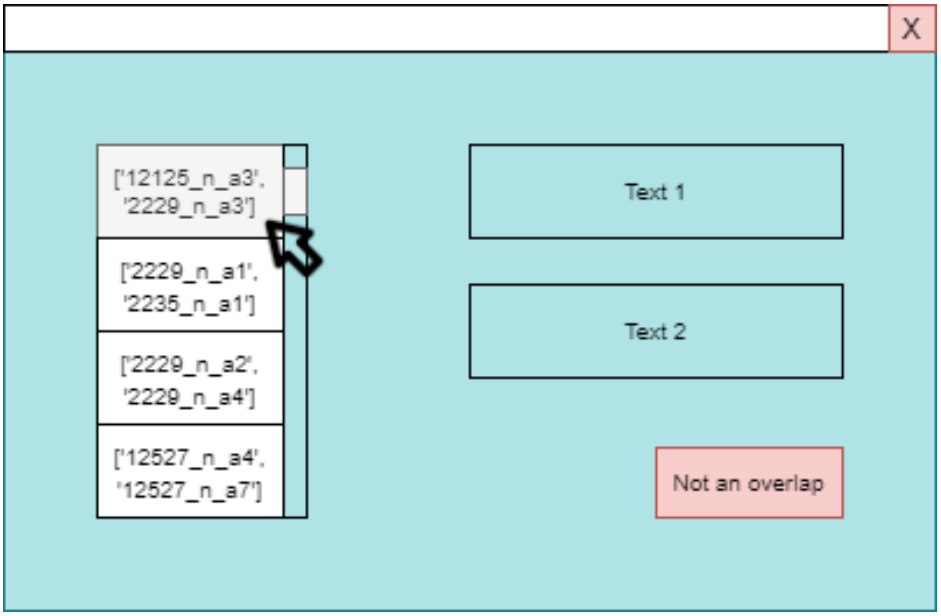

Figure 6.1: User Interface mock-up

6.3 Delivery against objectives

Aim 1: *"To create SADFace files from the same domain using a pre-existing format"*

This aim was the first met as a requirement for any implementation to actually be utilised.

Aim 2: *"Attempt the creation of a conversion tool from the chosen format to SADFace"*

The implementation was attempted based on the pseudocode written but was unfortunately not met

Aim 3: *"Hand select all identifiable overlaps within the SADFace files"*

This process was done by representing an overlap by ID in a text file, each line described an overlap as [id1, id2], this was originally implemented differently but was somewhat tricky to parse and also difficult to explain quickly so the more simplistic approach was used.

Aim 4: *"To implement a python script which will output identifiable overlapping nodes in SADFace"*

By chapter 3 - Implementation, the script functions as described and stayed as close as possible to the design made prior to implementation.

Aim 5: *"To use at least two string comparison algorithms"*

This goal was met by Hamming and Levenshtein, Jaro was then later implemented once the rest of functionality was implemented. More are recommended in the next section but as stated - this was out of scope.

Aim 6: *"Evaluate the output generated by string comparators against hand identified overlaps"*

Each algorithm creates its own respective list of overlaps which are then combined to create a list of occurrences across all lists, all of these are then compared through a function to identify which overlaps were correct and which were not - and providing a value for each.

Aim 7: *"The created script must be designed well enough to be understood by another developer for the purpose of improvements"*

The choice of Python made this aim easier to accomplish due to Python's inherent readability, this was also met through use of functions and appropriately named variables. These facets should allow any developer to begin interacting with the tool effectively.

Aim 8: *"Discuss what improvements can be made and what ones are believed to have the greatest effect if implemented"*

This aim is discussed in the next section.

6.4 Future work

Several recommendations for how this implementation could be further developed are listed:

- Word/Sentence 2Vec implementation, the ability to measure meaning between phrases would be of great benefit to this project - this would require access to a form of wordnet and may use a heavy amount of computational power however.
- A usage of token-based comparisons, this would also use more computational power but from results seen in other papers - the extra cost could be worth investing into.
- Development of the described UI, this would allow the tool to have a greater use by the argument community and would assist in analysis of the detected overlaps.

- The discussed Markov Edit Distance would be a strong inclusion to character-based comparisons.
- Like Markov, utilising NCS (number of common substrings) in edit distance could also help detect more overlaps.
- Advancement of the current algorithms i.e. Jaro-Winkler and Damerau–Levenshtein.
- The ability to determine and represent how many overlaps are counted to a singular node - if the script was applied to a large argument map and the same node has many overlaps - this could represent the importance of the argument within its relevant community.

6.5 Personal reflection

Personally I feel very satisfied by the work and development done for this project, while there are several features I wish I could have implemented they were outside of scope but I believe is also indicative of the creation of a solid base on which further enhancements can be added to. I am confident in the state of the deliverable as it currently stands, I feel the output generation could be enhanced to make analysis of the actual overlaps easier but this would have been somewhat unwieldy in a text document and could produce a large difficult to comprehend output.

I believe time in this project was managed effectively, with the project being complete a week prior to deadline which allowed additional development time. This allowed the implementation of Jaro-Winkler and Jaccard, AIFdb JSON compatibility, and the far more usable JSON output.

From my meetings with my supervisor along with my own development plan I feel like I have met the expectations for this dissertation and am able to be proud of the work done both in development and in writing.

Chapter 7

Appendices

7.1 Code - OverlapDetection.py

```
import os
import re
import json
import itertools
from math import floor
import datetime
import numpy as np
import string
import uuid
import time

#Read the hand detected overlaps file by line creating arrays
    ↪ of each line that contributes to an overlap
#Read in the atom nodes of the supplied SADFace json files
#Run atoms through each algorithm comparing them to each other
#Flag atoms which are "close enough" as overlap
#Return these
#Check against hand detected overlaps
#Return these


##########################################################################################
    ↪
#output data to textfile

def build_json(overlaps, arguments):
    jsonoverlaps = []

    for thisoverlap in overlaps:

        overlap_id = str(uuid.uuid4())
        overlaps = []
        for thisnode in thisoverlap:
            for each in arguments:
                if thisnode == each[1]:
                    overlaps.append(each)

        overlap = {
```

```
            "overlap_id": overlap_id,
            "node_id_1": overlaps[0][0],
            "node_text_1": overlaps[0][1],
            "node_id_2": overlaps[1][0],
            "node_text_2": overlaps[1][1]
        }
        jsonoverlaps.append(overlap)

    filepath = os.path.dirname(os.path.realpath(__file__)) +
        ↪ '\\output'
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    currenttime = datetime.datetime.now()
    filename = str(currenttime.day) + str(currenttime.hour) +
        ↪ str(currenttime.minute) + str(currenttime.second) +
        ↪ '.json'

    with open(os.path.join(filepath, filename), 'w') as f:
        json.dump(jsonoverlaps, f)

def output(multitrue, multitruecount, multifalse,
    ↪ multifalsecount, levtrue, levtruecount, levfalse,
    ↪ levfalsecount, hamtrue, hamtruecount, hamfalse,
    ↪ hamfalsecount, jarotrue, jarotruecount, jarofalse,
    ↪ jarofalsecount, jarowinklertrue, jarowinklertruecount,
    ↪ jarowinklerfalse, jarowinklerfalsecount, jaccardtrue,
    ↪ jaccardtruecount, jaccardfalse, jaccardfalsecount):
    currenttime = datetime.datetime.now()
    filename = str(currenttime.day) + str(currenttime.hour) +
        ↪ str(currenttime.minute) + str(currenttime.second) +
        ↪ '.txt'

    filepath = os.path.dirname(os.path.realpath(__file__)) +
        ↪ '\\output'
    if not os.path.exists(filepath):
        os.makedirs(filepath)

    with open(os.path.join(filepath, filename), 'w') as f:
        f.write('Detected by All:\n')
        f.write('\nCorrect Detection: ' + str(multitruecount) +
            ↪  '\n')
```

```
for overlap in multitrue:
    f.write(str(overlap) + '\n')
f.write('\nIncorrect Detection: ' + str(multifalsecount
    ↪ ) + '\n')
for overlap in multifalse:
    f.write(str(overlap) + '\n')
f.write('\n____________________\n\n')

f.write('\nDetected by Levenshtein:\n')
f.write('\nCorrect Detection: ' + str(levtruecount) +
    ↪ '\n')
for overlap in levtrue:
    f.write(str(overlap) + '\n')
f.write('\nIncorrect Detection: ' + str(levfalsecount)
    ↪ + '\n')
for overlap in levfalse:
    f.write(str(overlap) + '\n')
f.write('\n____________________\n\n')

f.write('\nDetected by Hamming:\n')
f.write('\nCorrect Detection: ' + str(hamtruecount) +
    ↪ '\n')
for overlap in hamtrue:
    f.write(str(overlap) + '\n')
f.write('\nIncorrect Detection: ' + str(hamfalsecount)
    ↪ + '\n')
for overlap in hamfalse:
    f.write(str(overlap) + '\n')
f.write('\n____________________\n\n')

f.write('\nDetected by Jaro:\n')
f.write('\nCorrect Detection: ' + str(jarotruecount) +
    ↪ '\n')
for overlap in jarotrue:
    f.write(str(overlap) + '\n')
f.write('\nIncorrect Detection: ' + str(jarofalsecount)
    ↪  + '\n')
for overlap in jarofalse:
    f.write(str(overlap) + '\n')
f.write('\n____________________\n\n')
```

```
        f.write('\nDetected by Jaro-Winkler:\n')
        f.write('\nCorrect Detection: ' + str(
            ↪ jarowinklertruecount) + '\n')
        for overlap in jarowinklertrue:
            f.write(str(overlap) + '\n')
        f.write('\nIncorrect Detection: ' + str(
            ↪ jarowinklerfalsecount) + '\n')
        for overlap in jarowinklerfalse:
            f.write(str(overlap) + '\n')
        f.write('\n____________________\n\n')

        f.write('\nDetected by Jaccard:\n')
        f.write('\nCorrect Detection: ' + str(jaccardtruecount)
            ↪  + '\n')
        for overlap in jaccardtrue:
            f.write(str(overlap) + '\n')
        f.write('\nIncorrect Detection: ' + str(
            ↪ jaccardfalsecount) + '\n')
        for overlap in jaccardfalse:
            f.write(str(overlap) + '\n')
        f.write('\n____________________\n\n')

################################################################################
    ↪
#If something is detected by all algorithms, might be worth
    ↪ noting what that is

def multipleOverlaps(multi_overlaps, levoverlaps, hamoverlaps,
    ↪ jarooverlaps, winkleroverlaps, jaccardoverlaps):

    dup_multi = []

    editoverlaps = itertools.product(levoverlaps, hamoverlaps)
    #Levenshtein and Hamming
    for a, b in editoverlaps:
        if a == b:
            dup_multi.append(a)

    ratiooverlaps = itertools.product(winkleroverlaps,
        ↪ jaccardoverlaps)
    #Winkler, and Jaccard
```

```
    for a, b in ratiooverlaps:
        if a == b:
            dup_multi.append(a)

    edit_and_ratio = [] #Everything but jaro

    #Get rid of duplicates
    for each in dup_multi:
        if each not in edit_and_ratio:
            edit_and_ratio.append(each)

    final_check = itertools.product(jarooverlaps,
        ↪ edit_and_ratio)
    #Finally, Jaro as our safe guard
    for a, b in final_check:
        if a == b:
            multi_overlaps.append(a)

##############################################################################
    ↪
#Compare detected overlaps to self detected overlaps

def compareOverlaps(handoverlaps, overlaps, correctoverlap,
    ↪ correctcount, incorrectoverlap, incorrectcount):


    for overlap in overlaps:
        for handoverlap in handoverlaps:

            match = 0

            if overlap[0] == handoverlap[0] or overlap[0] ==
                ↪ handoverlap[1]:
                match += 1
            if overlap[1] == handoverlap[0] or overlap[1] ==
                ↪ handoverlap[1]:
                match += 1
            if overlap[0] == overlap[1]: #this *should* never
                ↪ hit thanks to itertools but, better safe
                match = 0
```

```
            if match == 2:
                break


        if match == 2:
            correctoverlap.append(overlap)
            correctcount += 1
        else:
            incorrectoverlap.append(overlap)
            incorrectcount += 1

    return correctcount, incorrectcount


#################################################################################
    ↪
#ALGORITHMS HERE

def levenshtein(token1, token2):
    distances = np.zeros((len(token1) + 1, len(token2) + 1),
        ↪ dtype = int)

    for t1char in range(len(token1) + 1):
        distances[t1char][0] = t1char

    for t2char in range(len(token2) + 1):
        distances[0][t2char] = t2char

    a = 0
    b = 0
    c = 0

    for t1char in range(1, len(token1) + 1):
        for t2char in range(1, len(token2) + 1):
            if (token1[t1char-1] == token2[t2char-1]):
                distances[t1char][t2char] = distances[t1char -
                    ↪ 1][t2char - 1]
            else:
                a = distances[t1char][t2char - 1]
                b = distances[t1char - 1][t2char]
                c = distances[t1char - 1][t2char - 1]
```

```
                if (a <= b and a <= c):
                    distances[t1char][t2char] = a + 1
                elif (b <= a and b <= c):
                    distances[t1char][t2char] = b + 1
                else:
                    distances[t1char][t2char] = c + 1

    return (distances[len(token1)][len(token2)])

def hamming(token1, token2):
    distance = 0
    for char in range (len(token2)): #loop for the shortest
       ↪ word
        if token1[char] != token2[char]:
            distance += 1
    distance += (len(token1) - len(token2)) #Add the remaining
       ↪ length of the input
    return distance

def jaro(token1, token2):
    maxdist = floor(max(len(token1), len(token2)) /2) -1
    match = 0
    chars_t1 = [0] * len(token1)
    chars_t2 = [0] * len(token2)

    if (token1 == token2):
        return 1

    for t1char in range(len(token1)):
        for t2char in range (max(0, t1char-maxdist), min(len(
           ↪ token2), t1char + maxdist+1)):

            if(token1[t1char] == token2[t2char] and chars_t2[
               ↪ t2char] == 0):
                chars_t1[t1char] = 1
                chars_t2[t2char] = 1
                match += 1
                break

    if(match == 0):
```

```
        return 0
    transpositions = 0
    point = 0

    for t1char in range (len(token1)):
        if(chars_t1[t1char]):
            while(chars_t2[point] == 0):
                point += 1
            if(token1[t1char] != token2[point]):
                point += 1
                transpositions += 1
    transpositions = floor(transpositions/2)
    jaro_ratio = (match/ len(token1) + match / len(token2) + (
        ↪ match - transpositions + 1) / match)/ 3.0

    return(jaro_ratio)

def jaro_winkler(s1, s2, jaro_ratio):
    prefix = 0

    for i in range(min(len(s1), len(s2))):
        if (s1[i] == s2[i]) :
            prefix += 1
        else :
            break

    prefix = min(4, prefix) #Maximum 4 characters as indicated
        ↪ by the algorithms description
    winkler = jaro_ratio
    winkler += 0.1 * prefix * (1 - winkler)

    return(winkler)

def jaccard(token1, token2):
    token1chars = []
    for char in token1:
        token1chars.append(char)

    token2chars = []
    for char in token2:
        token2chars.append(char)
```

```
    intersection = len(list(set(token1chars).intersection(
        ↪ token2chars))) #AnB
    union = (len(set(token1chars)) + len(set(token2chars))) -
        ↪ intersection #AuB
    jaccard_similarity = (float(intersection) / union) #AnB /
        ↪ AuB
    return(jaccard_similarity)

##########################################################################################
    ↪
#Build array of arguments

def getArguments(arguments):
    #Read in the atom nodes of the supplied json files
    #Turn each json file into an array of strings holding the
        ↪ atoms + their ID

    domainpath = (os.path.dirname(os.path.realpath(__file__)) +
        ↪  '\\Pets\\SADFace\\Hand done\\') #<-------- Change
        ↪ directory here
    args = os.listdir(domainpath)


    for each in args:
            with open(domainpath + each) as arg:

                data = json.load(arg) #JSON to dict
                nodes = data.get('nodes') #Dict to list
                count = len(nodes) #How many nodes long is this
                    ↪ json?

                for i in range(count):
                    thisarg = [] #temp array to hold text and id,
                        ↪  will append once filled
                    try:
                        thisarg.append(data['nodes'][i]['text'])
                            ↪ #Both SADFace and AIFdb's JSON
                            ↪ output of AIF use 'text' under '
                            ↪ nodes', makes this a bit easier
                        try:
```

```
                        thisarg.append(data['nodes'][i]['id
                            ↪ '])
                        arguments.append(thisarg) #Only
                            ↪ append an argument with both
                            ↪ text and id
                    except KeyError:
                        try:
                            if(data['nodes'][i]['type'] == "I
                                ↪ "): #if this is AIF, we only
                                ↪  want i-nodes
                                thisarg.append(data['nodes'][i
                                    ↪ ]['nodeID'])
                                arguments.append(thisarg) #
                                    ↪ Only append an argument
                                    ↪ with both text and id
                        except KeyError:
                            print("no id") #uh oh, somethin
                                ↪ wrong with the json
                except KeyError: #Means no text is present -
                    ↪ its a scheme node likely
                    pass #If it ain't got text, we got no use
                        ↪  for it

################################################################################
    ↪
#Build array of self detected overlaps

def getHandOverlaps(handoverlaps):
    #Read the hand detected overlaps file by line creating
        ↪ arrays of each line that contributes to an overlap


    handoverlapsfilepath = os.path.dirname(os.path.realpath(
        ↪ __file__)) + '\\Pets\\overlaps.txt'
    with open(handoverlapsfilepath) as handoverlapsfile:
        handoverlaps.extend([line.split() for line in
            ↪ handoverlapsfile])

    handoverlapsfile.close()
```

```
#################################################################################################
    ↪
#Run

#Build array using getHandOverlaps
handoverlaps = []
getHandOverlaps(handoverlaps)
#Build arrays of atoms for each SADFace
arguments = []
getArguments(arguments)

#Arrays for each overlap for each algorithm
levoverlaps = [] #All detected overlaps for an algorithm
hamoverlaps = []
jarooverlaps = []
jarowinkleroverlaps = []
jaccardoverlaps = []

#For each atom in a SADFACE
#Compare to each atom in all SADFaces
#Using itertools combinations to easily compare all elements
    ↪ but only once, two for loops would double up
for a, b in itertools.combinations(arguments, 2):

    a_no_punctuation = a[0].translate(str.maketrans('', '',
        ↪ string.punctuation))
    a_clean = a_no_punctuation.lower()
    b_no_punctuation = b[0].translate(str.maketrans('', '',
        ↪ string.punctuation))
    b_clean = b_no_punctuation.lower()

    thislev = []
    #levenshtein
    levdist = levenshtein(a_clean, b_clean) #0 = the text, 1 =
        ↪ the id
    if (levdist <= 12): #ACCEPTABLE OVERLAP HERE - gotten
        ↪ through testing
        thislev.append(a[1])
        thislev.append(b[1])
        levoverlaps.append(thislev)
```

```
    thisham = []
    #hamming
    if len(a_clean) < len(b_clean):
        hamdist = hamming(b_clean, a_clean)
    else:
        hamdist = hamming(a_clean, b_clean)
    if(hamdist <= 13): #ACCEPTABLE OVERLAP HERE
        thisham.append(a[1])
        thisham.append(b[1])
        hamoverlaps.append(thisham)

    thisjaro = []
    thisjaro_winkler = []
    #jaro + jaro winkler
    jaroratio = jaro(a_clean, b_clean)
    jaro_winkler_ratio = jaro_winkler(a_clean, b_clean,
        ↪ jaroratio)
    if (jaroratio >= 0.71): #ACCEPTABLE OVERLAP HERE
        thisjaro.append(a[1])
        thisjaro.append(b[1])
        jarooverlaps.append(thisjaro)
    if (jaro_winkler_ratio >= 0.8): #ACCEPTABLE OVERLAP HERE
        thisjaro_winkler.append(a[1])
        thisjaro_winkler.append(b[1])
        jarowinkleroverlaps.append(thisjaro_winkler)

    thisjaccard = []
    #jaccard
    jaccard_index = jaccard(a_clean, b_clean)
    if (jaccard_index >= 0.8): #ACCEPTABLE OVERLAP HERE
        thisjaccard.append(a[1])
        thisjaccard.append(b[1])
        jaccardoverlaps.append(thisjaccard)

levtrue = [] #For all correctly identified overlaps
levtruecount = 0
levfalse = [] #For all incorrecntly identified overlaps
levfalsecount = 0
levtruecount, levfalsecount = compareOverlaps(handoverlaps,
    ↪ levoverlaps, levtrue, levtruecount, levfalse,
    ↪ levfalsecount)
```

```
hamtrue = []
hamtruecount = 0
hamfalse = []
hamfalsecount = 0
hamtruecount, hamfalsecount = compareOverlaps(handoverlaps,
    ↪ hamoverlaps, hamtrue, hamtruecount, hamfalse,
    ↪ hamfalsecount)

jarotrue = []
jarotruecount = 0
jarofalse = []
jarofalsecount = 0
jarotruecount, jarofalsecount = compareOverlaps(handoverlaps,
    ↪ jarooverlaps, jarotrue, jarotruecount, jarofalse,
    ↪ jarofalsecount)

jarowinklertrue = []
jarowinklertruecount = 0
jarowinklerfalse = []
jarowinklerfalsecount = 0
jarowinklertruecount, jarowinklerfalsecount = compareOverlaps(
    ↪ handoverlaps, jarowinkleroverlaps, jarowinklertrue,
    ↪ jarowinklertruecount, jarowinklerfalse,
    ↪ jarowinklerfalsecount)

jaccardtrue = []
jaccardtruecount = 0
jaccardfalse = []
jaccardfalsecount = 0
jaccardtruecount, jaccardfalsecount = compareOverlaps(
    ↪ handoverlaps, jaccardoverlaps, jaccardtrue,
    ↪ jaccardtruecount, jaccardfalse, jaccardfalsecount)


multi_overlaps = []
multipleOverlaps(multi_overlaps, levoverlaps, hamoverlaps,
    ↪ jarooverlaps, jarowinkleroverlaps, jaccardoverlaps)

multitrue = []
multitruecount = 0
```

```
multifalse = []
multifalsecount = 0
multitruecount, multifalsecount = compareOverlaps(handoverlaps,
    ↪  multi_overlaps, multitrue, multitruecount, multifalse,
    ↪ multifalsecount)


#Oops, all meaningful data! - this looks messy~~~~
output(multitrue, multitruecount, multifalse, multifalsecount,
    ↪ levtrue, levtruecount, levfalse, levfalsecount, hamtrue,
    ↪  hamtruecount, hamfalse, hamfalsecount, jarotrue,
    ↪ jarotruecount, jarofalse, jarofalsecount,
    ↪ jarowinklertrue, jarowinklertruecount, jarowinklerfalse,
    ↪  jarowinklerfalsecount, jaccardtrue, jaccardtruecount,
    ↪ jaccardfalse, jaccardfalsecount)
build_json(multi_overlaps, arguments)
print("done")
```

7.2 Code - OverlapDetectionUSER.py

```
import os
import re
import json
import itertools
from math import floor
import datetime
import numpy as np
import string
import uuid
import time

#Read the hand detected overlaps file by line creating arrays
    ↪ of each line that contributes to an overlap
#Read in the atom nodes of the supplied SADFace json files
#Run atoms through each algorithm comparing them to each other
#Flag atoms which are "close enough" as overlap
#Return these
#Check against hand detected overlaps
#Return these
```

```
#################################################################################
    ↪
#output data to textfile

def build_json(overlaps, arguments):
    jsonoverlaps = []

    for thisoverlap in overlaps:

        overlap_id = str(uuid.uuid4())
        overlaps = []
        for thisnode in thisoverlap:
            for each in arguments:
                if thisnode == each[1]:
                    overlaps.append(each)

        overlap = {
            "overlap_id": overlap_id,
            "node_id_1": overlaps[0][0],
            "node_text_1": overlaps[0][1],
            "node_id_2": overlaps[1][0],
            "node_text_2": overlaps[1][1]
        }
        jsonoverlaps.append(overlap)

    filepath = os.path.dirname(os.path.realpath(__file__)) +
        ↪ '\\output'
    if not os.path.exists(filepath):
        os.makedirs(filepath)
    currenttime = datetime.datetime.now()
    filename = str(currenttime.day) + str(currenttime.hour) +
        ↪ str(currenttime.minute) + str(currenttime.second) +
        ↪ '.json'

    with open(os.path.join(filepath, filename), 'w') as f:
        json.dump(jsonoverlaps, f)

#################################################################################
    ↪
```

```
#If something is detected by all algorithms, might be worth
    ↪ noting what that is

def multipleOverlaps(multi_overlaps, levoverlaps, hamoverlaps,
    ↪ jarooverlaps, winkleroverlaps, jaccardoverlaps):

    dup_multi = []

    editoverlaps = itertools.product(levoverlaps, hamoverlaps)
    #Levenshtein and Hamming
    for a, b in editoverlaps:
        if a == b:
            dup_multi.append(a)

    ratiooverlaps = itertools.product(winkleroverlaps,
        ↪ jaccardoverlaps)
    #Winkler, and Jaccard
    for a, b in ratiooverlaps:
        if a == b:
            dup_multi.append(a)

    edit_and_ratio = [] #Everything but jaro

    #Get rid of duplicates
    for each in dup_multi:
        if each not in edit_and_ratio:
            edit_and_ratio.append(each)

    final_check = itertools.product(jarooverlaps,
        ↪ edit_and_ratio)
    #Finally, Jaro as our safe guard
    for a, b in final_check:
        if a == b:
            multi_overlaps.append(a)

##########################################################################################
    ↪
#ALGORITHMS HERE

def levenshtein(token1, token2):
```

```
    distances = np.zeros((len(token1) + 1, len(token2) + 1),
        ↪ dtype = int)

    for t1char in range(len(token1) + 1):
        distances[t1char][0] = t1char

    for t2char in range(len(token2) + 1):
        distances[0][t2char] = t2char

    a = 0
    b = 0
    c = 0

    for t1char in range(1, len(token1) + 1):
        for t2char in range(1, len(token2) + 1):
            if (token1[t1char-1] == token2[t2char-1]):
                distances[t1char][t2char] = distances[t1char -
                    ↪ 1][t2char - 1]
            else:
                a = distances[t1char][t2char - 1]
                b = distances[t1char - 1][t2char]
                c = distances[t1char - 1][t2char - 1]

                if (a <= b and a <= c):
                    distances[t1char][t2char] = a + 1
                elif (b <= a and b <= c):
                    distances[t1char][t2char] = b + 1
                else:
                    distances[t1char][t2char] = c + 1

    return (distances[len(token1)][len(token2)])

def hamming(token1, token2):
    distance = 0
    for char in range (len(token2)): #loop for the shortest
        ↪ word
        if token1[char] != token2[char]:
            distance += 1
    distance += (len(token1) - len(token2)) #Add the remaining
        ↪ length of the input
    return distance
```

```
def jaro(token1, token2):
    maxdist = floor(max(len(token1), len(token2)) /2) -1
    match = 0
    chars_t1 = [0] * len(token1)
    chars_t2 = [0] * len(token2)

    if (token1 == token2):
        return 1

    for t1char in range(len(token1)):
        for t2char in range (max(0, t1char-maxdist), min(len(
            ↪ token2), t1char + maxdist+1)):

            if(token1[t1char] == token2[t2char] and chars_t2[
                ↪ t2char] == 0):
                chars_t1[t1char] = 1
                chars_t2[t2char] = 1
                match += 1
                break

    if(match == 0):
        return 0
    transpositions = 0
    point = 0

    for t1char in range (len(token1)):
        if(chars_t1[t1char]):
            while(chars_t2[point] == 0):
                point += 1
            if(token1[t1char] != token2[point]):
                point += 1
                transpositions += 1
    transpositions = floor(transpositions/2)
    jaro_ratio = (match/ len(token1) + match / len(token2) + (
        ↪ match - transpositions + 1) / match)/ 3.0

    return(jaro_ratio)

def jaro_winkler(s1, s2, jaro_ratio):
    prefix = 0
```

```
    for i in range(min(len(s1), len(s2))):
        if (s1[i] == s2[i]) :
            prefix += 1
        else :
            break

    prefix = min(4, prefix) #Maximum 4 characters as indicated
        ↪ by the algorithms description
    winkler = jaro_ratio
    winkler += 0.1 * prefix * (1 - winkler)

    return(winkler)

def jaccard(token1, token2):
    token1chars = []
    for char in token1:
        token1chars.append(char)

    token2chars = []
    for char in token2:
        token2chars.append(char)

    intersection = len(list(set(token1chars).intersection(
        ↪ token2chars))) #AnB
    union = (len(set(token1chars)) + len(set(token2chars))) -
        ↪ intersection #AuB
    jaccard_similarity = (float(intersection) / union) #AnB /
        ↪ AuB
    return(jaccard_similarity)

##########################################################################
    ↪
#Build array of arguments

def getArguments(arguments):
    #Read in the atom nodes of the supplied json files
    #Turn each json file into an array of strings holding the
        ↪ atoms + their ID
```

```
domainpath = (os.path.dirname(os.path.realpath(__file__)) +
    ↪  '\\Pets\\SADFace\\Hand done\\') #<-------- Change
    ↪ directory here
args = os.listdir(domainpath)


for each in args:
        with open(domainpath + each) as arg:

            data = json.load(arg) #JSON to dict
            nodes = data.get('nodes') #Dict to list
            count = len(nodes) #How many nodes long is this
                ↪ json?

            for i in range(count):
                thisarg = [] #temp array to hold text and id,
                    ↪  will append once filled
                try:
                    thisarg.append(data['nodes'][i]['text'])
                        ↪ #Both SADFace and AIFdb's JSON
                        ↪ output of AIF use 'text' under '
                        ↪ nodes', makes this a bit easier
                    try:
                        thisarg.append(data['nodes'][i]['id
                            ↪ '])
                        arguments.append(thisarg) #Only
                            ↪ append an argument with both
                            ↪ text and id
                    except KeyError:
                        try:
                            if(data['nodes'][i]['type'] == "I
                                ↪ "): #if this is AIF, we only
                                ↪  want i-nodes
                                thisarg.append(data['nodes'][i
                                    ↪ ]['nodeID'])
                                arguments.append(thisarg) #
                                    ↪ Only append an argument
                                    ↪ with both text and id
                        except KeyError:
                            print("no id") #uh oh, somethin
                                ↪ wrong with the json
```

```
                except KeyError: #Means no text is present -
                    ↪ its a scheme node likely
                    pass #If it ain't got text, we got no use
                        ↪  for it

##########################################################################################
    ↪
#Run

#Build arrays of atoms for each JSON
arguments = []
getArguments(arguments)

#Arrays for each overlap for each algorithm
levoverlaps = [] #All detected overlaps for an algorithm
hamoverlaps = []
jarooverlaps = []
jarowinkleroverlaps = []
jaccardoverlaps = []

#For each atom in a SADFACE
#Compare to each atom in all SADFaces
#Using itertools combinations to easily compare all elements
    ↪ but only once, two for loops would double up
for a, b in itertools.combinations(arguments, 2):

    a_no_punctuation = a[0].translate(str.maketrans('', '',
        ↪ string.punctuation))
    a_clean = a_no_punctuation.lower()
    b_no_punctuation = b[0].translate(str.maketrans('', '',
        ↪ string.punctuation))
    b_clean = b_no_punctuation.lower()

    thislev = []
    #levenshtein
    levdist = levenshtein(a_clean, b_clean) #0 = the text, 1 =
        ↪ the id
    if (levdist <= 12): #ACCEPTABLE OVERLAP HERE - gotten
        ↪ through testing
        thislev.append(a[1])
        thislev.append(b[1])
```

```
        levoverlaps.append(thislev)

    thisham = []
    #hamming
    if len(a_clean) < len(b_clean):
        hamdist = hamming(b_clean, a_clean)
    else:
        hamdist = hamming(a_clean, b_clean)
    if(hamdist <= 13): #ACCEPTABLE OVERLAP HERE
        thisham.append(a[1])
        thisham.append(b[1])
        hamoverlaps.append(thisham)

    thisjaro = []
    thisjaro_winkler = []
    #jaro + jaro winkler
    jaroratio = jaro(a_clean, b_clean)
    jaro_winkler_ratio = jaro_winkler(a_clean, b_clean,
        ↪ jaroratio)
    if (jaroratio >= 0.71): #ACCEPTABLE OVERLAP HERE
        thisjaro.append(a[1])
        thisjaro.append(b[1])
        jarooverlaps.append(thisjaro)
    if (jaro_winkler_ratio >= 0.8): #ACCEPTABLE OVERLAP HERE
        thisjaro_winkler.append(a[1])
        thisjaro_winkler.append(b[1])
        jarowinkleroverlaps.append(thisjaro_winkler)

    thisjaccard = []
    #jaccard
    jaccard_index = jaccard(a_clean, b_clean)
    if (jaccard_index >= 0.8): #ACCEPTABLE OVERLAP HERE
        thisjaccard.append(a[1])
        thisjaccard.append(b[1])
        jaccardoverlaps.append(thisjaccard)

multi_overlaps = []
multipleOverlaps(multi_overlaps, levoverlaps, hamoverlaps,
    ↪ jarooverlaps, jarowinkleroverlaps, jaccardoverlaps)

build_json(multi_overlaps, arguments)
```

```
print("done")
```

7.3 Code - AIF-to-SADF.py

```
import os
import re

#This script requires the AIF file to be in json format, its
    ↪ also finiky as we are just spliting strings so be sure
    ↪ to verify
#any outputs, the aim of this script was mostly just to speed
    ↪ along the process of hand converting rather than
    ↪ implementing a way
#to completely and reliably convert AIF to SADFace

Domain = '\\Pets'
AIFs = ['\\2229.json']

index = 0
nodes = []
edges = []

for thisfile in AIFs:


    aiffilepath = os.path.dirname(os.path.realpath(__file__)) +
        ↪  Domain + AIFs[index]
    aiffile = open(aiffilepath, 'r')
    aiffilestring = aiffile.read()
    aiffile.close()

    aif = aiffilestring.split('}')

    for each in aif:
        argument = each.split(',')

        for parts in argument:
            case = 0
            thisNode = []
```

```
        components = parts.split(',')
        print(components)
        if '"nodeID":' in components:
            thisNode.append('node: ' + components.split('"
                ↪ nodeID":"',1)[1:-2])
            case = 1
        elif '"text":' in components:
            thisNode.append(components.split('"text":', 1)
                ↪ [1:-2])
        elif '"type":' in components:
            thisNode.append(components.split('"type":', 1)
                ↪ [1:-2])
        elif '"edgeID":' in components:
            thisNode.append('edge: ' + components.split('"
                ↪ edgeID":"',1)[1:-2])
            case = 2
        elif '"fromID":' in components:
            thisNode.append(components.split('"fromID":"',1)
                ↪ [1:-2])
        elif '"toID":' in components:
            thisNode.append(components.split('"toID":"',1)
                ↪ [1:-2])

    if thisNode == []:
        case = 3

    print(thisNode)

    if case == 1:
        nodes.append(thisNode)
    elif case == 2:
        edges.append(thisNode)
    #elif case == 3:
        print('null line')
    #else:
        print('error, line failed to append')

sadfilepath = os.path.dirname(os.path.realpath(__file__)) +
    ↪ Domain + "\\SADFace" + (AIFs[index])
sadfile = open(sadfilepath, "w+").close()
sadfile.write("{\n\t ""edges"": [\n\t\t{")
```

```
    index += 1
```

7.4 SADFace JSON

The JSON files used are contained within the git repository (Direct Link:`https://github.com/ConnorNess/Argument-Overlap-Detection/tree/main/Pets/SADFace/Hand%20done`)

7.5 Reports and Plans

7.5.1 Research Proposal

```
Project Outline
        Title: Argument Mining âĂŞ Simple Argument Description
            ↪ Format (SADFace) Overlaps
    Supervisor: Dr Simon Wells

Background:
The field of argument mining is tied with natural language
    ↪ processing (NLP) due to the aim of automatic detection
    ↪ and analysis of arguments in real world interactions.
    ↪ NLP aims to retrieve and understand contextually what
    ↪ the meaning of real world spoken language automatically
    ↪ by computational methods. NLP assists the field of
    ↪ argument mining through its ability to supply analysed
    ↪ texts and trends in language which may contain
    ↪ structured arguments. Various methods are used within
    ↪ NLP to identify language, part of speech tagging (nouns,
    ↪  verbs, adjectives, etc.) or chunking âĂŞ labelled
    ↪ segments of sentences such as noun and verb phrases.

Argument mining aims to implement a solution which will
    ↪ automatically identify and analyse an argument within
    ↪ natural language. Mining requires the correct
    ↪ identification of what is âĂIJimportantâĂİ to an argument
    ↪  and what is not, able to remove the excess of an
    ↪ argument without the argument losing any depth or
    ↪ meaning in the process. Mining uses, adapts, and
```

```
    ↪ develops techniques in machine learning as methods of
    ↪ identification within written text.

Arguments are a form of critical thinking where the aim of the
    ↪ discussion is to convince an opposition to reach the
    ↪ same conclusion when both parties may share the same
    ↪ information.

SADFace is a type of argument description language similar to
    ↪ Argument Markup Language and Argument Interchange Format
    ↪  which describes an analysed structured argument with
    ↪ the aim to construct argument tools for the web that can
    ↪  be simply used without the requirement of specialist
    ↪ software. Arguments are constructed in strings stored as
    ↪  âĂIJatomsâĂİ linked in a directed graph format, atoms
    ↪ are not linked directly towards other atoms but via
    ↪ scheme nodes therefor every link an atom has must be
    ↪ valid in the form of a known argumentative relation,
    ↪ these relations between strings and their connectivity
    ↪ are stored in a JSON document.

Overlap within SADFace could represent different statements
    ↪ that express the same argument, the goal of this project
    ↪  will be to identify these similarities. A potential
    ↪ method of detection would be through string comparisons.
    ↪  Several string comparison algorithms can be used to
    ↪ compare strings more accurately in natural language than
    ↪  simply checking if a word is equal to another word in
    ↪ value. The Levenshtein Distance is a method of measuring
    ↪  the distance or âĂIJeditâĂİ between two strings, the
    ↪ output of this function is the number of edits to a
    ↪ string that must be performed in order to change one
    ↪ string into the other; for instance, the distance
    ↪ between âĂIJcatâĂİ and âĂIJcarbâĂİ is 2 âĂŞ transforming
    ↪ âĂIJtâĂİ into âĂIJrâĂİ and then adding âĂIJbâĂİ to the end
    ↪ will yield the desired string. This output of 2 may also
    ↪  be represented as a ratio âĂŞ this case being a 71.4%
    ↪ similarity ratio. Comparing the vector space
    ↪ representation between strings, converting sentences
    ↪ into vector values, and comparing different values to
    ↪ find the similarity represented as a ratio. Word
```

```
    ↪ comparisons can be tested through a metaphone algorithm,
    ↪  an algorithm which transforms individual letters in a
    ↪ string based on a series of rules to output the phonetic
    ↪  âĂIJspellingâĂİ of the word.

The proposed algorithms are potential ways to detect similarity
    ↪  in statements, to evaluate the effectiveness of the
    ↪ algorithm implementation a data set of arguments will be
    ↪  constructed and the output of these will be checked
    ↪ with the desired goal of correct identification of known
    ↪  similar arguments, alongside a reduction of false
    ↪ positives and negatives âĂŞ for example in natural
    ↪ language it can be common to use a word of affirmation
    ↪ such as âĂIJyesâĂİ followed by a disagreement, trends
    ↪ such as this could result in the algorithm falsely
    ↪ flagging statements as in agreement when the opposite is
    ↪  true. As a larger and more real-world test the
    ↪ algorithm will be executed on a large scale dataset âĂŞ
    ↪ multiple databases such as AIFdb or Araucaria Database
    ↪ exist as a corpus of analysed arguments which will
    ↪ supply this dataset.


Project Outline:
The idea for this research arose from:
Proposal on http://arg.napier.ac.uk/admin/studentprojects/

The aims of the project are as follows:
To develop an algorithm to assist in the automatic detection of
    ↪  overlaps in SADFace arguments.

The main research questions that this work will address include
    ↪ :
How overlap detection contributes to the field of argument
    ↪ mining.
Potential methods of overlap detection.
Further enhancements that could add depth to the algorithm.

The software development/design work/other deliverable of the
    ↪ project will be:
```

```
A contribution towards the SADFace repository for further
    ↪ development out with this project.
An adaptable algorithm designed to detect argument overlaps in
    ↪ the SADFace format.

The project deliverable will be evaluated as follows:
The effectiveness of the deliverable will be based on how
    ↪ likely the algorithm is able to identify these overlaps,
    ↪  The algorithm will be tested against several human
    ↪ identified overlaps and its conclusion will be noted as
    ↪ correct or not.

The project will involve the following research/field work/
    ↪ experimentation/evaluation:
Research into argument mining as a field to inform the reader
    ↪ and ensure they can comprehend the further research
    ↪ being completed.

This work will require the use of specialist software:
No

This work will require the use of specialist hardware:
No

The project is being undertaken in collaboration with:
None


References:
(Collobert, R. 2011, Natural Language Processing (Almost) from
    ↪ Scratch, pp. 1-4)
(Wells, S. 2014, Argument Mining: Was Ist Das?, pp. 1-3)
(van Emeren, 2014, Handbook of Argumentation Theory, pp. 4-6)
(Ayyadevara, K. 2018, Pro Machine Learning Algorithms, pp.
    ↪ 167-172)
(Visser, J. 2020, Argumentation in the 2016 US presidential
    ↪ elections, pp. 125-129)
```

7.5.2 Work Plan

7.1

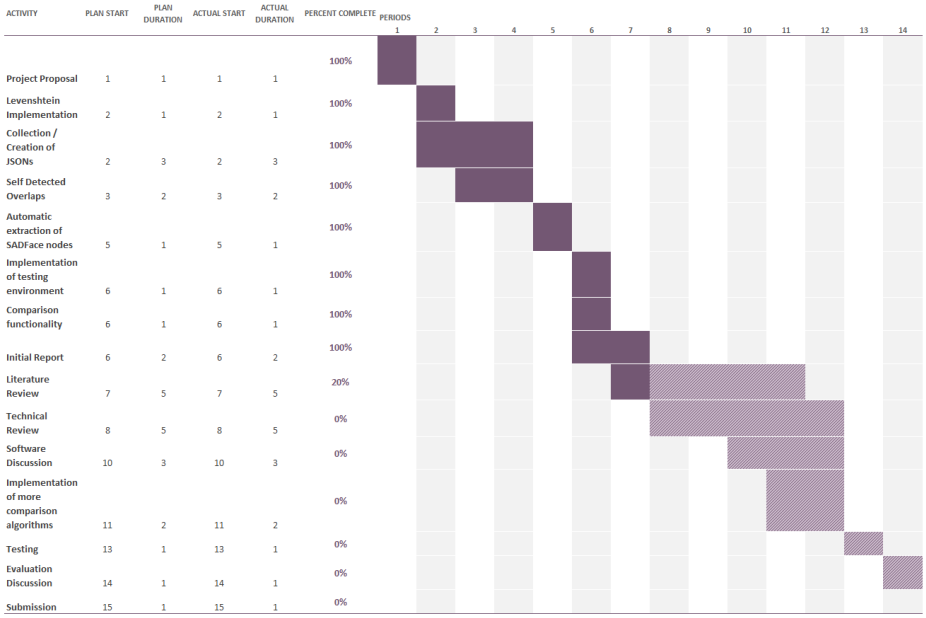

Figure 7.1: Work Plan Timeline

7.5.3 Initial Report

```
Project Title: Identifying Argument Overlaps in SADFace

Summary:
Several databases exist which aim to archive argument
    ↪ description markup languages by type such as AIF (
    ↪ Argument Interchange Format), these representation
    ↪ schemes exist to assist the inclusion of argument data
    ↪ in software. SADFace (Simple Argument Description Format
    ↪ ) is one of these representation schemes which
    ↪ constructs a JSON document connecting segmented argument
    ↪  structure.

A SADFace document will contain atom nodes âĂŞ these are
    ↪ uniquely identifiable statements within an argument, and
    ↪  edges âĂŞ the connections between these statements.
```

```
Within online databases such as AIFDB, several argument
    ↪ structures can contain multiple statements with the same
    ↪  meaning except these statements are expressed in
    ↪ different ways, these can be described as overlapping
    ↪ arguments.

This project aims to implement a methodology which can assist
    ↪ in the automatic detection of these overlaps through the
    ↪  use of several string comparison techniques.

Multiple hand collected or created SADFace documents from
    ↪ another argument representation such as AIF are
    ↪ collected from the same domain where known overlaps in
    ↪ statements exist, these overlaps are identified and
    ↪ collected into a document for evaluation purposes.

The implemented python script will extract the argument
    ↪ statements from supplied JSON files and then compare
    ↪ these statements to each other through several string
    ↪ comparison algorithms with the aim of detecting where
    ↪ two statements are similar.

For testing these identified overlaps are then compared to the
    ↪ previously collected known overlaps so the algorithms
    ↪ can be adjusted in order to more accurately detect an
    ↪ overlap.

Literature Review:
Natural Language Processing (NLP) is a linguistic based
    ↪ methodology utilising artificial intelligence and
    ↪ machine learning to extract, process, and analyse
    ↪ written text âĂŞ this could be from a book, online
    ↪ discussion, or transcript video. The functionality and
    ↪ utilisation of Natural Language Processing aims to
    ↪ convert natural language into data structures and code
    ↪ comprehendible by a computational structure. NLP has had
    ↪  major relevancy to modern implementations such as the
    ↪ rise in virtual assistants like Cortana for Windows 10,
    ↪ these assistants require the ability to take in natural,
    ↪  non-programmatically formatted inputs of language to
```

```
    ↪ then process the desired output and then relay this
    ↪ output to the user again in a naturally spoken way. NLP
    ↪ is a subfield of general speech and language processing
    ↪ which aims to model language on a rule-based system.
Difficulty in NLP comes from the nature of natural language
    ↪ being vague and non-conforming to all participants of a
    ↪ language. Several factors such as double meaning words,
    ↪ tone, the relation between those participating in
    ↪ conversation, even body language and misunderstandings
    ↪ in a conversation will affect the flow of a conversation
    ↪  which can be incredibly difficult to determine from a
    ↪ simple transcript of speech. The detection of these
    ↪ facets of language can be both essential to
    ↪ comprehension of a dialogue, and simultaneously unknown
    ↪ to a viewer of this dialogue. For example, if two radio
    ↪ hosts were discussing a topic and one was becoming
    ↪ visibly agitated by the conversation but not in tone, a
    ↪ listener would not be able to fully comprehend the
    ↪ emotion displayed by the speaker which could be
    ↪ important to understanding the meaning behind an
    ↪ argument or where the speaker is drawing their
    ↪ conclusions from. In a scenario such as this, a
    ↪ computational structure would have to be able to notice
    ↪ discrepancies such as tone difference, adjustments in
    ↪ speech, note when misunderstandings are made in order to
    ↪  exactly meet the needs of the user, these factors are
    ↪ not always able to be noted by any participant other
    ↪ than the speaker at times. In an online discussion, text
    ↪  cannot sufficiently relate linguistic tools such as
    ↪ sarcasm effectively and for a speaker to indicate a
    ↪ technique like this is being used can undermine the flow
    ↪  of an argument however the risk comes that the
    ↪ opposition will not notice this tool. Misunderstandings
    ↪ such as this can be difficult in certain scenarios for
    ↪ those who use the natural language to detect which
    ↪ translates to an algorithm having further difficulty in
    ↪ comprehension of a conversation.
Essentially the depth of communication between people
    ↪ contributes to the difficulty of implementations in NLP,
    ↪  language cognition aims to understand how the cognitive
    ↪  functions of the brain interpret and process language.
```

```
    ↪ Linguistics is the scientific study of language in use
    ↪ several topics within linguistics contribute to
    ↪ understanding of natural language such as semantics âĂŞ
    ↪ the meaning of words and their relationship to each
    ↪ other to form phrases.

Arguments are a form of critical thinking within natural
    ↪ language where the participants conduct a dialogue with
    ↪ the goal to convince an opposition dialogue to reach the
    ↪  same conclusion when the contributing parties may have
    ↪ conflicting viewpoints on a topic. Argument mining aims
    ↪ to implement solutions to automatically identify,
    ↪ process, and analyse an argument within natural language
    ↪ . Mining requires the correct identification of what is
    ↪ âĂIJimportantâĂİ to an argument and what is not within
    ↪ the context of the discussion, the ability to remove non
    ↪ -important dialogue or to extract the backbone of an
    ↪ argument without losing any depth, meaning, or relevancy
    ↪  in the process.

An issue presented by the processes of argument mining is the
    ↪ ability to present this automatic output in an
    ↪ analysable fashion. Argument markup languages aim to
    ↪ solve this issue by creating tools which assist in the
    ↪ visualisation and construction of processed arguments.
    ↪ Argument maps and diagrams are methods of this
    ↪ visualisation where statements are linked together by
    ↪ how each statement supports or is in opposition to
    ↪ another statement.
The Argument Markup Language (AML) is a simplistic tree format
    ↪ saved as an XML file which allows for the transmission
    ↪ of an argument between packages, AML formatted arguments
    ↪  populate the database AraucariaDB which is tied to the
    ↪ software tool for argument analysis âĂŞ Araucaria.
    ↪ Araucaria assists argument diagramming and
    ↪ reconstruction which ties nodes (which contain the
    ↪ actual text of an argument) to schemes (which represent
    ↪ how nodes relate to each other). However, the Argument
    ↪ Markup Language being in XML format can make it
    ↪ generally difficult to parse information from without
    ↪ the use of specialised software like Araucaria as
```

```
    ↪ mentioned which majorly aids the creation of argument
    ↪ maps.
Argument Interchange Format (AIF) was designed to help
    ↪ alleviate the shortcomings of the Argument Markup
    ↪ Language due to AML being designed for use in specialist
    ↪  software. Designed with the aim of facilitating the
    ↪ development of multiple-agent systems where individuals
    ↪ are capable of argument-based interactions and to
    ↪ facilitate data sharing with tools for argumentation
    ↪ manipulation and visualization. AIF has two node types
    ↪ âĂŞ information and scheme, which are used to represent
    ↪ information within an argument. Schemes can further
    ↪ contain multiple subtypes âĂŞ Inference or supporting
    ↪ statements, Conflict or opposition to a statement, and
    ↪ Preference. Preference schemes relate to when an
    ↪ argument is âĂIJpreferredâĂİ or viewed as stronger within
    ↪  an argument for example an argument from an expert will
    ↪  be preferred when compared to an argument from an
    ↪ uninformed participant. The âĂIJedgesâĂİ or links between
    ↪  nodes are inferred within AIF from which nodes are
    ↪ present and is not input by an analyst. The Argument
    ↪ Interchange FormatâĂŹs depth provides useful high-level
    ↪ representation in arguments, but this flexibility of
    ↪ this depth means users must understand the complexity
    ↪ that comes with AIF in order to work with the format.
Enter SADFace, the Simple Argument Description Format. A JSON
    ↪ based document format designed to be simple and capable
    ↪ of being adapted into pre-existing systems. SADFace
    ↪ being of a simplistic JSON format means it will not
    ↪ require specialised software in order to interact with
    ↪ and is laid out to represent an understandable model of
    ↪ argumentation structure.
The JavaScript Object Notation format (JSON) is a subset of
    ↪ JavaScript but is not itself a programming language but
    ↪ instead a data interchange format. JSON is designed only
    ↪  to carry information to be used in the communication
    ↪ between multiple processes requiring formatted data
    ↪ exchange. JSON is essentially a text representation
    ↪ structured as either a list of name/value pairs or a
    ↪ list of values where sets of pairs are separated by
    ↪ curled braces âĂIJ{ }âĂİ and individual values are named
```

```
    ↪ for example âĂIJtextâĂİ, divided by a colon âĂIJ:âĂİ and
    ↪ then followed by the value âĂIJgeneric sentence here.âĂİ
    ↪ Within sets, arrays or lists can be declared by use of
    ↪ square brackets âĂIJ[ ]âĂİ for example âĂIJ[âĂIJsentence 1
    ↪ âĂİ, âĂIJsentence 2âĂİ].
SADFace represents claims in an argument as argument atoms,
    ↪ notated as âĂIJatomsâĂİ within the document, an argument
    ↪ will consist of multiple of these atoms linked to form a
    ↪  flow of statements, these atoms have unique identifiers
    ↪ . In a structured argument a concluding atom will need
    ↪ to be supported by a premise atom. This support is
    ↪ displayed through âĂIJschemesâĂİ which is used to
    ↪ represent argumentative reasoning such as âĂIJsupportâĂİ
    ↪ or âĂIJconflictâĂİ as atoms will relate to each other in
    ↪ an argument, separate atoms may be in contention of a
    ↪ conclusion or may assist in the reasoning behind a
    ↪ conclusion. Atoms and schemes are connected by edges
    ↪ with a direction to assert the relation dynamic between
    ↪ nodes. These edges are uniquely identifiable and follow
    ↪ a source atom to a target which are retrieved by use of
    ↪ the atomâĂŹs unique identifier. Similar to AIF, atoms
    ↪ cannot be linked together directly and must go via a
    ↪ scheme node as separate statements must have a known
    ↪ relationship for the link to be meaningful in an
    ↪ argument.

In order to evaluate the difference between two nodes in
    ↪ SADFace, the text of each node can be evaluated through
    ↪ string comparison algorithms. These algorithms aim to
    ↪ measure the similarity between two strings through
    ↪ comparisons.
Levenshtein distance is one of these evaluations, the distance
    ↪ is calculated by how many edits must be performed on a
    ↪ string for it to match another. For example, the
    ↪ distance between âĂIJcarâĂİ and âĂIJmanyâĂİ is 3 âĂŞ the
    ↪ âĂŸcâĂŹ is converted to an âĂŸmâĂŹ, the âĂŸrâĂŹ is
    ↪ converted to an âĂŸnâĂŹ, and âĂŸyâĂŹ. This distance is
    ↪ simplistic but effective for similar length strings.
Jaro similarity measures a ratio of two strings between two 0
    ↪ and 1, where 0 is no similarity and 1 is an exact equal
    ↪ match. Jaro evaluates the number of matching characters
```

```
   ↪ against the number of changes needed to be made to have
   ↪ the strings equal with considerations to the lengths of
   ↪ each string. Say string 1 is âĂŸbusyâĂŹ and string 2 is
   ↪ âĂŸgasâĂŹ the algorithm will take the number of matching
   ↪  characters âĂŞ âĂŸmâĂŹ = 1, the number of changes âĂŞ
   ↪ âĂŸtâĂŹ = 3.
(1/3)*{(1(m)/4(string 1 length) + 1(m)/3(string 2 length) + 1(m
   ↪ )-3(t)/1(m)} = 0.8611
Jaro-Winkler advances Jaro by considering a prefix scale to
   ↪ calculate a more accurate output to where strings have
   ↪ common prefixes of set length.
Jaro-Winkler = Jaro + Scaling Factor * Length of prefix *(1 âĂŞ
   ↪  Jaro)
Sw = Sj + P * L * (1 âĂŞ Sj)

Python is utilised in a large amount of natural language
   ↪ processing topics due to its functionality in machine
   ↪ learning and numerous libraries contributing to NLP such
   ↪  as the Natural Language ToolKit (NLTK) have made it the
   ↪  de facto language for natural language processes. As a
   ↪ scripting language with powerful standard libraries and
   ↪ an extensive and growing number of APIs (Application
   ↪ Programming Interfaces) python is extremely useful as an
   ↪  environment to immediately start development of a
   ↪ solution or to test a new algorithm by use of functions
   ↪ which can also be attributed to its ease of text
   ↪ processing and creation of dictionaries or lists.
   ↪ PythonâĂŹs json library can be used to easily populate a
   ↪  dictionary (or dict) with simple commands such as
   ↪ âĂŸjson.load(file)âĂŹ which helps in this project as
   ↪ after loading a json file, we can simply use âĂŸfile.get
   ↪ (âĂŸnodesâĂŹ)âĂŹ to retrieve all our SADFace nodes.


*References are below 5. Work Plan*

Contents:
1. Introduction
a. Project Background
Description of the project - aims and objectives and potential
   ↪ constraints.
```

```
b. Lifecycle
Further use of the project, how the project can evolve with
    ↪ further development and practical use of the current
    ↪ implementation.
c. Structure of dissertation
Description of sections to be discussed.
2. Literature Review
a. Discussion of literature important to the project, what a
    ↪ reader will need to be informed of in order to fully
    ↪ comprehend the work being undertaken and how the
    ↪ literature researched has affected the project.
3. Technical Review
a. Discussion and breakdown of SADFace, how documents are
    ↪ represented and used.
b. Libraries utilised in the project and their functionality,
    ↪ justification for libraries used.
4. Software
a. Full description of the implementation and discussion of
    ↪ changes made during development, issues faced and how
    ↪ they were overcome.
5. Testing
a. How the final implementation has been adjusted to identify
    ↪ as many overlaps as possible with the methods used for
    ↪ comparison, how the software can process datasets not
    ↪ used for development.
6. Evaluation
a. How the implementation met or did not meet the desired aims.

References:
Beysolow, T. (2018). Applied Natural Language Processing with
    ↪ Python. pp1-12.
Kysela, J. (2018) . A comparison of text string similarity
    ↪ algorithms for POI name harmonisation.
Mochales, R. (2011). Argumentation Mining.
Sarkar, D. (2016). Text Analytics with Python. pp69-93.
Smith, B. (2015). Beginning JSON. pp37-80.
Lippi, M. (2015). Argument Mining: A Machine Learning
    ↪ Perspective.
Modgil, S. (2007). Towards Characterising Argumentation Based
    ↪ Dialogue in the Argument Interchange Format.
Rahwan, I. (2009). The Argument Interchange Format.
```

```
Sherborne, T. (2008). Argument Diagramming: The Araucaria
    ↪ Project.



The Argument Interchange Format (AIF) Specification (2018)
Open-Argumentation, 2017, SADFace (https://github.com/Open-
    ↪ Argumentation/SADFace)
```

7.6 Supervision Meetings

7.6.1 14-06-21

```
Write about potential methodologies
If sadface arguments line up, they could be merged to pull up
    ↪ context
large numbers arguments as a map how they chain together to
    ↪ arrive at a similar conclusion
tricky potential - what entails matching sentence (levenshtein
    ↪ distance)
potential start = surveying methods for comparing strings
potential difficult comparrison, where every string is
    ↪ different but the meaning is the same
    i.e. different ways to affirm agreement
vector space representations of strings
    sentece2vec

may be worth discussing AML or AIF
where different atoms can be merged

fork SADFace using open argumentation version
    implementation likely within python file or side file
potential function for sadface overlaps to return single
    ↪ document

starting with simple levenshtein distance implementation in the
    ↪  fork could be good
    python 3 standard library fine to limit number of installs
```

```
good evaluation
    test data set
        to avoid false positives and negatives, things I know
            ↪ can be true

potential workshop "late" publication 3-4 pages as a late
    ↪ submission
argtech aif repository

Monday @ 11am weekly meetings

messages

from Simon Wells to everyone: 3:11 PM
levenshtein distance
from Simon Wells to everyone: 3:12 PM
surveying methods for comparing strings
from Simon Wells to everyone: 3:14 PM
vector space representations of strings
from Simon Wells to everyone: 3:14 PM
sentence2vec
from Simon Wells to everyone: 3:20 PM
https://github.com/Open-Argumentation
```

7.6.2 21-06-21

```
same, similar, different arguments as markers
    same and similar should be defined differently
don't push too far past goals, just need automatic melding of
    ↪ SADFace
    if a potential further implementation is thought of, put it
        ↪  down to write about later
make a plan for easy stuff
    dataset ASAP
        maybe SADFace by hand in a matching domain (games,
            ↪ climate change)
            if I don't know the topic, may not notice the
                ↪ details
        gold standard ones later where I know what the outcome
            ↪ should be
```

```
    starting place at arg tech for datasets of argument maps
        convert to SADFace
            may be a nice side deliverable
            potential script to do so
    get tons of string comparrison algorithms
pip install sadface
    Own workspace, don't worry about breaking dependancies on
        ↪ SADFace as a whole
3 weeks till workshop deadline, get as much work done till then


messages

from Simon Wells to everyone: 11:14 AM
https://arg-tech.org
from Simon Wells to everyone: 11:15 AM
http://corpora.aifdb.org
```

7.6.3 05-07-21

```
Initial report really useful for a flagpost of where I should
    ↪ be/ looking to how much I think I have left buuuuuuut
    not marked, get it done and get back to work
finish off the first draft of implementation, publish it to git
    ↪ , add simon as collaborator
issues with current implementation
    .get(nodes) then get individual nodes
    comparison function bugging out for some reason

TODO:
Fix implementation and publish
get initial report done
    need a couple more hundred words for preview lit review
```

7.6.4 12-07-21

```
to mention: if have time may adapt levenshtein to damerau
    ↪ levenshtein
```

```
get number from putting multiple distances
    might get a nice curve, good for testing
might be worth measuring processing time for algorithms
changing how overlaps are written out
loading in monkey puzzle for good visualisation
make readme, how to run
```

7.6.5 19-07-21

```
tool useful for text overlaps, rather than completely manually.
tool good for assistance in human work rather than replacement
accuracy is good, but this tool could just flag things to be
    ↪ reviewed by an expert - people get lazy doing the same
    ↪ forever, lets try and streamline it
are all algorithms failing at same point? succeeding?
are some detected completely in one but not the other?
How to represent the data output - charts
    time of algorithms
    lengths of sentences
    number of overlaps
```

7.6.6 26-07-21

```
plot all on same graph
papers on algorithm comparisson
    any standard datasets
pass = implementation + output
good pass = depth in output discussion
```

7.6.7 09-08-21

```
current plan
-be done writing ideally friday (13th - 5 days before deadline)
-spend weekend creating a seperate non-testing version of the
    ↪ application (just remove all the extraction and hand
    ↪ comparisons),
    also creating my synonym antonym algorithm for funsies
-submit monday
```

```
all jsons in appendicies?? maybe - pages are free

intro good
    needs to be added to literature review
    replace with talking about the actual project

lit rev conclusion is more of a summary
potential diagram illustrating merged argument from conclusion
    ↪ of another being premise of a new on merged

First sentence of intro could be refined
    nlp methodology - change to approach
    extraction, process, analyses, and generation
    as discussed by, kind of clunky - [cite] claim, or [cite]
        ↪ point
    person isn't important - what they said is, we're just
        ↪ defending statment

title change, sadface not vital - current one doesn't do the
    ↪ work I've done justice
```

References

Ajjour, Y., Wachsmuth, H., Kiesel, J., Potthast, M., Hagen, M., and Stein, B. (2019). Data acquisition for argument search: The args.me corpus. In Benzmüller, C. and Stuckenschmidt, H., editors, *KI 2019: Advances in Artificial Intelligence*, pages 48–59, Cham. Springer International Publishing.

Ayyadevara, V. K. (2018). Word2vec. In *Pro Machine Learning Algorithms*, pages 167–178. Springer.

Bex, F., Lawrence, J., Snaith, M., and Reed, C. (2013). Implementing the argument web. *Communications of the ACM*, 56(10):66–73.

Beysolow II, T. (2018). *What Is Natural Language Processing?*, pages 1–12. Apress, Berkeley, CA.

Bird, S. (2006). Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, pages 69–72.

Bird, S., Klein, E., and Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Bornmann, L. and Mutz, R. (2015). Growth rates of modern science: A bibliometric analysis based on the number of publications and cited references. *Journal of the Association for Information Science and Technology*, 66(11):2215–2222.

Budanitsky, A. and Hirst, G. (2006). Evaluating wordnet-based measures of lexical semantic relatedness. *Computational linguistics*, 32(1):13–47.

Chen, G. and Zhang, S. (2018). Fcamapx results for oaei 2018. In *OM@ ISWC*, pages 160–166.

Chesnevar, C., Modgil, S., Rahwan, I., Reed, C., Simari, G., South, M., Vreeswijk, G., Willmott, S., et al. (2006). Towards an argument interchange format. *The knowledge engineering review*, 21(4):293–316.

Cohen, W., Ravikumar, P., and Fienberg, S. (2003). A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation*, volume 3, pages 73–78.

Droettboom, M. et al. (2015). Understanding json schema. *Available on: http://spacetelescope. github. io/understanding-jsonschema/UnderstandingJSONSchema. pdf (accessed on 14 April 2014)*.

Green, N. (2014). Towards creation of a corpus for argumentation mining the biomedical genetics research literature. In *Proceedings of the first workshop on argumentation mining*, pages 11–18.

Hearst, M. A. (2005). Teaching applied natural language processing: Triumphs and tribulations. In *Proceedings of the Second ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pages 1–8.

Kim, S.-M. and Hovy, E. (2006). Extracting opinions, opinion holders, and topics expressed in online news media text. In *Proceedings of the Workshop on Sentiment and Subjectivity in Text*, pages 1–8.

Kysela, J. (2018). A comparison of text string similarity algorithms for poi name harmonisation. In *International Conference on Articulated Motion and Deformable Objects*, pages 121–130. Springer.

Lauscher, A., Glavaš, G., and Ponzetto, S. P. (2018). An argument-annotated corpus of scientific publications. In *Proceedings of the 5th Workshop on Argument Mining*, pages 40–46.

Lawrence, J., Janier, M., and Reed, C. (2015). Working with open argument corpora. In *European Conference on Argumentation (ECA)*.

Lawrence, J. and Reed, C. (2014). Aifdb corpora. In *COMMA*, pages 465–466.

Lawrence, J. and Reed, C. (2020). Argument mining: A survey. *Computational Linguistics*, 45(4):765–818.

Lin, D. (1997). Using syntactic dependency as local context to resolve word sense ambiguity. In *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics*, pages 64–71.

Lin, D. et al. (1998). An information-theoretic definition of similarity. In *Icml*, volume 98, pages 296–304.

Lippi, M. and Torroni, P. (2015). Argument mining: A machine learning perspective. In Black, E., Modgil, S., and Oren, N., editors, *Theory and Applications of Formal Argumentation*, pages 163–176, Cham. Springer International Publishing.

Loper, E. and Bird, S. (2002). Nltk: The natural language toolkit. *arXiv preprint cs/0205028*.

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., and McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Mochales, R. and Moens, M.-F. (2011). Argumentation mining. *Artificial Intelligence and Law*, 19(1):1–22.

Modgil, S. and McGinnis, J. (2008). Towards characterising argumentation based dialogue in the argument interchange format. In Rahwan, I., Parsons, S., and Reed, C., editors, *Argumentation in Multi-Agent Systems*, pages 80–93, Berlin, Heidelberg. Springer Berlin Heidelberg.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

Rahwan, I. and Reed, C. (2009). The argument interchange format. In *Argumentation in artificial intelligence*, pages 383–402. Springer.

Reed, C. (2006). Preliminary results from an argument corpus. *Linguistics in the twenty-first century*, pages 185–196.

Reed, C. and Rowe, G. (2001). Araucaria: Software for puzzles in argument diagramming and xml. *Department of Applied Computing, University of Dundee Technical Report*, pages 1–21.

Reed, C., Wells, S., Devereux, J., and Rowe, G. (2008). Aifˆ+: Dialogue in the argument interchange format. *Frontiers in artificial intelligence and applications*, 172:311.

Resnik, P. (1995). Using information content to evaluate semantic similarity in a taxonomy. *arXiv preprint cmp-lg/9511007*.

Rowe, G. and Reed, C. (2008). Argument diagramming: The araucaria project. In *Knowledge cartography*, pages 164–181. Springer.

Sarkar, D. (2016). *Python Refresher*, pages 51–106. Apress, Berkeley, CA.

Smith, B. (2015). *Introducing JSON*, pages 37–47. Apress, Berkeley, CA.

Sun, Y., Ma, L., and Wang, S. (2015). A comparative evaluation of string similarity metrics for ontology alignment. *Journal of Information &Computational Science*, 12(3):957–964.

Thanaki, J. (2017). *Python natural language processing*. Packt Publishing Ltd.

Visser, J., Duthie, R., Lawrence, J., and Reed, C. (2018). Intertextual correspondence for integrating corpora. In *11th International Conference on Language Resources and Evaluation, LREC 2018*, pages 3511–3517. European Language Resources Association.

Wachsmuth, H., Potthast, M., Al Khatib, K., Ajjour, Y., Puschmann, J., Qu, J., Dorsch, J., Morari, V., Bevendorff, J., and Stein, B. (2017). Building an argument search engine for the web. In *Proceedings of the 4th Workshop on Argument Mining*, pages 49–59.

Wagner, W. (2010). Steven bird, ewan klein and edward loper: Natural language processing with python, analyzing text with the natural language toolkit. *Language Resources and Evaluation*, 44(4):421–424.

Walker, V., Vazirova, K., and Sanford, C. (2014). Annotating patterns of reasoning about medical theories of causation in vaccine cases: Toward a type system for arguments. In *Proceedings of the First Workshop on Argumentation Mining*, pages 1–10.

Wei, J. (2004). Markov edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(3):311–321.

Wei, M. (1993). *An analysis of word relatedness correlation measures*. PhD thesis, Faculty of Graduate Studies, University of Western Ontario.

Wells, S. (2020a). Datastores for argumentation data. In *CMNA@ COMMA*, pages 31–40.

Wells, S. (2020b). The open argumentation platform (oapl). In *Computational Models of Argument*, pages 475–476. IOS Press.

Znamenskij, S. (2015a). A belief framework for similarity evaluation of textual or structured data. In *International Conference on Similarity Search and Applications*, pages 138–149. Springer.

Znamenskij, S. (2015b). A substrings based approach to data similarity evaluation. pages 1–55.