Processing of ICARTT Data Files Using Fuzzy Matching and Parser Combinators

Matthew T. Rutherford[1], **Nathan D. Typanski**[2], **Dali Wang**[3], **Gao Chen**[4]
[1,2,3] Department of Physics, Computer Science & Engineering,
Christopher Newport University, Newport News, VA, United States
[4] Science Directorate, NASA Langley, Hampton, VA, United States

Abstract—*In this paper, the task of parsing and matching inconsistent, poorly formed text data through the use of parser combinators and fuzzy matching is discussed. An object-oriented implementation of the parser combinator technique is used to allow for a relatively simple interface for adapting base parsers. For matching tasks, a fuzzy matching algorithm using Levenshtein distance calculations is implemented to match string pairs, which are otherwise difficult to match due to the aforementioned irregularities and errors in one or both pair members. Used in concert, the two techniques allow parsing and matching operations to be performed which had previously only been done manually.*

Keywords: fuzzy matching, Levenshtein distance, parser combinator

1. Introduction

Enabling a computer to recognize and parse irregular and poorly formatted data can be handled in multiple ways. Many traditional parsing approaches include an initial lexical analysis phase, in which a given character sequence is tokenized, followed by a parse phase [1], [2]. These techniques suffice for compilers and other common parsing applications, but when faced with inputs in which unique edge cases are common, the grammar and parse rules must be adapted often in order to properly handle this kind of input. We found the lexxer-parser combination approach sluggish and cumbersome when novel errors in the input were arising frequently throughout the parsing process.

Parser combinators offered a viable solution, allowing the consolidation of multiple simple parsers to form a series of conglomerate parsers, which are then able to collectively address irregular, complex input. For our purposes, the combinator approach proved faster compared to frequently writing and rewriting lexxers and parsers when our program encountered input that would not parse properly. We could write a parser quickly, test it on the new input, and rewrite it as needed.

In order to produce the intended output, our program needed to reference an external knowledge base. However, this knowledge base was not guaranteed to correspond perfectly with the input data. Due to this varying degree of correspondence, fuzzy matching was used in multiple stages of this process to determine which portions of the knowledge base apply. The program had to ensure that no overlapping matches occurred: it needed a way to perform imperfect string matching such that the best one-to-one mapping with the knowledge base could be found.

1.1 The Extended ICARTT Data Format Standard

The International Consortium for Atmospheric on Transport and Transformation (ICARTT), a scientific airborne research group, created the Extended ICARTT Data Format Standard [3] in 2004 to provide a well-defined, text-based file format for atmospheric data to help ensure consistency across the several hundred files produced per data collection mission. ICARTT files are broken into two main sections: a data section and a corresponding metadata section. The metadata section is where all contextual information corresponding to the data section is given. This portion of the file is made up of several subsections. The data section is where the actual data recorded by the various aircraft-mounted instruments as well the corresponding time data are given. Raw data are given as either comma delimited or space delimited columns of decimal values. Each data column corresponds to a variable given in the above metadata section and can be several hundred to several thousand lines long, depending on the length of the data collection period for a given file.

1.2 Purpose of Research

The decentralized structure of the ICARTT data collection campaigns resulted in errors and inconsistencies—some quite subtle—among data files. Thousands of raw data files have been produced, which need to be edited and reformatted in order to meet the Extended ICARTT File Format Standard before being added to the central Toolset for Airborne Data (TAD) archive.

In the past, reformatting of ICARTT files had been done manually, file by file. The purpose of the program described here is to automate this process as much as possible in order to quickly reformat large sets of raw ICARTT files with little-to-no manual effort or editing. To this end, the aforementioned parser combinator and fuzzy matching algorithms were implemented.

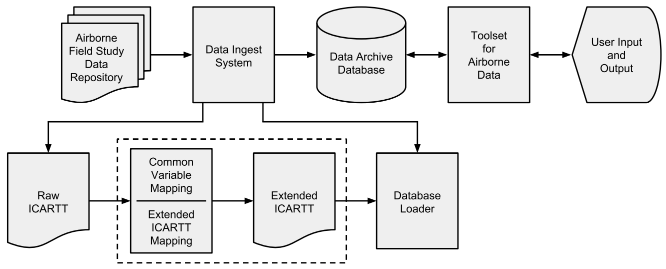

Fig. 1: Outline showing the process of converting a distributed set of raw ICARTT data files to the extended format standard. The role of our program is enclosed within the dotted box.

2. Related Literature

2.1 A Flexible Parser with Backtracking

In his 1963 paper [4], E.T. Irons describes his concept of an adaptive parsing schema, which could be used for pattern recognition as well as computer code correction and optimization. He points out weaknesses of the parsers in ALGOL and FORTRAN compilers at that time. Weak error handling being his main concern, he cites the frequent inability of compilers to correct and move past errors, specifically flawed object strings in the code. The schema he proposes would be capable of working through errors by using a more dynamic parsing structure.

He presents an approach which uses a slightly simplified equivalent of Backus-Naur Form (BNF) to describe the strings handled by the parser. Based on this pre-defined BNF-like grammar, the parser would work through and correct strings. When the parser encounters a string object where multiple parse sequences are applicable, it would attempt all possible sequences until one worked, and then move on to the next string. If, however, none of the attempted sequences work, the parser would backtrack to a previous point in the parsing sequence. This process of attempting different parse paths repeats until the parser works its way through the input.

Despite the age of Irons' paper, it is quite relevant to our chosen approach. He discusses an early conception of a recursive descent parser. Our program's parsing schema derives much of its functionality from recursive descent parsers, using varying combinations of parsers to work through the input.

2.2 USGS Metadata Pre-Parser

Peter N. Schweitzer of the United States Geological Survey addresses a somewhat similar parsing problem to our own with his Chew and Spit (cns) program [5]. The cns module precedes the primary parser module, mp [5], in cases where the input, a metadata file, is initially too poorly formatted to be parsed by mp.

When working through a metadata file, cns forms a parse tree, which it populates with parsed metadata elements based on user-defined element aliases as well as its own inference. These aliases are held in the alias file, which cns uses as a simple expert system. This alias file performs a somewhat similar role to that of the Extended Variable Map (see Section 3.2) in our program's parse and match system.

3. Description of Methodology

Our chosen approach had two distinct phases: 1. parsing and correcting raw file metadata contents based on the Extended ICARTT Data Format Standard, and 2. using fuzzy matching to properly identify and define variables from raw data files based on a corresponding variable map. We built the program in Python 3, which has multiple robust artificial intelligence libraries [6], [7] and parsing libraries [8], [9] available.

3.1 Using Parser Combinators

Due to the often irregular and poorly structured data found in many raw ICARTT files, a consistent, static grammar was impossible to define from the specification alone. Writing a parser capable of understanding the majority of these files required fast, iterative development. The maintenance costs of a separate lexical phase or library of dense regular expressions were unacceptable.

Ideally, we wanted a library that offered one of the benefits of monadic parsers [10], [11]: sufficient parser combinators that eliminate the need for a separate lexical phase. Pyparsing [8] is one such library. It approximates the compositional nature of monadic parsers in an object-oriented context. For example, `many1` from [11] corresponds with `OneOrMore()` in Pyparsing, `sepby1` with `delimitedList()`, and so forth for a number of other useful combinators.

The Pyparsing ParseResults object [8] offers a versatile data structure for the output of the parsing process, as it can be treated as a either a list or a dictionary containing parsed strings. These features gave our program the necessary level of flexibility to handle poorly formatted, inconsistent input from the raw ICARTT files. After a file has been parsed, it is rebuilt according to the specifications in the Extended ICARTT File Format Standard. For some parts of the file, this process is straightforward; though in other places, like the variable mapping described in Section 3.2, complicated inference techniques and the ability to reference a knowledge base is necessary.

3.2 Variable Mapping with Fuzzy Matching

The metadata subsection containing variables adheres to different specifications for content and formatting than the other metadata subsections. ICARTT variables proved to be particularly difficult to parse, as they were prone to subtle and irregular errors, and required our program to reference an external knowledge base for verification of content. As such,

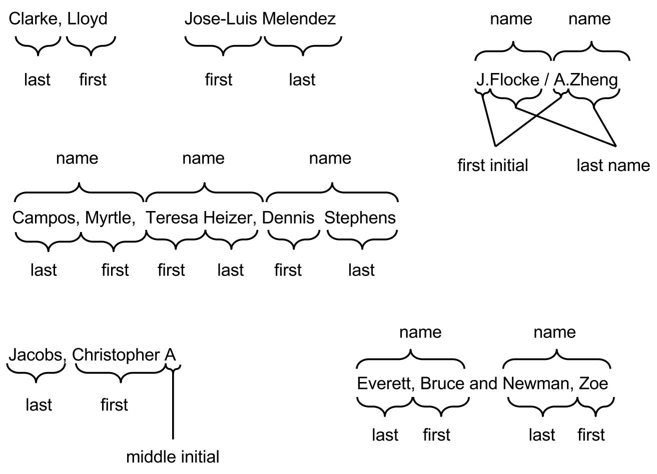

Fig. 2: Common input sequences for the "PI Names" field. These inputs can be automatically parsed and converted into a standardized format.

the implementation of a more intelligent, adaptable algorithm to properly handle the variable subsection was required.

ICARTT files contain two distinct variable types: time variables and dependent variables. Ideally, both have certain discernible traits denoting their type, but this is often not the case with raw files. In order to separate time variables from dependent variables, we use a configuration file against which fuzzy matching is performed to categorize the variable.

A corresponding Extended Variable Map was created as the aforementioned knowledge base for each of the ICARTT mission file sets. These maps contained all the information a human would normally need to correct the dependent variables in a file. All time variables are considered "independent variables" for these purposes and thus not included in the maps. Each row in an Extended Variable Map corresponds one-to-one to a variable in an ICARTT file, but both the raw files and the Variable Maps were created manually, resulting in irregularities stemming from human error. For example: the extended map and variable information lines often contained different names for the same variable, but the extended map would only list one of the two, or might not perfectly match either—perhaps using underscores where there could be hyphens, or zeros ("0") where there should be the letter "O". Thus, the information lines had to be matched to column headers before the extended map could be applied.

3.2.1 Fuzzy Match Pairing Algorithm

The Fuzzywuzzy Python library [6] gave our program the necessary matching capabilities to make it far more sensitive to subtle and odd edge cases resulting from human error. The library centers its operations around a fairly simple implementation of Levenshtein distance [12] to find the closest matches between pairs of strings.

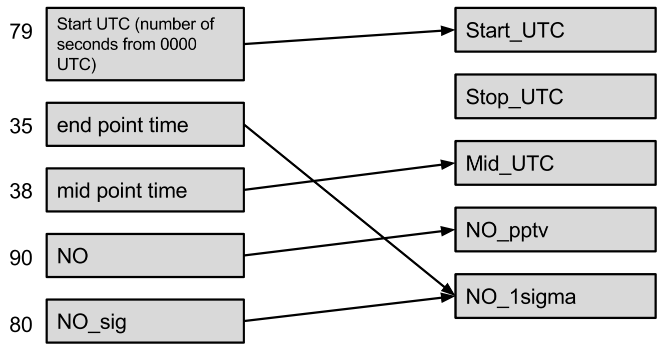

Fig. 3: Column header/variable info pairing, first pass. Confidence is shown to the left of variable info. Note contention for NO_1sigma.

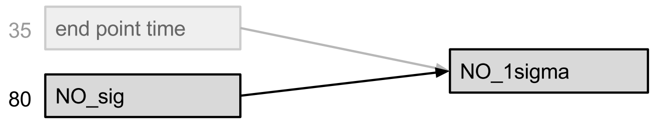

Fig. 4: NO_sig steals ownership of NO_1sigma because it has a greater confidence rating.

The following algorithm is used to find the optimal one-to-one fuzzy matching between two lists ("left" and "right") of strings:

1) If the right list is empty, pair each element in the left list with the null element.
2) For each element in the left list, find its best confidence match in the right list.
3) If any two or more elements in the left list matched to the same element in the right list, then choose the one with the greater confidence.
4) If an element in the right list only has one match in the left list, then pair those two elements.
5) If there are remaining elements that are unmatched in the left list, then recursively repeat this procedure on those lists.

Figures 3–5 illustrate the matching algorithm.

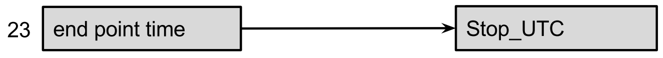

Fig. 5: Final step. The fuzzy match algorithm gets called again this time. Even though the confidence is low, this is the only match left—and it happens to be the right one. This is the recursive step. In this case the recursive step just instantly hits a base case. In more complex matches it might recurse a few times.

3.3 Results

Out of the 723 ICARTT files to be processed from the INTEX-A mission, our program was able to successfully parse and automate most of the formatting cleanup of 706 of them. Success in variable mapping varied depending on the quality of the variable names chosen. In cases where the name listed as the raw file's variable name, the variable info name, and its corresponding header name were all somewhat similar, the fuzzy matcher had little problem matching all of these together. In some cases, adding information to our knowledge base, e.g., a new line in the extended map or a "time variable name" listing in the configuration files, could be enough for the rest of the file cleanup to be fully automated.

These results are extremely promising, and a huge advantage over the previous manual cleanup of data files. Automated matching of variables, thanks to the fuzzy matching algorithms and a sufficiently advanced parser, meant even complex pairings of variables and column headers like "Carbon dioxide mixing ratio (ppmv)" and "CO2(ppmv)" could be performed automatically. Where this was not the case, and some manual editing was needed to guide the code toward the correct output, the speed benefits were still substantial compared to doing the same task by hand.

Overall, the program serves as an example of how artificial intelligence techniques and somewhat modern parsing techniques can automate a real, practical process previously performed only by manual human work.

4. Discussion and Future Work

Our program has successfully processed nearly 1,500 raw ICARTT data files with relatively little manual editing needed. Moving forward, we plan to continue enhancing the parsing and matching functionality for progressively increased automation and reliability in order to further minimize the need manual editing of raw files. Along with the continued ICARTT file conversion effort, we plan on exploring automation techniques for producing the knowledge base used here as well.

5. Acknowledgements

The authors would like to thank Lindsay Parker, Amanda Benson, and Aubrey Beach at NASA Langley Research Center, as well as Elliot Rieflin, Antonio Siochi, Clare Maliniak, and Anton Riedl at Christopher Newport University for their contributions, feedback, and support throughout the project.

References

[1] M. E. Lesk and E. Schmidt, *Lex: A Lexical Analyzer Generator*, 1975.

[2] S. C. Johnson, *Yacc: Yet Another Compiler Compiler*. Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[3] A. Aknan, G. Chen, J. Crawford, and E. Williams, *ICARTT File Format Standards V1.1*, 2013.

[4] E. T. Irons, "An error-correcting parse algorithm," *Communications of the ACM*, vol. 6, no. 11, pp. 669–673, 1963.

[5] P. E. Schweitzer, *A Pre-parser for Formal Metadata*, 2012. [Online]. Available: `http://geology.usgs.gov/tools/metadata/tools/doc/cns.html`.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: machine learning in python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] A. Cohen, *Fuzzy String Matching in Python*, 2014. [Online]. Available: `https://github.com/seatgeek/fuzzywuzzy/`.

[8] P. McGuire, *Getting Started with Pyparsing*. O'Reilly Media, Inc., 2007.

[9] A. Vlasovskikh, *Funcparserlib: Recursive Descent Parsing Library for Python Based on Functional Combinators*, 2013. [Online]. Available: `http://code.google.com/p/funcparserlib/`.

[10] G. Hutton and E. Meijer, "Functional pearl: monadic parsing in haskel," *Journal of Functional Programming*, vol. 8, no. 4, pp. 437–444, 1998.

[11] D. Leijen and E. Meijer, "Parsec: direct style monadic parser combinators for the real world," Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, Tech. Rep., 2001.

[12] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," in *Soviet Physics Doklady*, vol. 10, 1966, p. 707.