Using NLP to Simplify Claims in Legal Patents

Top Level Overview:

This document outlines the basics of using a fundamental approach to NLP for patent claim parsing. This document begins by defining what NLP is and the key concepts that are necessary to understanding NLP. It also covers the specific vocabulary within NLP, and more importantly, a specific possible set of steps to creating text that a computer is able to readily process and summarize. The ultimate goal here is to determine the lengths to which we can stretch true NLP techniques to simplify the content from a patent claim to something more readable. While true NLP would allow for simple information extraction and simple summaries, actual rewordings and complex summaries will likely require the use of an LLM. *Note: Throughout the document I make references to "true NLP", by that I mean NLP without the use of LLMs.* (Edited 7/9/25)

Definitions and Key Concepts:

1. **Natural Language Processing Definitions**
    a. Natural Language Understanding: Interpreting the meaning of text
    b. Natural Language Generation: Generating human-like text based on data
2. **Legal Claim Semantics:**
    a. Claims must contain the following, in order: A preamble, a phrase like "wherein the improvement comprises", and the elements that the applicant considers to be the new or improved portion.
3. **Fundamental concepts:**
    a. **Parts of Speech (POS):** We find these to be important because the assignment of the part of speech gives the machine a format that it can "fit"
        i. Nouns: Person, place, or thing | Appears after determiners and adjectives, can also be the subject of a verb
        ii. Pronouns: A word that is used in place of nouns (he, she, etc) | I suspect simple NLP struggles to connect a pronoun to the adjective or verb it is connected to
        iii. Verbs: Expresses action or being | Keep in mind that the plurality of the verb has to match the plurality of the noun it is referring to
        iv. Adjectives: Describes or modifies a noun/pronoun
        v. Adverbs: A modifier of a verb, adjective, or another adverb | NEVER MODIFIES A NOUN

vi. Prepositions: A word placed before a noun/pronoun that forms a link between the noun/pronoun and other words in the sentence

vii. Conjunctions: Joins words, phrases, or clauses | Examples include: and, but, or, while, because

viii. Interjections: A word used to express emotion | Often times identifiable via “!” suffix

ix. Determiner: Not necessarily a standard English POS, but important in the sense that the determiner refers to article

x. Numeral: Any numbers that are either written out in English or just as numbers.

xi. Punctuation: Periods, commas, exclamation points, etc.

xii. Notable Ambiguities:

   1. Lexical Ambiguity: A single word can be multiple things, i.e. Gold can either be the color gold or the metal gold.

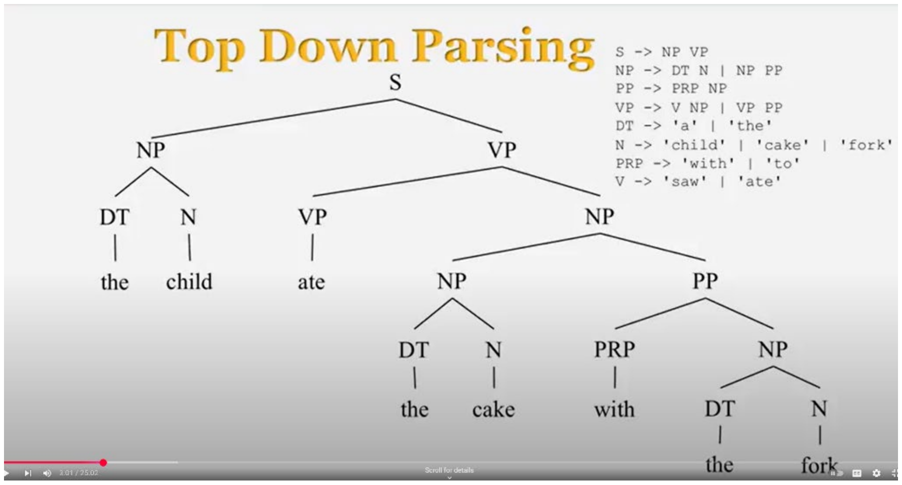
   2. Semantic Ambiguity: The meaning of words can be misinterpreted. “I saw her duck” could mean you saw someone’s pet or that you saw someone physically duck their head down.

b. Sentence Parsing in NLP

i. English often follows the pattern of subject-verb-object

ii. Generally, sentence parsing in true NLP is constrained by: The input sentence and the grammar used in the sentence

iii. Top-Down (goal-driven): Explores options that don’t match the full sentence



Top Down Parsing

```
S -> NP VP
NP -> DT N | NP PP
PP -> PRP NP
VP -> V NP | VP PP
DT -> 'a' | 'the'
N -> 'child' | 'cake' | 'fork'
PRP -> 'with' | 'to'
V -> 'saw' | 'ate'
```

1. Given a set of rules (S-> NP VP)

2. Try to build the sentence to fit one of the rules

iv. Bottom-Up (data driven): Explores options that won't result in a full parse | This style of parsing might be better for our use case because the NLP cannot return data that isn't directly a match to the source claim

1. You start by defining your sentence structure. Basically, this means you want to layout a template of what the sentence structure could possibly look like from the leaves up. So, for example, you could say: Noun Phrase (NP) = Determiner (DET) + Noun (NN), Verb Phrase (VP) = Verb (VB), Sentence (S) = NP + VP. This process gives the machine an idea of the sentence structure.

2. Begin with the tokenized paragraph

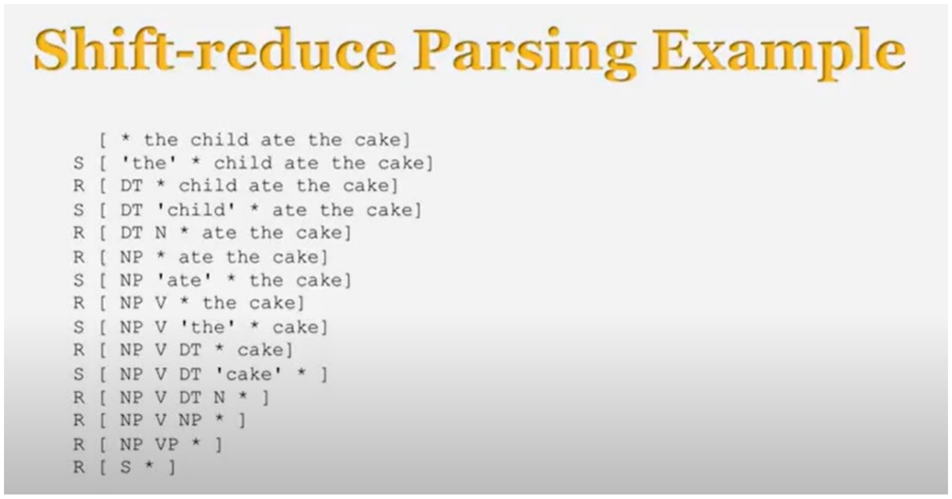
3. Each token should be a single word, each leaf of the tree is one token/word

4. Shift: As in, load the next token onto the stack

5. Reduce: As in, combine the applicable patterns

v. CKY parser: Bottom-up; Grammar has to be in a binarized form. http://lxmls.it.pt/2015/cky.html

vi. Shift-reduce parsing: Bottom-up that constructs a parse tree from individual words as tokens. To go through the words in the sentence, shift. Stopping conditions: The stack is empty, input is empty, sentence comes back.



Shift-reduce Parsing Example

```
  [ * the child ate the cake]
S [ 'the' * child ate the cake]
R [ DT * child ate the cake]
S [ DT 'child' * ate the cake]
R [ DT N * ate the cake]
R [ NP * ate the cake]
S [ NP 'ate' * the cake]
R [ NP V * the cake]
S [ NP V 'the' * cake]
R [ NP V DT * cake]
S [ NP V DT 'cake' * ]
R [ NP V DT N * ]
R [ NP V NP * ]
R [ NP VP * ]
R [ S * ]
```

c. Big picture understanding: True NLP basically assumes that language will have a format or pattern. English often follows as: subject, verb, object. NLP uses these patterns to assign the POS to the words in their respective positions. Then, once tagged, the individual POS are connected together where applicable to form noun-phrases or verb-phrases. These then are overall combined to create that tree. The tree is the machine way of understanding the text format and associations.

d. **Sentence Embedding: The process of converting text into numerical representations of itself using vectors that capture meaning** *NOTE: Code implementation I am working on does not use advanced sentence embedding yet, I just have a bag of words with value multipliers for nouns and verbs (because they are deemed to have more contextual value than conjunctions, identifiers, etc) NOTE2: I am finding it hard to find resources on actually building models, most of everything is about training preexisting models. Not sure if we need that level of customization though*

i. N-gram Vectors

ii. TF-IDF

iii. Word2Vec

iv. Bag of Words (less valid for our use case because each word has the same semantic importance in this method)

v. BERT

1. LegalBERT (https://huggingface.co/nlpaueb/legal-bert-base-uncased)

vi. We can use pretrained models/transformers for the time being like SpaCy en_core_web_lg or BERT/legalBERT

vii. FAISS (Facebook AI Similarity Search): Open source developed by Meta that can utilize GPU acceleration and large data sets that lie outside of memory

viii. ANNOY (Approximate Nearest Neighbor Oh Yeah): Open source developed by Spotify where one can quickly find "close enough" matches versus identical matches.

e. Text Summarization Techniques in NLP

i. Word frequency based:

1. Clean the doc
2. Tokenize into sentences
3. Count word occurrences
4. Word frequencies can be decided by dividing the word count by the highest word count
5. Sum sentence word frequencies
6. Return sentences by highest word frequencies

ii. TextRank:

1. Tokenize into sentences
2. Generate sentence embedding
3. Build a graph with sentences as nodes and edge weights are similarities of the above sentence embedding
4. Run PageRank on the graph
5. PageRank will give each sentence a relevance score, higher is better
6. Only keep certain sentences with a score above a threshold

f. Other notes:

i. Corpus is the actual set of machine-readable text that has been naturally produced. AKA the training set

ii. A treebank is a corpus annotated with syntactic structure

4. **NLP Steps:**

   a. **Claim Chunking Strategy:** Joe theorizes that the initial input text will likely be too large for the machine to handle, so the question arises: How best can we split the text into usable chunks?

      i. I think we can chunk the claims out via two main identifiers; 1. We can split on punctuation like periods, semicolons, dashes, etc. 2. WE can analyze legal texts and separate on common clause separators (lets say for now: comprising)

   b. Text Normalization: Normalize the text to the simplest format

      1. Convert all characters in the string to lowercase (Python: string.lower())

      2. Convert all numbers (1, 2, 3, …) to words (one, two, three, …) or remove them altogether

      3. Remove punctuation (Python: regex)

      4. Remove blank space (Python: strip())

      5. Remove stop words (a, on, is, all, …) (Python: NLTK)

   c. Tokenization: Make the text into smaller chunks

      i. **Word Tokenization: Text is divided into individual words (Python: text.split())**

      ii. Character Tokenization: Text is divided into individual characters

      iii. Sub-word Tokenization: Text is divided into chunks that are larger than a single character but smaller than a whole word

      iv. **Sentence Tokenization: Text is divided into sentences (Python: NLTK tokenizer)**

      v. N-gram Tokenization: Text is divided into n-sized chunks of data

   d. Lemmatization: Distil words to their fundamental forms based on context

      i. **Rule Based: Use linguistic rules to derive the base of a word (For our use, is a custom rule set best? My rationale is that we know the style of English that is used for these patent claims, so can we basically distill a few rules that should apply widely?**

      ii. Dictionary Based: Use predefined dictionaries or lookup table to find its base

        iii. LLM Based: Use an LLM to distill the base words
        iv. We can use NLTK: lemmatizer = WordNetLemmatizer()
    e. Stemming: Distil words into root form, might result in a non-word simplification
        i. Porter's Stemmer: Based on the idea that you can apply a specific pattern of rules to remove suffixes
            1. Snowball Stemmer: Better version of the above
        ii. Lancaster Stemmer: Repeatedly apply a set of rules until you can't anymore (I think we can add custom rules in NLTK using this)
        iii. Rule based: Define your own set of rules to stem by
    f. Stop word removal:
        i. Remove words that should be ignored (a, the, an, in, …)
        ii. Stop words are generally considered semantically uninportant
        iii. We can use NLTK again to remove stop words
    g. Parts of Speech (POS) tagging: Assign each word what it is (verb, adjective, etc.)
        i. We can use NLTK to tag
        ii. But the above discussed tagging algos might be better than what NLYK has to offer by default, I have to check it out

5. **Coreference resolution:** In text, we constantly work with references to the same entity but without directly addressing that identity. For example, I could say "Amruth plays the tabla because he likes it," we know that he refers to Amruth and it refers to the tabla, but how can we make the machine understand that?
    a. At a very basic level, our application of coreference resolution can basically replace all the pronouns with the nouns that they are referring to
    b. Anaphora: A reference to something earlier in the sentence Vs. Cataphora: A reference to something later in the sentence
    c. Rule-Based: A predefined set of rules or heuristics that define the pattern of speech Vs. Machine Learning
    d. Challenges: Ambiguity of speech, varied expressions, contextual nuances

e. Mention-pair: A classifier that is given a pair of mentions and makes a binary decision on if they are related

f. Papers:

    i. https://www.sciencedirect.com/science/article/pii/S1566253519303677?casa_token=aOKqlUgiZtkAAAAA:voF-90tw96TrHFWRbD1YghDK5NEvIVqF7cQl9X8z1ufspM1Kt-uofRs9E-TMazTs9uax2-oeTQ

    ii.

Code Libraries:

6. NLTK Functions

a. Tokenization: word_tokenize will tokenize the sentence into a list of individual words

b. POS tagging: pos_tag(word_tokenize(input))

c. Porter Stemming: stemmer = PorterStemmer() | stemmed = [stemmer.stem(word) for word in words]

d. Lemmatization: lemmatizer = WordNetLemmatizer()

7. Other libraries

a. SpaCy: Less flexible than NLTK, meant to be fast, not educational

    i. SpaCy comes with some trained pipeline models

    ii. Tokenization

    iii. POS tagging

    iv. Dependency parsing

    v. Lemmatization

    vi. Similarity (!!!)

    vii. Though SpaCy has its own set of custom pipelines that run by default, we can modify the pipeline by adding components if necessary. The link has an example of custom identification of animals using a component (www.course.spacy.io/en/chapter3)

    viii. We can also train SpaCy models

        1. Initialize the model weights randomly

2. Predict a few examples with current weights
3. Compare the prediction with true labels
4. Calculate how to change weights
5. Update
6. Go back to b

b. SpaCy/transformer Implementation

i. Summarizer()

1. Max_len: max length by char
2. Min_len: min length by char
3. Do_sample: Enables or disables random sampling
4. Top_k: Restricts data sampling to the k most common tokens
5. Top_p: Restricts data sampling to p percent of the tokens
6. Temperature: Determines randomness

c. AllenNLP: NLP research library I want to try to do coreference resolution with

8. **GitHub link:** https://github.com/rhit-annavaa/claims-parsing-test

a. One thing I am noticing with a lot of the samples I am using/making are that they miss the entire thing about the vertical mechanism (taken from the very first example in the randomclaims json). I am not sure why yet… (7/10/25)

b. Spacysimilarities.py: This is a rudimentary script that uses SpaCy's simple "en_core_web_lg" pretrained model to determine the similarities between two given sentences. In the code we just load the SpaCy model, run the SpaCy model on the text, and then use the similarity feature in SpaCy to determine the similarity. A score is given between 0 and 1, the higher the more similar. The algo works alright at best, some things it correctly identifies as pretty similar, but other things it incorrectly identifies as similar.

c. Spacydisplaytest.py: This is another rudimentary test script that utilizes SpaCy's built-in models and interpreters to generate a visualization of word dependencies and parts of speech. In the code, we again load the SpaCy model and run it on our text. The using SpaCy functions we can easily get a dependency visualization of the input text at localhost:5000.

d. Scratchnltktest.py: This is the big experimental program now. Instead of using SpaCy and it's easy-to-access built-in functions, I try to do each step "manually" in my own way.

    i. Clean(text): This method takes a text input and cleans it to my standard, this just means scrubbing white space and numbers

    ii. Makesometokens(text): This method takes a text input (ideally scrubbed with clean()) and tokenizes it. A custom splitting protocol is in place where sentences are defined between semicolons and commas. The we use NLTK's tokenize to split words (I believe via spaces and punctuation). Then we return the custom sentence chunks as well as the word tokens.

    iii. Buildfreqtable(words): This method takes a text input (that has been tokenized) and basically builds a table with each word and its corresponding score. Notably, certain words (that are predefined like a, the, is, etc) are ignored as they don't carry meaning. Nouns and verbs also receive extra score in the form of numerical weights. Then, the words and their scores are returned.

    iv. Setscores(text, freq_table): This method takes a text input as well as the set of word frequencies from buildfreqtable(). Now, using the individual word scores, this function combines the word scores to determine overall sentence scores. Then the program will give a standard ratio of the total number of words in a sentence over the number of important words found. The return is each sentence with its final score.

    v. Getsummary(sentence_scores, orig_sentence, threshold/top_k)

        1. Threshold based sorting: In this version of getsummary, we basically find the highest scoring sentence and multiply its score by the threshold. Any sentences with a value higher than that threshold will be included in the summary, any sentences with a lower value will not be.

        2. Top_K sentences: In this version, we start by finding the k-highest scoring sentences. We basically return all the sentences from highest to lowest score and only select the first k sentences. We then reorder the sentences back to how they originally were for output.

    vi. Steps for next week: 1. A non-static top_k algo that can dynamically decide the ideal length of summary 2. Maybe start investigating more abstractive techniques, leveraging LLMs.

<u>More Info & Old Data:</u>

9. **Index**

   a. **Cosine Similarity:** When using simple sentence transformers to get sentence embedding, the way that one can determine the similarities between two given phrases is by cosine similarity. The closer to 1, the more similar; The closer to 0, they're less similar. The basic idea is that one can calculate the cosine angle between two vectors. Data objects are treated as a vector; The formula to calculate goes as follows: . In this we can say that x.y is the dot product between vectors x and y. ||x|| refers to the magnitude of the vector and ||x|| x ||y|| refers to the regular product of the two vectors. Note this technique requires non-zero vectors.

   $$S_C(x, y) = x \,.\, y \,/\, ||x|| \times ||y||$$

   b. **SpaCy nlp()**

      i. Tokenizer tokenizes the text into chunks

      ii. Tagger tags all the parts of speech

      iii. Parser parses the dependencies present

      iv. Named entity recognizer does what it says

      v. Text classifier does what it says

   c. **BERT: Biderectional Encoder Representations from Transformers by Google**

      i. Encoder only architecture

      ii. BERT is first trained on a large volume of unlabeled text to learn contextual items (unsupervised pre-training), then it is trained on data specific to the NLP purpose it should serve

      iii. Masked Learning Model (MLM): A portion of words inputted is hidden and the machine has to guess what is supposed to go there based on context

      iv. Next Sentence Prediction (NSP): BERT predicts if a given sentence is connected to the previous one.

   d. **Weekly Conclusions:** I figure that if this is going to be longer term, I want to keep track of my weekly decisions and overall understanding

      i. Week 1 (7/7-7/11): I think I have a barely competent (as in far from good) claim parser using pure NLP techniques. In the Git, it is referred to as scratchnltktest. In my opinion (probably not as reliable as it needs to be in a legal context), the program is mostly able to capture big picture ideas. However, it entirely misses certain specific details. While I think that would be enough for a non-legal use case, in our context, it is not good enough, I suspect that every specific aspect matters, and this program just doesn't capture it. This also kind of gets into the big picture problem that I am encountering: I can't actually generate text. So basically, when using pure NLP techniques, you have to use the text as is and modify/change the order of it (extractive summarization). On one hand, this significantly limits me because I have to basically determine patterns and assume that those patterns will apply widely enough to work on any given claim. But on the other, it is one of the only real ways that we can safely guarantee that there will be no "hallucinatory" observations by the machine. The minute we introduce a transformer or LLM that is generating text, we move toward actual AI/LLM meaning that the output is no longer just a rephrase of the original content. Continuing, a lot of the techniques that I have become privy to tend to rely on frequency of words, word roles (POS), or predetermined relationships/entities. So fundamentally, without a transformer model, the machine does not understand the semantic meaning. Because of this, we cannot generate text based on it. And furthermore, the minute we introduce an LLM or transformer, that means the machine has to make inferences, which it might mess up on.

1. Deliverables for week 1:
   a. Spacydisplaytest
   b. Spacysimilarities
   c. Scratchnltktest
   d. This document

ii. Week 2 (7/14-7/18): We have some simple implementations working across the board.

e. OLD POSSIBLE PATH OF PRODECURE (MOVED FROM BP 4):

i. Possible path of procedure:

1. NLTK based:
2. Install NLTK
3. I will copy-paste a small sample from the claims json as a list of strings

4. Then I can create a bunch of functions that will normalize, tokenize, lemmatize, stem, and remove stop words.

5. Then, we can return an output that details the following: claim #, dependency, body text, components referred to, and verbs.

6. From here, I think we basically need to use an LLM to further extrapolate conclusions from the data? If we use pure NLP techniques, we can basically shrink the volume of text with rules, but if we want to summarize conclusions or reword things to be simpler, we probably need an LLM.

f. Spacy

i. **SpaCy implementations:**

1. Spacydisplaytest: In this, all I am trying to do is get the spacy visualizer to visualize the part of speech and the relationships between words

2. Spacysimilarities:

a. Under the hood: SpaCy uses it's own model to chunk the text and determine the POS. After those connections have been made, SpaCy uses Word2Vec or a comparable algorithm to convert the text into numerical vectors. These vectors can then be compared to each other using cosine similarity.

g. Manual Annotations on Claims JSON: Work in progress

10. Language Transformers:

Amruth's Extra Notes:

- https://ar5iv.labs.arxiv.org/html/1605.01744

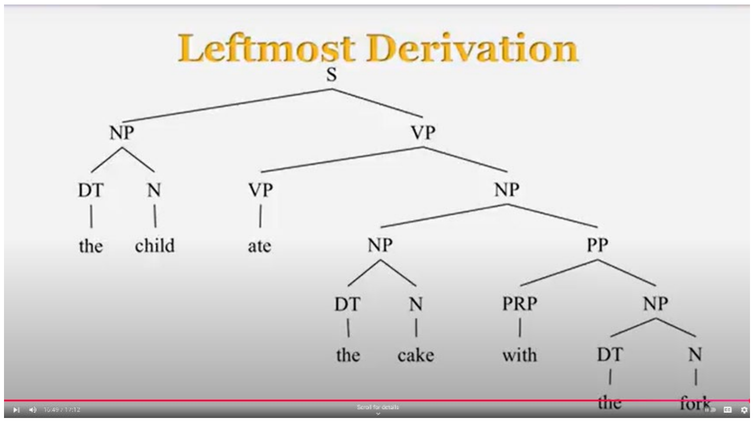
Can I use the techniques from the paper above to better the system? It looks like they used a real data set and tagged the entire set manually to define the rules for the NLP model. Then use this custom rule set to parse text with more accurate tags.

- General applications of sentence parsing: Grammar check, question answers, machine translation, information extraction, speech gen, speech understanding, interpretation

- We CANNOT use bag of words for embedding because sentences are not a bag of words, in other words, order matters!

- We can use/modify preexisting treebanks in a legal context to ensure better quality POS identification

- I am kind of theorizing that we can use known sentence structure (shown) to begin the NLP process. With the legal documentation, I am roughly outlining a concept that most legal documents flow in a specific way.



  One recurring pattern I am seeing is: Determiner->noun->preposition or conjunction. Lots of pattern recognition to be done here I think…

- Joe questions:
  - What is the context window for an LLM and how does it change if quantized? How does memory change with more or less tokens?
    - The context window is the number of tokens the LLM can take at once. For BERT that is 512.
    - Quantization is just reducing the accuracy of the multipliers in the LLM. So maybe instead of using 0.111222, the machine can just use 0.11. So, simply quantizing the multipliers will not change the maximum number of tokens that comes in, but it will change the final values of the vectors from said tokens because the multiplier is different.

        - Memory increases as you increase the number of tokens, that is because as you are able to access more tokens, you are able to access more content, thus you give the machine more context.

- Key Terms that I don’t know where to put elsewhere:
    - Basically when an LLM is generating text, it looks at previous tokens to create a most likely next word to select the best candidate.
        - Greedy decoding is when you just take the highest probability selection each time
        -

Extra resources:

- REMINDER TO LOOK AT: https://medium.com/@danielwume/getting-started-with-nltk-10-essential-examples-for-natural-language-processing-in-python-54451eae1366
- https://spacy.io/usage/spacy-101
- https://osanseviero.github.io/hackerllama/blog/posts/sentence_embeddings/
-