2 Question 2

Provide an overview of existing part-of-speech tagging algorithms, focusing on the question how unknown words (i.e., words not in the dictionary or the training data) can be handled by those algorithms.

Part-of-speech (POS) tagging refers to the task of annotating each word in a text with its appropriate POS tag, where the tags to be used are pre-defined in a tagset. A POS tagger takes a raw text \( W \) as input and produces a sequence of tags \( T \) as output. The tagging procedure can be formalized as follows:

\[
\phi : W \rightarrow T, \phi(w_i) = t_i, \quad t_i \in \{ g \mid g \text{ a possible POS tag for } w_i \}, \quad \forall i \ 1 \leq i \leq |W|
\]

where \( \phi \) denotes the tagging procedure, \( W \) denotes the raw text input, \( T \) denotes the sequence of POS tags assigned to \( W \), \( w_i \) denotes the \( i \)th word in \( W \), \( t_i \) denotes the \( i \)th tag in \( T \), \( g \) denotes a tag defined in the tagset used for tagging, and \( |W| \) denotes the length of \( W \).

Tagging ambiguities and unknown words constitute the two major sources of difficulty for POS tagging. Whereas some words may have only one allowable POS category, many words can have two or more possible POS categories. The POS tagger needs to determine the correct tag for each instance of an ambiguous word in the text. Unknown words call for special care, as we cannot directly obtain first-hand information on their POS categories from the training data and/or dictionary. Therefore, the POS tagger needs to have some sort of mechanism to guess the POS categories of unknown words. As discussed in section 1, the state-of-the-art POS taggers generally perform significantly worse on unknown words than on known words.

Many methods have been proposed for POS tagging in the literature. Depending on whether they require pre-tagged training data, they can be supervised or unsupervised (Manning and Schütze, 1999). In supervised methods, pre-tagged training data are used to train a POS tagger, which is then used to tag new texts. Unsupervised methods do not require pre-tagged training data, but rely on dictionary information and attempt to learn the regularities of tag sequences using some complicated computational model. Supervised methods generally perform better than unsupervised methods, but the construction of pre-tagged training data is often a costly and time-consuming process, and in cases where such data do not exist (e.g., for an understudied language) or are hard to find (e.g., for texts in a special domain), the need for unsupervised methods arises. In what follows, we discuss the commonly used supervised and unsupervised methods for POS tagging and how unknown words are handled in each of these methods.
2.1 Supervised Methods
2.1.1 Markov Model Taggers

Markov Model taggers (Church, 1988; DeRose, 1988; Weischedel et al., 1993; Charniak et al., 1993; Brants, 2000) generally seek to maximize two probabilities in determining a tag. These are 1) contextual probability, i.e., the probability of tag given its preceding tags, and 2) lexical probability, i.e., the probability of a tag given a word.

To compute the most likely tag sequence \( T \) for a given word sequence \( W \), we want to find \( \arg \max_T p(T|W) \), i.e., we need to maximize \( p(T|W) \). We can derive the following equation using Bayes Rule:

\[
(17) \quad p(T|W) = \frac{p(T)p(W|T)}{p(W)}
\]

where \( p(T) \) is the prior probability for \( T \), \( p(W|T) \) is the conditional probability of \( W \) given \( T \), and \( p(W) \) is the prior probability of \( W \). Given that \( W \) is fixed for any text input, we have

\[
(18) \quad \arg \max_T p(T|W) = \arg \max_T p(T)p(W|T)
\]

In other words, we only need to maximize \( p(T)p(W|T) \). Assuming the current tag depends on all its previous tags, and the current word depends on its current tag as well as its previous tags and words\(^{10} \), we can compute \( p(T)p(W|T) \) as follows:

\[
(19) \quad p(T)p(W|T) = \prod_{i=1}^{n} p(t_i|t_{i,i-1})p(w_i|t_{1,i}, w_{1,i-1})
\]

where \( t_{i,j} \) and \( w_{i,j} \) denote the \( i \)th tag to the \( j \)th tag in \( T \) and the \( i \)th word to the \( j \)th word in \( W \) respectively \((1 \leq i < j \leq |W|, |T| = |W|)\). As this is not easily computable in practice, the following further assumptions are made:

- the probability of a tag depends only on its previous tag (limited horizon), i.e., \( p(t_i|t_{i,i-1}) \) is reduced to \( p(t_i|t_{i-1}) \)\(^{11} \).
- words are independent of each other, and the probability of a word only depends on its tag, i.e., \( p(w_i|t_{1,i}, w_{1,i-1}) \) is reduced to \( p(w_i|t_i) \).

We can then derive the following equation:

\(^{10}\)The assumption about the current word may come in different variations. To name a few, we can assume it depends on the complete tag sequence, its previous tags and words (excluding the current tag), its previous words, etc. (Charniak et al., 1993).

\(^{11}\)For a first-order Markov model. More previous tags can be considered if higher-order models are desired.
In general, Markov models model a linear sequence of events as a sequence of states, and use the following three probabilities to compute the probability of a state sequence:

- probabilities of initial states,
- transition probabilities, i.e., the probabilities to go from one state to another, and
- generation probabilities, i.e., the probabilities for different symbols to be emitted by a state.

In the case of POS tagging, states are represented as tags, and symbols are represented as words. The probabilities of initial states are simply the probabilities for different tags in the tagset to appear in the beginning of a tag sequence. The two probabilities on the right hand side of equation (20) correspond to the transition probabilities, i.e., the probability of seeing one tag after another tag, and generation probabilities, i.e., the probability of seeing a word given a tag. These two probabilities can be estimated from the tagged training data as follows:

\[
p(t_k | t_j) = \frac{C(t_j, t_k)}{C(t_j)}
\]

\[
p(w_j | t_j) = \frac{C(w_j, t_j)}{C(t_j)}
\]

where \( t_i \) denotes the \( i \)th tag in the tagset, \( w_i \) denotes the \( i \)th word in the lexicon (which consists of all the words in the training corpus), \( C(t_j) \) denotes the frequency of \( t_j \), \( C(t_j, t_k) \) denotes the number of times \( t_k \) follows \( t_j \), and \( C(w_j, t_j) \) denotes the number of times \( w_j \) is tagged as \( t_j \) in the corpus.

Once we have estimated the transition and generation probabilities from the training data, we can try to find the best tag sequence for any text input. In theory, we need to compute \( p(W | T)p(T) \) for all possible tag sequences for the text input in order to find \( \arg_{T} p(W | T)p(T) \), but this is computationally expensive, as multiple ambiguous words in the input can give rise to a huge number of possible tag sequences. Therefore, in practice, the Viterbi algorithm (Viterbi, 1967) is used, which keeps the \( k \)-best paths at each state instead of considering all possible paths.

Two problems need to be addressed for Markov model taggers. The first of these is the data-sparseness problem. Regardless of the size of the corpus, some tag sequences and rare words may not appear in the corpus, and assigning them zero probability will cause a complete candidate tag sequence for the input to have a zero probability, which is not desirable. This problem is handled with smoothing techniques, which assign some small non-zero transition or generation probabilities to unseen cases. A number of smoothing
techniques exist, e.g., Good-Turing (Good, 1953; Church and Gale, 1991; Gale and Samp-
son, 1995), linear interpolation (Chen and Goodman, 1996; Brants, 2000), Adding One (Church, 1988), back-off (Katz, 1987), etc.

The second problem is unknown words. As these words are not found in the training
data, we need to estimate their lexical probabilities in some other way. This is generally
done using morphological, contextual, and other useful cues. We discuss two representa-
tive methods in detail here. Weischedel et al. (1993) proposed to use the following four
independent categories of features for this purpose: inflectional endings (-ed, -s, -ing),
derivational endings (e.g., -ion, -al, -ly, etc.), hyphenation, and capitalization (including
information on whether the word is the first word of a sentence). The following equation
is used to compute $p(w_i|t_i)$ for unknown words:

$$p(w_i|t_i) = p(\text{unknown word}|t_i) \ast p(\text{Capital - feature}|t_i) \ast p(\text{endings/hyph}|t_i)$$

The probabilities involved are estimated from tagged training data.

Brants (2000) used the linear interpolation of fixed length suffix model to estimate the
lexical probabilities for unknown words. Here, suffix is loosely defined as the last $k$
characters of a word. The probability distribution for various suffixes are estimated from words
in the training data that have the same suffix of a predefined maximum length. These prob-
obabilities are smoothed by successive abstraction. This calculates $p(t^j|l_{n-m+1,n})$, i.e., the
probability of a tag $t^j$ given the last $m$ letters of an $n$-letter word. Probabilities of $t^j$ given
increasingly shorter substrings of the suffix, i.e., $p(t^j|l_{n-m+2,n}), p(t^j|l_{n-m+3,n}), \ldots, p(t^j)$
are used for smoothing. The following recursion formula is used to calculate these proba-
bilities:

$$p(t^j|l_{n-i+1,n}) = \frac{\hat{p}(t^j|l_{n-i+1,n}) + \theta_i p(t^j|l_{n-i,n})}{1 + \theta_i}, \quad 0 \leq i \leq m$$

using the initialization calculated in (24a), maximum likelihood estimates $\hat{p}$ calculated
in (24b), and weights $\theta_i$ calculated in (24c):

$$p(t^j) = \hat{p}(t^j)$$

$$\hat{p}(t^j|l_{n-i+1,n}) = \frac{C(t^j,l_{n-i+1,n})}{C(l_{n-i+1,n})}$$

$$\theta_i = \frac{1}{s} \sum_{j=1}^{s} (\hat{p}(t^j) - \bar{p})^2, \quad 0 \leq i \leq m - 1$$

$$\bar{p} = \frac{1}{s} \sum_{j=1}^{s} (\hat{p}(t^j))$$

where $s$ is the size of the tagset. The weights are set to the standard deviation of the
maximum likelihood probabilities of the tags in the training corpus. These probabilities
are calculated for lower-case and upper-case respectively, and the various parameters, such
as the value of $m$, what words to include for training, etc., are determined empirically.
Feng (2001) used a Markov model for tagging Chinese and reported a precision of 96.87%. However, no mention was made on the performance on unknown words or how unknown words were handled. In principle, however, lexical probabilities for Chinese unknown words can also be estimated using various information about the unknown words, including morphological structures, prefixes and suffixes, the existence of special characters, etc.

2.1.2 Transformation-Based Error-Driven Learning

Transformation-based part-of-speech taggers (Brill, 1994, 1995a) automatically learn non-stochastic rules from corpus for tagging. These rules capture relevant linguistic knowledge in a direct and clear way, which has been argued to be an advantage over statistical approaches.

To train a transformation-based tagger, we need to have a tagged training corpus (the truth) as well its corresponding unannotated version. We then need to specify the following three components. The first component is the initial tagger, which is used to produce an initial tagging of the unannotated training corpus. There is no special requirement about the initial tagger: it could be a simple tagger that assigns each word its most likely tag, or some sophisticated statistical tagger.

The second component is the space of allowable transformations. The training process produces an ordered list of rules to be applied to new text for tagging, but we need to specify what kinds of transformations can be learned. Each transformation consists of a rewrite rule and a triggering environment. This can be represented in the form in (25), which specifies that Tag$_i$ should be changed to Tag$_j$ in context $C$.

(25) $\text{Tag}_i \rightarrow \text{Tag}_j$ in context $C$

A set of transformation templates are used to define the transformations the learner is allowed to examine. One example transformation template is given in (26):

(26) $\text{Tag}_i \rightarrow \text{Tag}_j$ when the word two before (after) is tagged $z$.

The transformation templates in Brill (1994) allowed tags to be changed based on the previous (following) three tags and the previous (following) two words. An example transformation from Brill (1994) is given in (27):

(27) Change the tag from preposition to adverb if the word two positions to the right is as.

The third component, the scoring function, is used to compare the corpus to the truth and determine which transformation should be learned. A common scoring function used is the number of errors reduced after applying a transformation.
The training process is iterative and takes place as follows. At each iteration, the learner applies each possible instantiation of the transformation templates to the text (starting with the text tagged by the initial tagger), counts the number of tagging error reductions each transformation can achieve, and chooses the transformation that achieved the greatest number of error reductions. That transformation is then applied to the text, and the learning process repeats, until no more transformations reduce errors beyond a pre-determined threshold. Once the system is trained, a new sentence is tagged by first applying the initial tagger and then by applying the learned transformations to the sentence in the right order.

In Brill (1994), the initial tagger naively tags each unknown word as proper noun if it is capitalized and common noun otherwise. He then built a transformation-based learner to learn rules specifically for tagging unknown words. This basically involves the design of a separate set of allowable transformations that will only be applied to unknown words. These transformations make use of three types of information: the unknown word’s prefix and suffix (loosely defined as the first or last $k$-letters of the word); the word immediately before or after the unknown word; and whether a particular character appears in the word. Some of the allowable transformations for unknown words from Brill (1994) are:

(28) Change the tag of an unknown word from $X$ to $Y$ if:
- Deleting the suffix $x$, $|x| \leq 4$, results in a word.
- The last $k$ characters of the word are $x$, $1 \leq k \leq 4$.
- Word $W$ ever appears immediately to the left (right) of the word.
- Character $Z$ appears in the word.

One example transformation quoted in Brill (1994) is given below:

(29) Change the tag from common noun to adjective if the word has suffix -al.

Meng and Ip (1999) used a transformation-based tagger for Chinese. Similar to Brill (1994), the allowable transformations for unknown words there also made use of information about 1) the prefix and suffix of the unknown word (loosely defined as the first and last character of the word), 2) the word immediately before or after the unknown word, and 3) particular characters that are components of the unknown word. An example template from Meng and Ip (1999) is given below:

(30) Change the tag from $X$ to $Y$ if word $W$ occurs to its left/right.

Meng and Ip (1999) reported that accuracy for unknown words was in the range of 40% to 50% (depending on the domain of the text), although the tagger achieved an overall accuracy of 87%.

Feng (2001) and Florian and Ngai (2001) have also developed a transformation-based tagger for Chinese, and they reported an accuracy of around 95% and 88% respectively, but neither of the papers discussed the treatment of or performance on unknown words.
2.1.3 Maximum Entropy

Like other taggers, the maximum entropy tagger (Ratnaparkhi, 1996, 1998) also attempts to use various pieces of contextual evidence to estimate the probability for a word to have a particular POS tag within a certain context. The philosophy underlying this model is that to make inferences based on partial information, we should use a probability model with maximum entropy subject to whatever is known, as assuming anything that is unknown makes the model biased (Jaynes, 1957; Good, 1963; Ratnaparkhi, 1996). Therefore, among the probability models that are compatible with the evidence we have, this approach seeks to find the model with maximum entropy beyond the evidence we have.

Evidence is represented with contextual predicates and features. Let $T$ denote the tagset, and $H$ denote the set of possible contexts (histories) we can observe from the data, contextual predicates and features are functions of the form in (31a) and (31b) respectively:

\[(31)\]
\[\begin{align*}
\text{a. } & \quad cp : H \rightarrow \{true, false\} \\
\text{b. } & \quad f : T \times H \rightarrow \{0, 1\}
\end{align*}\]

The set of contextual predicates that are used need to be specified by the experimenter. Similar to (Brill, 1994, 1995a), Ratnaparkhi (1996) used contextual predicates derived from templates. For example, two templates are shown in (32), where $w_i$ is the $i$th word, $t_i$ is the $i$th tag, and $X$ and $T$ refer to values to be filled in.

\[(32)\]
\[\begin{align*}
\text{a. } & \quad X \text{ is a suffix of } w_i, |X| \leq 4 \text{ & } t_i = T \\
\text{b. } & \quad t_{i-1} = X \text{ & } t_i = T
\end{align*}\]

A feature $f$ will be equal to 1 if the condition is met and 0 otherwise. A feature has access to any word or tag in the history $h$ of a given tag $t$. Features have the form:

\[(33)\]
\[f(t, h) = \begin{cases} 
1 & \text{if } cp(h) = true \text{ and } t = t' \\
0 & \text{otherwise}
\end{cases}\]

One example feature from Ratnaparkhi (1996) is given in (34):

\[(34)\]
\[f_j(t_i, h_i) = \begin{cases} 
1 & \text{if suffix-is-ing}(h_i) = true \text{ & } t_i = VBG \\
0 & \text{otherwise}
\end{cases}\]

To set the features, the model scans the training corpus asking yes/no questions about each item in $h \in H$ for a given tag $t \in T$. This gives each tag a probability of being correct based on its history.

In tagging a text, we are interested in the joint probability of a history $h$ together with a tag $t$, which is defined as:

\[(35)\]
\[p(t, h) = \pi \mu \prod_{j=1}^{k} \alpha_j f_j(t, h)\]
where $\pi$ is a normalization constant to ensure that the probabilities add up to one, and 
\(\{\mu, \alpha_1, ..., \alpha_k\}\) are positive model parameters. Each $\alpha_j$ corresponds to a feature $f_j$. Given

a sequence of words $\{w_1, ..., w_n\}$ and tags $\{t_1, ..., t_n\}$ as training data, and let $h_i$ denote

the history available for predicting $t_i$, the parameters are chosen to maximize the likelihood of

the training data, i.e., we want to find the model $p^*$

\[
(36) \quad p^* = \arg \max_p \prod_{i=1}^{n} p(t_i, h_i) = \prod_{i=1}^{n} \pi \mu \prod_{j=1}^{k} \alpha_j f_j(t_i, h_i)
\]

The parameters can be estimated using generalized iterative scaling. Interpreted under

the philosophy of maximum entropy, $p^*$ is the model that has maximal entropy among all

the models that are consistent with the training data. The entropy of the distribution is

defined as:

\[
(37) \quad H(p) = - \sum_{h \in H, t \in T} p(t, h) \log p(t, h)
\]

To ensure that the model is consistent with the training data, features are constrained to

have the same expected value in the model as in the training data (Berger et al., 1996).

This constraint is expressed in (38a), where $E f_j$ is the expected value of $f$ in the model

and $\tilde{E} f_j$ is the empirical expected value of $f$ in the training sample.

\[
(38) \quad \begin{align*}
& \text{a. } E f_j = \tilde{E} f_j \\
& \text{b. } E f_j = \sum_{t, h} p(t, h) f_j(t, h) \\
& \text{c. } \tilde{E} f_j = \sum_{t, h} \tilde{p}(t, h) f_j(t, h)
\end{align*}
\]

With this model, tagging proceeds by finding the conditional probability of a tag given

its history, i.e., $p(t| h)$. The algorithm is a beam search, which maintains the $N$ highest

probability sequences up to the current word. Ratnaparkhi (1996) reported an accuracy of

96.43%.

Like for transformation-based taggers, good feature selection is crucial for the perfor-
mance of the maximum entropy tagger. Likewise, unknown words are also handled using

features that capture the behavior and properties of unknown words. Ratnaparkhi (1996)
generated contextual predicates for tagging unknown words based on the hypothesis that

words with low frequency in the training data are similar to unknown words in the test

data in terms of how their spellings can help predict their tags. A set of special contextual

predicates are applied to both rare words and unknown words in test data. These predicates

consider the prefix and suffix of the word, whether it contains a number, whether it is cap-

tilized, and whether it contains a hyphen. Such information is used along with contextual

information, i.e., the tags before it and the words before and after it. Ratnaparkhi (1996)
reported an accuracy of around 86% for unknown word tagging.
Zhao and Wang (2002) and Ng and Low (2004) have applied the maximum entropy model to Chinese POS tagging. Zhao and Wang used some context predicates to handle reduplicated and derived words, but there is no discussion on how other types of unknown words are handled. They reported an overall tagging accuracy of 96.8%. Ng and Low’s experiments are primarily used to compare the word-based and character-based approaches to tagging. They reported a best tagging accuracy of 91.9%, but unknown word handling and performance were not discussed. As is the case with transformation-based learning, Chinese unknown words can also be handled using specially designed features that incorporates information about the morphological structure, affixes, special characters, etc., of the unknown words.

2.1.4 Memory-Based Learning

In memory-based learning (Daelemans et al., 1996; Zavrel and Daelemans, 1999), all training instances are kept in memory and new instances are classified by extrapolating directly from the remembered instances that are most similar to them. The memory-based learner has two components: a learning component and a performance component. The learning component is lazy in that it simply stores training instances in memory with little abstraction. The performance component performs similarity-based classification.

For POS tagging, training instances are referred as cases. Each case consists of the word, some lexical and contextual features about the word, and its tag. Given a tagged training corpus, the following data structures are extracted: a lexicon, which associates words with its possible tags; a case base for known words (words in the lexicon); and a case base for unknown words. The cases are represented by a number of features, whose relevance is determined by information gain weights.

In tagging new text, each word is looked up in the lexicon. If it is a known word, we first retrieve its lexical representation and determine its context, and then disambiguate its POS category by extrapolating from nearest neighbors in the case base for known words. If it is an unknown word, we first compute its lexical representations based on its form and determine its context. We then disambiguate its POS category by extrapolating from nearest neighbors in the case base for unknown words.

An information-gain weighted overlap metric is used to compute the distance between two instances $X$ and $Y$, represented by $n$ features. This is given in (39a), where $\Delta(X, Y)$ denotes the distance between $X$ and $Y$, and $\delta(x_i, y_i)$ is the distance between the values of feature $i$ for $X$ and $Y$.

\[
\Delta(X, Y) = w_i \sum_{i=1}^{n} \delta(x_i, y_i)
\]

\[
\delta(x_i, y_i) = \begin{cases} 
\text{abs}\left(\frac{x_i - y_i}{\max_i - \min_i}\right) & \text{if numeric, else} \\
0 & \text{if } x_i = y_i \\
1 & \text{if } x_i \neq y_i
\end{cases}
\]
c. \( w_i = H(T) - \sum_{v \in V_i} p(v)H(T|v) \)

d. \( H(T) = -\sum_{t \in T} p(t)\log_2 p(t) \)

In (39c) and (39d), \( T \) is again the tagset, \( V_i \) is the set of values for feature \( i \), and \( H(T) \) is the entropy of the tagset. Information gain is used to weight different features because not all feature contribute the same amount of information of the correct tag.

The memory-based tagger achieves an accuracy of 97% for English. As mentioned above, the POS categories of unknown words are disambiguated by extrapolating directly from similar cases in the case base for unknown words. The features used for this case base include the word’s suffix letters, prefix letters, capitalization, presence of hyphen, presence of numerals, as well as the POS tag to its left and right. Chinese unknown words can be handled in the similar fashion, but with a different set of features, such as the ones discussed in section 2.1.1.

2.1.5 Decision Trees

Schmid (1994) developed a probabilistic decision tree tagger, the TreeTagger. In a sense, it is similar to an n-gram tagger in that it also make use of transition probabilities and lexical probabilities to determine the probability of the current tag. It differs from n-gram taggers in that it estimates transition probabilities using a binary decision tree. Schmid (1994) argues that this addresses the sparse-data problem that Markov models are faced with. The nodes of the tree refer to one of the two previous tags and check whether that tag has a particular value, e.g., \( \text{tag}_{-1} = \text{ADJ?} \). The branches of the tree are either a yes or a no answer. The most likely tag can be determined by following the path down to the leaves of the tree, which are sets of (tag, probability) pairs.

The decision tree is constructed recursively. Each recursion finds the node that partitions the set of trigram samples maximally into two subsets with respect to the probability of the third tag. This is achieved by maximizing information gain (Quinlan, 1986, 1993) or minimizing the average amount of information still needed after the decision is made. The information gain is calculated as follows:

\[
I_q = -p(C_+|C) \sum_{t \in T} p(t|C_+)\log_2 p(t|C_+) - p(C_-|C) \sum_{t \in T} p(t|C_-)\log_2 p(t|C_-)
\]

where \( C \) is the context that corresponds to the current node, \( C_+/C_- \) is equal to \( C \) plus the condition that test \( q \) succeeds/fails, \( p(C_+|C)/p(C_-|C) \) is the probability that test \( q \) succeeds/fails, and \( p(t|C_+)/p(t|C_-) \) is the probability of the third tag \( t \) if the test succeeds/fails. These probabilities are maximum likelihood estimates from the data.

Once a decision tree is constructed, we can use it to derive transition probabilities for a given state in a Markov model. Lexical probabilities are obtained from a lexicon that
contains the a priori tag probabilities for individual words. The Viterbi algorithm is used to find the best sequence of tags. With this, Schmid (1994) reported an accuracy of 96.36%.

The lexical probabilities of unknown words are estimated using a suffix tree, which is constructed from the suffixes of length 5 of all words annotated with an open class tag. The tag frequencies of all suffixes are stored at the corresponding tree nodes. We first calculate the information measure, \( I(S) \) for each node as in (41a), and then use that to calculate the weighted information gain \( G(aS) \) for each node as in (41b)

\[
\begin{align*}
(41) & \quad a. ~ I(S) = - \sum_t p(t|S) \log_2 p(t|S) \\
& \quad b. ~ G(aS) = F(aS) \left( I(S) - I(aS) \right)
\end{align*}
\]

In (41a), \( S \) is the suffix that corresponds to the current node and \( p(t|S) \) is the probability of a tag given a word with suffix \( S \). In (41b), \( S \) is the suffix of the parent node, \( aS \) is the suffix of the current node, and \( F(aS) \) is the frequency of suffix \( aS \). A node is pruned if its information measure is smaller than a predetermined threshold. If a leaf is reached at the end of the path, the corresponding tag probability vector is returned. Otherwise some default leaf or default entry is used.

To estimate the lexical probabilities for Chinese unknown words using decision trees, we will need to use a richer feature set. As discussed above, these could include prefixes, suffixes, special characters, morphological structures, etc.

### 2.1.6 Neural Networks

Schmid (1994) built a POS tagger, the Net-Tagger, that consists of a multilayer perception (MLP) network and a lexicon. An MLP network consists of a large number of simple processing units that are arranged vertically in several layers, including an input layer, an output layer, and a certain number of hidden layers. The processing units are interconnected by directed weighted links, and each unit is associated with an activation value. During processing, activations are propagated from input units to output units through hidden units. The goal of the training process is to find the best network which correctly predicts the output units based on the input units.

In the case of tagging, each output unit represents one of the tags in the tagset. During training, the network learns to activate the output unit that corresponds to the correct tag while deactivating others. In the trained network, the output unit with the highest activation indicates which tag should be attached to the word being processed. The input contains all the available information about the parts of speech of the current word, the \( p \) preceding words, and the \( f \) following words. Specifically, for each POS tag \( t \) and each of the \( p + 1 + f \) words in the context, there is an input unit whose activation represents the probability that the current word has a certain part-of-speech. Therefore, the number of input units is \( n \times (p + 1 + f) \), where \( n \) is the number of possible tags. For the current word and its
following words, all we know is their prior tag probabilities, which are provided by the lexicon. For the preceding words, which are already tagged, we also have access to the activation values of their corresponding output units. The network is trained on a tagged corpus using a method called backpropagation, which feeds information from the corpus back to the input units. Using a network with no hidden layers, Schmid (1994) reported an accuracy of 96.22%.

The lexicon is organized in the same way as that for the decision tree. The lexical probabilities of unknown words are estimated by consulting the suffix tree, and if that fails, a default entry is used.

2.1.7 Support Vector Machines (SVMs)

Nakagawa et al. (2001) constructed a POS tagger using Support Vector Machines. SVMs (Vapnik, 1995) are binary classifiers on a feature vector space $R^L$. Given a set of training data, $\{(x_i, y_i) | x_i \in R^L, y_i \in \{\pm 1\}\}$, where $x_i$ is the $i$th sample in the training data, and $y_i$ is the label of the sample, SVMs find the optimal hyperplane that maximizes the margin between positive and negative samples among all hyperplanes that separate the training data into two classes. Giving a test example $x$, its label $y$ is determined by the sign of a discrimination function $f(x)$ given by the SVM classifier (Goh, 2003), as follows:

$$f(x) = \text{sgn}(\sum_{z_i \in SV} \alpha_i y_i K(x, z_i) + b)$$

where $b \in R$, $z_i$ is a support vector, which receives a non-zero weight $\alpha_i$, $K(x, z_i)$ is a polynomial kernel function of degree $d$ given by $K(x, z_i) = (xz_i + 1)^d$, which maps vectors into a higher dimensional space where all combinations of up to $d$ features are considered. The support vectors and the parameters are determined by quadratic programming. If $f(x) = +1$, then $x$ is a positive member, otherwise it is a negative member.

As SVMs are binary classifiers, Nakagawa et al. (2001) employed the one-versus-rest approach to extend them to multi-class classifiers to predict $k > 2$ tags. In training, $k$ classifiers $f_i(x) (1 \leq i \leq k)$ are created to classify class $i$ from all other classes. If $f_i(x) \geq +1$, $x$ belongs to class $i$, and otherwise if $f_i(x) \leq -1$. Given a test example $x$, its class $c$ is determined by the classifier that gives the largest discriminating function value, i.e., $c = \text{arg max}_i f_i(x)$.

Nakagawa et al. (2001) used the following features to predict the POS tag of an unknown word: 1) the POS tags of the two words before and after the unknown word, 2) the lexical forms of the two words before and after the unknown word, and 3) prefixes and suffixes of up to four characters of the unknown word and the existence of numerals, capital letters and hyphens in the unknown word. They reported an accuracy of about 97% for known words and about 87% for unknown words.
2.2 Unsupervised Methods

2.2.1 Hidden Markov Models (HMMs)

The supervised Markov models we saw earlier are also known as visible Markov models (VMMs) in that the values of the different states in the training data are visible to us. When no tagged training data is available, the real values of the states are unknown, and the Markov models used become hidden. As is the case with VMMs, the training process for HMMs (Jelinek, 1985; Cutting et al., 1992; Kupiec, 1992; Merialdo, 1994) also involves the estimation of the following three model parameters, i.e., initial probabilities, transition probabilities, and generation probabilities. Whereas VMMs estimate these parameters from the tagged training corpus, HMMs estimate these parameters using a dictionary and an unannotated raw corpus.

The training process consists of two steps: initialization and expectation. The parameters are initialized with information from a dictionary that lists all possible tags for each word, and we then attempt to find the parameters that maximize the probability of the training data through alternative expectation using the Baum-Welch, or forward-backward, algorithm (Baum, 1972). This algorithm proceeds by recursively defining the forward and backward probabilities. The forward probability is the joint probability of the state sequence from the beginning up to time \( t \) and the event that the Markov process is in a particular state \( i \) at time \( t \). The backward probability is the probability of seeing the state sequence from time \( t + 1 \) to the end given that the Markov process is in state \( i \) at time \( t \). The probability of the entire sequence \( S_{1,T} \) for any given \( t \) is defined in terms of the forward and backward probabilities at that time. At each iteration, the parameters are used to compute the forward and backward probabilities (and henceforth the probability of the entire tag sequence), and the new tag sequence is then used to reestimate the parameters. Iteration continues until convergence, i.e., when the probability of the entire tag sequence no longer improves. Once the model has been trained, the Viterbi algorithm is used to tag new texts, just as in VMMs.

Since HMMs rely on a dictionary to provide the ambiguity class for each word, the dictionary consists of three parts. The first part contains the ambiguity class of all known words, i.e., words that can be found in some existing dictionary. In the second part, a language-specific method is employed to guess the ambiguity classes for unknown words, i.e., words that are not found in existing dictionaries. The Xerox tagger (Cutting et al., 1992; Kupiec, 1992) used word suffixes for English. In this case, a separate word-category guessing mechanism developed using unsupervised methods can also be plugged in here (e.g., Mikheev, 1997; Cucerzan and Yarowsky, 2000). Mikheev (1997) learned a set of word-category guessing rules using word suffix, prefix, and ending from a lexicon and an unannotated corpus, and reported an improvement of around 6% on unknown word tagging for the Xerox tagger. A separate word-category guessing mechanism for Chinese unknown
words can also be implemented in a similar fashion. The third part of the lexicon assigns a default ambiguity class (usually all open class categories) to words that cannot be handled in the first two parts.

### 2.2.2 Transformation-Based Learning

The unsupervised version of transformation-based learning (Brill, 1995b) has the same three components as the supervised version, i.e., the initial state annotator, the space of allowable transformations, and the scoring function. However, given that we no longer have the tagged training corpus, the scoring function needs to be changed.

The unsupervised transformation-based learner assumes a dictionary that lists the ambiguity class for each word. The initial annotator now tags each word with a list of all its allowable tags, instead of one tag. Transformations now do not change one tag to another, but reduce the list of (two or more) tags to one tag. Learned transformations have the form in (43), where $\mathcal{X}$ is a list of two or more tags, $Y \in \mathcal{X}$, and $C$ is the context, which is limited to the previous (or following) word (or tag):

\[(43) \text{Change the tag of a word from } \mathcal{X} \text{ to } Y \text{ in context } C\]

During learning, we use information from the distribution of unambiguous words to reduce uncertainty about ambiguous words. At each iteration, the score of a rule is calculated as follows. For each tag $Z \in \mathcal{X}, Z \neq Y$, compute

\[(44) \frac{\text{freq}(Y)}{\text{freq}(Z)} \times \text{incontext}(Z, C)\]

where $\text{freq}(Y)$ is the frequency of words unambiguously tagged with $Y$ in the corpus (likewise for $\text{freq}(Z)$), and $\text{incontext}(Z, C)$ is the number of times a word unambiguously tagged with $Z$ occurs in context $C$ in the corpus. We then find $R$ such that

\[(45) R = \arg \max_Z \frac{\text{freq}(Y)}{\text{freq}(Z)} \times \text{incontext}(Z, C)\]

The score for the rule is then computed as:

\[(46) \text{incontext}(Y, C) - \frac{\text{freq}(Y)}{\text{freq}(R)} \times \text{incontext}(R, C)\]

$\frac{\text{freq}(Y)}{\text{freq}(R)}$ is used to adjust the relative frequency of $Y$ and $R$. The idea is that a good transformation should be one where one of the possible tags appears more frequently than all other possible tags in the context, as measured by unambiguously tagged words. The scoring function measures this by computing the difference between the number of unambiguous instances of tag $Y$ in context $C$ and the number of unambiguous instances of the most likely tag $R$ ($R \in \mathcal{X}, R \neq Y$) in context $C$. At each iteration, the transformation
with the maximum score is picked, and the procedure repeats until no transformation with a positive score can be found.

Brill (1995b) ignored unknown words, but mentioned that they could be handled either by assigning them all open class categories or developing an unsupervised version of the transformation-based unknown word tagger. As is the case for HMMs, an unknown word category guesser developed using unsupervised methods can also be plugged in to determine the possible categories of unknown words to be used by the initial annotator. For example, although Mikheev (1997) only tested his unknown-word guesser on the supervised transformation-based tagger, it could also work for the unsupervised version.

2.3 Combining Taggers

Following the line of research in classifier combination in the machine learning literature (Chan and Stolfo, 1995; Dietterich, 1997), there have been a number of recent attempts to combine taggers to achieve better tagging results (Brill and Wu, 1998; van Halteren et al., 1998, 2001). The general observation is that different taggers tend to incur different tagging errors, as they use different formalisms and/or contain different knowledge, and it is therefore possible to exploit the disagreement between the outputs of different taggers to reduce tagging errors. In general, classifier combination gives better results if the errors of the component classifiers do not correlate to a significant degree. This turns out to be the case for taggers. For example, Brill and Wu (1998) showed that the errors produced by a standard tri-gram tagger, the transformation-based tagger (Brill, 1995a) and the maximum entropy tagger (Ratnaparkhi, 1996) are strongly complementary.

There are two general approaches for exploiting tagger agreement (van Halteren et al., 2001). First, we can use the gang effect by letting the different taggers vote between their outputs. Methods for voting include simple voting (including majority and weighted voting) and pairwise voting. Second, we can create an arbiter effect, where we train a classifier to select the correct tag based on the patterns of co-occurrence of the various taggers’ outputs, a method referred to as stacking. Stacking makes it possible to identify the correct tag even when all taggers output the wrong answer, and generally performs better than voting. We briefly discuss these methods below.

**Majority voting** The most straightforward method is to take an unweighted $n$-way vote, where each tagger votes for the tag of its choice, and the majority win. Different tie breaking strategies can be used. For example, Brill and Wu (1998) took the vote from the tagger with the highest overall precision. With this method, they reported an error rate reduction of 6.9% over the best performing component tagger.

**Weighted voting** The vote from each taggers can also be weighted according to its quality and strength. For example, van Halteren et al. (1998) suggested three different weight-
ing schemes: using precision and recall, using accuracy (i.e., total precision), and using precision in relation to the particular tag the tagger suggests. They reported that the first of these achieved the best result among all three, improving the result from 97.34% (maximum entropy tagger) to 97.84%.

**Pairwise voting** Here, we first examine all situations where two taggers disagree and determine the best tag (which could differ from the ones suggested by the two taggers) out of all the possible tags in this situation based on the training data. In combining taggers, we let each pairwise combination vote for the best possible tag. Using this method, van Halteren et al. (1998) achieved a 19.1% error rate reduction over the best performing component tagger.

**Stacking** This refers to the practice of using the outputs of multiple classifiers as features to train a second level learner. These features can be complemented with additional information, e.g., about the original input pattern. Brill and Wu (1998) and van Halteren et al. (1998) both trained a memory-based (Daelemans et al., 1996) second level learner. Brill and Wu (1998) used the previous, current, and next word together with the output of each tagger for each of those words. They experimented with two methods for determining the best tag while tagging, i.e., to find either the most probable tag or the probable tagger for a particular context in the training data. They reported that using contexts to select the best tagger worked better and achieved an error reduction rate of 10.4% over the best performing component tagger. van Halteren et al. (1998) experimented with using the tagger outputs for the previous, current, and following words. The also experimented with using the tagger outputs and the correct tag for the current word (both with and without the form of the current word), but found that none of these worked better than pairwise voting.

van Halteren et al. (2001) argued that pairwise voting is also stacked, as the tag chosen is not always one of the tags suggested by the component taggers. They expanded the pairwise method by including word and context features and renamed the method Weighted Probability Distribution Voting (WPDV). In addition to pairs of features, this method can also use the probability distributions for all feature combinations observed in the training. They reported that this model resulted in an error reduction rate of 24.3%, higher than that of the memory-based, maximum entropy, and decision tree combinations.

Given that tagging accuracy for unknown words are generally significantly lower than that for known words and that unknown word tagging errors constitute a significant part of overall tagging errors, it would be relevant to examine how tagger combinations work for unknown words, and whether in training a second level classifier, we should handle unknown words differently from known words.