Named Entity Recognition in NLP

Real-world use cases, models, methods: from simple to advanced

Arun Jagota Follow
Jul 10 · 29 min read ★

Photo by Marianne Long on Unsplash

In natural language processing, named entity recognition (NER) is the problem of recognizing and extracting specific types of entities in text. Such as *people* or *place* names. In fact, any concrete "thing" that has a name. At any level of specificity. *Job titles*, *public school names*, *sport names*, *music album names*, *musician names*, *music genres*, … You get the idea.

As we will see below, NER is both an interesting problem in NLP and also has many applications. First, let's see a concrete example.

Example

From

```
Wimbledon is a tennis tournament held in the UK in the first two
weeks of July every year. In 2019, the men's singles winner was
Novak Djokovic who defeated Roger Federer in the longest singles
final in Wimbledon history.
```

we might extract entities

```
Person name(s): Novak Djokovic, Roger Federer
Sport name(s): tennis
Country name: UK
Month: July
Year: 2019
```

Let's muse on how we might use the extracted entities in the above example (and imagined extensions). This will reveal some ideas that will play out in various use cases of NER. We'll then discuss the use cases — "on the shoulders" of these ideas.

To discover the text's subject: This example is about the sport of tennis and the two players. It seems less about the country (*UK*), or the month

(*July*), or the year (*2019*). From an entire article, i.e. a lot more than two sentences, we might be able to sharpen (or refute) this assessment.

For example, if *Novak Djokovic* was frequently mentioned in the article, it might be about him. On the other hand, if *tennis* was mentioned frequently whereas each player only a few times, the article may be more about tennis. Even more so if there were other person names also mentioned in the article, each only a few times.

If *2019* was mentioned repeatedly in the article, perhaps the article is about the tournament that year.

To discover relationships among entities: Say we had a large corpus. Say the person name *Roger Federer* occurs frequently in it within the proximity of the sport *tennis*. It would be reasonable to infer that the two are related. In fact, this inference is just a specific instance of the type:

```
“statistical co-occurrence” (usually with some qualification added)
implies association.
```

In our case, the qualification is that we are limiting ourselves to entities, not to arbitrary words or phrases in the text. Plus we know the entity type of each.

What strength should we attach to an inferred association? We won't go into details here. Google *association rules*, *confidence*, and *lift* if you are curious.

So, from a large corpus and powered by a high-quality NER approach, we can build what is called an *entity graph*. The graph's nodes are entities, with attached attributes such as entity types. The graph is directed. Each arc on the graph has an attribute that captures the strength of the directed association. Arcs may have additional meta-data.

Next, let's discuss specific use cases.

Search

An important one is in web search. By understanding the entities in a query, search engines deliver better results. Google uses this approach.

Let's discuss this in more detail. Traditionally, search engines use an approach called *keyword search*. The document is indexed by the words that appear in it. The document is scored for relevance to a query by computing which query words appear in the document and weighing each hit suitably. A hit, i.e. a query word that appears in the document, is weighed by a factor called TF-IDF. This is based on the frequency of the word in the document adjusted for its rarity in the corpus. So if the query contains an uncommon word and this word appears very frequently in the document, the document's relevance to the query is high. Makes sense right?

Keywords search is definitely very useful. However, it can be improved further by augmenting it with entity recognition. This uses, in addition, knowledge about the main *entities* found in the document together with judgments on how significant of a role they play in that document.

Beyond improving relevance, this approach has another major benefit. If the query contains recognized entities, the search engine can in addition return key facts about them. The reader might have seen this at Google. Enter *Barack Obama* as the search term. The search engine understands this entity. It uses this understanding to return other key facts about Obama. Such as those gleaned from his Wikipedia page. And pictures of him.

Search engines backed by an entity *graph* (in addition to entity recognition) can improve search relevance even further. They can serve up better suggestions, e.g of the type one sees as “People also search for”.

Indexing documents

Another important one is in cataloging or indexing text documents such as web pages or online articles. This involves extracting the salient entities in a document and indexing the document using these. Online magazines and news publishers do this a lot. And of course search engines.

One way to think of the relevance value of this is the following. A document is more likely to be about frequently-mentioned entities in a document than about frequently mentioned arbitrary words. That is, an entity check acts as a quality filter.

Product reviews

A third use case is in extracting product names or other salient entities from online reviews in a retail setting. These reviews may be at eCommerce sites or at social media ones. Clearly it is useful to know which products and

features are being discussed in a particular review. As an example, consider the review

```
I like my new XYZ smartphone but the camera sucks
```

XYZ smartphone and *camera* are the entities we'd want to extract. We'd also want to extract the sentiment words *like* and *sucks* and associate each with the correct entity. Sentiment analysis is beyond the scope of the present post. Other than to say that sentiment could also be modeled as entities. For example, we might have two entity types *positive-sentiment* and *negative-sentiment* with *like* an instance of the former *sucks* an instance of the latter. Sentiment analysis has been studied extensively. More advanced approaches exist.

Reviews may then be automatically cataloged under the products or other salient entities they apply to. The resulting groupings can also reveal new insights. Such as which products or features people are complaining (or praising) the most.

Entities in emails

A fourth use case is in detecting and extracting entities from emails and automatically triggering certain actions. Many readers might have seen this happening on their own smart devices. As an example, say you get an email from an airline confirming your purchase of a flight ticket. State-of-the-art email engines can automatically extract the relevant information (airline, departure date and time, …) and create an entry in your digital calendar from it. The entities here are *airline name*, *date*, and *time*, with *date* and *time* tagged as *departure*.

Building a structured database from a corpus

A fifth use case is in building a structured database by recognizing entities of the desired type from a corpus. For example, we might build a database of medical disease names by scraping the web and recognizing entities of this type. (As an aside, a pure dictionary-based recognition approach wouldn't work because the result we seek is that dictionary.)

There is a market for specialized lists of this sort. There are niche vendors who've been making money for a while selling such lists.

Towards building an ontology

A sixth use case is essentially building an entity graph from a corpus of documents such as web pages or those internal to a company. It may also be seen as an enhanced version of the fifth use case in which relationships among entities are also inferred.

The nodes are the recognized entities. The arcs come from pairs of entities being in sufficiently close proximity frequently enough. For finer points on arcs, such as weights, see the previous discussion.

Beyond the uses of an entity graph mentioned earlier, there is a market for deep and rich domain-specific ontologies to be offered as data products, for example, to improve browsing and search at niche e-commerce sites. Some combination of automated entity recognition and arcs inference combined with human curation has a lot of potential. There are niche companies making money in this arena.

Entity Recognition Algorithms

Next, we discuss various approaches to recognizing and extracting entities, from the simplest to the more advanced. We also discuss their benefits and limitations. We also describe how and why to combine some of the simpler approaches that are complementary, to build more advanced ones.

Dictionary-based

This is one of the simplest. We have a dictionary of values for every entity type to be recognized. To recognize and extract the entities we simply scan the text and find hits in the various dictionaries. A hit also reveals the entity type as we know the dictionary that was hit.

Here is a simple example.

Say we have the following dictionaries:

```
Month = January, February, …, July, …, December
Country = USA, United States, UK, United Kingdom, France, …
Person First Name = John, Roger, Jim, …
Person Last Name = Smith, Green, …
```

Consider the text we saw earlier.

```
Wimbledon is a tennis tournament held in the UK in the first two
weeks of July every year. In 2019, the men's singles winner was
Novak Djokovic who defeated Roger Federer in the longest singles
final in Wimbledon history.
```

The extracted entities are

```
Month = July, Person First Name = Roger, Country = UK
```

Ah, surprise — the extraction is grossly incomplete! In fact, to a lesser extent, this happens even in real-world NER. We are getting a foretaste of reality.

Well, let's dig deeper … First of all, the entity recognition is only as good as the dictionaries. Novak was missing from our person's first name dictionary, so it was missed. The lack of completeness of dictionaries and the challenge of maintaining them is definitely a major issue in this approach. Second, we chose to have separate dictionaries for a person's first and last name rather than a single dictionary for a person's full name. Since person first and last names are somewhat independent, the latter type of dictionary would have been huge. Even more challenging to put together and maintain. Given this choice, we would need some mechanism for combining a person's first and last names into a full name. In fact, it would likely be pattern-based. That is, we are already getting a glimpse of the utility of combining dictionary-based and pattern-based approaches.

Another significant issue with dictionary-based approaches, albeit one that has not surfaced in our example, is that they don't accommodate ambiguity well. The same value being in multiple dictionaries. Consider *Carter*. This is typically a person's last name but on occasion can also be a person's first name.

We could, of course, place it in both dictionaries. We wouldn't be capturing the probabilities that it is more likely to be the last name than a first name though. And without capturing such probabilities, the more the number of ambiguous values there are, the more "polluted" the dictionaries will get. We won't be able to distinguish obscure memberships from significant ones.

Pattern-based

Say we wish to extract US phone numbers in some text. Such as *(123) 456–7890*. Using a dictionary-based approach doesn't make sense. That would require storing all possible US phone numbers in the dictionary. Including all formatting variations such as *123 456 7890*.

Such entities are better recognized via pattern matching. Typically, this is done using regular expressions.

Example

Consider the text

```
Recognize all these as US phone numbers: (123) 456 7890, 456 7890,
123–456–7890, +1 (123) 456 7890.
```

The point of this example is to reveal that the patterns to be recognized do start getting intricate. Now add to this list international phone numbers from across the globe. Now, this gets *very* intricate. In software engineering parlance, pattern matching in intricate scenarios has three issues involving the 'software', here regular expressions.

1. They are harder to debug in the first place
2. They are often opaque (difficult to understand).
3. They are hard to maintain, i.e. to update when changes are needed (e.g. when bugs are found or a new scenario, such as a phone number in a new country) needs to be added.

Despite 1–3 above, pattern-matching is the approach to use when, well, the entity is best described by structural patterns. Examples of other such entities are *email addresses*, *URLs*, and *US zip codes*.

Probabilistic Dictionaries

Previously we mentioned that dictionary-based approaches don't satisfactorily accommodate ambiguity. Our example was *Carter*. It can be a person's first name or a person's last name. It should, therefore, go into both dictionaries. Doing so however implicitly assumes that *Carter* is equally likely to be a person's first name or a person's last name. This unwarranted inference can adversely affect the quality of the entity recognition results.

Fortunately, the dictionary-based approach is easy to enhance to accommodate ambiguity. To every value in a dictionary, we attach an attribute, its frequency. If a value's frequency is unavailable, we default it to some suitably chosen value, say 1. The frequency is a positive number but can be a fraction.

Here is an example. Say, *Carter* has a 20% chance of being a person's first name and an 80% chance of being a person's last name. (These percentages are made up.) We could place *Carter* in both dictionaries and assign the

frequency 0.2 in the person's first name dictionary and 0.8 in the person's last name dictionary. Think of this as *Carter*'s default frequency of 1 proportionally distributed into the two entity types.

In some use cases, frequency information is already available. Great, when this is the case. This approach can then really exploit the available frequencies to make subtle distinctions.

Okay, so we have frequency attached to every value in a dictionary. How do we actually *use* it?

First let's introduce a key concept: *likelihood*. The likelihood of value v for dictionary D, written as $P(v|D)$, is simply the probability of v under the distribution of the various values that D has. (Note that we are using the terms *entity type* and *dictionary* interchangeably.)

The likelihood $P(v|D)$ is simply the frequency of v in D divided by the sum of the frequencies of all values in D.

Now, given a value v, inferences about v's dictionary are most flexibly done via Bayesian inference

$P(D|v) = P(v|D)P(D)/P(v)$

By "flexibly done" we specifically mean that the priors $P(D)$ can be adjusted to capture our preferences (or prior beliefs).

To classify v into a specific dictionary we pick the D that maximizes $P(D|v)$. This is called the maximum a posteriori (MAP) choice.

Two noteworthy examples of priors:

1. The uniform prior in which all entity types are equally likely
2. The data-driven prior in which $P(D)$ is the sum of the frequencies of the values in dictionary D divided by the sum of the frequencies of the values in all the dictionaries.

A note of caution. To set the data-driven priors, the frequencies of the various values in the various dictionaries need to be on the same scale.

Next, let's note that $P(v) = \text{sum\_}D' P(v|D')\text{P}(D')$. So we can compute $P(v)$ on-the-fly during the process of inferring v's likely dictionary.

Let's see a numeric example. Below are three dictionaries.

```
D1      D2     D3
a 10    a 1    e 1
b 20    c 5    f 5
e 15
```

Next, we'll walk through calculating the posterior probabilities of the three dictionaries for the value a. First, let's surface what we expect these to be, qualitatively speaking. D1 should have the highest posterior, D2 the next highest, and D3 the lowest. This is because a has frequency 10 in D1, 1 in D2, and 0 in D3.

Okay, now to the calculations. We'll assume all dictionaries are equally likely. So we can discard the $P(D)$ term in the numerator of $P(D|v)$. The numerators of $P(\text{D1}|a)$, $P(\text{D2}|a)$, and $P(\text{D3}|a)$ are $P(a|\text{D1}) = 10/30$, $P(a|\text{D2}) = 1/21$, and $P(a|\text{D3}) = 0$ respectively. Consequently, the posteriors $\text{P(D1}|a)$, $\text{P(D2}|a)$, and $\text{P(D3}|a)$ are $(10/30)/(10/30+1/21)$, $(1/21)/(10/30+1/21)$, and 0 respectively. These match up with our expectations. D1 is the winner by a wide margin.

Composite Entities

By composite entities we mean entities that are themselves composed of other entities. Here are two types of examples:

```
Person name: John K Smith | Dr. John Smith | John Smith, jr | John
Smith, PhD

Street Address: 10 Main St, Suite 220 | Hannover Bldg, 20 Mary St
```

In each case, the vertical bar separates entity values.

There is an important special case of composite entities that we call multi-token entities. We will cover this case separately, following the present section's (long) discussion on composite entities. We have structured the content this way because going deep into composite entities will benefit us in the subsequent discussion on multi-token entities. The reader curious about multi-token entities should scroll down to that section once they have had "enough" of the present one.

When working with composite entities, it helps to distinguish between two computational problems.

Parsing aka Decomposition: Break a composite entity down into its component entities. Here is an example.

```
Dr. John K Smith ⇒ {salutation =Dr., first_name = John, middle_name
= K, last_name = Smith}
```

Recognition: Here the composite entity is embedded in some unstructured text. The task is to recognize the boundaries of the entity as well as its type. And maybe also parse, i.e. decompose, it into its constituents. Here is an example.

```
I looked up Dr. John K Smith. His street address came up as 10 Main
St, Suite 220 in San Francisco.
```

We'd want our recognition to produce the following results:

```
Dr. John K Smith      ⇒ person_name
10 Main St, Suite 220 ⇒ street_address
```

One reason for distinguishing between parsing and recognition is that various use cases map to one or the other. For example, we may have a database of full person names and seek to parse each person's name into its parts (first name, last name, etc). This does not involve any recognition. Just parsing.

Another reason is the "divide-and-conquer". Say our task is to recognize composite entities in unstructured text and parse them. A two-stage approach — first recognize the entity boundaries and their types, then parse for the recognized entity — makes sense.

We are not implying that a two-stage approach will be more accurate than an integrated approach which simultaneously does recognition and parsing, only that it is worth considering when you quickly want to get a reasonably effective solution and you have easier means to solve the two problems separately.

In the rest of this post, we will focus on parsing composite entities.

Methods for parsing/decomposition

First, we'd like to observe that parsing composite entities may be framed as a sequence labeling problem. The composite entity is suitably tokenized, to yield a sequence of tokens. The label associated with a token is its constituent entity. Below is an example of the composite entity person's name.

```
Token sequence: Dr. John K Smith
Label sequence: salutation first_name middle_name last_name
```

We suppose we have available a training set in the form of (*token sequence*, *label sequence*) pairs. This is similar to training sets in supervised learning in fixed-dimensional spaces except that here our inputs and labels are (aligned) *sequences*.

Why model parsing of composite entities this way? The main reason is that there is information buried in the sequence of labels that can be fruitfully exploited.

Say we have a rich data set of (*token sequence*, *label sequence*) pairs available for learning to parse person names. There are useful signals buried in just

the label sequences. *Salutation* typically comes before *first_name*, not after. *First_name* typically comes before *last_name*, not after.

Training and Inference

Okay, so we have a training set of (*token sequence*, *label sequence*) pairs. Our task is to learn a machine learning model from this data set so that when a token sequence is an input, likely one that was never seen during training, the model outputs the best label sequence for it. Again, this is similar to supervised learning in vector spaces except that our inputs and outputs are *sequences*.

Hidden Markov Models

This is the most widely known method for modeling sequence labeling problems. It fits our needs quite well. Having said that, it does make a strong assumption, the Markovian one. At times this can be overly limiting. The reader interested in more advanced methods or having use cases in which there are longer-range interactions (i.e. composite entities with long token sequences in which tokens far apart influence each other) should also read about *conditional random fields*.

Okay, back to HMMs. We'll illustrate setting this up to parse person name entities. Structurally, an HMM is a directed acyclic graph with nodes (also called *states*) and arcs (also called *transitions*). Actually there's more. Nodes may emit observables from some alphabet.

An HMM has parameters attached to the various nodes and the arcs. These parameters form suitable probability distributions. In more detail, emissions from a state are governed by a probability distribution over the alphabet of observables. This distribution is state-specific. Transitions from a state are governed by a probability distribution over states. These distributions are specific to the starting state. In short, from a state you have to go somewhere. The probability distribution over the transitions from the state captures our preferences of where (which state) we should go next.

For Parsing Person Names

This is all a bit abstract. Let's make it concrete in our setting. This exercise will also reveal the 'H' in HMM, which stands for *Hidden*.

We will depict an HMM for parsing a somewhat simplified version of a person's full name. Specifically, we will only parse names that are a match to the following regular expression format.

```
[salutation_word] first_name_word [middle_name_word] last_name_word
```

Let's read out what this regular expression says. A full name optionally starts with a salutation, followed by a first name, then optionally a middle name, and finally the last name. All these entities must be single-word entities.

HMM vs Patterns-based Parsing

First question, why not just use this regular expression to parse? That is, why not apply a patterns-based approach to this parsing problem. You could if your aim was to quickly develop a light-weight approach.

The HMM-based approach is more powerful in the following ways.

Partial parses: Consider the input *Smith*. The HMM-based approach would likely parse it as last_name = *Smith*. This is because it models entity probabilities as well. (Details below.) The patterns-based approach is incapable of doing this. It has no way of knowing that *Smith* is more likely to be the last name than the first name.

Resolve ambiguities better: Consider the input *Dr. Smith*. The HMM-based approach would likely parse this as **salutation** = *Dr.* and **last_name** = *Smith*. The patterns-based approach would likely parse this as **first_name** = *Dr.* and **last_name** = *Smith*. Once again, since it doesn't model entity probabilities it has no way of knowing that *Dr.* is much more likely to be a salutation than a first name.

Attach a parse confidence score: Consider two inputs *John Smith* and *Smith John*. As would the patterns-based approach, the HMM-based one may very well parse these as {**first_name**=*John*, **last_name** = *Smith*} and {**first_name** = *Smith*, **last_name** = *John*} respectively. The HMM-based one would likely attach higher confidence to the first parse than the second one, as it knows that John is much more likely to be the first name than the last name, and Smith much more likely to be the last name than a first name.

Such confidence scores can be used to find low-confidence parses (and maybe treat them differently). They can also surface data issues. As in this example, in which the first and last names seem to have been reversed.

Person Names Parsing HMM: The Structure

Okay, let's come back to the regular expression that defines what the HMM will match.

```
[salutation_word] first_name_word [middle_name_word] last_name_word
```

From this, we will determine the states of the HMM to be *salutation_word*, *first_name_word*, *middle_name_word*, and *last_name_word*. To these, we will add two more states: *begin* and *end*.

We will use the regular expression to determine the arcs as well. Below are the arcs

```
begin → salutation_word
begin → first_name_word

first_name_word → middle_name_word
first_name_word → last_name_word

last_name_word → end
```

Computer scientists, think of the structure of the HMM as defining a finite-state automaton.

Before moving off this topic, we'd like to note that it is possible to infer the structure of the HMM (nodes and transitions) directly from the training set. This can be more flexible. As the training set evolves, so can the structure.

Our main reason for fixing the structure in advance based on the regular expression is pedagogical. It allows us to separate the discussion on the HMM's structure from a description of its learning (of the probability parameters on its emissions and transitions).

We'd also like to note that even when a training set can in principle determine the structure, it is helpful to consider initializing the structure, before seeing the training set, with prior domain knowledge if available. This lets one use domain knowledge (for Bayesians, prior beliefs) into the modeling. This can compensate for weaknesses in the training set.

Training

Next, we will illustrate training the parameters of the HMM from a training set.

As we mentioned earlier, the training set is a collection of (*token sequence*, *label sequence*) pairs. From such a training set, the emission and the transition probabilities are learned in the HMM. The training of these two types of probabilities decomposes, as we will see soon. We might call such a training *modular*. Following the training illustration, we will touch on some benefits of modular training.

Let's see an example. It has three training instances. Notice that we have added the *begin* and *end* states to the flanks. These will play a role in the training, as we will see soon.

```
         Dr.        John       K          Smith
begin salutation first_name middle_name last_name end

         John      Smith
begin first_name last_name end

         John       Kent       Smith
begin first_name middle_name last_name end
```

The *begin* and *end* states are called silent because they don't emit any tokens.

Training The Transition Probabilities

For training these, we only need to look at the state sequences. It is as if our training set was

```
begin salutation first_name middle_name last_name end
begin first_name last_name end
begin first_name middle_name last_name end
```

The probability of the transition from the state a to state b, i.e. on the arc $a \rightarrow b$, is simply the fraction of occurrences of state a (excluding any as the last state in a state sequence) in which it was succeeded by state b.

Let's see some of the learned transition probabilities.

```
begin → salutation       1/3
first_name → middle_name 2/3
first_name → last_name   1/3
```

One of three occurrences of the state *begin* is followed by the state *salutation*. Two of three occurrences of the state *first_name* are followed by

the state *middle_name*, one by the state *last_name*.

Training The Emission Probabilities

For training these, we throw away the sequence information and just track (*state*, *token*) pairs every time token *token* is emitted from state *state*. It is as if the training set were

```
state       emitted token
salutation  Dr.
first_name  John
first_name  John
middle_name K
middle_name Kent
last_name   Smith
last_name   Smith
```

The probability of emitting token *t* from state *s* is just the fraction of occurrences of state *s* in which token *t* was emitted.

Modular Training

The number of different values a state can plausibly emit is quite large. For example, state *first_name* can emit millions of different values. Each needs

to be explicitly modeled, so as to distinguish first names, however rare, from gibberish strings such as *xyz*.

It is unrealistic to expect that the training set of (*token sequence*, *label sequence*) pairs will be rich enough to cover all plausible first names from it. Fortunately, it doesn't have to be. The emissions can be trained independently of the transitions. So long as we have rich dictionaries of values for each entity (state), preferably augmented with frequency information, we can learn the emission probabilities from these.

Also note that, by contrast, this explosion does not apply to learn transition probabilities. A small number of curated state sequences can adequately capture the sequence structure of most full names. "Small" may not be as small as one might think if one wishes to be able to parse person names in all kinds of locales. Spanish and Arabic for example. Still, manual curation of a training set even in this situation is feasible.

Inference

Okay, now that we have the trained HMM, or at least we can imagine it, let's walk through an example of inference. The specific aim of inference is to input a sequence of tokens and output the best parse. In order for this

process to be well-defined, we need (i) a *score function* that evaluates the quality of a specific parse of the token sequence and (ii) a *search procedure* that finds a highest-scoring parse. The reason we need a search procedure is that the approach of exhaustive search — score all possible parses and pick one that scores highest — can be too slow, even when there are only a few tokens.

The gold standard search procedure is a form of dynamic programming called the Viterbi algorithm. The score function it implicitly maximizes is the joint probability of the (*token sequence*, *label sequence*) pair. The token sequence is fixed; the label sequence is variable, one per parse. The Viterbi algorithm operates *as if* it was doing an exhaustive search over all possible label sequences for the given token sequence, only it is much faster.

We will not describe the Viterbi algorithm formally here. It is too complex for this already long post. However, we will illustrate some aspects of its working on an example. Admittedly in a sketchy way. Our hope is that those reading about it for the first time will get curious enough to learn about it more from elsewhere (see a reference below), and those having already read about it will find some insight to take away from our illustration to add to their understanding.

The input we will illustrate it on is *Dr. John Smith*. The first key idea we'd like to depict is that the Viterbi algorithm operates by maintaining a matrix whose rows index the states of the HMM and whose columns index the tokens in the input. In our example, this matrix has the structure

		Dr.	John	Smith
begin	1			
salutation	0	P(b)P(s|b)P(Dr.|s)		
first_name	0		?	
middle_name	0			
last_name	0			
end	0	0		

What goes into each cell of the matrix and how does it get there? That's much too complicated to fully specify and explain in the space we have allocated for it. So instead we have put values or formulae in some of the cells, which we will explain below.

Let's start in the first column. The values there are easy to interpret. To parse an input, the HMM must start from the state *begin*. This is represented in the first column as "the probability of being in state *begin* is 1, being in in any other state is 0".

Next, let's look at the cell [*salutation*, *Dr.*]. The value in there is the probability of parsing up to the first token and ending up in state *salutation*. (In our case, "ending up in state *salutation*" would mean that *Dr.* was emitted from *salutation*.)

This probability is the probability of starting from the state *begin*, transiting to the state *salutation*, and emitting *Dr.* from *salutation*. This is what we see in that cell.

Now let's look at what goes into the cell [*first_name*, *John*]. We have placed a question-mark there because the expression is much too complicated to put in there in a way that captures all the important things that are happening under the hood. So we'll just talk through them here.

First, let's remind ourselves of what we *want* to go into this cell. It is the joint probability of the first two tokens and the best parse for these tokens. Say what?

Let's ease into it. First, from now on, let's abbreviate references to the aforementioned joint probability to "probability of the best parse of these tokens". This loses some precision but is much more readable. From inspection, what do you think is the best parse? [*begin*,*salutation*,

first_name] right? The probability of this parse is what would go into this cell.

Next comes a key observation. We won't prove it. It is essential to the understanding of how this algorithm works. The probability of this parse is the **probability of [*begin,salutation*] being the parse for [*Dr.*]** multiplied by the probability of extending this parse to cover *John* being emitted from *first_name*. We have already computed the portion in bold and deposited its result into the cell [*salutation*, *Dr.*] of the matrix. So we just need to multiply this cell's value with the probability of the aforementioned extension, which is the probability of transiting fro state *salutation* to state *first_name* multiplied by the probability of emitting *John* from *first_name*. This reuse of probabilities computed earlier is the key to the speedup the algorithm achieves over exhaustive search.

One issue remains. In the previous two paragraphs, to ease into the explanation, we assumed we knew that the best parse of [*Dr.*, *John*] was [*begin*, *salutation*, *first_name*]. The algorithm of course doesn't. How do we cope with this?

First, let's be clear on exactly what we *don't* know. We do know *John* must be emitted from *first_name* because we are trying to fill out that cell's value.

What we don't know is the best parse of [*Dr.*] and which state it ends up in. Let's call this unknown '*Dr.*'s state'. For each possible value of *Dr.*'s state, we know the **probability of the best parse of [*Dr.*] of the form [*begin*, *Dr.*'s state]**. To each of these, we multiply out the probability of extending the parse to become [*begin*, ***Dr.*'s state**, *first_name*]. The only variable here is Dr.'s state so since we want the probability of the best parse we just take the maximum of the probabilities over all values of *Dr.*'s state. Note that in these calculations we are reusing the probabilities in bold, as they have already been computed and stored in the matrix before then.

Recovering The Best Parse

We are not done yet. After we have filled in the matrix all we know is the probability of the best parse of the full input. This sits in the bottom-right cell of the matrix. We don't know the actual parse.

To recover an actual (highest-scoring) parse, first, during the "bottom-up" phase in which the matrix of scores is built, we capture some additional meta-data. Once the matrix has been built, we use this meta-data to find an actual parse.

Let's see the matrix of scores, with some cells augmented with the aforementioned meta-data. Let's depict this in its own matrix, which we will call the *back-pointers matrix*. This matrix has the same structure as the matrix of scores. The only difference is that its cells don't store scores, rather indices of other cells. Hence the name "back-pointers".

Below is the back-pointers matrix with some values filled in. Let's explain one. Cell [2,2] contains the value [1,1]. This captures the information that the best parse of [*Dr.*, *John*] in which state *first_name* is constrained to emit *John* emits *Dr.* from state *salutation*. This information emerges during the process of computing the score in the corresponding cell in the scores matrix. Specifically during the process of computing the maximum over the various terms.

	0	1	2	3
		Dr.	John	Smith
0	begin			
1	salutation	[0,0]		
2	first_name		[1,1]	
3	middle_name			
4	last_name			[2,2]
5	end			[4,3]

Next, let's walk through how this matrix is used to find an actual best parse. We start at the bottom-right cell, i.e. at cell [5,3]. From this cell, we follow a path back to the cell [0,0]. In our case, the path is

```
[5,3] → [4,3] → [2,2] → [1,1] → [0,0]
```

This path contains within it the information to recover a (best) parse in reverse. This is depicted below.

```
[5,3]    →    [4,3]    →    [2,2]    →    [1,1]   →    [0,0]
                      Smith          John         Dr.
       end        last_name      first_name   salutation     begin
```

In each back-pointer step, we uncover the *state* of one token in the reversed token sequence. In the step [5,3] → [4,3] we recover [,end]. The token is missing as *end* is a silent state. In the step [4,3] → [2,2] we are moving from state 4 (*last_name*) to another state (state 2) and simultaneously from token 3 (*Smith*) to token 2. These moves yield the pair (*Smith*, *last_name*). And so it goes.

The final step is to reverse the parse, which is easy to do with the help of a stack.

Multi-token entities

Multi-token entities are a special case of composite entities in which the value of an entity spans multiple tokens but the entity type itself is not decomposed into finer entity types. (Or we don't care to.)

Here are two examples of such entities: *national park names* and *medical disease names*. Let's see some instances of each:

```
national park names: Yellowstone national park, Yosemite national
park, Grand Canyon national park, ...

disease names: Parkinson's disease, Type I diabetes, heart disease,
alzheimer's disease
```

In multi-token entities, there is no parsing problem, only one of recognition. That is, the aim is to recognize entities and their boundaries in flowing text. As in

```
I traveled to Yellowstone National Park and after that went to
Colorado.
```

Methods

Dictionary-based

This is one of the simplest. However, it won't work when the dictionary is incomplete, as would be the case when we are trying to use entity recognition to build a rich dictionary in the first place. (See the use case **Building a structured database from a corpus** described earlier in this post.)

Supervised binary classification

This is an attractive approach to this problem. A training instance would be a short phrase labeled as *positive* (instance of the entity) or *negative* (not an instance of the entity). We'd also choose appropriate features to extract from the input phrase. (More on this later.)

An important consideration is how to choose the negative instances for the training set. The universe of negative instances is huge — all phrases up to a certain length (what should this be?). Randomly sampling from the universe may give us weak negatives. Consequently, the learned classifier may be weaker than we think it is, relative to our assessment on a train-test split of the labeled data set. Point being that the negative instances are not real — we constructed them via a certain process — and that process may not be representative of the actual negatives we get in the field, which would be sufficiently short phrases in the text that are not instances of this entity.

By contrast, adding human-curated negatives that superficially look like positives will likely yield a more accurate classifier. Here is an example of one such negative for the entity *national park name*: *Castle Rock State Park*. It is a park but not a national one.

The discussion of the previous paragraphs also implies that no matter how carefully we pick the negatives, the ultimate evaluation of the classifier's accuracy should be done on its entity recognition accuracy on actual text representative of how it will be used in practice.

Once again, this is because we are picking the negatives from a huge space, and we cannot rule out selection bias. A second important reason is that the class proportions in the field, specifically the fraction of phrases input to the classifier during operation that is negative, will heavily depend on the text to which it is applied. Consider disease names. The fraction of phrases in financial documents that are disease names will likely be far lower than in documents at WebMD. Consequently, the false positive rate on the two might be quite different.

Feature Engineering

Consider recognizing national park names, i.e. distinguishing them from other phrases. The examples we have seen suggest a for-purpose heuristic variant of bag-of-words as the features. Specifically, first, we tokenize the input into words, then combine the last two words into a bigram, together with capturing that they are the last two words. This is easiest illustrated with examples.

```
Yellowstone National Park --> {yellowstone, 'national_park_end'}
Castle Rock State Park    --> {castle,rock,'state_park_end'}
```

Why did we combine the last two words into a bigram? Because leaving them separate risks a higher false-positive rate. For example, *Yellowstone national bank* (a made-up example).

It's worth noting that this scheme offers some flexibility in that should we decide at a later stage to also deem national monuments as national parks, we need just add some to our positives. An example is Dinosaur National Monument. The features we have chosen will pick these up.

These features leveraged the structure of national park names. How well they work on some other multi-token problem depends on whether or not its structure matches that of national park names. Consider disease names. Sure, the last word is often a predictor of *positive* versus *negative*. (See disease names examples we previously listed.) That said, combining the last two words into a bigram may not generalize well enough. This is because in disease names only the last word may be a cue word. (E.g. *disease*.) Using the national park name features will not detect diseases that end in *word disease* unless the training set has some positives that end in *word disease*. *word* may be a very specific word. This is why we say these features don't generalize well to disease names.

The Classifier Algorithm

Now that we have our labeled training set and features defined, we may in principle use any binary classifier algorithm for this problem. Two that come to mind our logistic regression and naive Bayes. We won't delve into either.

Alternative Solution: the HMM

A different way to solve the problem is to formulate it as a sequence labeling problem, albeit with a twist. This merits consideration as multi-token entity recognition definitely has a sequential structure. That said, it does need some thought on what the primitive entities would be. Plus, how we will turn an HMM into a binary classifier.

Consider national park names. Take *Yellowstone National Park*. Say we model its corresponding state sequence as *name_word geo_level_word park_type_word*. That is, as a (token sequence, label sequence) instance this would be represented as

```
          Yellowstone  National        Park
begin     name_word    _national_      park_type_word     positive
```

Now consider a negative: *Castle Rock State Park*. Its labeled instance may look like this.

```
          Castle     Rock      State
begin     name_word  name_word _state_                  negative
```

Let's look at one more negative: *Mocca Ice Cream*. Its labeled instance may look like this.

```
          Mocca    Ice      Cream
begin     word     word     word                       negative
```

Our HMM will have the states we see in these three label sequences. The idea here is the following. On any particular input, we will force a path that ends in the state *positive* or *negative*. If both states are reachable from *begin* on some input, then the hope is that our HMM structure and training will favor the path that leads to the class (*positive* or *negative*) of the input.

Discussion

What is offered in the previous section are the key ingredients of such a solution. In practical settings, this approach may need further refinement.

For example in the states (for example further refinements of the state *word*, perhaps part-of-speech: *proper-noun*, *noun*, *verb*, *adjective*, *article*, …

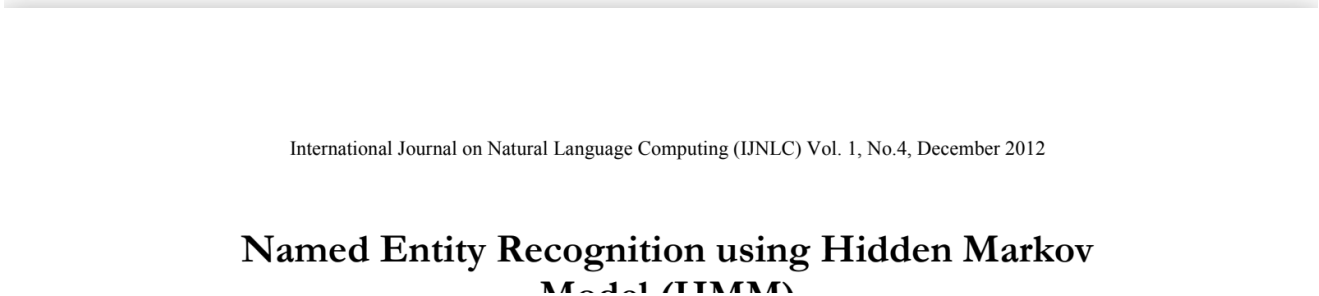
Or in relaxing the first-order Markovian assumption. Most generally, as the conditional random field does.

Further Reading

Named Entity Recognition(NER) using Conditional Random Fields in NLP

Its time to jump on Information Extraction in NLP after a thorough discussion on algorithms in NLP for pos tagging...

medium.com



International Journal on Natural Language Computing (IJNLC) Vol. 1, No.4, December 2012

Named Entity Recognition using Hidden Markov Model (HMM)

Model (HMM)

Sudha Morwal [1], Nusrat Jahan [2] and Deepti Chopra [3]

[1]Associate Professor, Banasthali University, Jaipur, Rajasthan-302001
.in

[2] M.Tech (CS), Banasthali University, Jaipur, Rajasthan-302001

[3] M. Tech (CS), Banasthali University, Jaipur, Rajasthan-302001

Abstract

Named Entity Recognition (NER) is the subtask of Natural Language Processing (NLP) which is the branch of artificial intelligence. It has many applications mainly in machine translation, text to speech synthesis, natural language understanding, Information Extraction, Information retrieval, question answering etc. The aim of NER is to classify words into some predefined categories like location name, person name, organization name, date, time etc. In this paper we describe the Hidden Markov Model (HMM) based approach of machine learning in detail to identify the named entities. The main idea behind the use of HMM model for building NER system is that it is language independent and we can apply this system for any language domain. In our NER system the states are not fixed means it is of dynamic in nature one can use it according to their interest. The corpus used by our NER system is also not domain specific.

Keywords

Named Entity Recognition (NER), Natural Language processing (NLP), Hidden Markov Model (HMM).

1.Introduction

Named Entity Recognition is a subtask of Information extraction whose aim is to classify text from a document or corpus into some predefined categories like person name, location name, organisation name, month, date, time etc. And other to the text which is not named entities. NER has many applications in NLP. Some of the applications include machine translation, more accurate internet search engines, automatic indexing of documents, automatic question-answering, information retrieval etc. An accurate NER system is needed for these applications.
Most NER systems use a rule based approach or statistical machine learning approach or a Combination of these. A Rule-based NER system uses hand-written rules frame by linguist which are certain language dependent rules that help in the identification of Named Entities in a document. Rule based systems are usually best performing system but suffers some limitation such as language dependent, difficult to adapt changes.

Machine-learning (ML) approach Learn rules from annotated corpora. Now a day's machine learning approach is commonly used for NER because training is easy, same ML system can be used for different domains and languages and their maintenance is also less expensive? There are various machine learning approaches for NER such as CRF (conditional Random Fields),

DOI : 10.5121/ijnlc.2012.1402

Hidden Markov Models Pattern Matching Dictionary Sequence Labeling