11-711: Algorithms for NLP
Assignment 1: Language Modeling

Due September 24, 2018

Collaboration Policy

You are allowed to discuss the assignment with other students and collaborate on developing algorithms at a high level. However, your writeup and all of the code you submit must be entirely your own.

Setup

First, make sure you can access the course materials. The components are:

1. `code1.zip` (the Java source code provided for this course)
2. `data1.zip` (the data sets used in this assignment)

The source archive contains four files: `assign1.jar` contains the provided classes and source code (most classes have source attached, but some do not). `build_assign1.xml` is an ant script that you will use to compile the .jar file you submit for grading. The other two files are stubs of the classes you will need to implement. You may wish to use an IDE such as Eclipse to link the `.jar` file and browse the source (we recommend it). In general, you are expected to be able to set up your development environment yourself, but for this first assignment, we will provide setup instructions using Eclipse:

1. Create a new Java Project.
2. Right click on the project and choose "Import".
3. The import type is "Archive File" (under General). Point to the downloaded source file.
4. Use the project Properties dialog to add assign1.jar to the Java Build Path.

At this point, your code should all compile, and if you are using Eclipse, you can browse the classes provided in `assign1.jar` by looking under "Referenced Libraries". You can also run a simple test by running

```
java -cp assign1.jar edu.berkeley.nlp.Test
```

You should get a confirmation message back.

The testing harness we will be using is `LanguageModelTester` (in the `edu.berkeley.nlp.assignments.assign1` package). To run it, first unzip the data archive to a local directory (in commands below, `$PATH` refers to this directory containing the `phrasetable.txt.gz`, `weights.txt`, among others. If you're using Eclipse, putting it in the root directory of the Eclipse project suffices). Then, build the submission jar using

```
ant -f build_assign1.xml
```

Then, try running the following (in Windows, replace the `:` (colon) with `;` (semi-colon))

```
java -cp assign1.jar:assign1-submit.jar -server -mx500m
  edu.berkeley.nlp.assignments.assign1.LanguageModelTester -path $PATH -lmType STUB
```

You will see the tester do some translation, which will take a couple of minutes and take up about 130M of memory, printing out translations along the way (note that printing of translations can be turned off with -noprint). The tester will also print BLEU, an automatic metric for measuring translation quality (bigger numbers are better; 60 is about human-level accuracy). For the stub, accuracy should be terrible (15-16). The next step is to include an actual language model. We've provided a model, which you can use by running

```
java -cp assign1.jar:assign1-submit.jar -server -mx500m
  edu.berkeley.nlp.assignments.assign1.LanguageModelTester -path $PATH -lmType UNIGRAM
```

Now, you'll see the tester read in around 9,000,000 sentences of monolingual data and build an LM. Unfortunately, the unigram model doesn't really help, so you'll need to improve it by writing a higher order language model.

Alternatives to Java You are free to submit any jar on which we can run the necessary commands; this means you can use any language that runs on the JVM, including Scala, Jython, and Clojure. You can package a Scala project into a single jar (including the Scala library) using the plugin sbt-assembly[1] for the build tool sbt.[2] However, for any of these alternative languages, you'll be expected to figure their usage out yourself.

[1] `https://github.com/sbt/sbt-assembly`

[2] `http://www.scala-sbt.org/`

Description

In this assignment, you will implement a Kneser-Ney trigram language model and test it with the provided harness. Take a look at the main method of `LanguageModelTester.java` and its output. Several data objects are loaded by the harness. First, it loads about 250M words of monolingual English text. These sentences have been tokenized for you. In addition to the monolingual text, the harness loads the data necessary to run a phrase-based statistical translation system and a set of sentence pairs to test the system on. The data for the MT system consists of a phrase table (a set of scored translation rules) and some pre-tuned weights to trade off between the scores from the phrase table (known as the translation model) and the scores from the language model. Once all the data is loaded, a language model is built from the monolingual English text. Then, we test how well it works by incorporating the language model into an MT system and measuring translation quality.

You will need to implement a language model that reports the Kneser-Ney trigram probabilities computed from the training data. You should modify the class `LmFactory` to generate language models of this type. You are welcome to create as many additional classes as you like, so long as `LmFactory` retains its names and continues to implement `LanguageModelFactory`.

Evaluation Each language model is primarily tested by providing its scores to a standard MT decoder and measuring the quality of the resulting translations. An MT decoder takes a French sentence and attempts to find the highest-scoring English sentence, taking both the translation and language models into account. The resulting translations are then compared to the human-annotated reference English translations using BLEU. For reference, our exact Kneser-Ney trigram model gets a BLEU score of about 25; if implemented correctly, your trigram model will get the same score. Note that the monolingual English data contains every English word in the phrase table. However, sometimes when translating a French word that hasn't been seen before, the decoder will need to make up a translation rule that includes an unknown English word and the language model will need to return some score. All translations of that sentence will include that rule, so the score you return doesn't really matter so long as it's consistent (for example, always returning a constant should be fine).

In addition to translation quality, your language model will be evaluated for its speed and memory usage. We will expect your language model to fit into about 900M. Note that around 300M is used by the phrase table and the vocabulary (when we run the unigram model, total memory usage is 348M), so at worst, you should aim to make your language model fit in 1.2G of memory (though up to 1.3G is acceptable). We will allow the JVM to use up to 2G of memory since some implementations may require additional scratch space during language model construction. For speed, we are measuring the speed of decoding with your language model, not building it (this is a standard metric, the idea being that you only have to build a language model once, but you have to decode with it for as many sentences as you wish to translate). Note that decoding speed depends heavily on the language model order, so it's typical for decoding with a trigram language model to be dramatically slower than decoding with a unigram model. For reference, on one particular testbed machine, decoding all 2000 sentences with the unigram language model took 9 seconds, but decoding with the exact trigram took 338 seconds.

When we autograde your submitted code we will do two things. First, we will measure BLEU, memory usage and decoding speed using the same testing harness as you by running the two commands

```
java -cp assign1.jar:assign1-submit.jar -server -mx2000m
  edu.berkeley.nlp.assignments.assign1.LanguageModelTester
  -path $PATH -lmType TRIGRAM
```

In addition, we will programmatically spot-check the stored counts for various n-grams.

Implementation Tips The main challenge of this assignment will be efficiently storing and retrieving the counts necessary to compute language model probabilities. You should not expect to be able to do this with standard Java data structures like maps and lists. A `HashMap` internally stores data in `Entry` objects, which maintain references to both their keys and values as well as a cached hash and a pointer to the next entry. Combined with the fact that in Java, values of type `int` have to be boxed inside `Integer` objects to be stored in a `HashMap`, a single key-value pair in a map requires on the order of 100 bytes of overhead. Arrays in Java have no such overhead; an array of type `long` takes space equal to its size times 64 bits. Arrays containing values taking less than 64 bits of space are packed in memory such that no space is wasted even on 64-bit architectures. You will need to be mindful of effects like this in order to fit the data into the allotted space.

As you start the project, you will also want to think about a plan for debugging your code. Because the final evaluation is time-consuming, you will probably want to run your model on smaller test cases along the way. These can either be formal unit tests (using a framework like JUnit) or simply your own testing harness with a `main` method that prints the results of some queries. Try manually computing the Kneser-Ney probabilities on a small example (for example, your corpus might be "a a a b") and make sure that your code behaves correctly on this before scaling it up to larger corpora.

Submission and Grading

Write-ups For this assignment, in addition to submitting a compiled jar for autograding according to the instructions, you should turn in a write-up of the work you've done. The write-up should be 2-3 pages in length and should be written in the voice and style of a typical computer science conference paper. We suggest using the ACL style files[3] but you may use your own style files as long as your submission has comparable length.

In your submission, you should describe your implementation choices for the language model and report performance (BLEU, memory usage, speed) using appropriate graphs and tables. In addition, we expect you to include some of your own investigation or error analysis: for example, you might investigate language model perplexity versus training data size, discuss some machine translation errors and analyze how a better language model might address them, or characterize how your language model's memory usage scales with vocabulary size and number of trigrams. We're

[3]Available at `https://acl2018.org/call-for-papers/`

more interested in knowing what observations you made about the models or data than having a reiteration of the formal definitions of the various models.

Submission You will submit `assign1-submit.jar` and a PDF of your writeup to an online system. Note that your jar must contain an implementation of LmFactory (and the other classes that you used to achieve this), but must not contain any modifications of the source code provided in `assign1.jar`. To check that everything is in order, we will run a small sanity check when you submit the jar. Specifically, we will run the command

```
java -cp assign1.jar:assign1-submit.jar -server -mx50m
  edu.berkeley.nlp.assignments.assign1.LanguageModelTester
  -path $PATH -lmType TRIGRAM -sanityCheck
```

The `-sanityCheck` flag will run the test harness with a tiny amount of data just to make sure no exceptions are thrown. Please ensure that these commands return successfully before submitting your jar.

Grading For this assignment, the following are required for successful completion of the project:

1. The memory usage printed before decoding must be no more than 1.3G.
2. Decoding must be no more than 50x slower than when decoding with the unigram model. no more than 375s.
3. Building your language model and decoding the test set must take no more than 30 minutes.
4. Your trigram BLEU score must be at least 23.

These are hard limits; additional improvements in memory usage and decoding speed will also affect your grade, as will your write-up. The highest-scoring submissions will be those that perform substantially better than the minimum requirements or do substantial investigation or extension.