Weight Vector Tuning in a Natural Language Filtered-Phrase Decoder

Matthew Hannah and **Conrad Yeung**
Simon Fraser University
8888 University Dr
Burnaby, BC V5A 1S6
mmh3@sfu.ca and conrady@sfu.ca

Abstract

We focus on varying methods used to find the weight vector for the core given features. We will compare and contrast different methods used, as well as an overview of the system that was built to conduct the experiments. We will start with a description of the main algorithm and some of its components and modifications, then discuss the code and data used for the project. We will also describe our experimental setup and conclude with analysis of our results and ideas for future development.

1 Motivation

The motivation for our project came from trying to understand the most effective ways in tuning the weight vector used in the filtered-phrase decoding system (Collins, 2013). Although the amount of information contained in a weight vector is relatively little (in our case, just four floats) the amount of computation required to find the optimal weights can be quite high. Finding ways of automatically finding weights over a wide search space, using different search methods, forms the core of our approach. The weight vector is a small but critical part of models with feature scores (Hopkins and May, 2011).

2 Approach

The noisy channel model forms a basis for our approach.

2.1 Noisy Channel Model

A noisy channel model is used to find the most probable target language translation.

$$e^* = \arg\max_e Pr(e|f)$$

The best translation is the one considered most probably given the input.

$$e^* = \arg\max_e \sum_a Pr_{\text{TM}}(f|a,e) * Pr_{\text{LM}}(e)$$

We can sum over all alignments to effectively ignore alignments, since every one is considered in the sum. The translation model is the model used to transform sentences from original to target languages, and the language model is the model used to judge the fluency and correctness of the candidate answers.

2.2 Phrase Based Lexicon

A phrase based lexicon is a set of tuples that each contain three components (Collins, 2013). These are a list of at least one word in the original language, a list of at least one word in the target language, and some sort of heuristic score that denotes the result of some processing on the text.

The two lists of words (original and target) do not need to have indentical numbers of words or characters, giving this type of model much more flexibility than simpler word-to-word models. Many sentences with the same semantic content can be represented in phrases of different lengths - being able to handle this is one of the main advantages compared to earlier models.

In a simple example of a phrase based lexicon, a rudimentary scoring system might simply be the observed count of each entry (Collins, 2011).

$$count(f, e) = \frac{count(e, f)}{count(e)}$$

We use log-likelihood probabilities to prevent underflow and allow addition of probabilities instead of multiplication. Since logarithm is monotonic, values that maximize the log-likelihood will also maximize the original objective function (Lopez, 2013). Since we have access to more feature scores than simple frequency, we can extend this and improve the performance of the system.

2.3 Language Model

An ARPA-format language model is used by the program to represent its idea of fluency in the target language. The ARPA format files contain the count of the total amount of ngrams of each length, along with a list of the ngrams and their associated scores, and a back-off weight for unigrams which allows easier processing at the edges of phrases. (Katz, 1987)

Sets of training data of different sizes were used in the development of our algorithm. Smaller sets were used for development and testing, and larger ones for producing final results. The English Gigaword Corpus was the largest source used.

2.4 Translation Model

The translation model used is a filtered phrase table. Unlike simple unigram or word-to-word models, whole phrases are contained in the table, often with meaningful semantic content in a single phrase. This increase in the size of text requires more processing and computational resources, but smooths out many inconsistencies in generating natural language output (MacCartney et al., 2008).

Each example contains a piece of target language text, a piece of original language text (in this case Chinese and English respectively) and the score for any number of features that apply to that phrase. The training data contains four feature scores per phrase. For more information on the language model and translation model, see the Data section.

2.5 Weight Model

If the translation model contained only only feature score per phrase pair, we could simply use that score to determine which candidate phrase is preferred. With multiple scores, we have the option of applying some weight vector to bias the results in favor of some weights over others. We use a separate file containing a list of floats that correspond to the weight of each feature. For more information, see the Code section.

2.6 Algorithm Description

Our system first loads working copies of the language model, translation model and weight model from their respective files. The working copies are also populated with any unknown words encountered in the data, initialized (in this case) with scores of 0.0.

The algorithm next examines different possible candidate phrases after pruning the parent stack, evaluating them and selecting a winner based on feature scores. The pruning size varies based on the size of the stack, in order to allow smaller stacks to require fewer resources. As mentioned before, we use a weight vector to place different priority on the weights. We will return to the weights vector later and discuss some methods for finding optimum values. At this point we can simply treat the weight vector as a series of *N* floats (one per feature), such that for each float n:

$$0 \leq n \leq 1$$

that are multiplied by their corresponding feature during scoring. The result is a target-language sentence that the algorithm has judged to be the most likely translation given the training data it was supplied.

2.7 Weight tuning

Each feature has a corresponding weight that determines how relatively important it is compared to the others. Having a well tuned weight vector improves performance; likewise, a poor vector will result in much worse performance. We compare results from using different search algorithms to find the optimal weights.

2.8 Reranking Feedback loop

Our reranker takes hypotheses from the decoder's output and ranks them according to their perceived accuracy. This information is fed back into the decoder, allowing it to alter its parameters, before providing new outputs to the reranker. Over time this will converge to a set of weights, providing a method for doing so. We also implemented a Pairwise Ranking Optimization algorithm (Hopkins and May, 2011) due to its versatility and ease of deployment.

3 Data

3.1 Language Models

Files used:

- en.gigaword.3g.arpa.gz
- en.gigaword.3g.filtered.train_dev_test.arpa.gz
- en.gigaword.3g.filtered.dev_test.arpa.gz
- en.tiny.3g.arpa

The language models used are from the English Gigaword Corpus (Graff and Cieri, 2003) which are taken from the Agence France Press English Service, The Associated Press Worldstream English Service, The New York Times Newswire Service and The Xinhua News Agency English Service. The full corpus is contained in data/lm/en.gigaword.3g.arpa.gz. The small version is used for convenience during development.

3.2 Translation Models

Files used:

- data/large/phrase-table/moses/phrase-table.gz
- data/medium/phrase-table/phrase-table.gz
- data/small/phrase-table/moses/phrase-table.gz
- data/toy/phrase-table/phrase_table.out

The translation models used are filtered phrase tables in a MOSES-compatible format, with each phrase having four feature scores given. Similar to the language models, the pruned versions are used for ease of development.

3.3 Input files

Files used:

- all-cn-en.cn
- all-cn-en.en0
- all-cn-en.en1
- all-cn-en.en2
- all-cn-en.en3

The input files consist of the Chinese text along with 4 independent translations.

4 Code

The decoding algorithm is of central importance to our system, and it is largely taken from the code we submitted for Assignment 4. Several modifications were made in order to make it as useful possible: simple ones like the ability to load large compressed files, and relatively more complicated ones like allowing variable weights. We also used the bleu.py file provided for Assignment 5 to calculate BLEU scores.

4.1 Examining a hypothesis in our decoder

A hypothesis is a named tuple containing the metadata and text of a phrase.

```
start_hypo = hypothesis(0.0, lm.begin(), None,
None, cover, 0, 0, [0])
```

Above is a hypothesis at initialization. lm.begin() is the probability of opening the phrase.

The logprob variable is used for single feature scores and is a result of first developing the system with only one score. It is kept in to allow it to interface with some of our older code. The lm_state variable is used for the language model score. The predecessor, coverage, start and end variables are used to keep track of positioning. The phrase variable contains the actual phrase text. The logprobarray variable is added to store an array containing the individual feature scores for the phrase if a multi-feature system is being used.

4.2 Weight Calculations

We use a weight vector as described early to bias the system in favor of some weights over others. The weight can be any number in the real number space. The weights are simply multiplied, so a weight of 1.0 has no effect on a feature, and a weight of 0 negates a feature completely.

```
weight_vector = [1.0, 1.0, 1.0, 1.0]
```

This default vector has no effect initially, but after training can determine which features are more informative at determining correct phrase translations. A key step in our learning process is modifying this weight vector based on the previous results and the particular search algorithm being used.

```
result = 0
for x in len(logprobs):
 result += logprobs[x] * weight_vector[x]
return result
```

This is an example of a straightforward weight vector model that takes account of individual weight scores. It would be possible for the behaviour in this section to become more intricate if necessary. The dot-product of the weight and logarithm probability arrays (or any other heuristic) other actions could act as thresholds to trigger additional processing. We will discuss this more in the concluding section.

4.3 Variable prune size

In the decoder, a limit on the amount of computing resources available necessitates pruning large stacks to avoid large exponential expansions. Although the system performs fine with a static prune size, it can be improved by variable contracting and expanding as required. Possible improvements to the pruning section will be discussed later.

```
pruneSize += int(round(len(stack) / 50))
```

This is a simple variable beam search that increases based on stack size. The following is a more complicated variable beam search that increases if sum of the feature scores of the first X elements is below some constant R.

```
pruneSize = opts.s
cumulative_score = 0
for x in range(X):
  cumulative_score += stack[x].logprob
if (cumulative_score < R):
  pruneSize = prunzeSize * 2
if (pruneSize > len(stack)):
  pruneSize = len(stack)
```

4.4 Output formatting

In order to produce output that was easy for our reranker to process, we output the sentence count, most probably english translation, and the log probabilities for applicable features.

Although we used four features and therefore four feature weights in our system, it would be trivial to extend this to an arbitrary number of features and weights, simply by increasing the size of the weight vector array.

The basis for both the reranker and decoder assignment code is developed from the default files provided by the course instructor. In order to calculate BLEU scores, we used the bleu.py module. To decompress .gz files we use the python gzip module. We also use a models file that is based on previous assignment work.

5 Experimental Setup

For our experiments, we used personal servers to process data and train the model. We wrote custom code to iteratively run the decoder on given data and weight vector, retrieve the BLEU scores from the algorithm, and use that information to make necessary adjustments to the weight vector for the next iteration. Our setup is quite standard and would be easily to replicate with any moderately powerful workstation.

5.1 Initial Weights

`[0, 0, 0, 0]`

With all weights at 0, no features will be counted and the system should perform randomly. As expected the system did not produce any meaningful output.

`[1.0, 1.0, 1.0, 1.0]`

For a baseline, we ran the system on initial weights (negating any effect) to see how much the results changed once weights were applied.

5.2 Manual tuning

`[0.1, 0.1, 0.1, 0.1]`

We select this vector to test, expecting that it should produce competency on the same level as the

initial vector. Although all weights are lower than the default weight vector, they are still relatively balanced and so should not have a large effect on the output of the algorithm.

`[0.1, 0.4, 0.3, 0.2]`

This vector is meant to create a hierarchy of feature scores. Features can be placed in relative importance to one another, denoting their influence on the overall score. Features that are consistently valuable can be prioritized over those that only apply in certain circumstances.

`[0.05, 0.85, 0.05, 0.05]`

Heavily favoring one feature over the others makes the system behave more like a single feature system, with the other features coming into play only in more unusual cases. Here, the second feature weight is much higher than the other three.

5.3 Automated Search

Consider the vectors *W* under the constraint

$$W = [w, 4w, 3w, 2w]$$

$$\{w \in \mathbb{R}, w > 0\}$$

the second weight vector in 6.2 fits this constraint, and so too would `[0.13, 0.52, 0.39, 0.16]` and infinitely many other vectors. The weight vector can be deconstructed into two components: the weight base *w* and the coefficients for each feature.

$$W = [\alpha w, \beta w, \gamma w, \delta w]$$

$$W = w[\alpha, \beta, \gamma, \delta]$$

This is highly useful going forward if we wanted to add more advanced weighting techniques based on feature score thresholds, with the weight base *w* acting as the scale of the vector, and the individual coefficients representing the direction in each dimension. For now we can treat them as simple floats in our iterative training.

The feedback loop is fairly simple since the output score is a real number that increases monotonically with accuracy according to the ranking algorithm (although not necessarily according to natural language users).

6 Results

6.1 Search Algorithms

There are many search algorithms that can deal with a problem that is so easily quantifiable. Brute force search is obviously suboptimal and dealing with floating point integers means it is a poor choice. Choosing a heuristic that takes into account the linguistic competency of the decoder allows us to avoid many of the pitfalls of more naive search methods. Using reranker feedback is a dependable technique that will tune weights (Hopkins and May, 2011). A simplified version of our iterative reranker feedback loop follows.

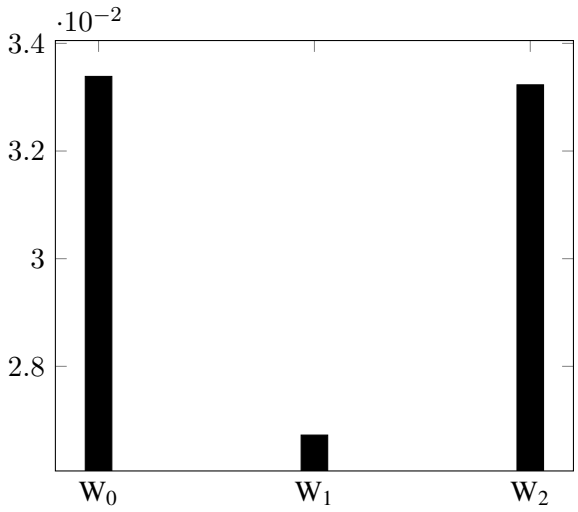
```
W < weights.file
for each epoch:
 decode.py(LM, TM, W, input) > nbest
 learn.py(nbest) > W
decode.py > output
weights.file < W
```

6.2 BLEU Scores for manually tuned weights

$$W_0 = [0.2, 0.2, 0.2, 0.2]$$

$$W_1 = [0.2, 0.45, 0.2, 0.2]$$

$$W_2 = [0.2, 0.1, 0.2, 0.2]$$

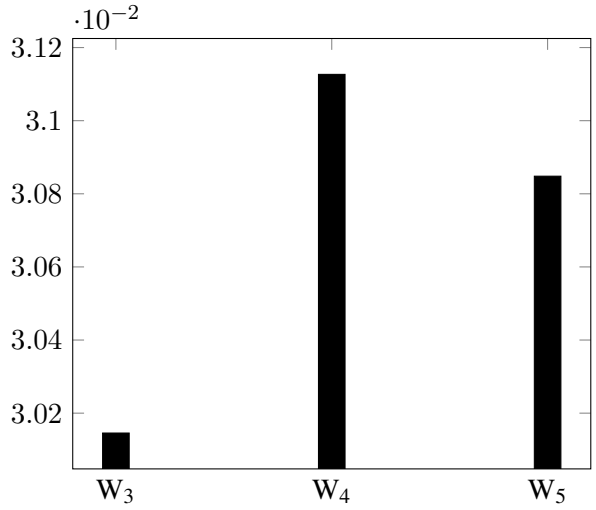


6.3 BLEU Scores for automatically tuned weights

$$W_3 = [0.42, 0.55, 0.26, 0.44]$$

$$W_4 = [0.15, 0.96, 0.34, 0.36]$$

$$W_5 = [0.47, 0.12, 0.22, 0.52]$$

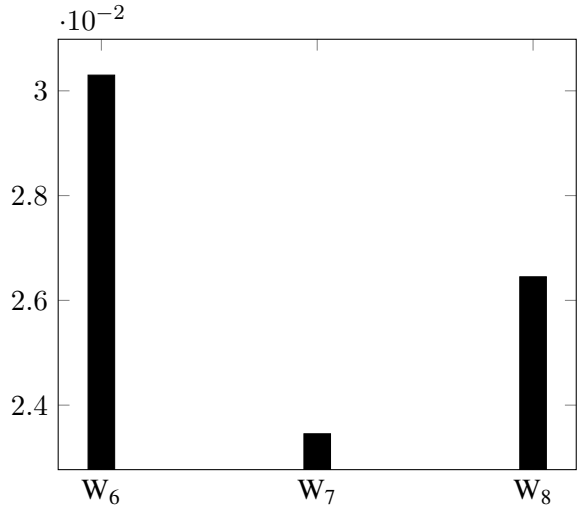


6.4 Random Walk

$$W_6 = [0.2, 0.25, 0.2, 0.2]$$

$$W_7 = [0.5, 0.4, 0.5, 0.4]$$

$$W_8 = [0.3, 0.5, 0.3, 0.5]$$



7 Analysis

Automated searching for weight vectors is clearly far superior to either random or manual design. The fact that a well performing weight vector requires the same amount of resources to compute than a relatively poor one means that even a small benefit in the weight tuning could result cumulative benefits over time.

8 Conclusion and Considerations for Future Work

Further work could be done to improve the system in several ways. In this paper we compared different search methods for finding weights, but there are many other possibilities that could be considered for further comparison. The decoder can increase the maximum allowable stack size, and also increase the number of elements that survive pruning. This can have a positive effect on the results at the cost of extra computation time. Other methods could be used to determine stack size besides the ones described earlier. One promising idea would be to use a neural net to tune the weights iteratively, but the demanding processing and data requirements left that out of reach. By allowing for complex processing and the addition of more features, it is possible that better results could be achieved.

References

Michael Collins. 2011. *Statistical Machine Translation: IBM Models 1 and 2*. Columbia University.

Michael Collins. 2013. *Phrase-Based Translation Models*. Columbia University.

Mark Hopkins and Jonathan May. 2011. *Tuning as Ranking*. SDL Language Weaver.

Slava M. Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(3).

Adam Lopez. 2013. *Word Alignment and the Expectation-Maximization Algorithm*. Johns Hopkins University.

Bill MacCartney, Michel Galley, and Christopher Manning. 2008. *A Phrase-Based Alignment Model for Natural Language Inference*. Stanford University.