**Massachusetts Institute of Technology**
**Department of Electrical Engineering & Computer Science**

6.345 Automatic Speech Recognition
Spring, 2003

Issued: 03/14/03
Due: 04/02/03

---

# Assignment 6
# Language Modeling

---

## Introduction

The goal of this lab is to improve your understanding of language modeling. The lab allows you to become familiar with aspects of $n$-gram modeling, using tools in the SUMMIT speech recognition toolkit. This lab and lab 9 (Segment-Based Automatic Speech Recognition) will help you learn the procedures involved in building the various components of SUMMIT. These skills will be very useful later when you are assembling a baseline recognizer for your term-project experiments.

We will be using data from the **Pegasus** domain which contains queries into flight arrival and departure information.

As in previous labs, the following tasks (marked by **T**'s) should be completed during your lab session. The answers to the questions (marked by **Q**'s) should be handed in on the due date.

## Part I: $N$-Gram Language Modeling

### Introduction

In this part of the lab, we will train and evaluate several $n$-gram statistical language models. In particular, we will do the following:

- train word $n$-gram language models,
- train class $n$-gram language models, and
- measure the perplexity of the language models on training and development data.

### Software

We will use a suite of programs that together comprise the SUMMIT recognition system. We will be using version 3.7 of the system which is specified by setting the environment variable `SLS_HOME` to `/usr/sls/sls/v3.7`. Note that this is done for you automatically in the `.cshrc` file of the class accounts.

A descriptive listing of the programs used in this lab is given in the **SUMMIT Tools** appendix at the end of this handout. Please look over the description of the programs before you start the lab.

## Getting Started

To get started for this lab, type the following command at the UNIX prompt:

```
% start_lab6.cmd
```

This will create a new subdirectory under your home directory called `lab6` and populate it with the set of files needed for this lab. These files contain data, models, and parameter specifications used during the lab. Descriptions of the files can be found in the **SUMMIT Files** appendix at the end of this handout. Do this lab within the `lab6` directory.

## Word $n$-gram Language Models

To train a word $n$-gram statistical language model, we need to specify the list of vocabulary words and a set of word level or orthographic transcriptions of the training sentences.

The file, `pegasus.vocab`, contains all the vocabulary words, sorted in alphabetical order. It also contains:

1. the sentence boundary marker, <>*,
2. the unknown word, <`unknown`>*,
3. <`pause1`>which appears at the beginning of a sentence,
4. <`pause2`> which appears that the end of a sentence.

You will be training the language models using training sentences from the file `pegasus.sents` and evaluating these models on development set sentences in `dev.sents`.

**T1:** Create a word lexicon file (`.wlex`) from the vocabulary file `pegasus.vocab` using the program `wlex_create` as follows:

```
% wlex_create -in pegasus.vocab -out pegasus.wlex
```

Use the program `ngram_create` to train a smoothed **bigram** language model as follows:

```
% ngram_create -wlex pegasus.wlex -in pegasus.sents  \
               -n 2 -out pegasus.bigram \
               -set_log_prob 50 50 0
```

Train a smoothed **trigram** language model as follows:

```
% ngram_create -wlex pegasus.wlex -in pegasus.sents \
               -n 3 -out pegasus.trigram \
               -set_log_prob 50 50 0
```

The `-set_log_prob` option specifies parameters for your language model. The first number is $K$ which controls the interpolation parameter $\lambda$:

$$\lambda = \frac{c(h_i)}{c(h_i) + K} \tag{1}$$

The second number determines a minimum count of tokens used in the unigram. The third number specifies if the <`unknown`> word will be ignored. We have chosen to include it in our case, so the probability of the <`unknown`> word will be greater than zero.

You can view all the options and their explanations by doing:

```
% ngram_create -help
```

## Perplexity

A quantitative way to evaluate a language model is to compute an information theoretic measure, the *perplexity*, of the model on a new set of sentences. The program `ngram_perplexity` can be used to do this.

**T2:** Measure the perplexity of the bigram and trigram models on the development set sentences (`dev.sents`). For example, to compute the perplexity of the word trigram language model, we can say:

```
% ngram_perplexity -ngram pegasus.trigram -in dev.sents
```

Try out several values for the smoothing parameter for the trigram model by changing the first argument of `-set_log_prob` in `ngram_create`. Find a smoothing parameter that will reduce the development set perplexity of the baseline $n$-grams.

**Q1:** **(a)** Give a plot of perplexity (on `dev.sents`) versus smoothing parameter for the bigram and trigram models.

**(b)** How do the perplexity measurements of the trigram models compare to those of the bigram models? Which model is better?

**(c)** How does smoothing affect the perplexity on the development sentences?

**T3:** We can also disregard the pauses at the beginning and end of the sentence as follows:

```
% ngram_perplexity -ngram pegasus.trigram -in dev.sents \
                   -discount_pauses
```

And we can evaluate the probability and perplexity of any word sequence using `ngram_print_sentence_score`. You can specify any sequence of words in the input and it will compute the probabilities based on the specified $n$-gram. Note that these computations are done using the natural logarithm. For example, type the following:

```
% ngram_print_sentence_score -ngram pegasus.bigram \
  "<pause1>" what flights "<pause2>"
```

**Q2:** **(a)** What happens to the perplexity measurement when we ignore the pauses at the beginning and end of the sentences?

**(b)** In using `ngram_print_sentence_score`, observe that the log probability of starting with <pause1> is not exactly zero and the probability of ending with <pause2> is also non-zero. Explain why.

**T4:** We can also output the sentences, reordered according to their probability with the following options:

```
% ngram_perplexity -ngram pegasus.bigram -in dev.sents \
                   -out out.sents
```

Examine the file `out.sents` which contains the sentences from `dev.sents` but in order of probability. And also using `ngram_print_sentence_score`, try out several word sequences. You should get a sense of the types of word sequences that the language model (both bigram and trigram) favors.

**Q3:** **(a)** What types of sentences or word sequences do the language models predict well?

**(b)** What types of sentences or word sequences do they predict poorly?

## Class $n$-gram Language Models

In situations where the training data is sparse for individual words, we may want to map sets of words down to *equivalence classes* to reduce the number of different words in the vocabulary, thereby reducing the number of parameters that have to be estimated. Classes can be created manually or automatically. Typically words in a class are related to one another in some meaningful way. Syntactic and semantic equivalence are popular choices. An example of a hand-crafted word class file is `pegasus.rules`.

**T5:** Examine the word class mappings specified in the `pegasus.rules` file.

To train a class $n$-gram language model, we first need to create a context-free grammar (`.cfg`) file from this class file using the program `cfg_create` as follows:

```
% cfg_create -rules pegasus.rules \
             -wlex pegasus.wlex -out pegasus.cfg
```

We can then use the program `ngram_create` to train smoothed class bigram and trigram language models as follows:

```
% ngram_create -cfg pegasus.cfg -in pegasus.sents \
               -n 2 -out pegasus.class_bigram \
               -set_log_prob 50 50 0
% ngram_create -cfg pegasus.cfg -in pegasus.sents \
               -n 3 -out pegasus.class_trigram \
               -set_log_prob 50 50 0
```

**Q4:** **(a)** How many different classes are defined in the `pegasus.rules` file?

**(b)** How do the numbers of unique $n$-grams change for different $n$ in the word $n$-gram models created in **T1**?

**(c)** How do these numbers compare to those of the class $n$-gram models in **T5**? Use `ngram_tool` to view the contents of a language model file. For example,

```
% ngram_tool -in pegasus.class_bigram
```

**T6:** Now instead of using word classes, lets have a look at phrase level classes. These rules operate at multiple levels. You are given a very simple set of phrase-class rules in `pegasus_phrase.rules` and you will be asked to make up some more to improve the development set perplexity further.

First, let's create the context-free grammar from the set of rules:

```
% cfg_create -rules pegasus_phrase.rules \
             -wlex pegasus.wlex -out pegasus_phrase.cfg
```

We will examine the effect of these rules using the function `cfg_reduce_sentences`. This function will take a file of sentences and print out how the rules have parsed them. Run the following and examine the output file `out.sents`.

```
% cfg_reduce_sentences -cfg pegasus_phrase.cfg -sents dev.sents \
                       -out out.sents -detail 0
```

Observe that in `out.sents` some of the words have been replaced by their class names. Run the function again with "2" as the argument for `-detail`. You will find the different levels of the rules printed out in detail.

Now use this grammar to train up a bigram and a trigram.

**Q5:** **(a)** How many different classes are defined in the `pegasus_phrase.rules` file?

**(b)** How do the numbers of unique $n$-grams in **T6** compare to those in **T5** and **T1**?

**T7:** Now examine the perplexity of the word class and phrase class language models on the development set, `dev.sents`.

**Q6:** How do the perplexities of the word class models compare with those of the phrase class models? Which models are the best?

**Q7:** You will find `cfg_reduce_sentences` helpful for these exercises.

**(a)** Can you create additional word classes to reduce perplexity on `dev.sents`? Retrain your word class language models on `pegasus.sents` and compute the perplexity of the sentences on `dev.sents`.

**(b)** Do the same thing for phrase class rules. Can you create additional phrase classes to reduce perplexity on `dev.sents`? Retrain the models and compute the perplexity.

## Random Sentence Generation

Another way to get a qualitative feel for how well the language model is working is to randomly generate sentences using the statistics in the trained language model and see if they make (some) sense. The program `ngram_create_random_sentences` can be used to do this.

**T8:** Use `ngram_create_random_sentences` to generate 10 random sentences from the bigram model `pegasus.bigram` as follows:

```
% ngram_create_random_sentences -ngram pegasus.bigram -num 10
```

Do the same thing with the trigram model `pegasus.trigram`. Examine the sentences.

**Q8:** Based on your examination of the generated sentences, is the bigram or the trigram language model better? Why?

## Part II: $N$-gram Interpolation with Expectation Maximization

In this part, we will compare the simple count-based interpolation method we used above to a deleted interpolation method that relies on using an expectation maximization (EM) algorithm on held-out data. This algorithm starts with some initial value for $\lambda$ (usually 0.5), and iteratively re-estimates its value to improve the overall likelihood on some held-out data.

In the SUMMIT system, we estimate $\lambda$ starting with the lowest-order $n$-gram and up to the desired $n$-gram. For example, assuming we have estimated the unigram probabilities, we use the following formula to estimate the probability of the bigram:

$$\tilde{P}(w_2|w_1) = (1 - \lambda_{w_1})P(w_2|w_1) + \lambda_{w_1}\tilde{P}(w_2) \tag{2}$$

Where $P(w_2|w_1)$ is the non-interpolated bigram probability, while $\tilde{P}(w_2|w_1)$ is the interpolated one. Note that this notation is slightly different from the one you have seen in class: $\lambda$ is used to weigh the unigram, while $(1 - \lambda)$ is used to weigh the bigram.

The held-out data is obtained through a Leave-One-Out procedure at the $n$-gram token level. In order to estimate the $n$-gram probability of a particular token, we discount that token from the probability computation and use it as a held-out token.

**T9:** Use the program `ngram_create` to train an EM smoothed **bigram** language model:

```
% ngram_create -wlex pegasus.wlex -in pegasus.sents  \
               -n 2 -out pegasus.em_bigram \
               -em_log_prob 0
```

Train an EM smoothed **trigram** language model as follows:

```
% ngram_create -wlex pegasus.wlex -in pegasus.sents \
               -n 3 -out pegasus.em_trigram \
               -em_log_prob 0
```

**Q9:** **(a)** What is the perplexity on `dev.sents` using the EM-smoothed bigram?

**(b)** What is the perplexity on `dev.sents` using the EM-smoothed trigram?

**(c)** How does the EM smoothing compare to the simple interpolation we did in **T1**?

**T10:** We will see how the two smoothing methods compare as we decrease the amount of training data for our bigram language model. For each method, we will train on 80%, 60%, 40%, and 20% of the data and compare how the perplexity degrades for the two methods as we decrease the amount of training data.

There are around 5,000 training sentences in the training data, `pegasus.sents`. To get 80% of the data, use:

```
% head -4000 pegasus.sents > pegasus.80.sents
```

This will create a file, `pegasus.80.sents`, with the first 4,000 sentences from the training data, (or 80% of the complete training). Follow the same method to create the smaller training sets.

**Q10:** **(a)** Plot on the same graph the perplexity as a function of the amount of training data for the two smoothing techniques.

**(b)** Which method performs better as we use less data?

**T11:** In this part, you will estimate the EM interpolation weight $\lambda$ for the bigram case. We make the following simplifying assumptions:

We are trying to estimate the interpolation weight $\lambda$ of the word *FOR* where we have observed only the three words *DALLAS*, *TODAY*, and *THE*. This is represented in the $n$-gram tree in Figure 1. Assume that the counts in the training data are as follows:

$C(FOR) = 14$
$C(DALLAS|FOR) = 10$
$C(TODAY|FOR) = 2$
$C(THE|FOR) = 2$

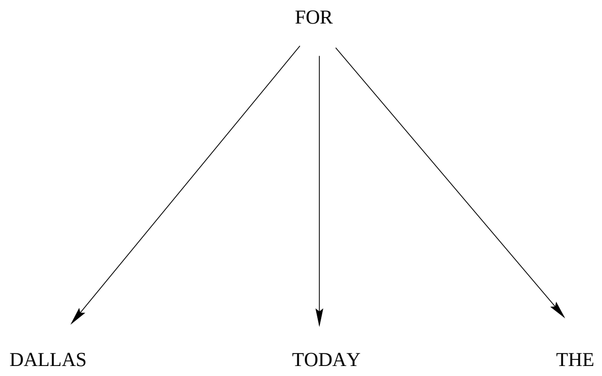

Figure 1: *A bigram tree for the word "FOR".*

In addition, assume that all unigram probabilities are already estimated and that the discounting does not impact the value of these unigrams. Assume the following values for the unigram probabilities:

$\tilde{P}(DALLAS) = 0.05$
$\tilde{P}(TODAY) = 0.15$
$\tilde{P}(THE) = 0.70$

Under these assumptions, the EM update equation is:

$$\lambda_{i+1} = \frac{1}{C(W)} \sum_{\forall W_j} \frac{\lambda_i \tilde{P}(W_j)}{(1-\lambda_i)P(W_j|W) + \lambda_i \tilde{P}(W_j)} C(W_j|W) \tag{3}$$

Where $P(W_j|W)$ is the discounted bigram probability of a token given by:

$$P(W_j|W) = \frac{C(W_j|W) - 1}{C(W) - 1} \tag{4}$$

Note that the index of $\lambda$ here indicates the iteration number (unlike the class notes where it indicates the $n$-gram order).

**Q11:** Use the update equation to estimate the value of $\lambda$ starting with an initial value of 0.5. You should re-estimate $\lambda$ until it does not change by more than 0.01. How many iterations are needed? Make a table of the values of $\lambda$ at each step.

**Q12:** There are two cases where this EM approach leads to a degenerate solution.

**(a)** Provide an example set of counts $C(W_j|W)$ which will lead to the degenerate case of $\lambda = 1$. Explain why this answer is degenerate.

**(b)** Provide an example set of counts $C(W_j|W)$ which will lead to the degenerate case of $\lambda = 0$. Explain why this answer is degenerate.

**(c)** Which case, (a) or (b), is worse for recognition? Why?

**(d)** Propose some upper bound and lower bound for $\lambda$ based on counts and vocabulary size.

## Appendix: SUMMIT Tools

This section gives pointers to some available online documentation for the SUMMIT tools, provides a descriptive index of the tools used in the lab, and mentions some of the essentials of what you need to know in order to complete the lab.

### Command Line Usage Help

Note that for most of the programs, supplying the command line argument `-help` will list the program's usage along with valid command line arguments. This is useful if you forget the exact usage syntax or want to see what other options are available.

### Program Index and Description

**cfg_create** Creates context free grammar (CFG) given a set of classes/rules (`.rules`) and a word lexicon (`.wlex`). The CFG is used in training the $n$-gram language models.

**cfg_reduce_sentences** Outputs a set of sentences redcued by a CFG, given an input set of sentences. This is useful for examining CFGs.

**ngram_create** Creates an $n$-gram language model file given a set of training utterances and either a `.cfg` file (for class $n$-gram) or a `.wlex` file (for word $n$-gram). The command line option `-set_log_prob` is used to set the log probabilities and to specify smoothing parameter values.

**ngram_create_random_sentences** Randomly generate sentences using the statistics in the specified $n$-gram language model. This is useful for qualitatively evaluating an $n$-gram model.

**ngram_perplexity** Compute the perplexity of a specified set of utterances using a given $n$-gram language model. Useful for getting a quantitative measure of the performance of an $n$-gram model.

**ngram_print_sentence_score** Prints out the perplexity and log probability for a given word sequence and language model.

**ngram_tool** Displays information about a given $n$-gram model file. Also allows checking of the parameter values and the setting of smoothing parameters.

**wlex_create** Creates a word lexicon (`.wlex`) file from a basic vocabulary (`.vocab`) file. The word lexicon file is used in building $n$-gram language models.

## Appendix: SUMMIT Files

This section contains a descriptive index of the files used in the lab. These files contain data, models, and parameter specifications needed to build a speech recognizer.

## File Index and Description

**pegasus.sents** List of utterances in the training set.

**dev.sents** List of utterances in the development set.

**pegasus.rules** Set of word to class mappings or rules. This is used to create a CFG file which is then used to build class $n$-gram language models.

**pegasus_phrase.rules** Set of phrase to class mappings or rules.

**pegasus.vocab** List of all the vocabulary words in the domain.