Language Models Conditioned on Dialog State

Karthik Visweswariah, Harry Printz

IBM
Thomas J. Watson Research Center
Yorktown Heights, NY 10598
e-mail: kv1@us.ibm.com, printz@us.ibm.com

Abstract

We consider various techniques for using the state of the dialog in language modeling. The language models we built were for use in an automated airline travel reservation system. The techniques that we explored include (1) linear interpolation with state specific models and (2) incorporating state information using maximum entropy techniques. We also consider using the system prompt as part of the language model history. We show that using state results in about a $\frac{1}{BE/BC/B1}$ relative gain in perplexity and about a $\frac{1}{BL/B1}$ percent relative gain in word error rate over a system using a language model with no information of the state.

1. Introduction

In this paper we report on various experiments that we conducted to improve recognition accuracy by dialog state in a DARPA communicator system. The system is meant to act as an automatic travel agent: A user can make travel plans and reserve flight tickets by calling the system. For more details on the task see [1, 2, 3]. We are concerned here with the speech recognition part of the system, and more specifically with the language model used in the speech recognizer.

When the user converses with the system it is intuitive that the state of the dialog strongly conditions the responses that one might expect from the user. The state of the dialog ideally would be the entire history of the dialog up to the current time. Of course this definition of state would lead to too large a state space to learn anything useful, given the amounts of training data. In the system we used, the NLU part of the system maintains a state (which is feedback to the parser). This state coarsely represents (state can be one of 10 values) the state that the dialog is in. Apart from this state we also consider using the prompt that the user made his reply in response to. We list a couple of the prompts which occurred in the NONE state:

Do you want to leave at 10.50 a.m or 10.50 p.m
Pardon me, what time would you like to leave?

In the remainder of the paper we describe our efforts at using the state information that we had for part of our language modeling training data to improve recognition performance.

2. Training data and experimental set up

The training data that we have consisted of two parts. The first contains 95 thousand sentences and no information of the state of the dialog. The second contains about 11 thousand sentences with information on the state of the system (as maintained by the NLU part of the system) and with the system prompt that the user made his reply in response to. In Table 1 we list the various parser feedback tags that appeared in the training data and we give the number of words that were observed in each state. There is in addition to the parser feedback tag a recognizer feedback tag which has two values: A default one and one telling the recognizer to expect a yes/no in reply. Most of these feedback tags occur when the parser feedback was NONE, but in our experiments we did not find this information to affect the error-rate and so we leave out this feedback information in the remainder of the discussion.

In addition to these tags we also have (for the 11k sentences) the system prompt that the user responded to. While we would not expect the system prompt to be very useful for some of the parser feedback states, we might expect to use some information from the prompt when the parser feedback tag is NONE. We list a couple of the prompts which occurred in the NONE state:

- Do you want to leave at 10.50 a.m or 10.50 p.m
- Pardon me, what time would you like to leave?

We first describe the language model training data that we worked with. We then describe the various methods that we used to incorporate state into the language model and finally we report results (measured by perplexity and speech recognition accuracy) of the various models.

<table>
<thead>
<tr>
<th>Feedback tag</th>
<th>Number of words</th>
<th>System expects</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE-ARR</td>
<td>864</td>
<td>arrival date</td>
</tr>
<tr>
<td>DATE-DEP</td>
<td>2527</td>
<td>departure date</td>
</tr>
<tr>
<td>DATE</td>
<td>194</td>
<td>departure or arrival date</td>
</tr>
<tr>
<td>DONE</td>
<td>335</td>
<td>confirmation</td>
</tr>
<tr>
<td>FROM</td>
<td>1568</td>
<td>departure city</td>
</tr>
<tr>
<td>LIST</td>
<td>5181</td>
<td>choice from presented list</td>
</tr>
<tr>
<td>NONE</td>
<td>36148</td>
<td>default-tag</td>
</tr>
<tr>
<td>NUM-FLT</td>
<td>24</td>
<td>flight number</td>
</tr>
<tr>
<td>TIME-ARR</td>
<td>22</td>
<td>arrival time</td>
</tr>
<tr>
<td>TIME-DEP</td>
<td>3585</td>
<td>departure time</td>
</tr>
<tr>
<td>TIME</td>
<td>567</td>
<td>departure or arrival time</td>
</tr>
<tr>
<td>TO</td>
<td>1202</td>
<td>departure city</td>
</tr>
</tbody>
</table>

Table 1: Description of the feedback tags
3. Linear interpolation

The simplest way to use the state information is to use the data that we have available for each state to build a model specific to that state. Looking at Table 1 it is clear that there is not enough data to build models for most of the states. It is also clear that many of the states would result in similar responses. We combine the data in these states while we build models. We then have six models: DATE, TIME, DONE, NONE, LIST. Even combining the model in this way it is not clear that we have enough data in each state to build models that will generalize. To avoid over training we linearly interpolate the state specific models with a model built from all the data. We used linear interpolation to smooth the models of various orders for the state specific data. For the state specific data we do not have enough data to use held out data and train the interpolation weights so these weights were chosen arbitrarily and we do not optimize over the weights. We also experimented with using the modified Kneser-Ney smoothing algorithm [6] which we can use to build a model without any held out data, as [6] indicates that when building models from small amounts of data the modified Kneser-Ney smoothing algorithm outperforms deleted interpolation (among other smoothing schemes).

3.1. Using unlabeled data

Since the number of sentences with state information that we have is much smaller sentences of data with state information. To exploit the unlabeled data we use the state specific models to infer state as follows. Given a sentence $s$ without a label we could compute the probability that the true state label of the sentence is $l$:  

$$ P(l|s) = \frac{P(s|l)P(l)}{\sum_{j \in L} P(s|j)P(j)} $$

where $L$ is the set of states and $P(s|l)$ is the probability of the sentence built using the model specific to the state and $P(j)$ is the a-priori probability that a sentence belongs to a class $j$. We could then label the sentence with label $l$ with probability $P(l|s)$ or we could use the label $l$ if $P(l|s)$ is above a certain threshold.

We now give examples (randomly picked) of some of the sentences obtained using the thresholding methods:

- **DATE**: the next day
- **DONE**: yes
- **TO FROM**: first class round trip airfare from Indianapolis to Memphis
- **TO FROM**: history lives

These examples indicate that labeled data can be used to learn some state information for unlabeled data. Of course there in learning labels from our smaller models there is a fundamental tradeoff: We want to pick sentences which are similar to those in the labeled training data but we want the sentences to be slightly different so that we build models different from the ones we already have.

The principled way of using the unlabeled data to build state based models is to build models to maximize the likelihood of the unlabeled data. Given sentences $s_1, s_2, \ldots, s_n$ their likelihood under the state based models is

$$ \prod_{i=1}^{n} \sum_{l} P(s_i|l)P(l). $$

This would be the likelihood of the data assuming that the state of a sentence is independent from the state of other sentences, $P(l)$ is the probability that the true state label is $l$ and the probability of a sentence given label $l$ is $P(s|l)$. The model $P(s|l)$ is initially built from counts obtained from data labeled $l$, we would then:

a) Compute $P(l|s_i)$ for each sentence b) Use

$$ C_l(w_1, w_2, w_3) = \sum_{i=1}^{n} P(l|s_i)C(w_1, w_2, w_3 \in s_i) $$

to rebuild models for each state label $l$. This is a method used in [7] to induce long range dependency in language models.

3.2. Using the system prompt

So far we assumed the state was the feedback tag to the parser, which mapped as described above to one of six values. The system prompt also contains some information which influences the user response. We feel that even when the parser feedback is NONE the system prompt could contain information that we could exploit in modeling the user utterance. To exploit this information we use the following approach: Find phrases that occur often in the system prompts, and divide data according to which of the phrases is present in the system prompt. Clearly we cannot have too many phrases as this would fragment the data too much. We could reduce some of this fragmentation by clustering together data which occurs in different states. We combine two states if the data in these two states have similar statistics. We measure similarity by the unigram distributions in the two states: Given two distributions $P_1$ and $P_2$ we can measure their similarity by $H(\pi_1 P_1 + \pi_2 P_2)$ where $\pi_1 H(P_1) + \pi_2 H(P_2)$ is the mutual information between the underlying state and the observed words. We combine the two most similar states repeatedly. This will give us a bottom up cluster tree, we can then use any subtree of this tree as our set of states. In Figure 1 we show the cluster tree we obtain for the eighteen states that we obtained from the system prompt phrases. Parentheses indicate the number of sentences in each of the states. The encircled states show the data that we would pool together if we wanted to reduce back to six states. Apart from the states we had earlier we have some more states:

<table>
<thead>
<tr>
<th>Feedback tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>change_time</td>
<td>system asks to change time</td>
</tr>
<tr>
<td>change_airline</td>
<td>system asks to change airline</td>
</tr>
<tr>
<td>init_gen</td>
<td>initial state: no guidance</td>
</tr>
<tr>
<td>init_spec</td>
<td>initial state: example user utterance</td>
</tr>
<tr>
<td>trouble_hearing</td>
<td>system complaint</td>
</tr>
<tr>
<td>cannot</td>
<td>system cannot perform action requested</td>
</tr>
</tbody>
</table>

Table 2: Description of extra states from prompt

Using the system prompt information in the framework we have used so far of splitting the data and building separate models for each part could fragment the data too much and lead to over training of our models.

4. Maximum entropy models using state information

Rather than building a separate model for each state (which results in fragmentation of data), we could use exponential models
with feature functions that have state information. Let us denote
our training set as: \( \{ p_i, \lambda_i \}_{i=1}^n \) where \( p_i \) is a set of words that
includes all words in the prompt, the parser feedback and any
other feedback that we might have. The class of binary feature
functions that we consider are as follows:

\[
    f_{uv}(p, w) = \begin{cases} 
        1 & : u \in p, v = w \\
        0 & : \text{otherwise}
    \end{cases}
\]

The class of models we consider is:

\[
    P_{\beta}(w|\tilde{p}) \exp(\sum_{v} \lambda_{uv} f_{uv}(p_i, w))
\]

where \( Z(h, p, \lambda) \) denotes the normalizer. We could then
find the model from this class that maximizes the log likelihood of
the training data:

\[
    \sum_{i=1}^n \sum_{(h, w) \in \lambda_i} \log P_{\beta}(w|\tilde{p}) \exp(\sum_{v} \lambda_{uv} f_{uv}(p_i, w)))
\]

This is equivalent to [8, 9] finding a model \( P(w|\tilde{p}, p) \) that
minimizes:

\[
    \sum_{i=1}^n \sum_{h \in \lambda_i} D(P(\cdot|p, \lambda_i)||P_{\beta}(\cdot|\tilde{p}))
\]

subject to satisfying the constraints

\[
    \sum_{i=1}^n \sum_{h \in \lambda_i} \sum_{v \in V} P(x|\tilde{p}, p_i) f_{uv}(p_i, x) - \sum_{i=1}^n \sum_{(h, w) \in \lambda_i} f_{uv}(p_i, w)
\]

for all features \( f_{uv} \). These constraints require that the empirical
number of times various features occur match the expected
values predicted by our model.

### 4.1. Selection of features and training

In all exponential modeling the key aspect is the selection of features. Once the features are selected training can be done by
the improved iterative scaling procedure [8, 9]. Smoothing of the
model can be achieved by using a Gaussian prior on the
weights without changing the essence of the training procedure
[10]. When we needed to pick features (i.e., when we would not
want to use all features) in our experiments we picked features
by ranking features by their gain in likelihood when used once at
a time, and picking the top \( N \) features.

### 5. Experimental results

We conducted our speech recognition experiments on two test
sets. Test set INT was a set of utterances internally collected
from travel-domain human computer interactions over the
telephone. Test set EXT was a test set obtained from interactions
with the IBM system during the evaluation organized by the
National Institute of Standards and Technology. The test set
INT contained 8160 words and 2082 utterances whereas EXT
contained 3075 words and 1173 utterances. We keep the acous-
tic models and all other parameters fixed across all our experiments.

All language models we describe are built on the same voc-
AB with 3k words. The training data is tokenized using a na-
uroversal language understanding classer [12]. Our base language
model is one build from the 95k unlabeled sentences. We build
the models using either deleted interpolation or using the
modified Kneser-Ney method [6]. We test our this lan-
guage model LM1 (or LM1_KNm for an language model built
using the modified Kneser-Ney method). The real baseline that
we should use for comparison is of course a language model
built from all data without using state information. We will re-
fer to this as LM_base (LM_base_KNm).

Our first state based model is built from data split according
to the parser feedback. We then interpolate each state based
with a base model built from all data with weight. The state
model and the base model are weighted equally. Let us call this
linearly interpolated state based model linS_LM (linS_LM_KNm).
For the state based models we did not want to further partition data into heldout data to train the weighting be-
tween models of various orders. Let us call the models we built
from data split according to the system prompt linSprmpt_LM (linSprmpt_LM_KNm). Starting with linS_LM we build models by
relabeling with states data that did not have state information
(linS_unb中医药). We also built a model using EM starting from
linS_LM, to use the unlabeled sentences but this did not help and the models were similar to just doing one step (which
is approximately what linS_unb中医药 does).

All our exponential models used LM1 as a base model. The
first model we built used all features of the form \( f_{uv}(s, w) \) which is binary valued and non-zero only when \( u = s \)
and \( v = w \) (The state \( s \) being the parser feedback tag). This
especially placed unigram constraints on the model expectations
within each state. The number of features in this case was 2k.
This model uses the same state information as model LM1. We
refer to this model as meS_LM. Since the base model LM1 is
class based we should also allow our features to use this class
information (the classes in the original model are handpicked,
C.g days of the week, months, etc.). We build a model from all
possible features of the form \( f_{uv}(s, w) \) which is 1 when \( u = s \)
and \( v \in C \) for all \( u \in S \) and classes \( C \) in the original model.
Let us refer to this model as meS_class_LM. We also built two
models with the full prompt as the history. In one case we used
a set of hand picked phrases \( \Phi \) and considered all features of
the form \( f_{uv}(p, w) \) which is 1 when \( u \in p \) and \( w \in \Phi \). This model was built with
2.6k features. We call this model meSprmptHP_LM. The final
model we report results for is a ME model with features \( f_{uv} \) which trigger when the system prompt contains \( u \) and the cur-
rent word is \( w \). The set of such features was of size 60k which
was then reduced to 5k by picking those with the best single
feature perplexity gain. We refer to this model as meSprmpt_LM.

We first present perplexity results for these models in Table
3. We have reported perplexity results on only the internal test

![Figure 1: Tree obtained by bottom clustering of data in various states](image-url)
set because we do not have the external test set semantically tokenized [12].

<table>
<thead>
<tr>
<th>Model</th>
<th>Perp. on INT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM1</td>
<td>31.13</td>
</tr>
<tr>
<td>LM1_KNm</td>
<td>30.65</td>
</tr>
<tr>
<td>LM_base</td>
<td>29.60</td>
</tr>
<tr>
<td>LM_base_KNm</td>
<td>28.27</td>
</tr>
<tr>
<td>linS_LM</td>
<td>24.49</td>
</tr>
<tr>
<td>linS_LM_KNm</td>
<td>22.10</td>
</tr>
<tr>
<td>linPrmpLt_LM</td>
<td>23.20</td>
</tr>
<tr>
<td>linPrmpLtClust_LM</td>
<td>23.01</td>
</tr>
<tr>
<td>linPrmpLtClust_LM_KNm</td>
<td>20.57</td>
</tr>
<tr>
<td>meS_LM</td>
<td>25.36</td>
</tr>
<tr>
<td>meS_LM</td>
<td>25.01</td>
</tr>
<tr>
<td>meS_class_LM</td>
<td>24.97</td>
</tr>
<tr>
<td>mePrmpLtHP_LM</td>
<td>24.89</td>
</tr>
<tr>
<td>mePrmpLtHP_LM</td>
<td>27.78</td>
</tr>
</tbody>
</table>

Table 3: Perplexity of various models on internal test set

Recognition results for various language models are presented in Table 4. For models which use the system prompt we have decoding results only for test set INT (we did not have the prompt information for test set EXT.)

<table>
<thead>
<tr>
<th>Model</th>
<th>WER (INT)</th>
<th>WER (EXT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM1</td>
<td>20.9</td>
<td>25.3</td>
</tr>
<tr>
<td>LM1_KNm</td>
<td>21.2</td>
<td>25.6</td>
</tr>
<tr>
<td>LM_base</td>
<td>20.5</td>
<td>25.2</td>
</tr>
<tr>
<td>LM_base_KNm</td>
<td>20.4</td>
<td>25.3</td>
</tr>
<tr>
<td>linS_LM</td>
<td>19.2</td>
<td>24.1</td>
</tr>
<tr>
<td>linS_LM_KNm</td>
<td>18.9</td>
<td>23.2</td>
</tr>
<tr>
<td>linPrmpLtClust_LM</td>
<td>19.2</td>
<td>-</td>
</tr>
<tr>
<td>linPrmpLt_LM</td>
<td>19.8</td>
<td>-</td>
</tr>
<tr>
<td>linPrmpLtClust_LM_KNm</td>
<td>18.5</td>
<td>-</td>
</tr>
<tr>
<td>linS_unbLj_LM</td>
<td>19.5</td>
<td>24.2</td>
</tr>
<tr>
<td>meS_LM</td>
<td>20.3</td>
<td>24.5</td>
</tr>
<tr>
<td>meS_class_LM</td>
<td>20.2</td>
<td>24.9</td>
</tr>
<tr>
<td>mePrmpLtHP_LM</td>
<td>20.3</td>
<td>-</td>
</tr>
<tr>
<td>mePrmpLtHP_LM</td>
<td>20.3</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4: Word error rates for various language models

6. Summary

We studied various ways of using the state of the dialog in improving the performance of a language model (as measured by perplexity, and WER in recognition experiments). Even with a very small amount of data labelled with state (31k compared to a total of 300k ) we obtain an improvement in word error rate of 9.3% relative (1.9% absolute) on test set INT and 8.3% relative (2.1% absolute) on test set EXT. The best performance is obtained using linearly interpolated models trained on data specific to the state (the state was parser feedback for test set EXT and based on the system prompt for test set INT). While using the prompt information it was important to cluster the states to achieve good performance. Also in all cases while using state information, Kneser-Ney modified smoothing used to build the language model helped performance. Exponential models using either the parser feedback state or system prompts in the features failed to help performance in terms of the recognition rate (although in terms of perplexity the model built on hand-picked features from the prompt is comparable to some of the linearily interpolated state based models).

7. References