Appendix B

Documentation of the Maximum Entropy Toolkit

One of the contributions of this dissertation is the production of a toolkit for the training and use of maximum entropy models. This appendix presents a brief design document for our toolkit.

Maximum entropy methods can be used to estimate a probability distribution on the joint space of histories $\mathcal{X}$ and futures $\mathcal{Y}$. However, our toolkit is designed to estimate and evaluate only conditional probability models, because we focus on applications in natural language processing, which need conditional models rather than joint ones. In Section B.1, we first describe the implementation details of the training and evaluation algorithms. The former is very complicated, whereas the latter is straightforward. Therefore, we focus on the former. In Section B.2, we present the major data structures used in our programs.

B.1 Implementation of Hierarchical Training Algorithms

The training procedure of an ME model is split into two phases: computing the feature expectations and updating the model parameters. We use a concrete example
of the composite model (6.1)

\[
p(w_i | w_{i-2}, w_{i-1}, h_{i-2}, h_{i-1}, n_{ti-1}, n_{ti-2}, t_i) = \frac{1}{z} \alpha_{w_i} g(w_i | w_{i-1}, w_i) \cdot \alpha_{w_{i-1}} g(w_{i-2}, w_{i-1}, w_i) \cdot \alpha_{h_{i-1}} g(h_{i-1}, w_i) \cdot \alpha_{h_{i-2}} g(h_{i-2}, h_{i-1}, w_i) \\
\alpha_{n_{ti-1}} g(n_{ti-1}, w_i) \cdot \alpha_{n_{ti-2}} g(n_{ti-2}, n_{ti-1}, w_i) \cdot \alpha_{t_i} g(t_i, w_i)
\]
described in Chapter 6 to show how our training programs estimate the model parameters.

B.1.1 Feature Expectation

In hierarchical training algorithms, the feature expectations (cf. Equations (3.44), (3.45) and (3.46)) and the normalization factors (cf. Equation (3.5)) are computed recursively. In principle, the recursive depth depends on the highest order of models. However, we fix the recursive depth to three for all models, so that the training program will not become too complicated. We will show in Section C.5 how to deal with features of order four or above. In this and the following appendices, we use \( x \) and \( w_i \) to represent the history and the future token, respectively.

When training the composite model, the computation in the first recursion is history independent and involves only unigram features, that in the second recursion depends on history equivalence classes \((w_{i-1}, h_{i-1}, n_{ti-1})\) and involves both unigram and bigram features, and that in the third one depends on the entire history \( x = w_{i-2}, w_{i-1}, h_{i-2}, h_{i-1}, n_{ti-1}, n_{ti-2} \) and involves all features. The feature expectations can be gathered from three parts, each of which is obtained from one step above. In our programs, we denote partial expectations obtained from these recursions as \texttt{PartEi.w}, \texttt{PartEi.phi} and \texttt{PartEi.x}, respectively. It is obvious that the computation of unigram feature expectations is involved in all three recursions and the expectations of bigram features and trigram features are obtained from two parts: \texttt{PartEi.phi} and \texttt{PartEi.x}, and one part \texttt{PartEi.x}, respectively.

It has also been noted in Chapter 2 that the normalization factor \( z \) is required for computation of feature expectations. The computation of \( z \) is also split into three summations involving only unigrams, unigrams and bigrams, and all features,
respectively. We denote the values obtained from these recursions as $z_{\text{of}, u}$, $z_{\text{of}, \phi i}$ and $z_{\text{of}, x}$, respectively.

Our programs compute the expectations for all features in four steps described below.

**Step 1: Loading global data and initialization.**

- Load all features, including $U/B/T$ regular unigrams/ bigrams/ trigrams, $B_h/T_h$ head-word bigrams/ trigrams, $B_n/T_n$ non-terminal bigrams/ trigrams, and $U_t$ topic-dependent unigrams, from eight different files. Features in these files are sorted alphabetically according to their coefficients.

  Assign index to each feature, e.g., index from 0 to $U - 1$ ([0 : $U - 1$]) for unigrams, [$U : U + U_t - 1$] for topic-dependent unigram features, $[U + U_t : U + U_t + B - 1]$ for regular bigrams and $[U + U_t + B : U + U_t + B + B_h - 1]$ for head-word bigrams, etc.

  Set the initial values of zero to the expectations$^1$ for all features.

- Read all model parameters $\alpha$ (of the last iteration) and their target expectations from the model file, whose format will be introduced in Appendix C, and then map features to their corresponding parameters.

Next, for each topic (or part of training set) $t = 1, \ldots, T$, do steps 2-4.

**Step 2: Computing $z_{\text{of}, u}$, the history-independent part of $z$, and, for each equivalence class $\phi(x) = w_{i-1}, h_{i-1}, n_{t-1}$ observed in the training data, computing $z_{\text{of}, \text{class}}$, the part of $z$ depending only on history equivalence classes.**

- Merge two unigram feature parameters, $\alpha_{w_i}$ and $\alpha_{t,w_i}$, to

  $$ \text{prod}_m \text{alphas} = \alpha_{w_i} \alpha_{t,w_i}. $$

  Compute the history-independent unigram term $z_{\text{of}, u}$ in the normalization factor $z$ by

  $$ z_{\text{of}, u} + = \text{prod}_m \text{alphas} $$

  for all $w_i$.

---

$^1$Since a feature $g$ in the composite model may apply simultaneously with seven other features, it needs $\max(g_{\#}) = 8$ accumulators for the expectation.
For each history class $\phi(x) = w_{i-1}, h_{i-1}, n_{i-1}$:

- Enumerate all $w_i$ that have any bigram feature activated, and for each $\langle \phi(x), w_i \rangle$, compute

$$\text{prod}_c\text{alphas} = \alpha_{w_{i-1}, w_i} \cdot \alpha_{h_{i-1}, w_i} \cdot \alpha_{n_{i-1}, w_i} - 1$$

where $\alpha$ equals one if the corresponding $g$ is zero.

- Compute $z_{\text{of class}}$ for the history class $\phi(x)$ by

$$z_{\text{of class}} + = \text{prod}_c\text{alphas} \cdot \text{prod}_c\text{alphas}$$

for all $w_i$.

**Step 3: Computing $z$ for each history, and gathering PartEi_\text{x}, the partial feature expectation.**

- For each history $x \in \hat{X}$:

  - Find all words $w_i$ that have any trigram feature activated, and for each $w_i$ compute

    $$\text{prod}_c\text{alphas} = \alpha_{w_{i-2}, w_{i-1}, w_i} \cdot \alpha_{h_{i-2}, h_{i-1}, w_i} \cdot \alpha_{n_{i-2}, n_{i-1}, w_i} \cdot (\alpha_{w_{i-1}, w_i} \cdot \alpha_{h_{i-1}, w_i} \cdot \alpha_{n_{i-1}, w_i} - 1)$$

  - Accumulate $z(x)$ by

    $$z_{\text{of x}} + = \text{prod}_c\text{alphas} \cdot \text{prod}_m\text{alphas}$$

    for all $w_i$, and then

    $$z_{\text{of x}} + = z_{\text{of class}} + z_{\text{of u}},$$

    where $z_{\text{of u}}$ and $z_{\text{of class}}$ are precomputed in Step 2.

  - For each $w_i$ enumerated, collect expectations for all features\(^2\) applied to $\langle x, w_i \rangle$ by

    $$\text{PartEi}_x + = \frac{\hat{p}(x)}{z_{\text{of x}}} \cdot \text{prod}_c\text{alphas} \cdot \text{prod}_m\text{alphas}.$$  

**Step 4: Updating feature expectations by history-independent part PartEi_\text{u} and class-dependent part PartEi_\text{u}.**

\(^2\)Not only trigram features, but also unigram and bigram ones.
• For each history class \( \phi(x) \), fetch all \( w_i^3 \) that has any bigram feature activated. Update expectations for unigram and bigram features applied to \( \langle \phi(x), w_i \rangle \) by

\[
\text{PartEi} \quad + \quad \sum_{x \in \phi} \frac{\hat{p}(x)}{z_{\text{of } X}} \prod_{c} \alpha_{\phi(x)} \cdot \prod_{m} \alpha_{\phi(x)}.
\]

• For all unigrams (including topic-dependent unigrams), update feature expectations by

\[
\text{PartEi} \quad + \quad \sum_{x \in X} \frac{\hat{p}(x)}{z_{\text{of } X}} \prod_{m} \alpha_{\phi(x)}.
\]

### B.1.2 Merging Partial Feature Expectations and Updating Model Parameters

It is worth noting here that the feature expectations \( \text{PartEi} \) gathered above are only for one topic (or part) of any training sample. The overall values of these expectations are summations over all parts. Since we may use many machines to train the model in parallel and save partial expectations in different files, we need to merge these partial feature expectations before we use Newton’s method as described in Section 2.5.3 to update model parameters.

### B.1.3 Memory Concerns

In the algorithm we described above, the set of tokens \( w_i \) with some features activated for each history \( x \) (or history class \( \phi(x) \)) are dynamically generated. It may be argued that these words \( w_i \) need to be generated only once in the first iteration and saved to disk, and then reused for the remaining iterations. Actually, the ME toolkit of Ristad (1997) adopts this implementation. The major drawback, however, is the enormous space required to save all combinations of histories and future tokens. The ME toolkit of Ristad (1997) cannot even train a regular trigram model for Switchboard without high cut-offs for N-grams. Another problem is the extremely long overhead time of loading the huge amount of data from disk. Since the disk access time is

\[3\text{Already generated in Step 2.}\]
thousands of times longer than the memory access time, reading these data is even slower than generating them dynamically. Therefore, we always generate the word set for each history dynamically in our programs.

B.1.4 Computing Conditional Probabilities Using ME Models

We have discussed details of the computational issues when ME models are used and have described the efficient algorithms in Chapter 7. Here, we only emphasize that the procedure of using ME models is very similar to that of training ME models. We summarize the steps of computing probabilities using ME models below.

Step 1: Loading data and initialization.
The same as in training.

Step 2: Computing $z_{of\ u}$, which is history independent.
Step 3: For each test tuple $(x, w_i)$, compute $z(x)$ and prod_m_alphas,
prod_c_alphas, and set

$$p(w_i|x) = \frac{\text{prod}_m_{\text{alphas}} \cdot \text{prod}_c_{\text{alphas}}}{z(x)}$$

It should be noted that if $z(x)$ is already available in cache, it need not be computed again; otherwise, it is calculated similarly as in the training.

B.2 Data Structures

We describe major data structures (C++ classes) for constraints, model parameters, histories and futures. We only show major data members and functions in these C++ classes, because the complete classes (in header files) total thousands of lines. We exemplify the role of class members in the comment lines preceding or following these members.

B.2.1 Model parameters

An ME model contains a list (or an array) of “constraints,” which are defined as
class Constraint{
    // Collect feature expectations.
    // Index j is defined in Equation (2.28).
    void UpdatePart_Ei(int j, double e) {PartEi[j] += e;}

    // Update model parameters by the Newton method.
    void UpdateAlpha() {alpha= alpha * Newton(max_e_num);}

    // Other member functions, such as reading and writing
    // data members, are easy to implement and thus are omitted.

private:

    double target; // Target expectations.
    double alpha; // Model parameters.
    double E_i; // Feature expectations under the current model.

    // PartEi[ ] is the partial sum for the expectation of feature i.
    // MAX_C_NUM: Maximum # of constraints active simultaneously in
    // the model.
    double PartEi[MAX_C_NUM];

    // max_e_num is the maximum # of constraints active simultaneously
    // with this constraint.
    // For example, MAX_C_NUM = 8 in the composite model. However,
    // max_e_num = 1 if this feature never applies with other features.
    int max_e_num;
};

B.2.2 Features

No features used in this dissertation are more complicated than trigram features. Therefore, all features are represented as N-grams in the ME toolkit. We show how to store and access these N-grams efficiently using the example of bigrams.

Bigrams \((w_i, w_j)\) and their count \(c(w_i, w_j)\) can be stored in a matrix \(C_{i,j}\), where the subscripts \(i\) and \(j\) are the indices of token \(w_i\) and \(w_j\), respectively. Since this matrix \(C\) is extremely sparse, \(e.g.,\) less than 2\% of elements are non-zero in Switchboard, this implementation is extremely inefficient. Of course, the matrix can be compressed
in a hash table in which the key is \((w_i, w_j)\) and the value is \(c(w_i, w_j)\). Although a hash table needs only \(O(1)\) data access time for any \((w_i, w_j)\), it is not efficient for the operation of enumerating all \(w_j\) following \(w_i\), which is often used in the hierarchical training method. Therefore, we design the following data structure to store the bigram matrix.

![Data structure for bigrams](image)

**Figure B.1: Data structure for bigrams.**

Figure B.1 illustrates the bigram data structure used in our programs. We use two one-dimensional arrays to represent bigrams, one for index and one for content. The index array itself is indexed by \(w_i\), and it saves the position of the last bigram starting by \(w_i\) in the content array. For example, the first element (for \(w_0\)) in the index array is 2, meaning that the first two elements in the content array correspond to bigrams starting with \(w_0. w_j\), and the count \(c(w_i, w_j)\) will be stored as an element in the content array. To search for a bigram \(w_i, w_j\), we first find positions for all bigrams starting with \(w_i\) (from \(\text{index}[i-1]+1\) to \(\text{index}[i]\)), and then use binary search to
locate the position of the pair \( (w_j, c(w_i, w_j)) \). This implementation has the advantage of efficiently finding all words following \( w_i \) (in elements from \( \text{index}[i-1] + 1 \) to \( \text{index}[i] \) stored in the content array). Further, this implementation saves at least one-third of memory compared to a hash table. The typical bigram class is shown below.

```cpp
class Bigram{
public:
    int Search(int w_i, int w_j); // Search for bigram (w_i, w_j).
    int ReadBigram(const char* filename); // Read bigrams from a file.
    int GetConstraintId; // Assign each bigram a feature index.

private:
    int *idx; // Index array, sorted by w_j
    int *w2; // w_j's
    float *freq; // #[w_i,w_j]
    int *c_idx; // constraint id for bigram (w_i, w_j)
};
```

Trigrams \((w_i, w_j, w_k)\) are stored in the same way, the only difference is that the indices of trigrams are bigram \((w_i, w_j)\) instead of single words.

### B.2.3 History

The history class is the most important and also the most complicated class in the toolkit. It will dynamically generate all future tokens for a given history, and compute the normalization factor \( Z \). The core of the history class is a group of \textit{Setby} functions. These functions are the only model-dependent ones in the toolkit. Users can adding their own \textit{Setby} functions for training and evaluating their own models. Here, we list some of these functions but focus only on \textit{SetbyHighNgrams}, which is used in the algorithms we have described in Section B.1.

```cpp
class History {
public:
    int SetbySymNgram;
    int SetbyNTSyncNgram;
    int Setbyonly3gram;
    int SetbyTestNgram;
    int SetCache3gramTest;
};
```
... // Function SetbyHighNgrams will be used in Step 3 in the training // algorithms in Section B.1. // For each history, it enumerates all words that have some trigram // features. // Create a future class instance for each of these words. int SetbyHighNgrams(NTWorkSpace &temperate_word_set, NTHList &history_class_list, NgramHList &list_of_two_preceding_words, NgramHList &list_of_two_preceding_heads, NgramHList &list_of_two_preceding_nts, Unigram &list_of_unigrams, Bigram &list_of_regular_bigrams, Bigram &list_of_headword_bigrams, Bigram &list_of_nt_bigrams, Trigram &list_of_regular_trigrams, Trigram &list_of_headword_trigrams, Trigram &list_of_nt_trigrams, TupleTwo &list_of_training_tuples, HistoryClassList &history_classes_and_future_words, int &index_of_history_class_for_the_current_history);

void UpdateHighNgramEi();  // For implementing Step 3 in the // training algorithm.

// LM() will compute log probability for all words following this // history in either training and test data. // This function is not required to train ME models. However, we can // evaluate the log probabilities of the training data and see // if the training procedure converges. double LM(int);

// Create future classes for each word enumerated for this history. void SetMeFuture(int,MeFuture&);

// Compute the history dependent part of normalization factor z. void SetHighNgramZ(double, int);

// Other functions are omitted.

private:
  float p_of_x;  // p(x)
  MeFutureList futures;  // Future list for the history.
double z; // z(x).
int hc_id; // History class_id.
... // Other unimportant data members.
};

The C++ class of the history equivalence class \(\phi(x)\) described in the training
algorithm is very similar to the class described above and is thus omitted.

### B.2.4 Future

The set of words with some conditional features activated for a given history is
represented as a list (or an array) of futures, which are described in the class below.

class MeFuture{
public:
   // Each word may have more than one marginal constraint, e.g., a
   // regular unigram and a topic-dependent unigram. This function will
   // add a new constraint to the marginal constraint list.
   void AddMConstraint(Constraint);

   // Similarly, Each word may have more than one conditional
   // constraint.
   // This function will add a new constraint to the conditional
   // constraint list.
   void AddCConstraint(Constraint);

   double SetCProd(ConstraintList&); // Compute \(\text{prod\_c\_alphas}\).
   double SetMProd(); // Compute \(\text{prod\_m\_alphas}\).

private:
   float count; // \([x, w_i]\]
   int c_n; // Number of conditional features activated for \(<x, w_i>\)
   int m_idx; // Unigram feature id.
   int* c_idx; // Conditional feature ids.
   double prod_c_alphas; // Product of all conditional features.
   double prod_m_alphas; // Product of all marginal features.
};

Figure B.2 shows the relations among the history class, the future class and the
constraint class. In this example, the history class has a list of \(M\) futures, and each
future has a list of pointers to the constraint list.

Figure B.2: History, future and constraint classes.