Proceedings of the 12th Conference on Language Resources and Evaluation (LREC 2020), pages 7060–7068
Marseille, 11–16 May 2020

Data Query Language and Corpus Tools for Slot-Filling and Intent Classification Data

Stefan Larson, Eric Guldan, Kevin Leach
Clinc, Inc.
Ann Arbor, Michigan, USA
stefan@clinc.com

Abstract

Typical machine learning approaches to developing task-oriented dialog systems require the collection and management of large amounts of training data, especially for the tasks of intent classification and slot-filling. Managing this data can be cumbersome without dedicated tools to help the dialog system designer understand the nature of the data. This paper presents a toolkit for analyzing slot-filling and intent classification corpora. We present a toolkit that includes (1) a new lightweight and readable data and file format for intent classification and slot-filling corpora, (2) a new query language for searching intent classification and slot-filling corpora, and (3) tools for understanding the structure and makeup for such corpora. We apply our toolkit to several well-known NLU datasets, and demonstrate that our toolkit can be used to uncover interesting and surprising insights. By releasing our toolkit to the research community, we hope to enable others to develop more robust and intelligent slot-filling and intent classification models.

Keywords: corpus, conversational systems/dialogue/chatbots/human-robot interaction, tools

1. Introduction

Contemporary task-oriented dialog systems rely on natural language understanding (NLU) models to classify a user's utterance into an intent class (e.g., *"what is my savings account balance"* → `get_balance`) as well as to extract important entities from the utterance (e.g., *"savings account"* → `BANK_ACCOUNT`). While the choice of particular NLU models for the tasks of intent classification and slot-filling (also called slot or entity extraction) can depend on factors such as the task domain, deployment environment, and computational constraints, one thing that all machine learning-driven NLU models have in common is that they require large amounts of carefully annotated and curated training data. Moreover, developing, debugging, improving, and maintaining such models in production environments requires rapidly searching through training corpora to gain insights from the data. Such insights can help the dialog system designer spot gaps and errors in the training data, potentially improving system performance drastically. As a motivating example, suppose a system designer/maintainer seeks to train an NLU system composed of intent classification and slot-filling models. Given a dataset annotated in a style similar to that shown in Figure 1, there are many valuable insights that the designer/maintainer might wish to understand, including:

A. The distribution of slot values for a given slot type. If a particular slot type has only a few unique slot values in a training set, then a slot-filling model trained on this data will likely be overfit to those few values.

B. The composition and distribution of context words for a particular slot type. For example, in the training utterance

book me a flight from detroit (ORIGIN) to new york (TARGET)

the left and right context 1-grams for the `ORIGIN` slot ("detroit") are "from" and "to", respectively. Given a slot type in a corpus, low diversity of left and right context windows could cause slot-filling models to become overfit to certain contextual cues, which would negatively impact model performance.

C. Distributions and frequency counts of tokens for each intent class, where a low number of unique tokens for an intent class could indicate that the data for that particular intent class is not diverse.

1. find me a flight to Paris (TARGET) with Ryanair (AIRLINE) on the 4th (DEPART) → book_flight
2. what time is DL582 (FLIGHT) scheduled to land → flight_status
3. i 'm looking to leave tuesday (DEPART) and fly back the fifteenth (RETURN) → book_flight
4. what do i do if i think my luggage was lost → lost_luggage
5. book a flight from detroit (ORIGIN) to new york (TARGET) please → book_flight
6. hello there can you check if DL123 (FLIGHT) is delayed → greeting, flight_status
7. find me airfare from ORD (ORIGIN) to LGA (TARGET) with United (AIRLINE) → book_flight
8. check what time UA682 (FLIGHT) leaves for newark (TARGET) → flight_status
9. missing bags at the carousel, my flight was DL188 (FLIGHT) → lost_luggage

Figure 1: Samples from a corpus of annotated slot-filling and intent classification data used for training task-oriented dialog systems. Any given training sample can contain a number of annotated slots. Training samples can also have membership to a single or multiple intent classes (e.g. in the case of multi-label intent classification). This paper presents a toolkit designed to enable the designer of task-oriented dialog systems a way to gain insights from such data.

This paper introduces a toolkit designed to aid in data curation tasks for NLU data (including the insight tasks listed

above). Our toolkit consists of several key features:

1. A data annotation format designed to be lightweight and human readable. Our data format enables annotation for intent and slot/entity data, all on a single line.

2. A query language to efficiently search and retrieve subsets of an NLU dataset. Our query language is modeled after our data format, allowing the user of the toolkit to formulate queries using a familiar syntax.

3. A suite of data insight tools designed to aid the developer in understanding the nature of their dataset.

We demonstrate the utility of our toolkit by performing several analyses on several existing intent classification and slot-filling datasets. Using our toolkit, we uncover surprising patterns and errors in several widely-used NLU datasets, indicating the need for a careful reappraisal as to how NLU datasets are collected and curated.

2. Data Query Language

This section introduces our query language, which is specifically tailored to NLU corpora.

2.1. Data Format

Our query language mimics the file format we use for NLU corpora, so this format is worth introducing here. In our file format, each annotated sample is represented by a single line. Labeled slots are denoted using the special characters { and }, and slot labels are specified using the first token inside the {·} expression. For example, the annotated sample

```
please transfer { AMOUNT 40 bucks } -> transfer
```

indicates that the tokens "40 bucks" are labeled as the `AMOUNT` slot. The `IOB` labels are implied by this format: the first token after the slot label in each {·} expression corresponds to the `B` (begin) label, while everything after (but still within the {·} expression) corresponds to the `I` (inside) label. All other tokens are implied to be labeled with `O` (outside). We denote intent classification annotations with the `->` symbol. Intent annotations can take the form of single- or multi-class labels. For example the annotated sentence

```
hello what is my balance -> greeting, balance
```

has the multi-class intent label of both the `balance` and `greeting` intents.

Figure 2 shows several example annotated sentences in our data format. Compare the brevity of this format against sentences annotated in the MIT `IOB` (Liu et al., 2013) and Snips (Coucke et al., 2018) formats, shown in Figures 3 and 4, respectively. The MIT `IOB` format (Figure 3) requires one line per token, and does not provide intent information. The Snips format (Figure 4) uses a JSON structure that requires one line per label span. In contrast, our format allows for the annotation of one sample per line.

2.2. Query Language

Our toolkit includes a query language to aid in the management of slot-filling corpora. The query language allows the user to query an annotated NLU corpus using a syntax that closely resembles the data format introduced in Section 2.1. The query language allows the user to search for token strings and sequences, slot types, slot-value pairs, and slot sequences. Searching for combinations of these query types is also possible using boolean expressions. Table 1 displays several example queries using our query language. Our query language is regular: we use bottom-up parsing on input query strings to build a parse tree consisting of filtering criteria, which are successively applied to gather a list of samples meeting the specified criteria (i.e. syntax-directed translation).

We built a standalone parser in Python for this query language. The language can be thought of as a layer atop a regular expression engine, though with some extensions (e.g., logical operators). Moreover, as we traverse the parse tree, we interface with an efficient representation of our corpus data using inverted indices, which enables rapid data filtering and, in turn, a convenient tool for developing corpus insights and debugging data issues. (Note that the parse tree can be consumed by other applications and translated into other query languages, such as SQL.) We describe the syntactic features of our query language in this subsection.

2.2.1. Slot Query Syntax

To search for occurrences of labeled slots, the {·} syntax introduced in Section 2.1. can be used. For instance, the query

```
{ AMOUNT "40 dollars" }
```

would return all data samples where the token span "40 dollars" is labeled as the `AMOUNT` slot.

Wildcards and regular expressions can be used in lieu of specific slot values. For example,

```
{ AMOUNT "* dollars" }
```

searches for all samples where an `AMOUNT` slot fits the pattern of any token(s) followed by "dollars". Our query language also supports regular expressions. For instance, the query

```
{ AMOUNT r'checkings?|savings?' }
```

searches for all samples where the `AMOUNT` slot is "checking", "checkings", "saving", or "savings".

The wildcard can also be used to search for *any* occurrence of a particular slot, no matter the annotated tokens. An example of this is

```
{ AMOUNT * }.
```

Boolean expressions can also be constructed within slot queries using the `not`, `and`, and `or` operators. For example, we could use the `or` operator and query

```
{ TARGET "checking" or "savings" }
```

in order to search for cases where the `TARGET` slot is either "checking" *or* "savings",

```
hi there what is my balance -> greeting, balance
please book me a spot for { PARTY_SIZE two } at { RESTAURANT City Tavern } -> book_restaurant
are there any { RESTAURANT arbys } { LOCATION with in ten miles of here } -> find_restaurant
```

Figure 2: Our data format for representing annotated NLU sentences. Our format allows for one line per annotated sentence.

```
O                    please
O                    book
O                    me
O                    a
O                    spot
O                    for
B-Party_Size         two
O                    at
B-Restaurant_Name    City
I-Restaurant_Name    Tavern
```

Figure 3: Example annotated NLU sentence in the MIT IOB format (Liu et al., 2013). This format requires one line per token, and does not allow for intent classification labels.

```
{
  "data": [
    {
      "text": "please book me a
         spot for"
    },
    {
      "text": "two",
      "entity": "party_size"
    },
    {
      "text": "at "
    },
    {
      "text": "City Tavern",
      "entity": "restaurant_name"
    }
  ]
}
```

Figure 4: Example annotated NLU sentence using the Snips file format (Coucke et al., 2018). This format uses one line per annotated token span.

2.2.2. String Query Syntax

Strings, or sequences of tokens, can be searched using quote marks. For instance,

```
"make a transfer"
```

searches for samples that contain that exact sequence of tokens.

Slot queries can be used within string queries. The query

```
"transfer { AMOUNT "40 dollars" } now"
```

specifies to search for that exact sequence of four tokens and specifies that the span "40 dollars" must be labeled as an AMOUNT slot. Boolean expressions can be used in string queries as well, for instance

```
not "balance" and "money"
```

retrieves all samples that contain the token "money" but not "balance".

2.2.3. Combination Query Syntax

Searching for combinations of slots and strings can be accomplished by joining the two query types using boolean expressions. For instance, the query

```
not "transfer to" and { AMOUNT * }
```

would return all samples without the string "transfer to" but also containing an AMOUNT slot.

2.2.4. Sequence Query Syntax

Wildcards can be employed to mine a slot-filling corpus for all samples that manifest a specific pattern of slots. Suppose we were interested in finding all samples where AMOUNT, TARGET, and SOURCE slots occurred in that specific order. Then we could query

```
"* { AMOUNT * } * { TARGET * } * { SOURCE * } *".
```

2.2.5. Number Query Syntax

To retrieve all samples with a specific number of slots, one can use the <, >, and = operators. For instance, to query for all samples with more than one AMOUNT slot, the user can query

```
{ AMOUNT * } > 1.
```

Wildcards can also be used for slot names. To retrieve samples that contain *any* slot, but less than 2 slots, one can query

```
{ * * } < 2.
```

Strings can also be used along with the <, >, and = operators. For example,

```
"this" = 2
```

would search for all samples containing two occurrences of the token "this".

2.2.6. Similarity Search

The toolkit provides an interface for users to define custom similarity measures for string searches. Once a user defines a similarity function that operates on two strings, the similarity function is applied when the $\sim$ operator is used prior to a string. For example,

```
{ AMOUNT ~"40 dollars" }
```

searches for all data samples that have an AMOUNT slot similar to "40 dollars" (as defined by the similarity function).

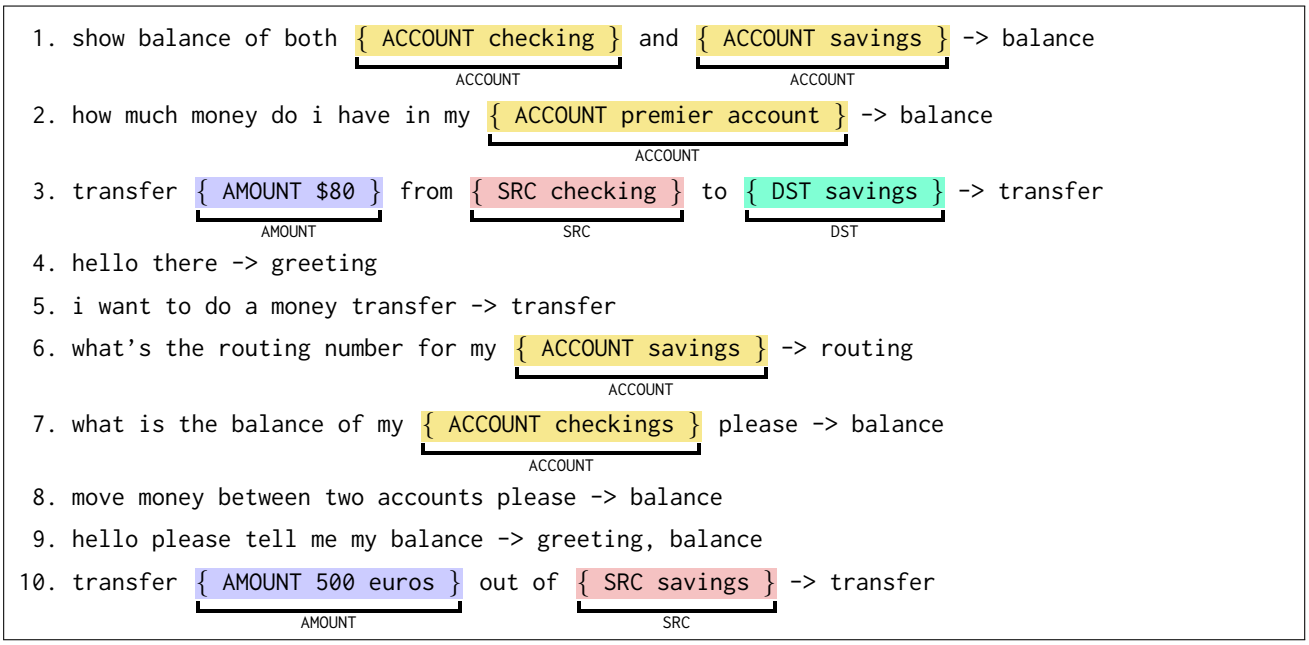


Figure 5: Example annotated NLU corpus represented by our data format. The colored annotations have been added to clarify syntactic elements of our format.

Query	Retrieved	Description of Retrieval Criteria
`{ ACCOUNT "savings" }`	1, 6	samples with an `ACCOUNT` slot with value "savings"
`{ ACCOUNT * } > 1`	1	samples that have more than one `ACCOUNT` slot
`{ * * } = 0`	4, 5, 8, 9	samples with no slots
`"from { SRC * } to { DST * }" -> transfer`	3	samples in the `transfer` intent with a specified slot pattern
`"* { DST * } * { SRC * } * " -> transfer`	Ø	samples in the `transfer` intent with a `DST` followed by a `SRC` slot
`{ AMOUNT "500 euros" or "$80" }`	3, 10	samples with an `AMOUNT` whose value is "500 euros" or "$80"
`"*" -> balance and greeting`	9	samples that belong to both `balance` and `greeting` intent classes
`{ AMOUNT * } and not "transfer" -> transfer`	Ø	samples in the `transfer` intent containing an `AMOUNT` slot, but not containing the string "transfer"
`"hello" -> greeting sortby:length`	9, 4	samples belonging to the `greeting` intent class and containing the string "hello", sorted by length of sample

Table 1: Example queries showing the versatility of our data query language. The Query column indicates the syntax to specify a query, whose behavior is summarized in the Description column. The Retrieved column refers to which samples would be returned by that Query from the dataset shown in Figure 5.

2.2.7. Intent Search

If a corpus contains multiple intents, or contains samples belonging to multiple intent classes, then it is natural for the user to want to select from a subset of the intent classes. The query language provides a way to do this using the `->` operator, which mimics how intents are specified in the data format introduced in Section 2.1. To retrieve all samples belonging to the `transfer` intent, for instance, the user can query

```
* -> transfer.
```

Boolean operators can be used on the right side of the `->` operator, for instance

```
* -> balance and transfer
```

would return all multi-intent samples that belong to both the `balance` and `transfer` intents.

2.2.8. Sorting Results

Retrieved samples can be sorted by appending the search query with the `sortby` command. As an example, the query

```
* -> greeting sortby:length
```

would return all samples belonging to the `greeting` intent sorted by length. The sorting functionality supports several out-of-the-box sorting functions, including sorting alphabetically, sorting by length, and sorting by the number of slots. Users can define additional sorting functions as well.

3. Corpus Tools

Our toolkit provides several tools for analyzing NLU corpora. Included are data summarization measures (discussed in Section 3.1.) as well as tools for pattern mining (discussed in Section 3.2.).

Dataset	Num. Samples	Num. Intents	Num. Slots	Vocab Size	TTR	Diversity	Coverage
ATIS (Hemphill et al., 1990)	5,871	18	83	950	0.297	0.880	0.276
MIT Movie (Liu et al., 2013)	12,218	—	12	7,481	0.060	0.976	0.325
MIT Restaurant (Liu et al., 2013)	9,181	—	8	4,166	0.049	0.979	0.350
Snips (Coucke et al., 2018)	14,484	7	39	12,027	0.136	0.939	0.373
AskUbuntu (Braun et al., 2017)	162	5	3	471	0.486	0.961	—
WebApplications (Braun et al., 2017)	89	8	3	294	0.658	0.949	—
Chatbot (Braun et al., 2017)	206	2	7	184	0.134	0.837	—
Liu et al. (2019)	25,716	68	56	7,955	0.178	0.948	—
Larson et al. (2019)	23,700	150	—	8,376	0.149	0.914	0.361

Table 2: Summary statistics on several NLU benchmark datasets included in our toolkit. Coverage is computed on the datasets with train-test splits. We set $N = 3$ in our computations of Diversity and Coverage (see Section 3.1.).

3.1. Summary Statistics

Functions for computing summary statistics of several facets of NLU corpora are provided in the toolkit. These summary statistics are centered around both intent and slot-filling data, and include simple measures like average sentence length and slot frequency counts.

The toolkit also supports several more advanced measures, such as type-token ratio (ttr), which is defined as the number of unique token types divided by the number of tokens in a document (Templin, 1957). This metric can be applied to a corpus of data by averaging the type-token ratios for each intent class (treating each intent class as a single document by joining together all samples). That is, we define the type-token ratio (TTR) for an NLU corpus C to be

$$\text{TTR}(C) = \frac{1}{|C|}\sum_{i=1}^{|C|}\text{ttr}(X_i)$$

where $|C|$ is the number of classes (i.e. number of intents) that partition the corpus, and X_i is a class of the corpus. The toolkit also provides functions for computing the *diversity* of a corpus, defined as

$$\text{Diversity}(X) = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|X_i|^2}\left[\sum_{a\in X_i}\sum_{b\in X_i} D(a,b)\right]$$

where a and b are samples from the corpus and

$$D(a,b) = 1 - \frac{1}{N}\sum_{n=1}^{N}\frac{|n\text{-grams}_a \cap n\text{-grams}_b|}{|n\text{-grams}_a \cup n\text{-grams}_b|}$$

where N is a positive integer and n-grams$_a$ is the set of n-grams in sample a. We also include a metric for measuring how much overlap there is between two datasets X and Y, called coverage, which is defined as

$$\text{Coverage}(X,Y) = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{|Y_i|}\sum_{b\in Y_i}\max_{a\in X_i}(1 - D(a,b)).$$

Both Diversity and Coverage metrics were introduced in Kang et al. (2018) and are useful for measuring the spread and overlap of datasets, and in particular can be used to measure how well a training set "covers" a test set. Table 2 lists summary statistics measured on several publicly available NLU corpora.

3.2. Pattern Mining Tools

We include several pattern mining tools that offer ways of gaining insights from corpora beyond what the summary statistics provide. These pattern mining tools include tools for analyzing the relative weight of tokens in a dataset (e.g. using *tf-idf*, as well as a slot context analysis tool aimed at providing insights into the structure of texts in a corpus with respect to slots.

As a concrete example, consider the following text:

book me flights from Paris (ORIGIN) to New York (TARGET).

The context neighborhood of a slot is simply the left and right n-grams surrounding the slot. In the example above, the left 2-gram for the `TO_LOC` slot is "flights from", and the right 1-gram is "to".

Knowledge of the frequency of context neighborhoods of a particular slot can shed light on the robustness of a corpus. For instance, if the context neighborhoods of a given slot are limited to very few sets of left- and right n-grams, then models trained on the corpus are likely to be overfit to those patterns. Examples of using these pattern mining tools are discussed in Section 5.

4. Technical Details

Both the Data Query Language and Corpus Tools are implemented as a package in Python. Both tools support Python 2 and Python 3. The tools are indexed in PyPI and can be installed via `pip` by executing the following:

```
pip install nlu-dql
```

or by cloning the repository at `www.github.com/clinc/nlu-dql`.

The tool suites come with several pre-existing datasets out-of-the-box, including ATIS, Snips (Coucke et al., 2018), and those from Liu et al. (2013), Braun et al. (2017), Liu et al. (2019), and Larson et al. (2019). These datasets are summarized in Table 2. Users can load a pre-existing dataset using the `Corpus` class constructor:

```
atis = Corpus('atis')
```

The `Corpus` class provides an interface for filtering samples within by using our data query language. Alternatively,

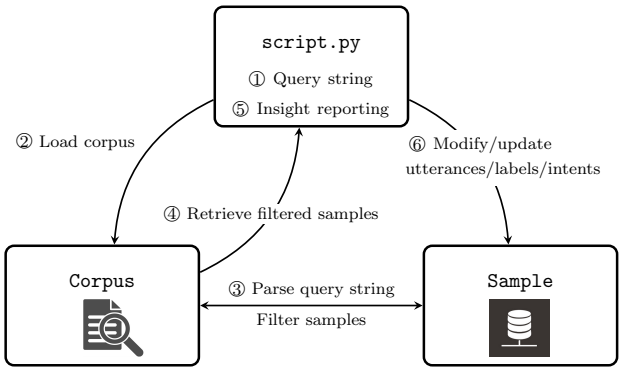


Figure 6: Workflow for our toolkit. A developer writes a script that imports our library, where they develop a Query Language string (annotation ①). This script uses our `Corpus` class to load a corpus of utterances following our data format in ②. As the `Corpus` object is created, it parses the query string and filters the samples in that corpus (③). The `Corpus` object then exposes a list of filtered samples (④) from which the developer can build insights about samples in that query (⑤). Finally, the `Sample` class exposes an interface for modifying utterances, slot information, and intent classes directly (⑥).

these datasets can be installed independently of the Data Query Language and Corpus Tools by executing

```
pip install nlu-data
```

or by cloning the repository at `www.github.com/clinc/nlu-data`.

4.1. System Architecture

Figure 6 illustrates a standard workflow for using our corpus tools. In particular, we provide `Corpus` and `Sample` classes that expose interfaces for loading corpora, specifying input queries following our query format (Section 2.), and modifying token, intent, and slot information directly. Using these corpus tools, developers can readily develop insights from and debug issues with large corpora of NLU data.

5. Example Use Cases

In this section we demonstrate both our data query language and data insights tools on a suite of commonly used NLU datasets. We demonstrate the utility of our tools by uncovering annotation errors, analyzing common utterance structure, and performing token frequency analysis.

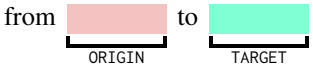
5.1. Analyzing Slot Context

The Airline Transportation Information System (ATIS) (Hemphill et al., 1990) dataset[1] has been used as a benchmark for the task of slot-filling for almost 3 decades. The ATIS dataset contains utterances targeted at a flight-booking intelligent digital assistant. Example slot types from this dataset include origin and destination airports. As of writing, slot-filling performance on the ATIS dataset has been dominated by deep learning models such as RNNs, LSTMs, and transformers, with state-of-the-art models achieving F_1 scores in the mid-to-high 90s (Qin et al., 2019).

While there has been some analysis on the structure and composition of ATIS itself, such analyses have been limited to categorizing output errors (Tur et al., 2010; Béchet and Raymond, 2018). Here we provide a short program to analyze the structure of two ATIS slot types: the `FROMLOC.CITY_NAME` and `TOLOC.CITY_NAME` slots.

```
atis = Corpus('atis-train')
# Find how many samples contain both
# TOLOC.city_name and FROMLOC.city_name
q = '{ FROMLOC.CITY_NAME * } and' \
    '{ TOLOC.CITY_NAME * }'
print(len(atis.query(q)))
# Find how many samples have the pattern
# "from FROMLOC.city_name to
# TOLOC.city_name"
q = '"from { FROMLOC.CITY_NAME * }' \
    ' to { TOLOC.CITY_NAME * }"'
print(len(atis.query(q)))
```

In this program snippet, we first find all the samples in ATIS that contain both `FROMLOC.CITY_NAME` and `TOLOC.CITY_NAME` slots. This number turns out to be 3,621. Then, we retrieve all samples that contain the pattern:



This number is 2,922, which is 80.7% of all the samples that contain `TOLOC.CITY_NAME` and `FORMLOC.CITY_NAME`. Such a high percentage indicates that ATIS contains a large amount of common patterns that are easily learned by modern slot-filling models.

Similarly, one can use our toolkit's slot context analysis tool to see the distribution of context words for a given slot. The short program below performs context analysis on the `FROMLOC.CITY_NAME` and `TOLOC.CITY_NAME` slots from the ATIS corpus.

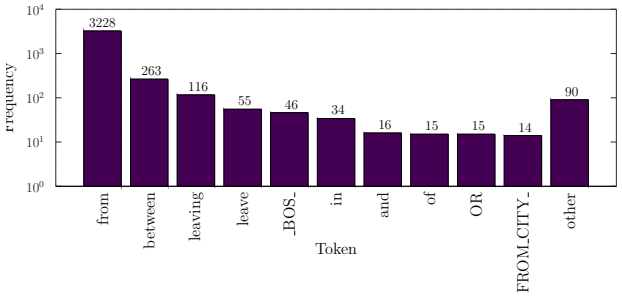
```
atis = Corpus('atis-train')
dist = atis.context('FROMLOC.CITY_NAME')
print(dist)
dist = atis.context('TOLOC.CITY_NAME')
print(dist)
```

The distributions of context words for these two slots are shown in Figure 7 (overleaf). Again, we see that an overwhelming portion of context words for these two slots are either "to" or "from", which could mean one reason that models do so well on the ATIS dataset is because the there are relatively few contextual cues for certain slots.

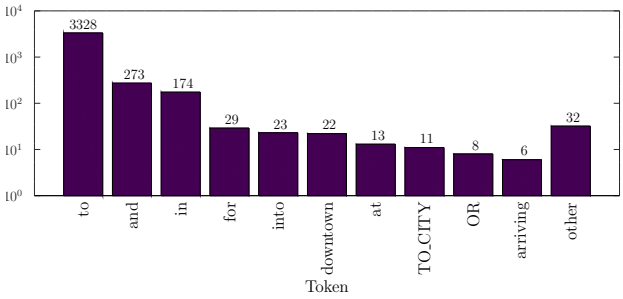
5.2. Slot Value Analysis

The previous examples investigated the *context* of particular slots, but suppose the user instead wanted to look at distributions of slot *values* instead. Here we present a Python snippet that accomplishes this on the `FROMLOC.CITY_NAME` slot from the ATIS (train) dataset:

[1]Here we investigate the ATIS configuration as used by Tur et al. (2010).



(a) ATIS FROMLOC.CITY_NAME query response.



(b) ATIS TOLOC.CITY_NAME query response.

Figure 7: Distribution of left 1-grams for the FROMLOC.CITY_NAME (Subfigure 7a) and TOLOC.CITY_NAME (Subfigure 7b) slots for the ATIS dataset, as produced by our toolkit. Note that the data is plotted on a logarithmic y-axis. We observe that an overwhelming majority of left 1-grams are "from" and "to", which can cause slot-filling models to overfit to these patterns.

```
atis = Corpus('atis-train')
slot = 'FROMLOC.CITY_NAME'
d = atis.slot_value_distribution(slot)
print(d)
```

We present the output of this snippet as a histogram in Figure 8, in which we observe that over half of the dataset's FROMLOC.CITY_NAME slot occurrences are due to just five values. Given that the space of possible city names (even those with airports) is very large, this finding helps indicate to the system developer that more slot value diversity might be necessary in order to develop a more robust slot-filling model.

5.3. Finding Annotation Errors

High quality training data is important in developing robust NLU models, yet annotation errors and inconsistencies can and do often arise, especially when using multiple annotators (e.g. in the case of crowdsourcing). In this section, we introduce a way of finding annotation inconsistencies for annotated slot-filling corpora.

The following code snippet analyzes the GENRE slot in the MIT Movie (Liu et al., 2013) corpus. In this code snippet, we first search for all samples where there is a GENRE slot followed by the token "movie".

```
mit_movie = Corpus('MIT_Movie')
q = '"{ GENRE * } movie"'
print(mit_movie.query(q))
```

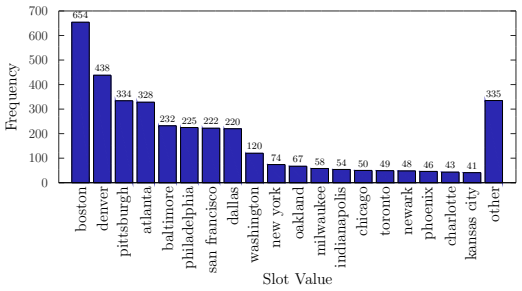


Figure 8: Distribution of slot values for the FROMLOC.CITY_NAME slot in the ATIS dataset. We can see that slot values tend to be biased by very large or populous cities, which may lead to overfitting in models trained on such data. Our toolkit enables finding such insights efficiently.

```
q = '{ GENRE "* movie" }'
print(mit_movie.query(q))
```

In the second search, we find all samples that have the "movie" token inside of the annotated GENRE slot. It turns out that there are 1,159 of the former and 22 of the latter case, indicating that these are not labeled consistently. The MIT Movie corpus is not the only dataset with annotation errors: we can do a similar analysis on the AskUbuntu corpus from Braun et al. (2017) on the UBUNTUVERSION slot:

```
askubuntu = Corpus('AskUbuntu')
q = '~"ubuntu { UBUNTUVERSION * }"'
print(askubuntu.query(q))
q = '{ UBUNTUVERSION ~"ubuntu *" }'
print(askubuntu.query(q))
```

Here we are using the $\sim$ similarity operator to find tokens similar to "ubuntu". Using this code snippet, we find that there are 5 cases where "ubuntu" or similar (e.g. "lubuntu" and "xubuntu") are part of the UBUNTUVERSION slot and 20 cases where they are not, indicating annotation errors.

5.4. Fixing Annotation Errors

Suppose we wanted to programmatically fix the annotation mistakes from Section 5.3. We can use the sub-corpus returned from the query on the AskUbuntu dataset seen earlier to find all the samples that contain the annotation error. Then we can iterate through these erroneous samples and simply re-label the mis-labeled "ubuntu" token as a UBUNTUVERSION slot, taking care to ensure that what follows the "ubuntu" token remains part of the original UBUNTUVERSION slot. The following Python code accomplishes just that:

```
askubuntu = Corpus('AskUbuntu')
q = ' "ubuntu { UBUNTUVERSION * }"'
samples = askubuntu.query(q)
slot = 'ubuntuversion'
for s in samples:
  annos = s.annotated_tokens
  n = len(annos)
  for i, a in enumerate(annos):
    if 'ubuntu' in a.token and i+1<n:
      if annos[i+1].label=='b-'+slot:
```

```
        a.set_label('b',slot)
        annos[i+1].set_label('i',slot)
  s.refresh()
askubuntu.refresh()
```

One could programmatically fix multiple annotation mistakes by wrapping the above code into a function, and running it for multiple token-slot combinations.

5.5. Finding Key Tokens in Intent Classification Data

Thus far the example use cases have been tailored toward slot-filling applications. Here we give an example of finding important tokens in intent classification data by inspecting the *tf-idf* weights of the Snips dataset. This code snippet is shown below:

```
snips = Corpus('snips')
# compute tf-idf weights for tokens in
# the getweather intent in the
# snips corpus.
weights = snips.tfidf('getweather')
print(weights)
# see if the token "forecast" is in any
# intent besides the getweather intent.
q = '"forecast" -> not getweather"'
print(len(snips.query(q)))
```

In this snippet, we inspect the *tf-idf* weights for the `getweather` intent. The top 15 weights are displayed in Figure 9, where we clearly see that the tokens "forecast" and "weather" are dominant, with "forecast" holding slightly more importance. The snippet then checks to see if "forecast" appears in any intent other than `getweather` using the query language. It turns out that "forecast" is exclusive to `getweather`.

This type of insight—that is, finding key tokens—can help find tokens on which an intent classification model might be prone to overfitting. Moreover, knowledge of key tokens can help inform and guide data collection and curation for future datasets and model development.

6. Related Work

The development of query languages and search tools for linguistic corpora has a rich history. Prior work include GUI tools and languages for querying linguistically annotated tree structures (Steiner and Kallmeyer, 2002; Resnik and Elkiss, 2005; Zeldes et al., 2009; Augustinus et al., 2012; Singh, 2012; Krause and Zeldes, 2016; Bladier et al., 2018), but these languages and tools are tailored to general annotation schemes (e.g. parse trees), rather than NLU data.

Tools that are more closely aligned with areas that are similar to NLU corpora include GUI tools for searching FrameNet constructions (Sato, 2012) and Wikipedia entities (Klang and Nugues, 2018). The Corpus Query Language[2] is similar to our query language in that it allows for word and contextual search, but its goal is in the retrieval of words and not whole documents (i.e. annotated training samples) like our retrieval system. Other general tools for searching and managing annotated corpora include the Corpus Query Processor (Christ, 1994; Evert and Hardie, 2011), GATE (Cunningham, 2002) and Nothman et al. (2014). However, none of the prior work discussed here are specifically tailored to NLU corpora, nor offer ways of extracting analyses from data.

[2] `https://www.sketchengine.eu/documentation/corpus-querying/`

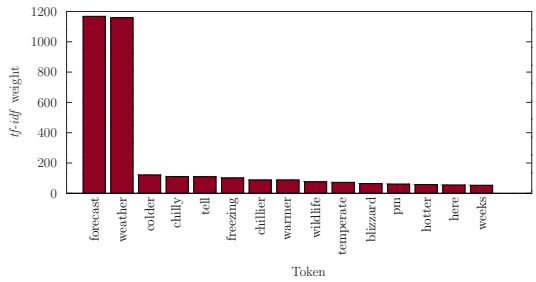


Figure 9: Distribution of *tf-idf* weight values for the Snips `getweather` intent.

7. Conclusion

In this paper we introduced tools to aid in the data exploration and management work of curating datasets for the tasks of slot-filling and intent classification. We showed that our data query language can be used to uncover annotation errors as well as over-abundant patterns in slot-filling data. By releasing our tools to the public, we hope that our tools can be used by others to speed up the development process of making quality, robust, useable, and intelligent task-oriented dialog systems.

Acknowledgements

This work benefited greatly from the help and ideas from—and discussions with—Adrian Cheung, Connor Witt, Jonathan Kummerfeld, Cooper Lower, Yiping Kang, Alex Shye, Parker Hill, and Thomas Wenisch. We also thank the anonymous reviewers for their insightful and well-written feedback.

8. Bibliographical References

Augustinus, L., Vandeghinste, V., and Eynde, F. V. (2012). Example-based treebank querying. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC)*.

Béchet, F. and Raymond, C. (2018). Is ATIS too shallow to go deeper for benchmarking spoken language understanding models? In *Proceedings of InterSpeech 2018*.

Bladier, T., Seyffarth, E., Hellwig, O., and Petersen, W. (2018). AET: Web-based Adjective Exploration Tool for German. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC)*.

Braun, D., Hernandez-Mendez, A., Matthes, F., and Langen, M. (2017). Evaluating natural language understanding services for conversational question answering systems. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue (SIGDIAL)*.

Christ, O. (1994). A modular and flexible architecture for an integrated corpus query system. In *Proceedings of the 3rd Conference on Computational Lexicography and Text Research (COMPLEX)*.

Coucke, A., Saade, A., Ball, A., Bluche, T., Caulier, A., Leroy, D., Doumouro, C., Gisselbrecht, T., Caltagirone, F., Lavril, T., Primet, M., and Dureau, J. (2018). Snips voice platform: An embedded spoken language understanding system for private-by-design voice interfaces. *CoRR*, abs/1805.10190.

Cunningham, H. (2002). GATE, a general architecture for text engineering. *Computers and the Humanities*, 36(2):223–254, May.

Evert, S. and Hardie, A. (2011). Twenty-first century corpus workbench: Updating a query architecture for the new millennium. In *Proceedings of the 2011 Corpus Linguistics Conference*.

Hemphill, C. T., Godfrey, J. J., and Doddington, G. R. (1990). The ATIS spoken language systems pilot corpus. In *Speech and Natural Language: Proceedings of a Workshop*.

Kang, Y., Zhang, Y., Kummerfeld, J. K., Hill, P., Hauswald, J., Laurenzano, M. A., Tang, L., and Mars, J. (2018). Data collection for dialogue system: A startup perspective. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

Klang, M. and Nugues, P. (2018). Linking, Searching, and Visualizing Entities in Wikipedia. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC)*.

Krause, T. and Zeldes, A. (2016). ANNIS3: A new architecture for generic corpus query and visualization. *Literary and Linguistic Computing*, 31:118–139, 03.

Larson, S., Mahendran, A., Peper, J. J., Clarke, C., Lee, A., Hill, P., Kummerfeld, J. K., Leach, K., Laurenzano, M. A., Tang, L., and Mars, J. (2019). An evaluation dataset for intent classification and out-of-scope prediction. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Liu, J., Pasupat, P., Cyphers, S., and Glass, J. (2013). Asgard: A portable architecture for multilingual dialogue systems. In *Proceedings of IEEE Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

Liu, X., Eshghi, A., Swietojanski, P., and Rieser, V. (2019). Benchmarking natural language understanding services for building conversational agents. In *Proceedings of the Tenth International Workshop on Spoken Dialogue Systems Technology (IWSDS)*.

Nothman, J., Dawborn, T., and Curran, J. R. (2014). Command-line utilities for managing and exploring annotated corpora. In *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*.

Qin, L., Che, W., Li, Y., Wen, H., and Liu, T. (2019). A stack-propagation framework with token-level intent detection for spoken language understanding. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Resnik, P. and Elkiss, A. (2005). The linguist's search engine: An overview. In *Proceedings of the ACL 2005 on Interactive Poster and Demonstration Sessions*.

Sato, H. (2012). A search tool for framenet constructicon. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC)*.

Singh, A. K. (2012). A concise query language with search and transform operations for corpora with multiple levels of annotation. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC)*.

Steiner, I. and Kallmeyer, L. (2002). VIQTORYA – a visual query tool for syntactically annotated corpora. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC)*.

Templin, M. C. (1957). *Certain Language Skills in Children: Their Development and Interrelationships*, volume 26. University of Minnesota Press.

Tur, G., Hakkani-Tür, D., and Heck, L. (2010). What is left to be understood in ATIS? In *Proceedings of Spoken Language Technology Workshop (SLT)*.

Zeldes, A., Lüdeling, A., Ritz, J., and Chiarcos, C. (2009). ANNIS: a search tool for multi-layer annotated corpora. In *Proceedings of Corpus Linguistics*.