Contents

Intro

Motivation

Word and sentence embeddings are an incredibly powerful tool for natural language processing. Google Translate's error rate decreased by over 50% when it switched to a model that used embeddings for translations [("A Neural Network for Machine Translation, at Production Scale" n.d.).

The options for telling how good embeddings are before running them through a downstream task like Google Translate are less amazing, however: Researchers only have a few small datasets, which are largely based on linguistic survey data, for this task (Schnabel et al. 2015).

One of the reason so few datasets for evaluating embeddings exist are difficulties in determining when two words or sentences are different or similar. Consider how difficult it is for a machine learning model to determine that the phrase "the red-faced man" can be equivalent to "the embarrassed man" in the right context.

Mathematics is a type of language without this type of ambiguity about whether two clauses mean the same thing. The equals sign tells us with certainty when

two expressions are equivalent to each other. Therefore mathematics is a perfect test bed for new embeddings models, since every mathematics text contains tens or hundreds of equivalencies with which to evaluate embeddings.

As far as I am aware, no one has yet taken advantage of the equals sign for creating and evaluating embeddings. Throughout the course of this semester, I have created the first ever dataset of equivalent equations.

Applications

This project will enable researchers and data scientists who work with mathematics texts to evaluate the intrinsic success of their embeddings. It may also enable better understanding of embedding in general. If I am able to create a large enough dataset of equivalent equations, the equivalent equations could also be used to train embeddings in a supervised fashion.

Background

This fall, I started working with a research group that Prof. John Lafferty helped found called https://github.com/hopper-project. The Hopper Project has created a dataset of academic articles from arxiv and coded up a pipeline that extracts LaTeX from these academic articles. I will extract equivalent equations from this dataset.

Outcome: What I've Actually Created

A dataset for evaluating intrinsic similarities

I have created a dataset of equivalent equations from these texts and provide benchmarks for various embeddings models using the equivalencies, which will be made available to the public via github. In early tests, I have found millions of candidates for generating equivalent equations in the arxiv dataset.

A set of benchmarks using this dataset

I have created a set of benchmarks for working with this dataset.

A suite of tools for working with equation embeddings

Equations texts are harder to work with than normal english texts because they are written in LaTeX. I started working on a set of tools for visualizing mathematics in embeddings models this fall (Helms 2017).

{WRITE MORE HERE}

These tools include a tool for visualizing any embeddings for things that aren't easily represented as text in T-SNE plots {INS_CITE} and a method for using KaTeX in jupyter notebooks for a rich representation of math.

The Code

In this section, I will go over the code I have written to tokenize TeX equations at a high level and explain why I wrote it the way I did. Note that this is only a high level discussion: look to the readme and comments in the code base for specific instructions on how to use the pipeline. Before I jump into the specifics of the code, I will give a brief background on TeX to make it clearer why I had to go to the lengths I did to get a aligned few math equations.

Identifying and using equal equations in machine learning models requires two things: One, being able to properly identify tokens (so that the LaTeX code `\\int` is treated as an atomic unit and not a sequence of the characters "\", "i", "n", and "t"); Two, being able to say whether a subexpression is substantive enough to use in the model.

You could determine substantiveness (the second requirement) by doing things like setting a string length curoff for the subexpression and taking tf-idf scores between two expressions on either side of an equals, but this provides no guarantee of substantiveness–the expression `x + y` is arguably just as substantive as `\\frac { x_7 } { y_i }`, with one operator and two variables, but it would be hard to determine so based on tf-idf score or the length of the expression.

As a result of this difficulty in determining substantiveness, I decided to implement a heuristic function that would make a shallow parse of the TeX expression and determine if it contained at least one operator and two variables and/or numbers at a high level. I arbitrarily decided on this cutoff for the number of variables and expressions, and wrote the code so this could be adjusted. By "high level" I mean not within a subexpression or exponent, or a function argument–so f(x + y) would not count as a substantive expression.

A brief history of TeX

You might be wondering why tokenization and parsing similar equations was hard enough a task to spend a whole semester doing it. You would be right to wonder: it should not be very hard to do, if not for the fact that the core layout algorithms for TeX are all written in a virtually extinct language called WEB written by the creator of TeX, Donald Knuth, himself. {INS_CIT} (https://en.wikipedia.org/wiki/WEB).

Because WEB has been all but forgotten, the core TeX algorithms implemented in it by Knuth are automatically cross-compiled into C code before being used by TeX tools like texlive. This makes it extremely difficult to hack into TeX's parsing algorithms and access their internal data structures. Without access to the TeX algorithms or internal data structures (or even source code that is easy to navigate and/or understand), it is hard to even tokenize LaTeX codes, not to mention parse them.

As a result, I have written my own tokenization function. I have not written a complete TeX parser, but I have implemented a shallow parse that can skip along the highest level of TeX code and check if it contains certain types of operations.

A group at Harvard wrote a wrapper around KaTeX for tokenizing LaTeX math expressions. I decided to write my own so that I could be absolutely certain about the assumptions underlying the data and avoid accidentally introducing bias into the data. KaTeX's priority is parsing as many expressions to something that can render on a website as possible, and doing so quickly; it still only parses about {INS_BENCH} equations correctly.

Tokenization

I implemented a finite state machine to tokenize TeX codes. It takes a stream of characters from an equation from an arxiv text as input and iterates over it, changing state with each new character. Depending on the state and the character, the FSM will build a token or create a new one.

Example:

Location in codebase:

On the first pass, the tokenizer follows some extremely simple rules, like building every character a-z that comes after a \ into a single token. This creates some incorrect tokens, like `\\intx`. So the tokenizer makes another pass along the now partially-tokenized expressions and attempts to break up any string starting wiht '\' into a known macro. I acquired and merged two collections of known macros: 1 from {INS_CIT}, and the other from KaTeX.

The … can't find about

Example:

Location in codebase:

Further preparation

The following section goes over a series of functions I have written to further prepare the tokenized equations for a machine learning model. Keep in mind

some, none, or all of these functions need not be used: You could modify them, write your own, or just use the output of the tokenization process for a model.

Splitting

Splitting equations comes in two steps: first, splitting equations that are in aligned environments, like the one below, into multiple expressions. These splits occur whenever a "\", ",", or ";" character that is not within a subexpression is encountered. A character is considered to be within a subexpression whenver it is within a two characters like "(" and ")" or "{" and "}". The full list of these characters comes from KaTeX's list of right and left bracket-type characters.

```
\begin{aligned}  f(x) &= x + y^2 \\ g(x) &= ax + b \end{aligned}
```

=>

```
f(x) = x + y^2 \text{ and } g(x) = ax + b
```

If something like the following gets encountered, the bottom line gets "folded in" to the top one.

```
\begin{aligned}  f(x) &= x + y^2 \\      &= ax + b \end{aligned}
```

folding in =>

```
f(x) = x + y^2 = ax + b
```

The same applied to inline equations like $ax + b = 700x + z, ax + c = \theta + z$, which would be split into $ax + b = 700x + z$ and $ax + c = \theta + z$

After such expressions have been handled, the equations are split on equalities. I use a list of {INS_NUM} equalities from KaTeX to find {CONTINUE} equalities to split on. I only split when the equality is not within a left and right bracket-type character (like "{" and "}").

Example:

"f(x) + y = 100x^2"

Completeness heuristics.

TF-IDF versus a shallow parse

Stoplisting

There are some tokens that add nothing to the mathematical meaning of the expression, such as `\begin{align}`. In addition, comments inside of `\text` tags, though perhaps pertaining to the equation, end up distracting from the

meanining as well, since, to the model, anything character inside the `\text` looks like a variable.

One avenue to explore could be tokenizing text inside of `\text` tags as words, but I have decided to just exclude `\text` and all children for the purposes of this project. Symbols and variables are closer to the heart of meanining in mathematical expressions, and seem to me to be safer for the time being.

Example:

Location in codebase:

Normalization

LaTeX allows you to write expressions like `\int_{x+y}^5` and `\int^5_{x+y}` and get equivalent representations. `normalize` makes it so that the `^` and any of its dependents always come before `_` and its dependents.

The Results

- table with: number of equations parsed, total number of equations (success rate). Maybe compare a few methods.
- matt's embeddings (graph)
- tf-idf (graph)

Bibliography

"A Neural Network for Machine Translation, at Production Scale." n.d. *Research Blog*. Accessed February 2, 2018. https://research.googleblog.com/2016/09/a-neural-network-for-machine.html.

Helms, Samuel. 2017. "Mathviz: A Python Package for Examining Mathematics Equation Embeddings." https://github.com/samghelms/mathviz.

Schnabel, Tobias, Igor Labutov, David Mimno, and Thorsten Joachims. 2015. "Evaluation Methods for Unsupervised Word Embeddings." In, 298–307. https://doi.org/10.18653/v1/D15-1036.