Al Barrentine Follow

Feb 25, 2016 · 22 min read · Listen

Statistical NLP on OpenStreetMap

Toward a machine-interpretable understanding of place

Street addresses are among the more quirky artifacts of human language, yet they are crucial to the increasing number of applications involving maps and location.

Last year I worked on a collaboration with Mapzen with the goal of building smarter, more international geocoders using the vast amounts of local knowledge in open geographic data sets.

The result is libpostal: a multilingual street address parsing/normalization library, written in C, that can handle addresses all over the world.

Libpostal uses machine learning and is informed by tens of millions of real-world addresses from OpenStreetMap. The entire pipeline for training the models is open source.

Since OSM is a dynamic data set with thousands of contributors and the models are retrained periodically, improving them can be as easy as contributing addresses to OSM.

Each country's addressing system has its own set of conventions and peculiarities and libpostal is designed to deal with practically all of them.

It currently supports normalizations in 60 languages and can parse addresses in more than 100 countries. Geocoding using libpostal as a preprocessing step becomes drastically simpler and more consistent internationally.

The core library is written in pure C, which means that in addition to having a small

But let's rewind for a moment.

Why we care about addresses

Addresses are the unique identifiers humans use to describe places, and are at the heart of virtually every facet of modern Internet-connected life: map search, routing/directions, shipping, on-demand transportation, delivery services, travel and accommodations, event ticketing, venue ratings/reviews, etc. There's a $1B company in almost every one of those categories.

The central information retrieval problem when working with addresses is known as geocoding. We want to transform the natural language addresses that people use to describe places into lat/lon coordinates that all our awesome mapping and routing software uses.

Geocoding's not your average document search. Addresses are typically very short strings, highly ambiguous, and chock full of abbreviations and local context.

There is usually only one correct answer to a query from the user's perspective (with the exception of broader searches like "restaurants in Fort Greene, Brooklyn"). In some instances we may not even have the luxury of user input at all e.g. batch geocoding a bunch of addresses obtained from a CSV file, the Web or a third-party API.

Despite these idiosyncrasies, we tend to use the same full-text search engines for addresses as we do for querying traditional text documents. Out of the box, said search engines are terrible at indexing addresses. It's easy to see how a naïve implementation could pull up addresses on St Marks Ave when the query was "St Marks Pl" (both the words "Ave" and "Pl" have a low inverse document frequency and do not affect the rank much). Autocomplete might yield addresses on the 300 block of Main Street for a query of "30 Main Street". Abbreviations like "Saint" and "St" which are not simple prefix overlaps might not match in most spellcheckers since their edit distance is greater than 2.

Typically we employ all sorts of heuristics to help with address matching: synonyms lists, special tokenizers, analyzers, regexes, simple parsers, etc. Most of these methods require changing the search engine's config, and make US/English-centric, overly-

Geocoding in 2016

Libpostal began with the idea that geocoding is more similar to the problem of record linkage than text search.

The question we want to be able to answer is: "are two addresses referring to the same place?" Having done that, we can simultaneously make automated decisions in the batch setting and return more relevant results in user-facing geocoders.

This decomposes into two sub-problems:

1. **Normalization**: the easiest way to handle all the abbreviated variations and ambiguities in addresses is to produce canonical strings suitable for machine comparison, i.e. make "30 W 26th St" equal to "Thirty West Twenty-Sixth Street", and do it in every language.

2. **Parsing:** some components of an address are more essential than others, like house numbers, venue names, street names, and postal codes. Beyond that, addresses are highly structured and there are multiple redundant ways of specifying/qualifying them. "London, England" and "London, United Kingdom" specify the same location if parsed to mean city/admin1 and city/country respectively. If we already know London, there would be no point in returning addresses in Manchester simply because it's also in the UK.

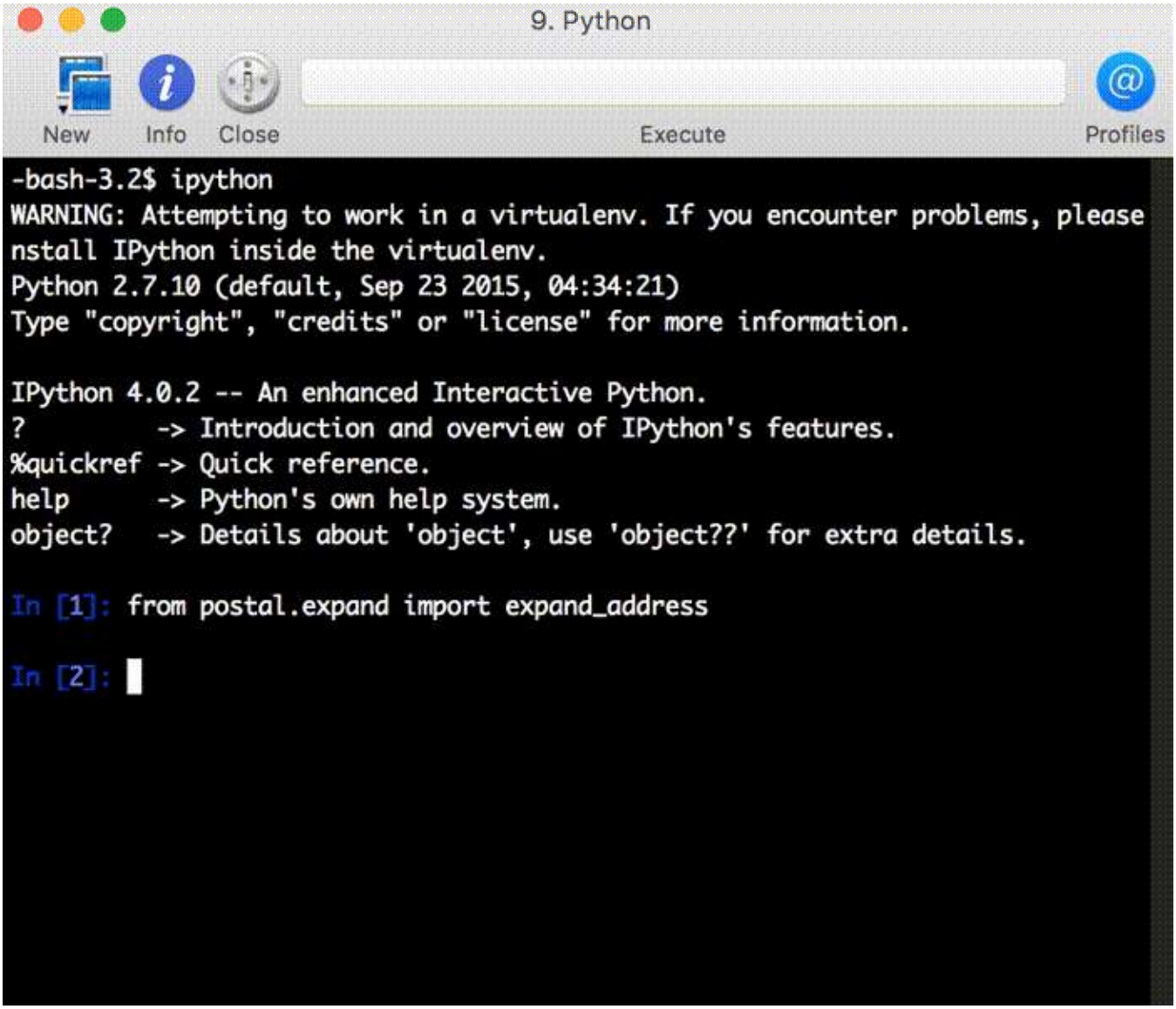
Once we've got canonical address strings segmented into components, geocoding becomes a much simpler string matching problem, the kind that full-text search engines and even relational/non-relational databases are good at handling.

With a little finesse one could conceivably geocode with nothing but libpostal and a hash table.

To see how that's possible, the next two sections describe in detail how libpostal addresses (pun very much intended) the normalization and parsing problems respectively.

Multilingual address normalization

Normalization is the process of converting free-form address strings encountered in



Address normalization using libpostal's Python bindings

There are several steps involved in making normalization work across so many different languages. I'll mention the notable ones.

Multilingual tokenization

Tokenization is the process of segmenting text into words and symbols. It is the first step in most NLP applications, and there are many nuances. The tokenizer in libpostal is actually a lexer implementing the Unicode Consortium's TR-29 spec for unicode word segmentation. This method handles every script/alphabet, including ideograms (used in languages not separated by whitespace e.g. Chinese, Japanese, Korean), which are read one character at a time.

equivalent. Indeed, tokenization is quite fast, chunking through > 2 million tokens per second.

Abbreviation expansion

Almost every language on Earth uses abbreviations in addresses. Historically this had to do with width constraints on things like street signs or postal envelopes. Digital addresses face similar constraints, namely that they are more likely than other types of text to be viewed on a mobile device.

Abbreviations create ambiguity, as there are multiple ways of writing the same address with different degrees of verbosity: “W St Johns St”, “W Saint Johns St”, “W St Johns Street”, and “West Saint Johns Street” are all equivalent. There are similar patterns in most languages.

For expanding abbreviations to their canonical forms, libpostal contains a number of per-language dictionaries, which are simple text files mapping “Rd” to “Road” in 60 languages. Each word/abbreviation can have one or more canonical forms (“St” can expand to “Street” or “Saint” in English), and one or more dictionary types: directionals, street suffixes, honorifics, venue types, etc.

Dictionary types make it possible to control which expansions are used, say if the input address is already separated into discrete fields, or if using libpostal’s address parser to the same effect. With dictionary types, it’s possible to apply only the relevant expansions to each component. For instance, in an English address, “St.” always means “Saint” when used in a city or country name like “St. Louis” or “St. Lucia” and will only be ambiguous when used as part of a street or venue/building name.

The dictionaries are compiled into a trie data structure, at which point a fast search algorithm is used to scan through the string and pull out matching phrases, even if they span multiple words (e.g. “State Route”). This type of search also allows us to treat multi-word phrases as single tokens during address parsing.

Ideographic languages like Japanese and Korean are handled correctly, even though the extracted phrases are not surrounded by whitespace. So are Germanic languages where street suffixes are often appended onto the end of the street name, but may

At the moment, libpostal does not attempt to resolve ambiguities in addresses, and often produces multiple potential expansions. Some may be nonsensical ("Main St" expands to both "Main Street" and "Main Saint"), but the correct form will be among them. The outputs of libpostal's expand_address can be treated as a set and address matching can be seen as a doing a set intersection, or a JOIN in SQL parlance. In the search setting, one should index all of the strings produced, and use the same code to normalize user queries before sending them to the search server/database.

Future iterations of expand_address will probably use OpenStreetMap (where abbreviation is discouraged) to build classifiers for ambiguous expansions, and include an option for outputs to be ranked by likelihood. This should help folks who need a "single best" expansion e.g. when displaying the results on a map.

Address language classification

Abbreviations are language-specific. Consider expanding the token "St." in an address of unknown language. The canonical form would be "Sankt" in German, "Saint" in French, "Santo" in Portuguese, and so on.

We don't actually want to list all of these permutations. In most user-facing geocoders, we likely know the language ahead of time (say from the user's HTTP headers or current location). However, in batch geocoding, we don't know the language of any of our input addresses, so will need a classifier to predict languages automatically using only the input text.

Language detection is a well-studied problem and there are several existing implementations (such as Chromium's compact language detector) which achieve very good results on longer text documents such as Wikipedia articles or webpages. Unfortunately, because of some of the aforementioned differences between addresses and other forms of text, packages like CLD which are trained on webpages usually expect more/longer words than we have in an abbreviated address, and will often get the language wrong or fail to produce a result at all.

So we'll need to build our own language classifier and train it specifically for address data. This is a supervised learning problem, which means we'll need a bunch of address-related input labeled by language, like this:


```
sv  Tollare Träskväg
nl  Johannes Vermeerstraat Akersloot
it  Strada Provinciale Ca' La Cisterna
da  Østervang  Vissenbjerg
nb  Lyngtangen Egersund
en  Wood Point Road
ru  улица Солунина
ar  جادة صائب سلام
fr  Rue De Longpré
he  השלום
ms  Jalan Sri Perkasa
cs  Jeřabinová  Rokycany
ja  山口秋穂線
ca  Avinguda Catalunya
es  calle Camilo Flammarión
eu  Mungialde etorbidea
pt  Rua Pedro Muler Faria
```

Sounds great, but where are we going to find such a data set? In libpostal, the answer to that question is almost always: use OpenStreetMap.

OSM has a great system when it comes to languages. By default the name of a place is the official local language name, rather than the Anglicized/Latinized name. Beijing's default name for instance is "北京市" rather than "Beijing" or "Peking."

Some addresses in OSM are explicitly labeled by language, especially in countries with multiple official street sign languages like Hong Kong, Belgium, Algeria, Israel, etc. In cases where a single name is used, we build an R-tree polygon index that can answer the question: for a given lat/lon, which official and/or regional language(s) should I expect to see? In Germany we expect addresses to be in German. In some regions of Spain, Catalan or Basque or Galician will be returned as the primary language we expect to see on street signs, whereas (Castilian) Spanish is used as a secondary alternative. In cases where languages are equally likely to appear, the language dictionaries in libpostal are used to help disambiguate. Lastly, street signs are always be written in the languages spoken by the majority of people, a vestige of linguistic imperialism, and the language index accounts for this as well.

All said and done, this process produces around 80 million language-labeled address strings. From there we extract features (informative attributes of the input which help

or Hebrew. Specific to our use case, we also include entire phrases matching certain language dictionaries from libpostal.

We then train a multinomial logistic regression model (also known as softmax regression) using stochastic gradient descent and a few sparsity tricks to keep training times reasonably fast. Logistic regression is heavily used in NLP because unlike Naïve Bayes, it does not make the assumption that input features are independent, which is unrealistic in language.

Another nice property of logistic regression is that its output is a well-calibrated probability distribution over the labels, not just normalized scores that look like probabilities if you "close one eye and squint with the other." With real probabilities we can implement meaningful decision boundaries. For instance, if the top language returned by the classifier has a probability of 0.99, we can safely ignore the other language dictionaries, whereas if it makes a less confident prediction like 0.62 French and 0.33 Dutch, we might want to throw in both dictionaries. Though the latter type of output should *not* be interpreted as the distribution of languages in the address itself (as in a multi-label classifier), results with multiple high-probability languages are most often returned in cases like Brussels where addresses actually are written in two languages.

Numeric expression parsing

In many addresses, particularly on the Upper East Side of Manhattan it seems, numbers are written out as words e.g. "Eighty-sixth Street" instead of "86th Street." Libpostal uses a simplified form of the Rule-based Number Format (RBNF) in CLDR which spells out the grammatical rules for parsing/spelling numbers in various languages.

Rather than try to exhaustively list all numbers and ordinals that might be used in an address, we supply a handful of rules which the system can then use to parse arbitrary numbers.

In English, when we see the word "hundred", we multiply by any number smaller than 100 to the left and add any number smaller than 100 to the right. There's a recursive structure there. If we know the rule for the hundreds place, and we know how to parse

Numeric spellings can get reasonably complicated in other languages. French for instance uses some Celtic-style numbers which switch to base 20, so “quatre-vingt-douze” (“four twenties twelve”) = 92. Italian numbers rarely contain spaces so “milleottocentodue” = 1802. In Russian, ordinal numbers can have 3 genders. Libpostal parses them all, currently supporting numeric expressions in over 30 languages.

Roman numerals can be optionally recognized in any language (so “IX” normalizes to 9), though they’re most commonly found in Europe in the titles of popes, monarchs, etc. In most cases Roman numerals are the canonical form, and can be ambiguous with other tokens (a single “I” or “V” could also be a person’s middle initial), so a version of the string with unnormalized Roman numerals is added as well.

Transliteration

Many addresses around the world are written in a non-Latin scripts such as Greek, Hebrew, Cyrillic, Han, etc. In these cases, addresses can be written in the local alphabet or transliterated i.e. converted to a Latin script equivalent. Because the target script is usually Latin, transliteration is also sometimes known as “Romanization.”

For example, “Тверская улица” in Moscow transliterates to “Tverskaya ulitsa.” A restaurant website would probably use the former for its Russian site and the latter for its international site. Street signs in many countries (especially those who’ve at some point hosted a World Cup) will typically list both versions, at least in major cities.

Libpostal takes advantage of all the transliterators available in the Unicode Consortium’s Common Locale Data Repository (CLDR), again compiling them to a trie for fast runtime performance. The implementation is lighter weight than having to pull in ICU, which is a huge dependency and may conflict with system versions.

Each script or script/language combination can use one or more different transliterators. There are for instance several differing standards for transliterating Greek or Hebrew, and libpostal will try them all.

There’s also a simpler transliterator, the Latin to ASCII transform, which converts “œ” to “oe”, etc. This is in addition to standard Unicode normalization, which would

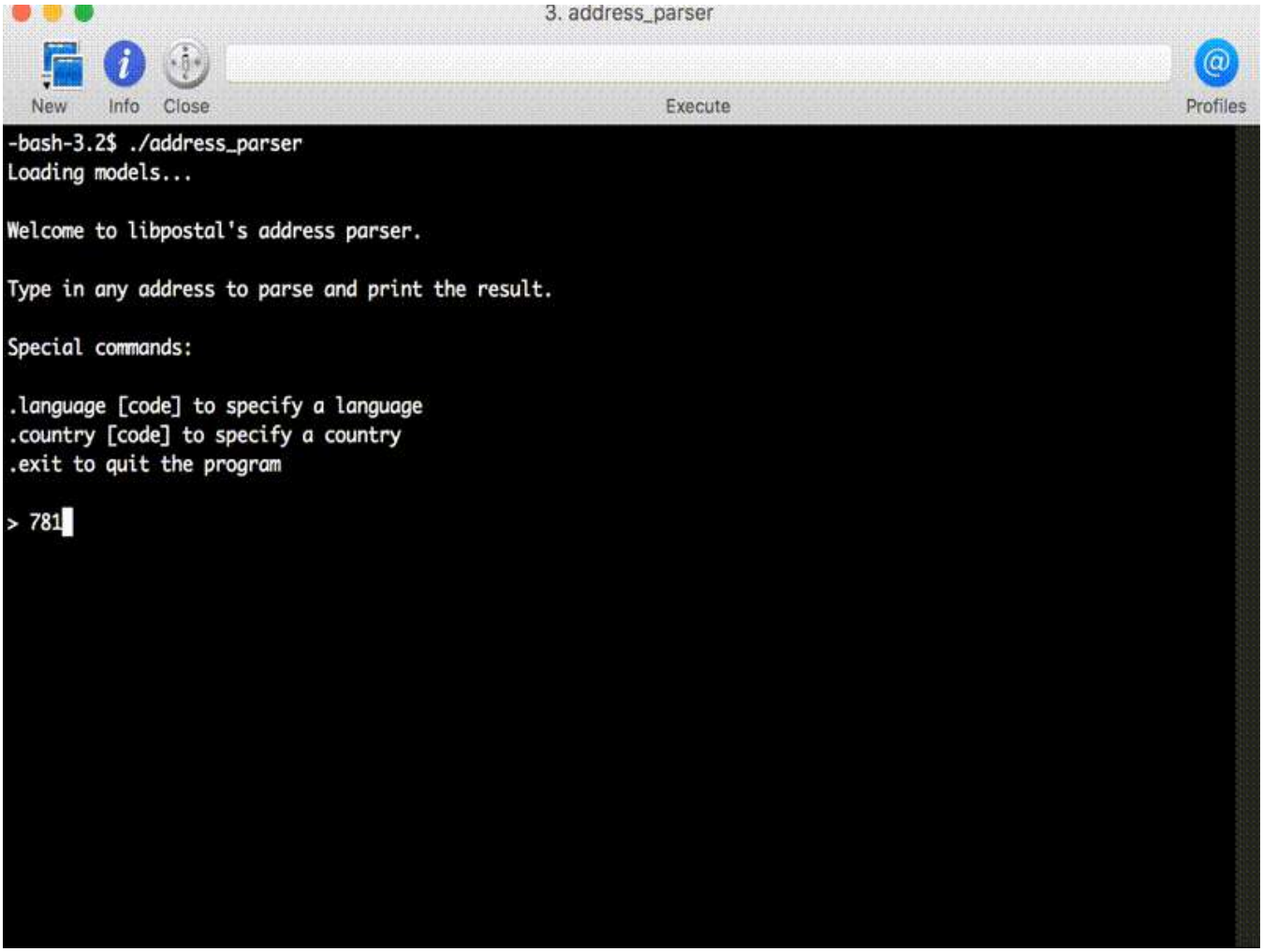
 

words. Still, sometimes addresses have to be written in an ASCII approximation (because keyboards), especially with travel-related searching, so we do strip accent marks by default, with an optional flag to prevent it.

Some countries actually translate addresses into English (something like “Tverskaya Street”), creating further ambiguity. At the cost of potentially adding a few bogus normalizations, libpostal can handle such translations by simply adding English dictionaries as a second language option for certain countries/languages/scripts.

International address parsing

Parsing is the process of segmenting an address into components like house number, street name, city, etc. Though many address parsers have been written over the years, most are rule-based and only designed to handle US addresses. In libpostal we develop the first NLP-based address parser that works well internationally.



International address parsing is something we could never possibly hope to solve deterministically with something like regex. It might work reasonably well for one country, as addresses tend to be highly structured, but there are simply too many variations and ambiguities to make it work across languages. This sort of problem is where machine learning, particularly in the form of structured learning, really shines.

Most NLP courses/tutorials/libraries focus on models and algorithms, but applications on real-world data sets are not in great abundance. Libpostal provides an example of what an end-to-end production-quality NLP application looks like. I'll detail the relevant steps of the pipeline below, all of which are open source and published to Github as part of the repository.

Creating labeled data from OSM

OpenStreetMap addresses are already separated into components. Here's an example of OSM tags as JSON:

```
{
    "addr:housenumber": "30",
    "addr:postcode": "11217",
    "addr:street": "Lafayette Avenue",
    "name": "Brooklyn Academy of Music"
}
```

This is exactly the kind of output we want our parser to produce. These addresses are hand-labeled by humans and there are lots of them, more than 50 million at last count.

We want to construct a supervised tagger, meaning we have labeled text at training time, but only unlabeled text (geocoder input) at runtime. The input to a sequence model is a list of tagged tokens. Here's an example of the for the address above:

```
Brooklyn/HOUSE Academy/HOUSE of/HOUSE Music/HOUSE 30/HOUSE_NUMBER
Lafayette/ROAD Avenue/ROAD Brooklyn/CITY NY/STATE 11217/POSTCODE
```

At runtime, we'll only expect to see "Brooklyn Academy of Music, 30 Lafayette Avenue,

Notice that the original OSM address has no structure/ordering, so we'll need to encode that somewhere. For this, we can use OpenCage's address-formatting repo, which defines address templates for almost every country in the world, with coverage increasing steadily over time. In the US, house number comes before street name ("123 Main Street"), whereas in Germany or Spain it's the inverse ("Calle Ruiz, 3"). The address templates are designed to format OSM tags into human-readable addresses in every country. This is a good approximation of how we expect geocoder input to look in those countries, which means we have our input strings. I've personally contributed a few dozen countries to the repo and it's getting better coverage all the time.

Also notice that in the OSM address, city, state, and country are missing. We can "fill in the blanks" by checking whether the lat/lon of the address is contained in certain administrative polygons. So that we don't have to look at every polygon on Earth for every lat/lon, we construct an R-tree to quickly check bounding box containment, and then do the slower, more thorough point-in-polygon test on the bounding box matches. The polygons we use are a mix of OSM relations, Quattroshapes/GeoNames localities, and Zetashapes for neighborhoods.

Making the parser robust

Because geocoders receive a wide variety of queries, we then perturb the address in several ways so the model has to train on many different kinds of input. With certain random probabilities, we use:

- **Alternate names:** for some of the admin polygons (e.g. "NYC", "New York", "New York City") so the model sees as many forms as possible
- **Alternate language names:** OSM does a great job of handling language in addresses. By default a tag like "name" can be assumed to be in the local official language, or hyphenated if there's more than one language. Something like "name:en" would be the English version. In countries with multiple official languages like Hong Kong, addresses almost always have per-language tags. We use these whenever possible.
- **Non-standard polygons:** like boroughs, counties, districts, neighborhoods, etc. which may be occasionally seen in addresses


- **Component dropout:** we usually produce 2–3 different versions of the address with various components removed at random. This way the model also has to learn to parse simple "city, state" queries alongside venue addresses, so it won't get overconfident e.g. that the first token in an address is always a venue name.

Structured learning

In the structured learning, we typically use a linear model to predict the most likely tag for a particular word given some local and contextual attributes or features. What differentiates structured learning from other types of machine learning is that in structured learning, the model's prediction for the previous word can be used to predict the current word. In similar tasks like part-of-speech tagging or named entity recognition, we typically design "feature functions" which take the following parameters:

1. The entire sequence of words
2. The current index in that sequence
3. The predicted tags for the previous two words

The function then returns a set of features, usually binary, which might help predict the best tag for the given word.

The tag history is what makes sequence learning different from other types of machine learning. Without the tag history, we could come up with the features for each word (even if they use the surrounding words), and use something like logistic regression. In a sequence model, we can actually create features that use the predicted tag of the previous word.

Consider the use of the word "Brooklyn." In isolation, we could assume it to mean the city, but it could be many other things e.g. Brooklyn Avenue, The Brooklyn Museum, etc. If we see "Brooklyn" and the last tag was HOUSE_NUMBER, it's very likely to mean Brooklyn the street name. Similarly, if the last tag was HOUSE (our label for place/building name), it's likely that we're inside a venue name e.g. "The Brooklyn Museum."


helpful in a case like “Brooklyn Avenue” where knowing that the next word is “Avenue” may disambiguate words used out of their normal context, or help determine that a rare word is a street name. In a French address, knowing that the previous word was “Avenue” is equally helpful as in “Avenue des Champs-Élysées.”

Training the model for multiple languages entails a few more ambiguities. Take the word “de.” In Spanish it’s a preposition. If we’re lowercasing the training data on the way in, it could also be an abbreviation for Delaware (“wilmington de”) or Deutschland (“berlin de”). Again, knowing the contextual words/tags is quite helpful.

In libpostal, we make heavy use of the multilingual address dictionaries used above in normalization as well as place name dictionaries (aka gazetteers) compiled from GeoNames and OSM. We group known multiword phrases together so e.g. “New York City” will be treated as a single token. For each phrase, we store the set of tags it might refer to (“New York” can be a city or a state), and which one is most likely in the training data. Context features are still necessary though as many streets take their name from a proper place like “Pennsylvania Avenue,” “Calle Uruguay” or “Via Firenze.”

We also employ a common trick to capture patterns in numbers. Rather than consider each number as a separate word or token, we normalize all digits to an uppercase “D” (since we’re lowercasing, this doesn’t conflict with the letter “d”). This allows us to capture useful patterns in numbers and let them share statistical strength. Some examples might be “DDDDD” or “DDDDD-DDDD” which are most likely US postal codes. This way we don’t need many training examples of “90210” specifically, we just know it’s a five digit number. GeoNames contains a world postal code data set, which is also used to identify potential valid postal codes. Some countries like South Africa use 4-digit postal codes, which can be confused for house numbers, and the GeoNames postal codes help disambiguate.

The learning algorithm

We use the averaged perceptron popularized by Michael Collins at Columbia, which achieves close to state-of-the-art accuracy while being much faster to train than fancier models like conditional random fields. On smaller training sets, the additional

The basic perceptron algorithm uses a simple error-driven learning procedure, meaning if the current weights are predicting the correct answer, they aren't modified. If the guess is wrong, then for each feature, one is added to the weight of the correct class and one is subtracted from the weight of the predicted/wrong class. The learning is done online, one example at a time. Since the weight updates are very sparse and occur only when the model makes a mistake, training is very fast.

In the averaged perceptron, the final weights are then averaged across all the iterations. Without averaging it's possible for the basic perceptron to spend so much of its time altering the weights to accommodate the few examples it gets wrong that it produces an unreasonable set of weights that don't generalize well to new examples (a.k.a. overfitting). In this way, averaging has a similar effect to regularization in other linear models. As in stochastic gradient descent, the training examples are randomly shuffled before each pass, and we make several passes over the entire training set.

Though quite simple, this method is surprisingly competitive in part-of-speech tagging, the existing NLP task that's closest to address parsing, and has by far the best speed/accuracy ratio of the bunch.

Evaluation

In part-of-speech tagging, simple per-token accuracy is the most intuitive metric for evaluating taggers and is used in most of the literature. For address parsing, since we'll want to use the parse results downstream as fields in normalization and search, a single mistake changes the JSON we'll be constructing from the parse. Consider the following mistake:

```
Brooklyn/HOUSE Academy/HOUSE of/ROAD Music/HOUSE 30/HOUSE_NUMBER
Lafayette/ROAD Avenue/ROAD Brooklyn/CITY NY/STATE 11217/POSTCODE
```

In a full-text search engine like Elasticsearch, it might still work to search the name field with ["Brooklyn Academy", "Music"] plus the other fields and still get a correct result, but if we want to create a structured database from the parses or hash the fields and do a simple lookup, this parse is rendered essentially useless.

On held-out data (addresses not seen during training), the libpostal address parser currently gets **98.9% of full parses correct**. That's a single model across all the languages, variations, and field combinations we have in the OSM training set.

Future improvements

The astute reader will notice that there's still an open question here: how well does the synthesized training set approximate real geocoder input? While that's difficult to measure directly, most of the decisions in constructing the training set thus far have been made by examining patterns in real-world addresses extracted from the Common Crawl, as well as user queries contributed to the project by a production geocoder.

There's still room for improvement of course. Not every country is represented in the address formatting templates (though coverage continues to improve over time). Most notably, countries using the East Asian addressing system like China, Japan, and South Korea are difficult because the address format depends on which language/script is being used, necessitating some structural changes to the address-formatting repo. In OSM these addresses are not always split into components, possibly residing in the "addr:full" tag. However, since each language uses specific characters to delimit address components, it should be possible to parse the full addresses deterministically and use them as training examples.

The libpostal parser also doesn't yet support apartment/flat numbers as they're not included in most OSM addresses (or the address format templates for that matter). The parser typically labels them as part of the house number or street field. For geocoders, apartment numbers aren't likely to turn up much as people tend to search at the level of the house/building number, but they may be unavoidable in batch geocoding. Supporting them would be relatively straightforward either by adding apartment or floor numbers to some of the training examples at random (without regard to whether those apartments actually exist in a particular building or not), or by parsing the "addr:flats" key in OSM. The context phrases like "Apt." or "Flat" can be randomly sampled from any language in libpostal with a "unit_types" dictionary.

Conclusions

I'm hoping that libpostal will be the backbone for many great geocoders and apps in

2. Technology and stack independent

3. Based on open data sets and fully open source

International by design, not as an afterthought

Almost every geocoder bakes in various myopic assumptions e.g. that addresses are only in the US, English, Latin script, the Global North, the bourgeoisie, etc.

Fully embracing L10N/I18N (localization/internationalization) means that there is no excuse for excluding people based on the languages they speak or the countries in which they live. An extra degree of rigor is required in recognizing and eliminating our own cultural biases.

There are of course always constraints on time and attention, so libpostal prioritizes languages in a simple, hopefully democratic way. Languages are added in priority order by the number of world addresses they cover, approximated by OpenStreetMap.

Usable on any platform

Libpostal is written in C mostly for reasons of portability. Almost every conceivable programming language can call into C code. There are already libpostal bindings for Python and NodeJS, and it's quite easy to write bindings for other languages.

Informed completely by open data

Libpostal makes use of several great open data sets to construct training examples for the address parser and language classifier:

- OpenStreetMap is used extensively by libpostal to create millions of training examples of parsed addresses and language classifications.
- GeoNames is used by the address parser as a place name and postal code gazetteer, and will also be used for geographic name disambiguation in an upcoming release.
- Quattroshapes and Zetashapes polygons are used in various places to add additional administrative and local boundary names to the parser training set. Zetashapes neighborhood polygons were particularly useful since neighborhoods are simple points in OSM.


The beauty of using these living, open, collaboratively edited data sets is that the models in libpostal can be updated and improved as the data sets improve. It also provides a great incentive for users of the library to support and contribute to open data.

Fin

You made it! The only thing left to do, if you haven't already, is check out libpostal on Github: https://github.com/openvenues/libpostal.

If you want to contribute and help improve libpostal, you don't have to know C, or any programming language at all for that matter.

For non-technical folks, the easiest way to contribute is to check out our language dictionaries, which are simple text files that contain all the abbreviations and phrases libpostal recognizes. They affect both normalization and the parser.

Find any language you speak (or add a directory if it's not listed) and edit away. Your work will automatically be incorporated into the next build.

Libpostal is already scheduled to be incorporated into at least 3 geocoding applications written in as many languages. If you're using it or considering it for your project/company, let us know.

Happy geocoding!