Statistical Machine Translation: Alignment and Training*

Programming Assignment #1

Due: Wednesday, October 9th at 5pm

IMPORTANT NOTE #1: You may complete this assignment individually or in pairs. *We strongly encourage collaboration*. Your submission must include a statement describing the contributions of each collaborator. See the collaboration policy on the course website: `http://cs224n.stanford.edu/grading.html`.

IMPORTANT NOTE #2: Please read through this whole document before starting on the assignment. We are eager to answer your questions, but please make sure that your question about section 2.3 isn't answered in section 2.4. You should also read the comments and Javadocs in the starter code files.

IMPORTANT NOTE #3: If you have never logged into Farmshare, then please make sure you can well before the assignment deadline: `http://farmshare.stanford.edu/`

1 Introduction

In this assignment, you will build a modern phrase-based statistical machine translation (MT) system. The assignment has two sequential components: implementation of the IBM word alignment models and feature engineering for an MT decoder. Recall from lecture that even the most sophisticated MT systems are based on the IBM word alignment models from the early 1990s. Consequently, you will start by implementing IBM Models 1 and 2. These models learn word-to-word alignments from parallel sentences in an **unsupervised** fashion: the training data does not contain alignments. The IBM models rely on the iterative **Expectation Maximization** (EM) to learn parameters from this incomplete training data. Although imperfect, you will nonetheless be amazed at what these models produce. If you were to visualize the alignments using a grid representation, you might see something like Figure 1:

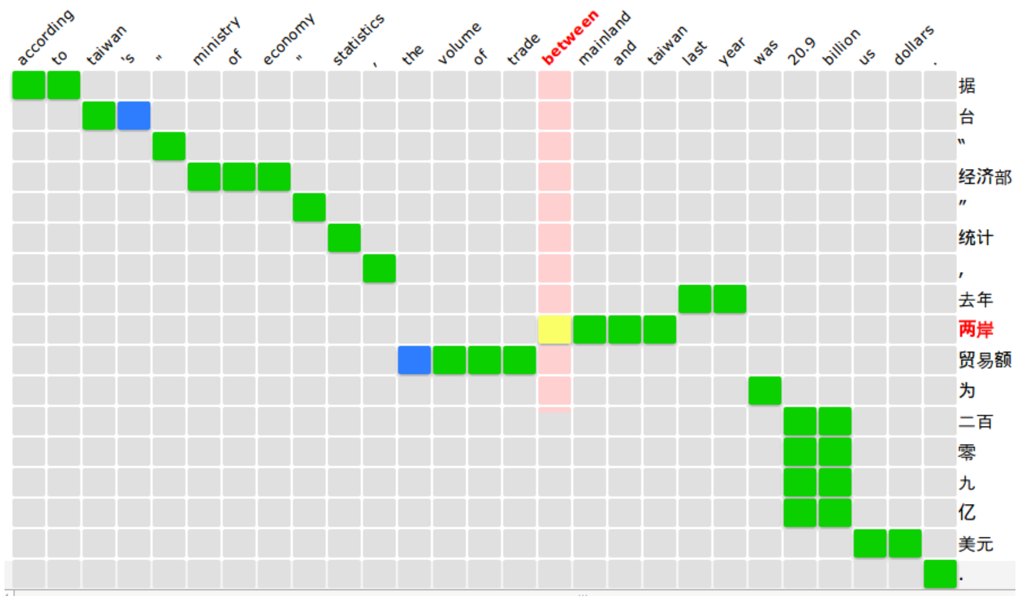

Figure 1: Chinese-English word alignment grid. These alignments were generated from the synthesis of separate Chinese-English and English-Chinese alignment models. Blue cells indicate output of the Chinese-English alignment model; yellow cells indicate the English-Chinese model; and green cells indicate alignments from both models. The green cells typically correspond to higher precision alignments, and thus lead to better translations.

*Revision: September 25, 2013

Once you have a working alignment model, you will be ready for the second part of the assignment: training a full MT system. We will use **Phrasal**, an Java-based system developed at Stanford. You will not write much code in the second part of the assignment, focusing instead on model tuning and feature engineering. MT system builders spend far more time analyzing experimental results and tweaking model parameters than they do cutting new code. You will be rewarded for documenting your experiments and clearly communicating your results.

How to Parse this Document Quickly This document is long. Here's how to quickly parse it and get to work. Sections 2 and 3 describe the two parts of the assignment: word alignment and MT system training. Within each part are two subsections that setup (a) your brain and (b) your coding environment: **Reading and References**, **Setup**. Then there are several subsections marked **Your Job** that tell you what to do. Complete them, write a report, and submit. *Don't wait until the last day to start this assignment*. Both parts require a significant amount of setup. We're ready to help, but we're least effective at 3am the day before the assignment is due. We're (usually) sleeping.

1.1 **Important Setup for Farmshare**

We encourage your to complete this assignment on Farmshare (`http://farmshare.stanford.edu`). This assignment is written for the bash shell, which—despite this being 2013—is not the default on Farmshare. If you aren't running bash, then you can switch to it by typing `bash`.[1]

Farmshare was upgraded over the summer. The new Ubuntu installation doesn't set a default file encoding. This leads to unpredictable encoding problems for our data files, which contain Unicode. Make sure to set this environment variable each time you login, either via your `.bash_profile` or manually:

```
export LANG=en_US.utf8
```

2 Word Alignment

The first task is to implement three word alignment models: a simple model based on pointwise mutual information (PMI), IBM Model 1, and IBM Model 2.

2.1 Reading and References

The lecture notes explain everything you need to implement the IBM models. An additional reference is:

- Michael Collins. *Statistical Machine Translation: IBM Models 1 and 2*. `http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/ibm12.pdf`

You might also skim the Knight workbook, especially the *Efficient Model 1 Training* (hint!) section:

- Kevin Knight. *Statistical MT Workbook*. `http://www.isi.edu/natural-language/mt/wkbk.rtf`

2.2 Setup

Login to Farmshare and copy the Java starter code to your workspace. Then build it using ant:

```
cd
mkdir -p cs224n/pa1
cd cs224n/pa1
cp -R /afs/ir/class/cs224n/pa1/java .
cd java
ant
```

Please take a moment to read through the comments and Javadocs in the starter code. In particular, please study:

`cs224n.assignments.WordAlignmentTester`	Test harness for word alignment models
`cs224n.wordaligner.WordAligner`	Interface that your models must implement
`cs224n.wordaligner.BaselineWordAligner`	Deterministic baseline model
`cs224n.wordaligner.Alignment`	Data structure for alignment predictions

[1] To set bash as your default, see: `http://answers.stanford.edu/solution/can-i-change-default-shell-farmshare-hosts`

The data for this assignment is located in `/afs/ir/class/cs224n/pa1/data`. To avoid exhausting your AFS quota, you should not copy the data to your workspace.

We provide a testing harness and Java interface for your alignment models. To execute the test harness with the baseline deterministic word aligner that simply pairs words along the diagonal of the alignment grid, run:

```
java -cp ~/cs224n/pa1/java/classes cs224n.assignments.WordAlignmentTester \
  -dataPath /afs/ir/class/cs224n/pa1/data \
  -model cs224n.wordaligner.BaselineWordAligner \
  -evalSet miniTest -verbose
```

The `-verbose` flag enables printing of ASCII alignment grids. The synthetic `miniTest` dataset has only four sentence pairs, so it is useful for sanity checking. If your code compiled correctly, the baseline model should report:

```
### Evaluation Results ###
Precision: 0.8889
Recall: 0.8889
AER: 0.1111
```

We'll explain Alignment Error Rate (AER) in section 2.4.

2.3 Your Job: Implementation

Once you have copied and compiled the baseline code, you are ready to implement the three word alignment models. Your models should implement the interface `cs224n.wordaligner.WordAligner`. When you have implemented the required methods, you can run your models with the test harness. For example, suppose you named your model `cs224n.wordaligner.PMIModel`:

```
java -cp ~/cs224n/pa1/java/classes cs224n.assignments.WordAlignmentTester \
  -dataPath /afs/ir/class/cs224n/pa1/data/ \
  -model cs224n.wordaligner.PMIModel -evalSet miniTest -verbose
```

PMI Model

The baseline word aligner did not utilize the `miniTest` data. We can build a better aligner by learning a model from a **corpus** (training data). Let's first define some notation. Let $f = \{f_1, f_2, \ldots, f_m\}$ be a sequence of **source** words from finite vocabulary $\mathcal{F}$ and $e = \{e_1, e_2, \ldots, e_n\}$ be a sequence of **target** words from finite vocabulary $\mathcal{E}$. Define alignment variables $a = \{a_1, a_2, \ldots, a_n\}$ such that $a_i = j$ indicates that target word e_i is aligned to source word f_j. One way to set the alignment variables is to calculate a measure of association between specific source and target words. **Pointwise mutual information (PMI)** is one such measure:

$$a_i = \arg\max_j \frac{p(f_j, e_i)}{p(f_j)p(e_i)} \tag{1}$$

where $p(f_j, e_i)$ is the probability that f_j and e_i appear together in an aligned sentence pair, and $p(f_j)$ and $p(e_i)$ are just the probability of the two different word types in the corpus. You should also reserve f_0 as a special `NULL` word to account for null alignments.

You'll need counters to collect the sufficient statistics required to estimate the model parameters. Primitive arrays might be very sparse, so you might want to use these classes:

`cs224n.util.Counter`	1-dimensional counter
`cs224n.util.CounterMap`	2-dimensional counter
`cs224n.util.Counters`	Various helper methods (normalization, sorting, etc.)

Finally, you'll need to train your model on a larger corpus than `miniTest`. For now, try loading 200 sentences of French-English data using the `-trainSentences` flag:

```
java -cp ~/cs224n/pa1/java/classes cs224n.assignments.WordAlignmentTester \
  -dataPath /afs/ir/class/cs224n/pa1/data/ \
  -model cs224n.wordaligner.PMIModel -evalSet dev \
  -trainSentences 200
```

Section 2.4 explains the `-evalSet dev` argument, and describes how to load corpora for other languages.

IBM Model 1

The PMI parameters did not depend on assignments to the alignment variables. You simply counted how many times words f_j and e_i appeared in the same parallel sentences. You could thus estimate the PMI parameters by looping once through the training data. In contrast, the IBM model parameters are learned iteratively, meaning that we compute new parameter estimates based on variable assignments from previous parameter estimates. We will use the Expectation Maximization (EM) algorithm that we discussed in class. An iterative learning algorithm will run forever if you allow it, so you will need to define a convergence criterion (based on the likelihood of the data) or set a hard iteration limit. We leave that decision to you.

Let's look at IBM Model 1, which is described by the following conditional probability:

$$p(e_1, \ldots, e_n, a_1, \ldots, a_n | f_1, \ldots, f_m, n) = \prod_{i=1}^{n} q(a_i|i, n, m) t(e_i|f_{a_i}) \tag{2}$$

Recall that Model 1 is a special case of Model 2 in that the $q(\cdot)$ parameter has a very simple form:

$$\begin{aligned} q(a_i|i, n, m) =& \frac{1}{m+1} \\ t(e_i|f_j) =& \frac{count(f_j, e_i)}{count(f_j)} \end{aligned}$$

In the first E-step, you should *uniformly* initialize $t(e_i|f_j)$.

IBM Model 2

To implement Model 2, you just need to estimate the parameter $q(a_i|i, n, m)$:

$$q(a_i = j|i, n, m) = \frac{count(j, i, n, m)}{count(i, n, m)} \tag{3}$$

Section 5.2 and Figure 2 of the Collins handout describe the iterative updates for this parameter in detail. As before, you need to initialize your parameters in the first E-step. Since Model 1 converges to a global optimum, the final Model 1 $t(e_i|f_j)$ parameters provide good initial values for Model 2. You should randomly initialize the $q(a_i = j|i, n, m)$ parameters.

2.4 Your Job: Evaluation

In NLP, and machine learning in general, we want to quantify how well our systems generalize to unseen data. Therefore, we evaluate our final/best model on a **test set**. Don't debug on it; don't tune on it; don't even look at it. But if you can only run your model on the held-out set once, how will you measure improvement during model development? We use a separate **development set** for this purpose. To access the development set, pass the `-evalSet dev` argument to `WordAlignmentTester`. When you have settled on a final model, you can access the held-out (test) set with the `-evalSet test` argument.

We also want to see how well our models generalize across languages. With the `-language` argument you can load corpora for other source languages. Valid values are `french` (default), `hindi`, and `chinese`. Development and held-out sets exist for each language.

You have your models, development and test sets, and corpora for different languages. Quantify the accuracy of your models. The standard evaluation metric is **Alignment Error Rate (AER)**,[2] which WordAlignmentTester executes for you. You should minimize AER, with a score of 0.0 indicating perfection, and 1.0 indicating total failure.

Include tables in your report like Table 1 for both the development and test sets:

	French-English	Hindi-English	Chinese-English
PMI			
Model 1			
Model 2			

Table 1: Example AER results table for word alignment experiments.

[2]See: F. J. Och and H. Ney. 2000. Improved statistical alignment models. In ACL.

For comparison, our implementation of Model 1 trained on 10k French-English sentences yielded the following AER results:

```
### Evaluation Results ###
Precision: 0.5295
Recall: 0.7209
AER: 0.4106
```

In addition to the two AER tables, your report should catalog interesting details that arose during development. What convergence criterion did you use? For how many iterations did your models run? If you are interested in code optimization, then you might also want to comment on runtime as a function of corpus size. Finally, if you are bilingual, then you could visualize and analyze the alignments with Picaro.[3] Run `WordAlignmentTester` with the `-outputAlignments filename` flag to print out your alignments for Picaro.

3 MT System Training and Feature Engineering

Now it's time to build an MT system. We'll focus on French-to-English. The word alignments enable extraction of a **phrase table** from the parallel text, also known as a **bitext**. The phrase table consists of short French strings that translate to short English strings. Here's an example from the phrase table that you'll extract:

```
international ||| international ||| (0) ||| (0) ||| -0.2585107 -0.20294084 ...
but ||| mais ||| (0) ||| (0) ||| -0.5131655 -0.33553126 -0.73683476 -0.37852663 ...
he ||| celui-ci ||| (0) ||| (0) ||| -2.8763855 -2.9311938 -6.4187727 -6.175867 ...
```

The first field is French, the second is English, the third and fourth are alignments between the two, and the final is a list of feature scores. This section of the assignment is about those features.

The MT system we'll train is **Phrasal**, which is used for all of the MT research at Stanford. It's the same statistical model that's behind Google Translate. Whereas they train on the whole web, we'll start with 40k sentences. To speed up your job, and to ensure that you can complete this part even if you have problems with the previous section, we provide word alignments for the MT bitext.

3.1 Reading and References

Moses is another popular phrase-based MT system (from the University of Edinburgh) that has (admittedly!) better documentation. Moses and Phrasal are equivalent models. Take a few moments to skim the Moses primer, which explains at a high-level the model that we'll train:

- Statistical MT Background. `http://statmt.org/moses/?n=Moses.Background`

3.2 Setup

We strongly advise you to use Farmshare for this part of the assignment. A full run of Phrasal requires 6GB of RAM and four cores, and lasts about six minutes. *If you choose to run elsewhere, then we cannot help you with configuration problems.* To get started, copy and unpack the source code and training files:

```
cd
mkdir -p cs224n/
cd cs224n/
cp /afs/ir/class/cs224n/pa1/pa1-mt.tgz .
tar xzf pa1-mt.tgz
```

Next, you'll need to set a few environment variables **each time you login to Farmshare**. You can do this manually, or use the simple bash script that we provide:

```
cd pa1-mt
source setup.sh
```

[3]`http://nlg.isi.edu/demos/picaro/`

Verify the Phrasal environment variables as follows:

```
echo "$CORENLP"
echo "$CLASSPATH"
echo "$PATH"
```

You should see paths that exist on the file system. Next, build Phrasal by running the ant script that we provide:

```
cd code
ant
```

If the code compiled and your environment is configured correctly, you can execute Phrasal and its associated run script:

```
java edu.stanford.nlp.mt.Phrasal
phrasal-train-tune.sh
```

You should see the following output:

```
Usage: java edu.stanford.nlp.mt.Phrasal (model.ini) [OPTS]

Usage: phrasal-train-tune.sh var_file steps ini_file sys_name

Use dashes and commas in the steps specification e.g. 1-3,6

Step definitions:
  1  Extract phrases from dev set
  2  Run tuning
  3  Extract phrases from test set
  4  Decode test set
  5  Output results file
  6  Generate a learning curve from an online run
```

If you don't see this output, **STOP**! Your environment isn't configured correctly. Double-check the steps above. If you are still struggling, and Google+StackOverflow fails you, then ping us at office hours or on Piazza.

Take a look at the MT training data:

```
zcat /afs/ir/class/cs224n/pa1/data/mt/corpus.en.gz | head -n 5
zcat /afs/ir/class/cs224n/pa1/data/mt/corpus.fr.gz | head -n 5
zcat /afs/ir/class/cs224n/pa1/data/mt/corpus.align.gz | head -n 5
```

Each of these files has the same number of lines. Each line of the alignments file contains entries like "0-1", which indicates that index 0 in the associated source (French) sentence corresponds to index 1 in the target (English) sentence. Had you run your word alignment models, they would have produced precisely this type of information.

3.3 Your Job: Build a Baseline MT System

Switch to the system directory and take a look at the French development and test sets:

```
cd cs224n/pa1-mt/system
head newstest2011.fr # dev set
head newstest2012.fr # test set
```

The data—including the bitext—is lowercased and tokenized to reduce data sparsity.

`phrasal-train-tune.sh` automates the MT training process for you. Here is a more complete description of the steps, which you already saw when you configured Phrasal:

1. Extract a phrase table from the bitext for newstest2011

2. Iteratively learn the log-linear model weights $w \in \mathbb{R}^m$ on newstest2011 (using a stochastic gradient descent algorithm)

3. Extract phrases from the bitext for newstest2012

4. Translate newstest2012 using the model from step 2 and the phrase table from step 3
5. Run the BLEU evaluation metric on the newstest2012 translations
6. Generate a learning curve from the weights of each iteration of the learning algorithm (we won't use this step)

Now let's train a system called "baseline" by running steps 1–5. In the command below, the vars file configures the script, and the ini file configures the Phrasal Java code:

```
phrasal-train-tune.sh cs224n.vars 1-5 cs224n.ini baseline
```

This job should take about six minutes on the corn machines. When it completes, inspect the following files:

```
newstest2012.newstest2011.baseline.trans # The English translation!
newstest2012.BLEU # Your BLEU score, marked as 'baseline'!
newstest2012.tables/phrase-table.gz # The phrase table for newstest2012
newstest2011.baseline.online.log # Learning log file (grep for "BLEU")
```

Verify that your BLEU score is within 0.5 of the "cs224n-staff" system. Finally, let's look at the feature weights of the model you've learned. We provide a tool that should be in your CLASSPATH:

```
java edu.stanford.nlp.mt.base.IOTools print-wts newstest2011.baseline.online.final.binwts
```

The careful reader will notice that we haven't explained the baseline model features! Here's a description:

- LM—The log probability assigned by the n-gram language model. Our system uses a trigram model.
- LinearDistortion—The total cost of the word reordering in the translation (e.g., switching the order of adjectives and nouns in French->English translation).
- "TM" (translation model) features—Simple conditional probabilities (e.g., $p(\text{red house}|\text{maison rouge})$) estimated from the bitext.
- WordPenalty—The number of tokens in the translation. This balances the LM's preference for short translations. Do you see why? Longer sentences will yield longer sequences of multiplied n-gram probabilities, hence yielding lower total probabilities.

3.4 Your Job: Write an MT Feature and Rebuild the System

Those baseline features are pretty simple...and boring. You need to implement a more interesting feature. A **feature** is simply some characteristic of the data to which we would like to assign a weight. In NLP, we mostly deal with **indicator features**, which are usually realized as truth conditions. Take the word *Sebastian*. A few indicator features of this word are: "starts with S", "is capitalized", "length is more than four characters", "contains b". If the condition is true, then the feature fires. You could check each condition with just a line of code. Then it's the learning algorithm's job to decide how informative each feature is.

Before digging into Phrasal MT features, let's define some notation. A **translation rule** $r = \langle f, e\rangle$ translates a source string f to a target string e. The phrase table that we extracted contains the translation rules. A **derivation** $d = r_1 r_2 \dots r_n$ is a sequence of rules that form a translation. For example, suppose that the French input is $f' = \langle$*la maison rouge est grand*$\rangle$ and that the phrase table has the rules:

- $r_1 = \langle \text{la}, \text{the}\rangle$
- $r_2 = \langle \text{maison}, \text{house}\rangle$
- $r_3 = \langle \text{rouge}, \text{red}\rangle$
- $r_4 = \langle \text{est grand}, \text{is big}\rangle$

Then the candidate derivation $d' = r_1 r_3 r_2 r_4$ defines both the source input and the target translation $e' = \langle$*the red house is big*$\rangle$.

Phrasal scores candidate derivations according to their features, a vector of which we'll define as $\phi(d) \in \mathbb{R}^m$, where m is the number of features in the model. The simple log-linear model score for our example is then:

$$p(e'|f') \propto \exp[w^\top \phi(d')] \tag{4}$$

For this assignment we'll add a new rule feature. Switch to the code directory and look at the following interface:

```
  cd cs224n/pa1-mt/code
  cat src/edu/stanford/nlp/mt/decoder/feat/RuleFeaturizer.java
```

The usage of this interface is best demonstrated by example. Here's a simple indicator feature:

```
public class TargetRuleDimension implements RuleFeaturizer<IString, String> {

  private static final String FEATURE_NAME = "TGTD";

  @Override
  public void initialize() {}

  @Override
  public List<FeatureValue<String>> ruleFeaturize(
      Featurizable<IString, String> f) {

    List<FeatureValue<String>> features =
      new LinkedList<FeatureValue<String>>();

    features.add(new FeatureValue<String>(
      String.format("%s:%d",FEATURE_NAME, f.targetPhrase.size()), 1.0));

    return features;
  }
}
```

This feature simply indicates the length of the target side of each rule. It implements the truth condition "is target side of length x". For example, if $x = 2$, then it will fire "TGTD:2" with a value of 1.0, which specifies that this is an indicator feature. The intuition is that small rules don't capture much bilingual information, and that large rules are probably very sparse. We'd really like modest rules of size 2 or 3. If this intuition is borne out by the data, then the learning algorithm can adjust the feature weights accordingly.

Please read the comments and javadocs in the `Featurizable` class and its associated fields for feature ideas. You can incorporate any property of `Featurizable.rule`. For memory efficiency, Phrasal stores all strings as integers, so you'll see variables of type `Sequence<IString>`. To get an individual word `i`, call `Sequence.get(i).toString()`. To convert the entire sequence to a `String`, call `Sequence.toString()`.

Keep it simple! Your feature should be no more than a few lines of code, but it should capture some empirical insight about the data.

Now that you've written your feature, we need to recompile Phrasal and build a new model with new feature weights. First, copy your feature into the Phrasal code tree and recompile:

```
  cp MyFeature.java src/edu/stanford/nlp/mt/decoder/feat/
  ant
```

Now switch to the system directory and enable your feature in the Phrasal ini file:

```
  cd ../system
  cp cs224n.ini myfeature.ini
  vim myfeature.ini
```

Look for the TODO in the ini file to enable your feature. Save the file. Now we're ready to re-train Phrasal. Thankfully, we can re-use the phrase tables from the baseline system since we haven't changed the data. We just need to run step 2 for model training, step 4 for test set decoding, and step 5 for evaluation:

```
  phrasal-train-tune.sh cs224n.vars 2,4-5 myfeature.ini myfeature
```

When this process completes, you can look at the translations, BLEU score, and feature weights as before. Check the feature weights to make sure that your feature was assigned a weight.

In your report, please include the BLEU scores of both the baseline and the system with your feature. Identify a few interesting differences in the translation output. Give a few sentences of motivation for your feature. What property of the data were you targeting?

4 Administrative Information

4.1 Extra Credit

You may earn up to 10% extra credit on this assignment by completing one of the following assignments:

- Scale up your IBM Model 2 implementation to 100k sentences (French-English). Plot learning curves for runtime as a function of training set with 10k step sizes. You will need to optimize your implementation in some interesting way and show that your optimization actually reduces training time.
- Compare your best translations to the output of Google Translate. You'll need to sample about 30 sentences, define a set of error categories, and then evaluate your system relative to Google's English output. Bilingual analysis is preferable, but you could just analyze the English output. Be sure to include the Google Translate output in your submission folder (but not your report).
- Implement a derivation feature in Phrasal. This is a more sophisticated feature over the whole sequence of rules. You'll need to read about `DerivationFeaturizer` in the Phrasal feature API tutorial here: `/afs/ir/class/cs224n/pa1/phrasal-featureapi.pdf`
- Maximize BLEU on newstest2012. **We'll give full extra credit to the top three submissions**. You could add new features, tune other system parameters (e.g., the distortion limit or maximum phrase length), or even write manual rules to post-process your English output.

We will assess extra credit based on the quality of your work.

4.2 Grading Criteria

Please write a report describing your models and features, and error analysis. Careful error analysis is essential. There is a **hard limit of 4 pages + 1 page for extra credit**. *We may deduct 10% for each page that exceeds this limit.*

You should have three functional word alignment models: the PMI model, an implementation of Model 1, and an implementation of Model 2. You should also have a Phrasal system that can be executed on Farmshare. Your grade will depend on:

- Clear, working code.
- Effective implementation of the models and reasonable results.
- Concise yet clear presentations of the word alignment algorithms and MT features.
- *Insightful error analysis and discussion*. **Every year, students seem to overlook error analysis on the first assignment**. That's understandable because many courses at Stanford emphasize implementation. But this is human language! You have a natural language processor in your head! Take some time to look at the output. You might consider these questions:
  - To what degree do the word alignment models cope with word order differences? Do you see any interesting successes or failures?
  - Does the word alignment model wrongly align function words (*the, a, of*, etc.) with content words? This is called **garbage collection**.
  - What are the most common alignment errors? Aligning too many words to a word, or completely wrong words?
  - For MT system building, what differences did you notice between the baseline and the system with your feature?

4.3 Submission Instructions

Please note:

- **We only accept electronic submissions**. *Do not submit a paper copy of the report*.
- If you are working in a group, then **only one member of the group needs to submit the assignment**.

You will submit your program code using a Unix script that we've prepared. To submit your program, first put your files in one directory on Farmshare. This should include all source code files, but should not include compiled class files or Phrasal phrase tables (the *.tables directories). **Put your report PDF in the root of your submission directory**. Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```

This script will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to re-submit your assignment, you can run:

```
/afs/ir/class/cs224n/bin/submit -replace
```

We will compile and run your program on Farmshare using `ant` and our standard `build.xml` to compile. If you did not complete the assignment on Farmshare, then please verify your code compiles and runs on it before submission. If there's anything special we need to know about compiling or running your program, please include a `README` file with your submission.