Source Code: 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 supplements (Madnani, 2009) and provides the entire set of source code listings along with appropriate technical comments where necessary. Some of the listings may already be included with the primary article (in complete or excerpted form) but are reproduced here for the sake of completeness.

1 Overview

Natural Language Processing (NLP) is a very active area of research and development in Computer Science and with good reason. NLP applications such as machine translation and automatic speech recognition pervade almost every area of our lives. An important part of such applications is a statistical n-gram language model which is used to assess the fluency of an utterance in the chosen language. These models are estimated by using large amounts of monolingual text. 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. To address this issue, we develop a native Python interface to the toolkit using popular the Simplified Wrapper and Interface Generator (SWIG) tool. A more complete description of language modeling and the process undertaken to build the interface and the server may be found in the accompanying article (Madnani, 2009).

2 Requirements

The work presented in this article and (Madnani, 2009) is based on creating a native Python interface to the SRI Language Modeling (SRILM) toolkit that is most commonly used to build large-scale n-gram language models (Stolcke, 2002). The interface is constructed using the Simplified Wrapper and Interface Generator or SWIG (Beazley, 1996). Before delving into the source code, it is important to list the salient requirements for this project:

1. **SRILM Toolkit**: The source for the SRILM toolkit was obtained from the web. The version used in this work is the latest stable release (v1.5.8). It was compiled and installed on both 32-bit and 64-bit machines. It requires gcc version 3.4.3 or higher along with GNU make. Although the platform used in this paper is Linux-x86, SRILM can be compiled on a variety of platforms including Mac OS X and Win32. The reader is referred to the installation instructions included with the source for more details.

2. **SWIG**: The source for SWIG was also obtained from the web and compiled manually. The version used in this work is the latest stable release (v1.3.39). It was also compiled and installed on both 32-bit and 64-bit machines. SWIG is easy to install from source on almost any flavor of UNIX (including Mac OS X). In addition, pre-compiled binaries of the latest version for the Win32 platform are available from the SWIG web site.

3. **Python**: Python version 2.5 was used for this work and was installed on both 32-bit and 64-bit machines. Since the end-product is a Python interface, the Python header files must be installed. On most Linux distributions, these can be installed by either building Python from source or installing the `python-devel` package in addition to the standard `python` package.

3 Code

This section provides the complete listings of source code that are necessary for the reader of the primary article (Madnani, 2009) to build the Python interface on her own. Listings are grouped by subsections that correspond directly to the sections in the primary article. Only brief technical descriptions are included for each listing. For more details on creating the Python interface, the reader should refer to the primary article.

Section 1: Introduction

In this section, statistical n-gram language models are introduced and the reader is shown how to build a simple unsmoothed unigram language model using tools that are very easily available on any machine. Listing 1 shows how to find the most frequent words from Jane Austen's *Persuasion*. Listing 2 shows how to write a Python script that uses this corpus to build a very simple unigram language model. Note that we ignore all casing information when computing the unigram counts to build the model. Finally, Listing 3 shows how to use this unigram language model to evaluate some given test sentences. Note that the script requires the input file to contain one sentence per line.

However, this model is unsmoothed and will assign a zero probability to any sentence that contains words that were not seen in the training text. Listing 4 shows how to modify `unigram.py` to incorporate Laplace smoothing. Since Laplace smoothing requires a 1 to be added to each unigram count, the sum of *all* counts (which forms the denominator for the maximum likelihood estimation of unigram probabilities) increases by $1 * N$ where N is the number of unique words in the training corpus. The results of using this smoothed model to evaluate the given test sentences are shown in Listing 5.

Trying to train language models by writing our own scripts is not ideal since it is very difficult to scale up to higher order n-grams and use more intelligent smoothing techniques. It is much easier to use a toolkit written specifically for this purpose: the SRI Language Modeling toolkit. Listing 6 shows how to build and use a smoothed trigram language model with this toolkit. The model is trained on Leo Tolstoy's *War and Peace* and can compute both probability and perplexity values for a file containing multiple sentences as well as for each individual sentence.

Section 2: A Python Interface for Language Models

This section describes how to use SWIG to build a Python interface for querying language models built with the SRILM toolkit. The first step is to learn how SWIG works. Listings 7 and 8 illustrate how to use SWIG for generating a Python interface for a simple C program that computes the volume and surface area of a cylinder, given the radius r and height h. Listing 9 shows how to compile and use the Python interface based on the wrapper files that are generated by SWIG. Note that the files `_cylinder.so` and `cylinder.py` together constitute the `cylinder` Python module.

Once we know how to write a SWIG interface file for a given C program, we can use the same procedure to create the SRILM Python module. We first create a C program containing all the functionality that the Python module should have. For the purpose of this article, this set of functionalities includes **introspection** (how many n-grams are in the model) and **evaluation** (score any sentence or a file containing multiple sentences in terms of both probability and perplexity values). Listing 10 shows this C program. Salient points regarding each function are provided as comments. 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. The code only includes very minimal error checking due to space considerations. In situations where perplexity cannot be calculated (all the words in a sentence are unseen, for example), this code returns a value of -1.0. Listing 11 shows the corresponding header file `srilm.h`. The SRILM toolkit is written in C++ which means a C++ compiler must be used. However, since our interface functions are written in C, we need to wrap our function declarations in an `extern C` block.

Given the C program and the header file declaring the functions, a corresponding SWIG interface file can now be created. This file must include the appropriate header files from the SRILM toolkit so that SWIG has knowledge of the data structures and other supporting functions from SRILM. In addition, it is important to note that the functions `getSentenceProbPpl()` and `getFileProbPpl()` require a pre-allocated array of size 3 as input which are then populated with the probability and perplexity values. SWIG provides the `carrays.i` library file which makes this quite easy. The `%array_class(<type>, <name>)` macro creates wrappers for an array object that can be passed around between C and Python easily. Listing 12 shows this interface definition file.

With the C program, the header file and the Python interface definition in hand, the interface itself can now be compiled using SWIG as shown in Listing 13. While compiling the Python shared library module (line 4), it is necessary to pass three SRILM static libraries (`liboolm.a`, `libdstruct.a` and `libmisc.a`) to the linker. These libraries contain the basic classes such as `Ngram` (defined in `$SRILM/lm/src/LM.cc`) and `TextStats` (defined in `$SRILM/lm/src/TextStats.cc`) that come bundled with the toolkit for third-party development. Note that the environment variable `$SRILM` here refers to the top level directory where the SRILM toolkit was installed.

At this point the Python SRILM module is compiled and ready to use. Listing 14 shows a Python script that outputs information similar to the output of the SRILM program `ngram` that we looked at earlier. The script is fairly self-explanatory with the provided comments. The script also generates 5 random sentences using the trigram language model and writes them out to a file. The output of the script is shown in Listing 15. From manual inspection, we can easily verify that the output of this script is identical to `ngram`. This confirms that the module has been implemented correctly and works as intended.

Section 3: Serving Language Models with Python

This section details using the above SRILM Python module to build a language model server that can service multiple clients. To build such a server, we rely on the XML-RPC server functionality that comes bundled with Python in the `SimpleXMLRPCServer` module.

Listing 16 shows an example script that creates a very simple XML-RPC server. This particular server has just one function that is callable by remote clients. All this method does is takes in a list of numbers and returns a list of their squares. The same listing also shows how to register an instance of a class that reverses the string passed to it. Note that in this example, we are starting the server on port 8585 on the local machine itself. In a production environment, we would use the hostname of the machine on which we want the server to run.

Once the server is up and running, writing a client to talk to the server is also very simple. Listing 17 shows a simple client script: a proxy object is created that manages the communication with the actual LM server and all the remote methods are called directly on this proxy object. Listing 18 runs both the client and the server, and also shows the output of the client. We see that both methods were called successfully and produced the expected output. Since remote introspection was also turned on, the server can also be asked to list all the methods that are available for remote invocation. Note also the lines that are interspersed between the various outputs. These are logs of the various requests that were sent to the server. The number 200 represents a status code definition in the HTTP protocol and indicates that the POST request succeeded. These logs are useful for debugging purposes but can be turned off in a production setting by setting the `logRequests` parameter to `False` when instantiating the server. We will do this for the language model server.

Now let's see how we can combine the `srilm` module and the `SimpleXMLRPCServer` module to create a language model server. Since we are writing the server in Python, we need to be able to interface to an SRILM language model in our Python code. For this purpose, we will reuse the SWIG-generated Python module from our toolbox. In order to use it, we need to install the `srilm` module somewhere in the Python search path. There are several ways to do this:

(a) Keep the relevant module files (`srilm.py` and `_srilm.so`) in the same directory as the script using them.

(b) Install the module files in an appropriate place and then add that path to the search path by using the `sys.path.append()` method.

(c) Add the module installation path to the PYTHONPATH environment variable.

Listing 19 shows the code for the language model server. This server assumes that the `srilm` module generated by SWIG is already in the search path. The server is structured such that instead of taking a bunch of input arguments on the command line, the arguments can be specified in a configuration file, which is the only input to the server. Another neat thing about this server is that it's remotely stoppable, i.e., it can be cleanly shut down from inside a remote client. The server code can be dissected to highlight some important points:

(a) The parameter `allow_reuse_address` is set to `True` since we want to allow the server socket to be reused after the particular server instance using it has finished running.

(b) Rather than registering each function from the `srilm` module individually with the server, it is much easier to create a wrapper class and register an instance of this class with the server. Doing this makes all the instance's methods remote-callable.

(c) To allow a client to stop the server remotely, the server's `serve_forever()` method is modified such that before serving any request, it first checks to see if a shutdown request has been made by a remote client.

(d) If there is indeed such a request (as indicated by the instance variable `quit`), the server calls its `stop_server()` method which performs a clean shutdown including freeing the memory allocated to the language model data structure (which is why the `StoppableServer` class is designed to contain a pointer to this structure).

(e) Finally, note that the methods `getSentenceProbPpl()` and `getFileProbPpl()` can't simply return the `floatArray` instances since the XML-RPC protocol does not know how to marshal non-builtin datatypes into XML. We get around this limitation by unpacking the individual array elements and returning those instead.

An example configuration file for this server is shown in Listing 20. A configuration file should specify four things:

- The hostname of the machine on which the server will run.
- The port number which the server should use for communication.
- The path to the actual SRILM language model file that needs to be served.
- The order of the language model that's contained in this file.

A corresponding client for the LM server is shown in Listing 21. This client can run on *any* machine that's on the same network as the server machine. For the purpose of illustration, both the server and the client are being run on the same machine here. This can obviously be changed for a networked scenario by setting the appropriate hostnames in the server configuration and the client script. Listing 22 shows how to run the server and the client and also shows the output of running the client. Note that the server needs a couple of minutes after the command is issued to load the LM into memory server. It does not accept any connections until the LM is loaded. However, the server log should indicate when the LM has been successfully loaded.

For the purpose of convenience and easy access, the index below lists all source code listings in this manuscript and indicates the respective page on which it appears.

Listings

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)*.

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

Madnani, Nitin. 2009. Querying and Serving N-gram Language Models with Python. *The Python Papers*. Under Review.

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/`.

Source Code Listings

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
```

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: Incorporating Laplace smoothing into the unigram model (`unigram_smooth.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()) + len(counts)

for sent in open(inputfile, 'r'):
    prob = 1.0
    for w in sent.strip().lower().split():
        prob = prob * (float(counts.get(w, 0)+1)/total)
    print prob
```

Listing 5: Evaluating sentences with the smoothed unigram model.

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

Listing 6: 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
```

Listing 7: A C program that computes the volume and surface area of a cylinder (`cylinder.c`).

```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); }
```

Listing 8: 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 9: 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
```

Listing 10: C program with the desired LM query functions 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);
}
```

srilm.c (contd.)

```
/* Get the probability and perplexity values for a given sentence */
void getSentenceProbPpl(Ngram* ngram, char* sentence, float ans[3]) {
  char* words[80];
  unsigned numparsed;
  TextStats stats;
  char* scp;

  /* Create a copy of the input string to be safe */
  scp = strdupa(sentence);

  /* Parse the sentence into its constitutent words */
  numparsed = Vocab::parseWords(scp, (VocabString *)words, 80);
  if (numparsed > 80) {
    fprintf(stderr, "Error: Number of words in sentence should be <= 80.\n");
  }

  /* Calculate the sentence probability and store it in stats */
  ans[0] = ngram->sentenceProb(words, stats);

  /* Now calculate the denominator for perplexity */
  if (stats.numWords + stats.numSentences > 0) {
    int denom = stats.numWords - stats.numOOVs - stats.zeroProbs + 1;

  /* Calculate ppl value */
  if (denom > 0)
    ans[1] = LogPtoPPL(stats.prob/denom);
  else
      ans[1] = -1.0;

  /* calculate ppl1 value */
  denom -= 1;
  if (denom > 0)
    ans[2] = LogPtoPPL(stats.prob/denom);
  else
    ans[2] = -1.0;
  }
}
```

srilm.c (contd.)

```
/* Run the file through SRILM and store its statistics into a TextStats instance */
unsigned fileStats(Ngram* ngram, const char* filename, TextStats &stats) {
  File givenfile(filename, "r");
  if (!givenfile) {
    fprintf(stderr,"Error:: Could not open file %s\n", filename);
      return 1;
  }
  else {
    ngram->pplFile(givenfile, stats, 0);
    return 0;
  }
}

/* Compute probability and perplexity over entire file */
void getFileProbPpl(Ngram* ngram, const char* filename, float ans[3]) {
  TextStats stats;

  if (!fileStats(ngram, filename, stats)) {

    /* calculate the file logprob */
    ans[0] = stats.prob;

    /* calculate the perplexity values if we can */
    if (stats.numWords + stats.numSentences > 0) {
      int denom = stats.numWords - stats.numOOVs - stats.zeroProbs + stats.numSentences;

      /* calculate ppl value */
      if (denom > 0)
        ans[1] = LogPtoPPL(stats.prob/denom);
      else
        ans[1] = -1.0;

      /* calculate ppl1 value */
      denom -= stats.numSentences;
      if (denom > 0)
        ans[2] = LogPtoPPL(stats.prob/denom);
      else
        ans[2] = -1.0;
    }
  }
}
```

`srilm.c` (contd.)

```
/* Generate random sentences using the 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++) {
        /* call the built-in SRILM method to generate a random sentence */
        sent = ngram->generateSentence(50000, (VocabString *) 0);

        /* write the generated sentence to the file */
        swig_srilm_vocab->write(outFile, sent);
        fprintf(outFile, "\n");
    }
    outFile.close();
}
```

Listing 11: The accompanying header file for the SRILM C program (`srilm.h`)

```
#ifndef SRILMWRAP_H
#define SRILMWRAP_H

extern "C" {
  Ngram* initLM(int order);
  void deleteLM(Ngram* ngram);
  int readLM(Ngram* ngram, const char* filename);
  int howManyNgrams(Ngram* ngram, unsigned order);
  void getSentenceProbPpl(Ngram* ngram, char* sentence, float ans[3]);
  unsigned fileStats(Ngram* ngram, const char* filename, TextStats &stats);
  void getFileProbPpl(Ngram* ngram, const char* filename, float ans[3]);
  void randomSentences(Ngram* ngram, unsigned numSentences, const char* filename);
}

#endif
```

Listing 12: The SWIG interface definition file (`srilm.i`)

```
/* We want our Python module to be called 'srilm' */
%module srilm

/* Include all needed header files here */
%{
 #include "Ngram.h"
 #include "Vocab.h"
 #include "Prob.h"
 #include "srilm.h"
%}

/* We need to generate interfaces for all functions defined in srilm.h */
%include srilm.h

/* We need a C to Python array bridge and SWIG comes with one */
%include "carrays.i"
%array_class(float, floatArray);

/* Since we are allocating memory when creating random sentences */
/* make sure that we tell SWIG so that it can free it */
%newobject randomSentences;
```

Listing 13: Compiling a Python interface to `srilm.c`.

```
$ swig -python srilm.i
$ g++ -c -fpic srilm.c srilm_wrap.c -I/usr/local/include \
   -I/usr/local/include/python2.5 -I$SRILM/include
$ g++ -shared srilm.o srilm_wrap.o -loolm -ldstruct -lmisc \
   -L$SRILM/lib/i686 -o _srilm.so
```

Listing 14: 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

# First go through our test file and get the individual sentence probabilities
# and perplexity values
infile = 'input.sentences'
for sentence in file(infile):
  svals = floatArray(3)
  print sentence,
  getSentenceProbPpl(n, sentence, svals)
  print 'logprob = %.6f, ppl= %.6f, ppl1= %.6f\n' % (svals[0], svals[1], svals[2])

# Now get overall values
fvals = floatArray(3)
getFileProbPpl(n, infile, fvalues)
print 'Overall:'
print 'logprob = %.6f, ppl= %.6f, ppl1= %.6f\n' % (fvals[0], fvals[1], fvals[2])

# Generate 5 random sentences
randomSentences(n, 5, 'faux-tolstoy.txt')

# Free model variable (to avoid a memory leak)
deleteLM(n);
```

Listing 15: Output for `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.

  this is the first sentence
  logprob = −12.661089, ppl= 128.878830, ppl1= 340.578888

  this is the second one
  logprob = −11.670806, ppl= 88.132133, ppl1= 215.854584

  this sentence gets a zero score
  logprob = −27.259981, ppl= 7839.404297, ppl1= 34940.585938

  Overall:
  logprob = −51.591873, ppl= 519.232483, ppl1= 1676.841675

$ 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 .
```

Listing 16: Creating a Python XML-RPC server (`example-server.py`).

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer

# Instantiate a server on the local machine on port 8585
# Note that the input is a single argument: a tuple
server = SimpleXMLRPCServer(('localhost', 8585))

# Tell the server to support introspection by clients
server.register_introspection_functions()

# Define a method that takes in a list of numbers and returns
# another list with the square of each corresponding number
def squarer(l):
  return [x**2 for x in l]

# Register this function with the server
server.register_function(squarer)

# Define a class with a single method
class reverser:
  # Define a reverse method which returns the reverse of the input string
  def reverse(self, s):
    return s[::-1]

# Register an instance of the above class with the server
server.register_instance(reverser())

# Start the server's main loop
server.serve_forever()
```

Listing 17: A client for `example-server.py` (`example-client.py`).

```python
import xmlrpclib

# Connect to the server
s = xmlrpclib.ServerProxy('http://localhost:8585')

# Call the squarer function
print s.squarer([1,2,3,4,5])

# Call the reverse method
print s.reverse('reverse this')

# Get a list of all the methods that are available on this server
print s.system.listMethods()
```

Listing 18: Output for `example-client.py`.

```
# Run the server forever
$ python example-server.py &

# Run the client
$ python example-client.py

localhost - - [28/May/2009 22:40:55] "POST /RPC2 HTTP/1.0" 200 -
[1, 4, 9, 16, 25]
localhost - - [28/May/2009 22:40:55] "POST /RPC2 HTTP/1.0" 200 -
siht esrever
localhost - - [28/May/2009 22:40:55] "POST /RPC2 HTTP/1.0" 200 -
['reverse', 'squarer', 'system.listMethods', 'system.methodHelp', 'system.methodSignature']
```

Listing 19: A language model server (`lmserver.py`).

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
import srilm, sys

class StoppableServer(SimpleXMLRPCServer):
  allow_reuse_address = True

  def __init__(self, addr, lm, *args, **kwds):
    self.myhost, self.myport = addr
    self.lm = lm
    SimpleXMLRPCServer.__init__(self, addr, *args, **kwds)
    self.register_function(self.stop_server)
    self.quit = False

  def serve_forever(self):
    while not self.quit:
      try:
        self.handle_request()
      except KeyboardInterrupt:
        break
    self.server_close()

  def stop_server(self):
    self.quit = True
    srilm.deleteLM(self.lm)
    return 0, "Server terminated on host %r, port %r" % (self.myhost, self.myport)

class LM:
  def __init__(self, lm):
    self.lm = lm
    self.tmparray = srilm.floatArray(3)

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

  def getSentenceProbPpl(self, s):
    srilm.getSentenceProbPpl(self.lm, s, self.tmparray)
    return self.tmparray[0], self.tmparray[1], self.tmparray[2]

  def getFileProbPpl(self, filename):
    srilm.getFileProbPpl(self.lm, filename, self.tmparray)
    return self.tmparray[0], self.tmparray[1], self.tmparray[2]
```

lmserver.py (contd.).

```
# Get command line arguments
args = sys.argv[1:]
if len(args) != 1:
  sys.stderr.write('Usage: lmserver.py <configfile>\n')
  sys.exit(1)
else:
  configfile = open(args[0],'r')

# Initialize all needed variables to None
address = None
port = None
lmfilename = None
lmorder = None

# Source the config file
exec(configfile)

# Make sure all needed variables got set properly
missing = []
for var in ['address', 'port', 'lmfilename', 'lmorder']:
  if eval(var) is None:
    missing.append(var)
if missing:
  sys.stderr.write('The following options are missing in \
                    the configuration file: %s\n' % ', '.join(missing))
  sys.exit(1)

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

# Read the LM file into this data structure
try:
  sys.stderr.write('Loading language model ...')
  srilm.readLM(lmstruct, lmfilename)
except:
  sys.stderr.write('Error: Could not read LM file\n')
  sys.exit(1)
finally:
  sys.stderr.write(' done.\n')
```

lmserver.py (contd.).

```
# 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
try:
  server = StoppableServer((address, port), lmstruct, logRequests = False)
except:
  sys.stderr.write('Error: Could not create server\n')
  sys.exit(1)

# Tell server to support remote introspection
server.register_introspection_functions()

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

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

Listing 20: An example configuration file for lmserver.py (server.conf).

```
# The address of the server. If you want to run the server and the client
# on a local machine that's not on a network, set this to 'localhost'.
address = "localhost"

# Port number. You might want to make sure what ports you have access to
port = 8585

# The LM file you wish to use. You can also use gzipped LMs.
lmfilename = "warpeace.lm"

# The order of the LM
lmorder = 3
```

Listing 21: An example client for the LM server (`lmclient.py`).

```
import xmlrpclib, socket, sys

# Connect to the server (use hostname of server if on network)
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

  # First go through our test file and get the individual sentence
  # probabilities and perplexity values
  infile = 'input.sentences'
  for sentence in file(infile):
    print sentence,
    svals = s.getSentenceProbPpl(sentence)
    print 'logprob = %.6f, ppl= %.6f, ppl1= %.6f\n' % (svals[0], svals[1], svals[2])

  # Now get overall values
  fvals = s.getFileProbPpl(infile)
  print 'Overall:'
  print 'logprob = %.6f, ppl= %.6f, ppl1= %.6f\n' % (fvals[0], fvals[1], fvals[2])

except socket.error:
  sys.stderr.write('Error: could not connect to the server.\n')
  sys.exit(1)
except xmlrpclib.Fault, fault:
  sys.stderr.write('Error %d: %s\n' % (fault.faultCode, fault.faultString))
  sys.exit(2)
except xmlrpclib.ProtocolError, err:
  sys.stderr.write('Error %d: %s\n' % (err.errcode, err.errmsg))
  sys.exit(3)

# Stop the server. NOTE: You will probably do this in the last client
# (if you know which one that is) or in a clean-up script when you are
# absolutely sure that all clients are finished.
s.stop_server()
```

Listing 22: Output for `lmclient.py`.

```
# Run the server and background it
$ python lmserver.py server.conf > server.log &

# Run the client
$ python lmclient.py

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

  this is the first sentence
  logprob = -12.661089, ppl= 128.878830, ppl1= 340.578888

  this is the second one
  logprob = -11.670806, ppl= 88.132133, ppl1= 215.854584

  this sentence gets a zero score
  logprob = -27.259981, ppl= 7839.404297, ppl1= 34940.585938

  Overall:
  logprob = -51.591873, ppl= 519.232483, ppl1= 1676.841675
```