Natural Language Math Problem Solving

CSCI 4152 - Natural Language Processing

Stanford Lockhart
Geoff Caven
Niclas Skaalum
Chelsey Childs

Abstract

The goal of our application is to be capable of interpreting and solving naturally expressed mathematical statements. The scope of our application was to handle elementary level mathematical expressions and solve ambiguities by using BEDMAS order of operations. The application was developed in python and uses an open-source toolkit called Natural Language Toolkit (nltk) to stem and parse the user's input. The resulting application is able to handle numerical values between -9999 and 9999 and passes the majority of mathematical expressions. The application solution can be further developed in the future projects to translate and solve verbally spoken mathematical expressions.

Introduction

As our project for CSCI 4145, we implemented a software solution capable of interpreting and solving naturally expressed mathematical statements. The solution accepts naturally expressed math, converts it to numerically expressed math, then solves the expression, outputting a numerical result.

The solution handles only written input (in the form it might be spoken as, for example), and mathematical expressions are typically written using purely numerical language. These expressions would not require the solution, however it nevertheless provides baseline functionality for expanded projects, such as solving voice-input queries or interpreting other forms of natural language.

This report aims to elaborate on the value of our software solution by discussing related works and demonstrating the underlying methodology and experimental results achieved by the solution.

Related Work

The paper *Bringing Machine Learning and Compositional Semantics Together* (Liang, P., & Potts, C. , 2015) has significant conceptual similarity with our solution, though it extends beyond the scope of our project. The paper describes a method by which natural language math problems (utterances) represented as sequences of strings can be transformed into a consistent logical form by the application of a Context-Free Grammar (CFG). The desired logical form is a representation of cardinal numbers and math operations. This approach has the potential to produce ambiguous results, with utterances that can be parsed into multiple logical forms. This paper does not propose a method by which to collapse these ambiguities into a single, correct form.

The CFG-based approach described in this paper was a direct inspiration for this project, with the grammar described in Table 1 serving as a beginning point for the development of the more complex PCFG used in this project.

Problem Definition

The project objective was to consistently interpret naturally expressed mathematical expressions and produce numerical results. For example, the natural expression “two plus two” would interpret to the purely mathematical expression “2 + 2” and generate the numerical result “4.”

Examples of existing products capable of converting and solving naturally expressed mathematical expressions include Wolfram Alpha and Google Search. Entering the aforementioned example into Google, for instance, produces an interactive calculator with the corresponding expression “2 + 2” already entered and solved. Similarly, Wolfram Alpha outputs “4” for the same input expression.

Bearing in mind the limited time, resources and developer experience behind the project, we reduced our scope to handle only a specific subset of naturally expressed operations, namely: addition, subtraction, division, and multiplication. Additionally, the problem definition requires correct handling of ambiguous cases and adherence to the order of operations and correct treatment of negative values. For example, the expression “negative two minus two times three” would yield “-2 - 2 * 3” and, in turn, “-8.” Though the scope does not consider non-integer inputs, it is capable of emitting non-integer values as output.

Potential extra functionality intended for this project included variable handling and function plotting, as offered by tools such as Wolfram Alpha. Due to time constraints, these features were not developed. Furthermore, while this type of functionality would have increased the complexity and potential utility of the project, it would likely not introduce much greater handling of natural language.

Methodology

The scope of the project is to be able to parse and solve elementary level utterances. The application will be able to solve questions with a combination addition, subtraction, multiplication, and division. The numerical values acceptable in the utterances will be $9999 > n > -9999$. The values were restricted due to the complexity of the grammatical rules increasing with the size of the number.

The application is written in Python version 2.7 and uses Natural Language Toolkit (NLTK). NLTK is an open source toolkit that provides multiple libraries that for stemming, parsing, and classification.

Parsing

A handbook of mathematical discourse by Charles Wells (2003) was used to determine a set of words equivalent to each of the operations supported by our solution. In the language of mathematics, there are many words that have the same meaning; for example the expressions “two subtract one” and “two minus one” have the same meaning. Once these sets were created, we are able to easily lemmatize these words into the forms directly supported by our solution, as seen in Figure 1. In the case in which a user has entered “two minus one”, this will be changed to “two subtract one”.

```
OPERATIONS = {
    "add": [
        "add",
        "plus",
        "sum"
    ],
    "subtract": [
        "subtract",
        "minus",
        "difference of",
        "negative"
    ],
    "divide": [
        "divide by",
        "divide",
        "over"
    ],
    "multiply": [
        "multiply by",
        "time",
        "multiply",
        "product of",
        "by"
    ]
}
```

Figure 1. Translating rules to used to create consistent utterances by eliminating synonyms.

Using NLTK, we constructed a probabilistic context free grammar (PCFG) to parse utterances into a consistent mathematical form (See Appendix A). The original intention behind the use of a PCFG was to utilize probabilities to make the application compute the operations in a specific order. Instead, the division and multiplication operations were put deeper into the tree by adding another level to the grammar, which in turn defines what order that operations should be evaluated. This resulted in artifacts being left in the code for support for probabilities in the productions. The probabilities are ultimately set so each production has an equal chance to be selected, which negates the PCFG and makes it a regular CFG in practice. In the created grammar, each production for a non-terminal has equal weights. The productions for S and T were created to enforce BEDMAS rules by embedding multiplication and division deeper in the resulting parse tree, as mentioned previously. The grammar is then passed to a parser provided by NLTK. Specifically, the parser used for this application was NLTK's `pchart.InsideChartParser()`.

Using this parser and grammar, the application is able to create a parse tree from the user's inputted utterance. For example if a user entered "eight plus two times four", this will first be translated using the rules in Figure 1 to the utterance "eight add two multiply four". This translated utterance is then applied to the parser to generate the tree in Figure 2.

```
(S
  (S (THOU (HUN (TEEN (CD eight)))))
  (ADD add)
  (S
    (T
      (T (THOU (HUN (TEEN (CD two)))))
      (MUL multiply)
      (T (THOU (HUN (TEEN (CD four)))))
    )
  )
)
```

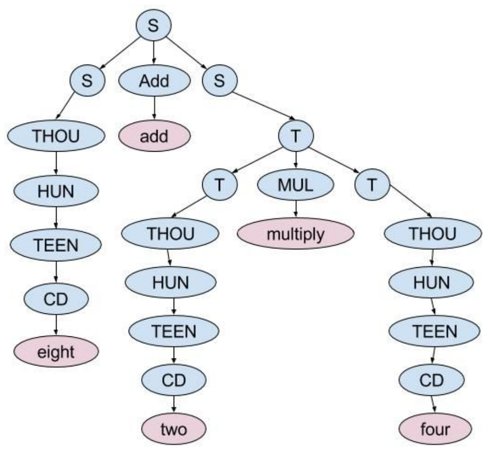

Figure 2. Parse tree for the utterance "eight add two multiply four", shown on the left in bracketed notation and on the right in a graphical format.

Semantics

When a parse tree is created, it represents a recursive structure consisting of values and operations applied to those values, either to a single value in the case of a negative number, or to pairs of numbers. Knowing this, we can determine the semantic meaning of a parse tree, and by extension the utterance that generated it, by calculating the nodes of the tree recursively, and returning the resulting value that appears in the root node.

The recursive algorithm works by starting at the root of the parse tree, and for every node, determining what it is working with - an operation or a value. If it is a value, it is immediately returned. Otherwise, the values of the two operands are calculated recursively, and those values then have the target node's operation applied to them, and the resulting value is returned as the value of the operation.

The largest source of ambiguity when determining the semantic meaning of a mathematical utterance is the order of operations. When traversing the tree to determine the value, the order must be respected. When designing the grammar, this was taken into account - the operations for multiplication and division appear deeper in the tree than those for addition and subtraction. Without this distinction being present in the grammar, the resulting trees would have ambiguities in their semantic meaning.

The other main ambiguity is created by the presence negative numbers. A negative sign in mathematics can be attached to either a number or an operation. With this ambiguity, it can be difficult when traversing the tree to determine which value the sign should apply to. Because our grammar does not allow for parentheses in this iteration, it was decided that negative signs should only apply to the value, not to the operation.

Experiment Design

In order to evaluate the fitness of our grammar and parsing algorithm, a list of utterance test cases was created for the program to take as input, with corresponding expected outputs. These input cases were then fed into the program, and the resulting output was automatically evaluated as to whether or not it was correct, incorrect, or produced ambiguous results. As a baseline, the Wolfram Alpha service was used to take in the utterances and return a value, that could then be compared to our own algorithm.

Because the test cases and the experiment could be repeated as development continued, ambiguous results in which the algorithm returned multiple different values were eventually eliminated in all of our test cases as improvements were made. At the beginning of testing, the most common result for utterances that contain ambiguities due to BEDMAS was to have a possible value returned for when multiplication and division are done first, and another returned for when addition and subtraction were done first. As the grammar was tweaked to ensure that BEDMAS was followed correctly, these ambiguities were resolved.

The test cases contained a number of difficult problems with regards to order of operations, and for the most part the algorithm was able to parse and return a value that matched what would be returned from Wolfram Alpha. Of the test cases that were given, only a single returned an incorrect value: “three minus negative twenty seven times three times three thousand one hundred and twenty seven minus one”. As the utterance increases in length, the number of possible errors increases - given that this is the only error found in our test suite, the algorithm would appear to have a high fitness for the problem.

Conclusion

Mathematical utterances are the middle ground between a user speaking, and the mathematical result being returned. Of the two problems that make up that workflow, parsing speech and solving the utterance, the second is what can be solved using grammar and trees. This implementation set out to try and find a general means of solving such utterances, using Python and the NLTK package.

The largest hurdle when trying to parse mathematical utterances was the ambiguities between what was meant and what was uttered. After lemmatizing and normalizing an utterance, it can be trivial to create a parse tree that could represent the semantic meaning when traversed, but ensuring that the parse tree is the correct one is the largest problem faced in the course of developing this implementation.

By developing a grammar that takes these ambiguities into account, an algorithm can be developed that will parse the majority of simple utterances, as seen in our experiment. The implementation did have a number of drawbacks - namely, there was a limit to the size of the numbers allowed, and it did not allow for parentheses, decimals or additional operations other than addition, subtraction, multiplication, and division. However, when given the wide range of utterances in our tests, it was seen to have a high fitness, with some remaining issues due to the high ambiguity of lengthy mathematical utterances.

References

http://www.abstractmath.org/Handbook/handbook.pdf
Wells, C. (2003, March). A handbook of mathematical discourse. PA: Infinity.

http://annualreviews.org/doi/pdf/10.1146/annurev-linguist-030514-125312
Liang, P., & Potts, C. (2015). Bringing machine learning and compositional semantics together. *Annu. Rev. Linguist.*, *1*(1), 355-376.

http://www.foo.be/cours/dess-20122013/b/Natural%20Language%20Processing%20with%20Python%20-%20O'Reilly2009.pdf
Bird, S., Klein, E., & Loper, E. (2009). *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.

Appendix A

```
GRAMMAR = nltk.PCFG.fromstring("""
    S -> S ADD S                        [0.2]
    S -> ADD S 'and' S                  [0.2]
    S -> 'subtract' THOU                [0.2]
    S -> THOU                           [0.2]
    S -> T                              [0.2]

    T -> T MUL T                        [0.25]
    T -> MUL T 'and' T                  [0.25]
    T -> 'subtract' THOU                [0.25]
    T -> THOU                           [0.25]

    ADD -> 'add'                        [0.5]
    ADD -> 'subtract'                   [0.5]
    MUL -> 'multiply'                   [0.5]
    MUL -> 'divide'                     [0.5]

    THOU -> HUN                         [0.25]
    THOU -> CD 'thousand' HUN           [0.25]
    THOU -> CD 'thousand' 'and' HUN     [0.25]
    THOU -> CD 'thousand'               [0.25]

    HUN -> TEEN                         [0.25]
    HUN -> CD 'hundred' TEEN            [0.25]
    HUN -> CD 'hundred' 'and' TEEN      [0.25]
    HUN -> CD 'hundred'                 [0.25]

    TEEN -> CD                          [0.076923077]
    TEEN -> TEN CD                      [0.076923077]
    TEEN -> TEN                         [0.076923077]

    TEEN -> 'ten'                       [0.076923077]
    TEEN -> 'eleven'                    [0.076923077]
    TEEN -> 'twelve'                    [0.076923077]
    TEEN -> 'thirteen'                  [0.076923077]
    TEEN -> 'fourteen'                  [0.076923077]
    TEEN -> 'fifteen'                   [0.076923077]
    TEEN -> 'sixteen'                   [0.076923077]
    TEEN -> 'seventeen'                 [0.076923077]
    TEEN -> 'eighteen'                  [0.076923077]
    TEEN -> 'nineteen'                  [0.076923077]

    TEN -> 'twenty'                     [0.125]
    TEN -> 'thirty'                     [0.125]
    TEN -> 'forty'                      [0.125]
    TEN -> 'fifty'                      [0.125]
    TEN -> 'sixty'                      [0.125]
    TEN -> 'seventy'                    [0.125]
    TEN -> 'eighty'                     [0.125]
    TEN -> 'ninety'                         [0.125]

    CD -> 'zero'                            [0.1]
    CD -> 'one'                             [0.1]
    CD -> 'two'                             [0.1]
    CD -> 'three'                           [0.1]
    CD -> 'four'                            [0.1]
    CD -> 'five'                            [0.1]
    CD -> 'six'                             [0.1]
    CD -> 'seven'                           [0.1]
    CD -> 'eight'                           [0.1]
    CD -> 'nine'                            [0.1]
""")
```