Querying and Serving N-gram Language Models with Python

Nitin Madnani
Laboratory for Computational Linguistics and Information Processing
Institute for Advanced Computer Studies
University of Maryland, College Park
nmadnani@umiacs.umd.edu

Abstract

Statistical n-gram language modeling is a very important technique in Natural Language Processing (NLP) and Computational Linguistics used to assess the fluency of an utterance in any given language. It is widely employed in several important NLP applications such as Machine Translation and Automatic Speech Recognition. However, the most commonly used toolkit (SRILM) to build such language models on a large scale is written entirely in C++ which presents a challenge to an NLP developer or researcher whose primary language of choice is Python. This article first provides a gentle introduction to statistical language modeling. It then describes how to build a native and efficient Python interface (using SWIG) to the SRILM toolkit such that language models can be queried and used directly in Python code. Finally, it also demonstrates an effective use case of this interface by showing how to leverage it to build a Python language model server. Such a server can prove to be extremely useful when the language model needs to be queried by multiple clients over a network: the language model must only be loaded into memory once by the server and can then satisfy multiple requests. This article includes only those listings of source code that are most salient. To conserve space, some are only presented in excerpted form. The complete set of full source code listings may be found in (Madnani, 2009), the source code manuscript accompanying this primary manuscript.

1 Introduction

In 1950, Alan Turing—one of the founding fathers of modern computer science—proposed a *gedankenexperiment* to delineate the limits of artificial intelligence (AI). A human judge carries on a natural language conversation with a human and a computer in text form. If the judge is unable to tell the difference between the human and the computer based on this conversation, the computer is said to have passed the **Turing test** (Turing, 1950). Even though this was simply proposed as a thought experiment, efforts have been made worldwide to try and pass this test in practice. So far, no machine has ever passed since the test was formulated.

This anecdote relates to a group of the most difficult and fascinating problems that the Artificial Intelligence (AI) community is working on—making a computer understand, process and analyze human language. During the latter half of the past century, several theories and techniques have been developed to search for solutions to the problems in this group. As a result, a field of research has emerged—Natural Language Processing (NLP). This article focuses on one of

the most useful and well-known NLP techniques known as **statistical language modeling**. This technique attempts to measure how likely any given piece of text is, in any given language, by building a probabilistic model of the language in question and assigning a probability value to the text using this model. The language models are usually estimated from very large collections of text in the chosen language and tend to be very large. Given the easy availability of such text today, n-gram language models are now used widely in NLP research. However, the toolkit that is most commonly used to construct these models is written in C++ and does not provide bindings in any other language. Given the rising popularity of dynamic languages as the primary language for building NLP applications and teaching NLP courses around the world—especially Python (Madnani, 2007; Madnani and Dorr, 2008; Bird et al., 2008)—the lack of such bindings represents a challenge. This article illustrates how to write a Python module that allows for efficiently querying such language models directly in Python code. In addition, it also describes how to build a Python language model server. Such a server can prove to be extremely useful when the language model needs to be queried used by multiple clients over a network: the language model must only be loaded into memory once by the server and can then satisfy multiple requests.

1.1 What is Language Modeling?

Before discussing the details of the Python implementations, it would be useful to understand the motivation behind language models and how they generally work. Let us assume that we are building a machine to pass the Turing test. In order for our machine to communicate in a natural language such as English, it must first be able to distinguish between good English and bad English. For example, it must be able to determine that the utterance "`He went to the store`" is syntactically correct whereas the utterance "`He go store`" is not. Of course, the information required cannot just be restricted to syntax. The machine must also be able to infer that the utterance "`The store went to him`", while syntactically correct, is not very likely to be uttered by an English speaker. Note that I used the words *not very likely*. This is important because even though I may never formulate that sentence, it is entirely conceivable that someone else might. Therefore, rather than building a model that simply classifies utterances as *good* and *bad*, it will be much more useful for the model to have a notion of the degree of the *goodness* (or *badness*) of any given utterance. The best way to represent this degree of goodness is as a probability value p ($0 \leq p \leq 1$). Therefore, if the model assigns a probability value of 0.9 to a given sentence, it can be easily inferred that this sentence is much more likely to be a fluent English sentence as opposed to a sentence which is assigned a probability of 0.1.

It should be obvious that building such a model is not as simple as it might sound. Such a model must not only know *all* the words in the English language (currently at $500,000$ and growing) but also *all* the rules of grammar that an English speaker learns during her formative years. There is another, even bigger, problem. Even if one had access to all the words and grammar rules of the English language, how would such massive amounts of information possibly be encoded into this black box?

Instead of trying to program exhaustive information about the English language into a black box, statistical language modeling takes a different approach. It relies on the massive amounts of English text that already exist and are freely available for it to learn from. Essentially, given some already existing English text, it extracts some useful statistics from it and then uses these statistics to compute the (approximate) probability of any given English sentence. The reason that this probability is an approximation is because there exists *no* English text that contains *all* English

words and covers *all* English syntactic phenomena. The next section describes these extracted statistics in more detail.

1.2 A Mathematical Formulation

The statistics extracted by language modeling techniques are commonly known as **n-gram statistics**. An n-gram can simply be defined as an overlapping sequence of n words. An example is more instructive. For the sentence "`He went to the store`", the following sets of n-grams can be obtained (the cardinality of each set is indicated by the corresponding number in parentheses):

1-grams (unigrams): `He`, `went`, `to`, `the`, `store` (5)

2-grams (bigrams): `He went`, `went to`, `to the`, `the store` (4)

3-grams (trigrams): `He went to`, `went to the`, `to the store` (3)

4-grams: `He went to the`, `went to the store` (2)

5-grams: `He went to the store` (1)

An n-gram is said to have order n. Therefore, the five unigrams have order 1, the four bigrams have order 2 and so on.

The motivation behind n-grams is best explained by applying the chain rule of probability to decompose the desired probability of a sentence. This rule says that the probability of our sentence "`He went to the store`" can be calculated as follows:

$$p(\texttt{He went to the store}) = p(\texttt{He}) * p(\texttt{went}|\texttt{He}) * p(\texttt{to}|\texttt{went He}) * p(\texttt{the}|\texttt{to went He}) * p(\texttt{store}|\texttt{the to went He}) \quad (1)$$

Equation 1 says that the probability of the whole sentence can be broken down into a number of smaller probabilities. The first term is the probability of the unigram `He`. The second term is the *conditional* probability of the bigram "`He went`". It answers the question that given the word `He`, how likely is it that the next word will be "`went`"? The third is the conditional probability of the trigram "`He went to`" and so on.

Note that this is an equation and not an approximation. If we could estimate each of the above conditional probabilities exactly, the final sentence probability would be exact as well. Unfortunately, the higher the order of an n-gram, the harder it is to estimate its conditional probability because of data sparsity. This is where the approximation enters the picture—to simplify computation, we replace the higher order n-grams with lower order ones. For example, the simplest approximation would use just unigrams:

$$p(\texttt{He went to the store}) \approx p(\texttt{He}) * p(\texttt{went}) * p(\texttt{to}) * p(\texttt{the}) * p(\texttt{store}) \quad (2)$$

This approximated version of the sentence probability can be computed by counting how often each of the five unigrams occurred in the training text, compute the probability of each of those unigrams via maximum likelihood estimation and take their product. For example, if the word `He` occurs 10 times in a text of 1000 words, the probability $p(\texttt{He})$ is just 10/1000 = 0.01.

However, the unigram approximation is extremely crude and noisy. A better approximation can be had by using bigrams instead of unigrams:

$$p(\texttt{He went to the store}) \approx p(\texttt{went}|\texttt{He}) * p(\texttt{to}|\texttt{went}) * p(\texttt{the}|\texttt{to}) * p(\texttt{store}|\texttt{the}) \tag{3}$$

Resorting to MLE, it is also straightforward to compute the conditional probability of a bigram. For example, we can calculate $p(\texttt{went}|\texttt{He})$ as follows:

$$p(\texttt{went}|\texttt{He}) = \frac{\#(\texttt{He went})}{\#(\texttt{He *})} \tag{4}$$

That is, the conditional bigram probability $p(\texttt{went}|\texttt{He})$ is simply the ratio of the number of times (denoted by #) the bigram **"He went"** occurs in the training text, to the number of times the word **"He"** appears as the first word in *any* bigram in the same text. Once all the constituent conditional probabilities have been computed in such a manner, the sentence probability can be estimated approximately by their product. As we increase the order of the n-grams used to compute the approximation, the estimated probability value of the sentence approaches the true probability.

Most practical language models use trigrams, at the very least, since they provide the best balance between approximation accuracy and computational tractability. However, it is not uncommon to see 4-gram and 5-gram language models being used when computational resources are not a bottleneck.

1.3 Building Language Models

This section briefly describes the techniques used to build a statistical language model. As an instructive exercise, the first language model discussed is a very simple unigram language model that can be built using only the simplest of tools that are available on almost every machine. This simple model can be used to explain the concept of *smoothing* which is a technique frequently used in NLP to help improve the estimation of statistical models. The next step is to examine how real language models are estimated using a freely available toolkit.

1.3.1 A Language Model On Training Wheels

Tools that are available on every machine can be used to build simple n-gram language models. Listing 1 shows a simple bash session that show how to extract the words in a given *corpus*[1] (Jane Austen's *Persuasion*, freely available from Project Gutenberg[2]) along with their frequencies (Church, 1989). The most frequent unigram in this English corpus is, not surprisingly, the word **"the"**.

Once the statistics for the unigram model have been collected, they can be used to assign probabilities to sentences by using the unigram approximation. Listing 2 shows a Python script that computes the unigram model probabilities for sentences in a given file by using the Persuasion corpus as the training data. The script essentially computes the maximum likelihood estimates for the probability of each unigram and then applies the unigram approximation from Equation 2 to compute the sentence probability. The script requires that the input file contain one sentence per line. The output of the script, shown in Listing 3, is a probability value for each of the sentences in the input file. These numbers indicate that this particular model considers the second sentence

[1] A *corpus* (pl. *corpora*) is the most commonly used term in NLP literature for a body of text.

[2] The reader can find a pre-processed, ready-to-use version of this corpus at: `http://www.umiacs.umd.edu/~nmadnani/python-papers/persuasion.txt`

Listing 1: Using unix tools to find unigram frequencies (Church 1989).

```
$ tr -sc 'A-Za-z' '\012' < persuasion.txt | sort \
    | uniq -c | sort -nr > persuasion.1grams

$ head persuasion.1grams
  3277 the
  2849 to
  2805 and
  2669 of
  1586 a
  1401 in
  1331 was
  1177 had
  1159 her
  1124 I
```

Listing 2: Computing sentence probabilities using a unigram model (`unigram.py`).

```
inputfile = sys.argv[1]
counts = {}
for line in open('persuasion.txt','r'):
    for word in line.strip().lower().split():
        counts[word] = counts.get(word, 0) + 1

total = sum(counts.values())

for sent in open(inputfile, 'r'):
    prob = 1.0
    for w in sent.strip().lower().split():
        prob = prob * (float(counts.get(w, 0))/total)
    print prob
```

a more likely English sentence than the first one. This is expected because the training corpus used is very small (only about 6,500 unique words and 100,000 word tokens) and, on top of that, the model is order 1. The most interesting number, however, is that for the third sentence. This sentence gets a zero score. Therefore, according to the unigram model, it should *never* occur in the English language. However, it is a perfectly valid sentence and has, in fact, been used previously in this paragraph (Did you notice?). Therefore, despite what the model says, the sentence does represent fluent English.

The reason that the unigram model failed for the third sentence is easy to verify. The training corpus does not contain the unigrams "`zero`" and "`score`". Therefore, sparsity in the training data led to a situation where a sentence that is perfectly acceptable was judged to be incorrect. This is

Listing 3: Evaluating sentences with the unigram model.

```
$ cat input.sentences
  this is the first sentence
  this is the second one
  this sentence gets a zero score

$ python unigram.py input.sentences
  1.8318e−14
  1.8656e−13
  0.0
```

Listing 4: Evaluating sentences with the smoothed unigram model.

```
$ python unigram_smooth.py input.sentences
  1.7138e−14
  1.4629e−13
  3.1679e−24
```

a fairly common issue in statistical modeling and is addressed by using a mathematical technique called **smoothing**. The motivation behind all smoothing techniques is rooted in common sense: if an event has never been observed to occur in the past, its occurrence in the future should only be deemed unlikely, not impossible. The most basic type of smoothing is **Laplace smoothing** which just adds 1 to all unigram counts. This ensures a very small non-zero likelihood for unigrams that were not observed in the training corpus. Therefore, the model no longer assigns zero scores to fluent sentences just because they contain a word that was not seen during model training. Since a 1 is added to each unigram count, the sum of all counts (which forms the denominator for maximum likelihood estimation) increases by $1 * N$ where N is the number of unique words in the training corpus. Given this, we can implement Laplace smoothing in our simple unigram model above by just modifying just two lines in `unigram.py`. Line 7 should be changed to:

```
total = sum(counts.values()) + len(counts)
```

and Line 12 should be changed to:

```
prob = prob * (float(counts.get(w, 0)+1)/total)
```

If the three example sentences are re-evaluated using this smoothed model, the numbers obtained are shown in Listing 4. The model still considers the last sentence very unlikely to occur in the English language but at least it no longer rejects it outright by assigning it a zero score.

1.3.2 Scaling up with the SRILM Toolkit

While it is possible to write programs from scratch to train unigram or even bigram language models with simple forms of smoothing, such programs will not be efficient enough to scale up to higher order language models; especially if the model needs to be smoothed in more sophisticated such as those described in (Chen and Goodman, 1998). Ideally, much larger training corpora

should be used to build higher-order models (trigrams, at the very least) and smarter smoothing techniques should be used so as to provide the best possible model of the English language.

This is where the SRI Language Modeling toolkit (Stolcke, 2002) comes in. The SRILM toolkit is designed to build large-scale language models for use in NLP systems. It is developed, maintained and distributed under an open source community license by SRI International's Speech Technology and Research Laboratory in California. The SRILM toolkit can handle massive English texts, use n-grams of any order and employs state-of-the-art smoothing techniques to deal with data sparsity. In addition, it also provides an API and a set of memory-efficient data structures that can be leveraged to incorporate a language model directly into any NLP application.

The SRILM toolkit runs on all UNIX platforms and is distributed free of charge under an open source community license, which means that it can be used freely for non-profit purposes as long as any changes made are shared with the rest of the user community. The toolkit is not available in pre-compiled form and must be compiled and installed manually, which is not very difficult. The source code can be downloaded after filling a standard form which stipulates that the user agrees to the SRI licensing terms. Once the source tarball has been downloaded, detailed installation instructions can be found in the `INSTALL` file included in the top-level directory. After installation, the `SRILM` environment variable must be set to the top-level directory where the toolkit was installed. In addition, the `bin` directory created by the installation under the top-level directory must be added to the search path so that all the SRILM binaries can be used without having to specify the absolute path.

Once the toolkit is installed, it can be used to build language models. For this purpose, we use Leo Tolstoy's *War and Peace* instead of the *Persuasion* since the former is much larger[3]). The English translation of this text is available from Project Gutenberg and it contains 20,000 unique words and more than half a million word tokens. Note that this is still puny compared to the corpora that large-scale NLP systems routinely use for the purpose of language modeling (as large as two billion word tokens!) but it suffices for the purpose of this article. Line 1 in Listing 5 shows how to build a smoothed trigram language model using the `ngram-count` command[4]. The resulting language model is saved in the file called `warpeace.lm`. The SRILM toolkit not only makes it simple to build n-gram language models, it also makes it very easy to use these models to evaluate any given set of sentences. However, it is important to discuss some salient details about the SRILM toolkit before the model can be used:

1. SRILM internally inserts a start of sentence marker <s> and an end of sentence marker </s> at the beginning and end of each training and test sentence respectively.

2. All *unseen* words in the test sentences (i.e., words that did not occur in the training corpus and have, therefore, never been seen by the language model) are mapped to one single dummy word <UNK> since all unseen words are equally useless to the model. Technically, these words are known as **OOVs** or **O**ut **O**f **V**ocabulary words. By default, all OOVs are discarded from the training corpus as well as the test sentences.

3. If, for any reason, the language model was not trained correctly, it might assign zero probabilities to known words (words that it has seen in the training data). This is an anomalous

[3]The reader can find a pre-processed, ready-to-use version of this corpus at: `http://www.umiacs.umd.edu/~nmadnani/python-papers/warandpeace.txt`

[4]The `tolower` option indicates that all words should be lowercased before collecting the statistics. Other options are self-explanatory.

Listing 5: Building and using a smoothed trigram language model using the SRILM toolkit.

```
$ ngram-count -order 3 -text warandpeace.txt -tolower -lm warpeace.lm

$ cat input.sentences
  this is the first sentence
  this is the second one
  this sentence gets a zero score

$ ngram -order 3 -lm warpeace.lm -ppl input.sentences
  file input.sentences: 3 sentences, 16 words, 0 OOVs
  0 zeroprobs, logprob= -51.5919 ppl= 519.232 ppl1= 1676.84

$ ngram -lm warpeace.lm -order 3 -ppl input.sentences -debug 1
  reading 20200 1-grams
  reading 185625 2-grams
  reading 67713 3-grams

  this is the first sentence
  1 sentences, 5 words, 0 OOVs
  0 zeroprobs, logprob= -12.6611 ppl= 128.879 ppl1= 340.579

  this is the second one
  1 sentences, 5 words, 0 OOVs
  0 zeroprobs, logprob= -11.6708 ppl= 88.1321 ppl1= 215.855

  this sentence gets a zero score
  1 sentences, 6 words, 0 OOVs
  0 zeroprobs, logprob= -27.26 ppl= 7839.4 ppl1= 34940.6

  file input.sentences: 3 sentences, 16 words, 0 OOVs
  0 zeroprobs, logprob= -51.5919 ppl= 519.232 ppl1= 1676.84
```

condition and indicates incorrect training of the model. If any such words do occur, they are referred to as **zeroProbs**.

4. SRILM also employs smoothing, i.e., the technique used to deal with sentences containing unseen or OOV words. The previous section described the very rudimentary Laplace smoothing technique. However, SRILM uses a much more sophisticated form of smoothing known as **backoff smoothing**. The technical details of backoff smoothing are outside the scope of this article and the reader is referred to (Katz, 1987) and (Chen and Goodman, 1998).

5. With the unigram model, the probability values obtained for the test sentences were usually extremely small, e.g., for the first sentence in our test file `input.sentences`, it was $1.7138e{-}14$. Since it is not convenient to interpret and manipulate numbers of such small magnitude,

SRILM uses the base-10 logarithm of these probabilities or **logprobs** instead. Therefore, instead of $1.7138e-14$, SRILM would output $log(1.7138e-14) = -13.7660$ for the first sentence. Logprobs are of a more manageable magnitude and larger (less negative) logprobs still indicate more fluent English.

6. There is another information-theoretic measure more commonly used by NLP researchers to indicate the fluency of a given text. This measure is called **perplexity** and it is calculated in SRILM as follows:

$$ppl = 10^{\frac{-logprob}{(N-j-k+1)}}, \text{where}$$

   - ppl is the perplexity value that we wish to calculate,
   - $logprob$ is the base-10 logarithm of the sentence probability,
   - N is the total number of words in the sentence,
   - j is the number of OOVs in the sentence, and,
   - k is the number of zeroProbs.

   An intuitive way to look at perplexity is to think of it as a measure of how "confused" the language model is for a given sentence. The higher the perplexity value, the more uncertain the language model is about picking the next word in the sentence at any point. The formula also indicates an inverse relationship between logprob and perplexity. Therefore, a text with higher perplexity is less likely to be fluent English. The perplexity measure also allows a way to compare language models: if the set of test sentences is kept fixed, a language model that is "less confused" (has lower perplexity) for this set of sentences is a better model of English than another which may have higher perplexity. In the formula, 1 is added to the exponent denominator in order to account for the end of sentence marker </s> which SRILM considers to be a part of the vocabulary. Perplexity can also be computed without counting </s>, in which case the denominator is simply $(N - j - k)$. This particular perplexity value is referred to by SRILM as $ppl1$.

Keeping these details in mind, let us look at how the trigram model `warpeace.lm` can be used to evaluate the test set of three sentences (shown by lines 3-6 in Listing 5). Lines 8-10 show that using the model is also very simple and can be done with a single commend (`ngram`). This command evaluates the sentence using the specified model and prints out the number of sentences, the number of words (including OOVs and zeroProbs) and, finally, the logprob and both perplexity values. If a detailed output is desired individually for each sentence instead of the full set, the `debug` option can be used to increase the level of detail in the output. When used with a value of 1, this option causes `ngram` to print out the number of unigrams, bigrams and trigrams in the model as well as the statistics for each individual sentence (shown in lines 12-30) .

2 A Python Interface for Language Models

The SRILM toolkit makes it very simple to build large language models and use them in standalone mode. Another advantage is that it also exposes all of its memory-efficient data structures and utility functions in the included header files and libraries. However, these libraries and header files are all written in C++. This presents a challenge if the primary development language for an NLP application is Python and the application needs to use an SRILM language model. An elegant

Listing 6: A C program that computes the volume and surface area of a cylinder (`cylinder.c`).

```
/* Compute the volume and surface area of a cylinder of radius r and height h */
#include <math.h>

/* Define pi */
float pi = 3.1415;

float volume(float r, float h) { return pi*r*r*h; }

float surface(float r, float h) { return 2*pi*r*(r+h); }
```

solution to address this challenge is to employ the Simplified Wrapper and Interface Generator (SWIG) to create an interface around the SRILM API that would allow loading and querying any SRILM language model directly in the Python code. This section first describes the basics of SWIG and then illustrates how it can be used to build a language model interface.

2.1 Introducing SWIG

SWIG (Beazley, 1996) is an extremely useful tool for creating wrappers to connect C- and C++-centric toolkits—such as SRILM—that don't come with built-in language bindings to a variety of common scripting languages such as Python, Perl, PHP and Ruby. SWIG is an open source project which may be freely used, distributed and modified for commercial and non-commercial use.

Using SWIG is fairly simple. If a C program that performed the desired calculations already exists, all that remains to be done is to write an *interface* file for SWIG. Such an interface file would list the appropriate functions and variables from the C program that the Python interface should be able to access. This interface file is then passed to the SWIG compiler which then auto-magically generates the rest of the code that is necessary to package the interface as a Python module.

An example is the best way to illustrate how SWIG works. Listing 6 shows a very simple C program that calculates the volume and surface area of a cylinder of radius r and height h. Of course, this could be implemented very easily in Python but it makes a for a good illustrative example nonetheless. In order to use these C functions in Python, an interface file `cylinder.i` shown in Listing 7 is created. The salient details of this interface file are as follows:

- The `%module` directive defines the name of the module that will be created by SWIG.
- The `%` ... `%` block provides a location for inserting additional code such as C header files or additional C declarations.
- Next the functions that should be exposed in the Python interface are appropriately declared.

Given the C program and the SWIG interface file, a Python module can now be easily compiled and used as shown in Listing 8. This listing includes comments that explain each command in sufficient detail. For a more advanced technical description, the reader is referred to the SWIG web site.

Listing 7: A SWIG interface file for `cylinder.c` (`cylinder.i`)

```
%module cylinder
%{
  #include <math.h>
  extern float volume(float r, float h);
  extern float surface(float r, float h);
%}

extern float volume(float r, float h);
extern float surface(float r, float h);
```

Listing 8: Compiling and Using the Python interface to `cylinder.c`.

```
# Tell SWIG that we need a Python module. This will generate a wrapper cylinder_wrap.c
# and a Python module file called cylinder.py
$ swig -python cylinder.i

# Compile both the original program cylinder.c and the wrapper cylinder_wrap.c.
# Make sure to tell the compiler where the Python header files are.
$ gcc -c -fpic cylinder.c cylinder_wrap.c -I/usr/local/include \
  -I/usr/local/include/python2.5

# Link the two object files together into _cylinder.so, the companion to cylinder.py.
$ gcc -lm -shared cylinder.o cylinder_wrap.o -o _cylinder.so

# Try out the newly compiled cylinder module in Python (interactive mode)
$ python
>>> import cylinder
>>> cylinder.volume(2,2)
25.131999969482422
>>> cylinder.surface(2,2)
50.26544189453125
```

2.2 SWIG-ging a Python Interface to SRILM

This section describes how to create a Python SRILM interface using SWIG. The first step is to create a C program that provides all the functionality that the Python interface requires. Listing 10 show an excerpt from such a program (`srilm.c`). The full listing for this program can be found in the accompanying source code manuscript (Madnani, 2009). Generally speaking, this file defines functions to read in an SRILM file and map it to an internal data structure. Once this mapping is complete, the structure can be queried for various pieces of useful information.

Listing 9: An excerpt from a C program that can query an LM using the SRILM API (`srilm.c`).

```
#include "Prob.h"
```

```
#include "Ngram.h"
#include "Vocab.h"
#include "srilm.h"
#include <cstdio>
#include <cstring>
#include <cmath>

Vocab *swig_srilm_vocab;

/* Initialize the ngram model */
Ngram* initLM(int order) {
  swig_srilm_vocab = new Vocab;
  return new Ngram(*swig_srilm_vocab, order);
}

/* Delete the ngram model */
void deleteLM(Ngram* ngram) {
  delete swig_srilm_vocab;
  delete ngram;
}

/* Read the given LM file into the model */
int readLM(Ngram* ngram, const char* filename) {
  File file(filename, "r");
  if (!file) {
      fprintf(stderr,"Error:: Could not open file %s\n", filename);
      return 0;
  }
  else return ngram->read(file, 0);
}

/* How many ngrams are there? */
int howManyNgrams(Ngram* ngram, unsigned order) {
  return ngram->numNgrams(order);
}
```

Once the C program has been created, a SWIG interface definition file must be created. Writing such a definition is a simple task once the set of functions to be exposed in the module has been determined. After the interface definition is complete, steps similar to those shown in Listing 8 are taken to compile the SRILM Python module that can be imported directly into Python code like any other module. The details of the interface definition file and the compilation process are discussed in the source code manuscript.

At this point, the Python interface for querying any language model built with the SRILM toolkit is compiled and ready for use in Python code. Listing 10 shows how we can use this interface to output the information that we previously obtained in Listing 5 by using the SRILM program

ngram. The output of this Python code is shown in Listing 11. It is easy to verify by simple manual inspection that the output obtained from the Python code is identical to a portion of the output from ngram in Listing 5. A more comprehensive example using the Python SRILM module as well as its output is included in the source code manuscript. That output is exactly identical to the output of the ngram command.

We have verified that our interface is correctly implemented and works as intended. This interface now makes it extremely simple for any Python-based NLP application to use an SRILM language model directly in Python without jumping through unnecessary hoops. Furthermore, the interface we used is by no means the most powerful one that one could come up with. SRILM comes with a large number of classes and libraries which can be leveraged to build much more powerful interfaces.

Now that we have the SRILM module in our Python toolbox, it will be instructive to look an application that uses this module to make the lives of NLP researchers and developers even easier.

Listing 10: Using the Python interface to SRILM (test_srilm.py).

```
# Use the srilm module
from srilm import *

# Initialize a variable that can hold the data for a 3-gram language model
n = initLM(3)

# Read the model we built into this variable
readLM(n, "warpeace.lm")

# How many n-grams of different order are there ?
print "There are", howManyNgrams(n, 1), "unigrams in this model."
print "There are", howManyNgrams(n, 2), "bigrams in this model."
print "There are", howManyNgrams(n, 3), "trigrams in this model."
print
```

Listing 11: The output of test_srilm.py.

```
$ python test_srilm.py

  There are 20200 unigrams in this model.
  There are 185625 bigrams in this model.
  There are 67713 trigrams in this model.
```

3 Serving Language Models with Python

Currently, one of the most popular trends in large-scale NLP applications is the move towards parallelization (Brants et al., 2007; Lin, 2008). In such a scenario, the processing is broken down into smaller chunks each running in parallel and then a collection stage where the results of each chunk are collected and merged together. If the NLP application requires that each processing chunk consult a language model in order to process the allocated data, then it is clearly wasteful to have each client load the same language model in memory on a different machine. A more elegant

solution would be to use a client-server model where a language model server loads the large language model into memory and the clients are able to send their queries to the server over the network. A server-based solution would allow multiple clients to query the same language model without incurring any overhead since the language model would be loaded into memory exactly once. In fact, it is possible to imagine a level of parallelism at an even higher level. It may prove very useful to have a dedicated machine running the LM server all the time and allowing multiple *different* applications to query the same language model over the network. Such a scenario is common in NLP research laboratories where a single language model is built only once using very large corpora (billions of words) and then used by all the researchers in their respective experiments and applications. Of course, this assumes that the network latency is not a bottleneck. If the latency is not within acceptable limits, a more effective solution would be to just trade off disk space for elegance and allow each client to use a localized copy of the large language model. However, networks in research institutions usually have very reasonable latency characteristics and can indeed benefit from the client-server model.

In this section, we will look at how to create an LM server around an existing SRILM language model. For a reasonably effective server solution, the Python standard library, combined with our SRILM interface, is all we need.

3.1 XML-RPC and Python

The XML-RPC protocol[5] is a popular **R**emote **P**rocedure **C**all protocol, i.e., it allows remote clients to call procedures defined in a server process. This particular protocol uses XML to encode the calls and HTTP as the transport mechanism for the calls. While there are several other RPC protocols that have evolved (e.g. SOAP), XML-RPC remains an attractive option because it is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

Since version 2.2, Python comes bundled with the `SimpleXMLRPCServer` module which allows us to easily create a basic XML-RPC server written entirely in Python. This module provides a class (also named `SimpleXMLRPCServer`) that creates a server instance and then register remote-callable functions with this instance. It also allows registering instances (of other, user-defined classes) with the server which, essentially, means that *all* the methods of that instance will be available as remote-callable XML-RPC functions in the server instance. The source code manuscript provides a simple example that illustrates the simplicity of XML-RPC server and clients as written in Python.

3.2 Everything Works Together to Serve

This section allows us to bring together all of the work that we have done so far to create an effective Language Model server written entirely in Python. For the server component, we will use `SimpleXMLRPCServer` and to interface with the existing SRILM language model that is to be served, we will use the SRILM Python module that we compiled in Section 2.

Listing 12 shows an excerpt from the Python script that implements an LM server and an excerpt from the client that will query the LM server is shown in Listing 13. The complete listing for both the server and the client can be found in the source code manuscript. The output of the client excerpt is shown in Listing 14, assuming that the server is serving `warpeace.lm` which we constructed earlier. From manual inspection, we can verify that this output matches the corre-

[5] `http://www.xmlrpc.com`

sponding portion of the output as the generated SRILM program *ngram* for this language model. The full output is, again, only included in the source code manuscript.

Thererfore, our LM server is working as intended and can now efficiently serve multiple clients that wish to query the SRILM language model being served. However, several practical considerations must be mentioned:

- The SRILM toolkit contains its own implementation of a client-server architecture. Therefore, it is certainly tenable that exposing the SRILM client-server implementation to Python using the SWIG toolkit might prove to be more efficient than using a simple XML-RPC implementation, although at the cost of increased code complexity. However, such an exercise lies outside the scope of this article.

- Network latency can prove to be a significant issue when using this client-server architecture. It is not far-fetched to imagine that a single client for some application in production might require a very large number of language model lookups, resulting in upwards of tens of thousands of calls to the LM server. This implies that there is a trade-off between the initial load time of the LM data and the total number of calls made by the clients to the server.

- When using such a model in the real world, it might also be advisable to enable local caching at the client by combining the proposed client-server setup with a form of distributed in-memory data cache (Google, 2009).

4 A Bit of Fun: Writing Faux Tolstoy

So far, we have only used n-gram language models to estimate probabilities of given sentences. However, it is also possible to use the language model to generate new sentences! In this section, we see how we can have a bit of fun with our language model by asking it to generate new sentences that it considers fluent. Listing 15 shows the C function that must be added to our SRILM wrapper (`srilm.c`) to accomplish this. Other changes that must be made in order to use this function are detailed in the source code manuscript. Listing 16 shows how to use this function and Listing 17 shows its output when generating 5 random sentences. It is evident that these are not completely fluent sentences but they are certainly more fluent than word salad: at least every sequence of three words (a trigram) seems to be fluent. This is the best one could expect from a trigram language model. Of course, faux Tolstoy can never be as readable as the real thing; just a bit of fun as promised!

5 Conclusion

Statistical language modeling is a very useful technique used widely in several NLP applications. However, given that the most commonly used toolkit for constructing and using such language models does not provide any bindings for any of the scripting languages including Python, using such language models in Python-based NLP applications is a significant problem. This paper provides an efficient and simple solution to this problem by using a popular interface generator (SWIG) to construct a Python interface such that any SRILM language model can be directly used in Python code. While another alternative interface generator could have been used, e.g., Pyrex (Ewing, 2009), the advantage of a SWIG-based solution is that once the C program and the

Listing 12: An excerpt from a Python LM Server (`lmserver.py`).

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import srilm, sys

class StoppableServer(SimpleXMLRPCServer):
  def __init__(self, addr, lm, *args, **kwds):
    self.myhost, self.myport = addr
    self.lm = lm
    SimpleXMLRPCServer.__init__(self, addr, *args, **kwds)

class LM:
  def __init__(self, lmstruct):
    self.lm = lmstruct

  def howManyNgrams(self, type):
    return srilm.howManyNgrams(self.lm, type)

# Initialize the LM data structure
lmstruct = srilm.initLM(lmorder)

# Read the LM file into this data structure
srilm.readLM(lmstruct, lmfilename)

# Create a wrapper around this data structure that exposes all
# the functions as methods
lm = LM(lmstruct)

# Create an instance of the stoppable server with a pointer to
# this LM data structure
server = StoppableServer((address, port), lmstruct, logRequests = False)

# Register the LM wrapper instance with the server
server.register_instance(lm)

# Start the server
server.serve_forever()
```

interface definition file have been written, it is trivial to generate an interface for any of the several language that SWIG supports. This Python interface for querying SRILM language models is used by the authors in their own NLP research as well as by several other NLP researchers and developers across the world.

In addition to illustrating how to build a Python interface for querying language models, this paper goes one step further and shows how to use the same module to build a language model

Listing 13: An excerpt from a Python LM Client (`lmclient.py`).

```
import xmlrpclib, socket, sys

# Connect to the server (on local machine for now)
s = xmlrpclib.ServerProxy("http://localhost:8585")

# Make the remote procedure calls on the server
try:
  # How many n−grams of different order are there in this LM ?
  print "There are", s.howManyNgrams(1), "unigrams in this model."
  print "There are", s.howManyNgrams(2), "bigrams in this model."
  print "There are", s.howManyNgrams(3), "trigrams in this model."
  print
except socket.error:
  sys.stderr.write('Error: could not connect to the server.\n')
  sys.exit(1)
```

Listing 14: The output from the LM client `lmclient.py` once the LM server is running.

```
$ python lmclient.py

  There are 20200 unigrams in this model.
  There are 185625 bigrams in this model.
  There are 67713 trigrams in this model.
```

Listing 15: A function for `srilm.c` that generates new sentences using a trained language model.

```
void randomSentences(Ngram* ngram, unsigned numSentences, const char* filename) {
    VocabString* sent;
    File outFile(filename, "w");
    unsigned i;

    /* Set seed so that new sentences are generated each time */
    srand48(time(NULL) + getpid());

    for (i = 0; i < numSentences; i++) {
        sent = ngram->generateSentence(50000, (VocabString *) 0);
        swig_srilm_vocab->write(outFile, sent);
        fprintf(outFile, "\n");
    }
    outFile.close();
}
```

Listing 16: Greating faux Tolstoy sentences.

```
from srilm import *

# Initialize and read the language model
n = initLM(3)
readLM(n, "warpeace.lm")

# Generate 5 random sentences in Tolstoy style
randomSentences(n, 5, "faux-tolstoy.txt")

# Free memory
deleteLM(n)
```

Listing 17: Random sentences generated by a trigram language model trained on *War and Peace*.

```
$ cat faux-tolstoy.txt

  it was about to go to the movement of his soldiers , dolokhov .
  he had not seen looking at the movement of a dangers vasili suddenly jumped up .
  " why are you ill ?  " and suddenly surrounded added , as soon as the fifteenth .
  found the rooms opened she said , indeed make a moment , asked rostov .
  he went to the chief .
```

server. There are several advantages of using such an LM server in scenarios where there are multiple consumers that wish to query a single language model. Rather than having each consumer load the large language model in its allocated memory, a more elegant solution is to use a server that loads the language model in memory only once and serves the queries from the consumers over the network. To build such a server, we use the Python LM query interface combined with an XML-RPC server built right into the Python standard library. While this is a very simple and elegant solution, it must be noted that such the communication in such a set up is of a synchronous nature, i.e., if the LM server has started processing one query, it cannot process any others until the processing of this query is finished. Therefore, if the processing takes a long time, then the server will keep waiting and waste time when it could be processing a different query. While this is not a serious disadvantage since most LM queries can be processed reasonably fast, it may become an issue if, for example, the sentence being evaluated by a client is very long. To deal with this issue, it may be necessary to use an asynchronous RPC framework such as Twisted (Lefkowitz, 2009).

References

Beazley, David M. 1996. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop*. http://www.swig.org.

Bird, Steven, Ewan Klein, Edward Loper, , and Jason Baldridge. 2008. Multidisciplinary instruction with the Natural Language Toolkit. In *Proceedings of the Third Workshop on Issues in Teaching*

Computational Linguistics (TeachCL).

Brants, Thorsten, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of EMNLP*.

Chen, Stanley F. and Joshua Goodman. 1998. An Empirical Study of Smoothing Techniques for Language Modeling. Technical Report TR-10-98, Harvard University.

Church, Ken. 1989. Unix for poets. `http://research.microsoft.com/users/church/wwwfiles/tutorials/unix_for_poets.ps`.

Ewing, Greg. 2009. Pyrex: A Language for Writing Python Extensions. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`.

Google. 2009. Memcache Python API. `http://code.google.com/appengine/docs/python/memcache/`.

Katz, S. M. 1987. Estimation of probabilities from sparse data for the language model component of a speech recogniser. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401.

Lefkowitz, Glyph. 2009. Twisted: An Event-driven Network Programming Framework Written in Python. `http://www.twistedmatrix.com`.

Lin, Jimmy. 2008. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce. In *Proceedings of the EMNLP*.

Madnani, Nitin. 2007. Getting Started on Natural Language Processing with Python. *ACM Crossroads*, 13(4).

Madnani, Nitin. 2009. Source Code: Querying and Serving N-gram Language Models with Python. *The Python Papers Source Codes*, 1.

Madnani, Nitin and Bonnie J. Dorr. 2008. Combining Open Source with Research to Re-engineer a Hands-on Introductory NLP Course. In *Proceedings of TeachCL*.

Stolcke, Andreas. 2002. SRILM - An Extensible Language Modeling Toolkit. In *Proc. Intl. Conf. Spoken Language Processing*. `http://www.speech.sri.com/projects/srilm/`.

Turing, A. M. 1950. Computing machinery and intelligence. *Mind*, 59:433–460.