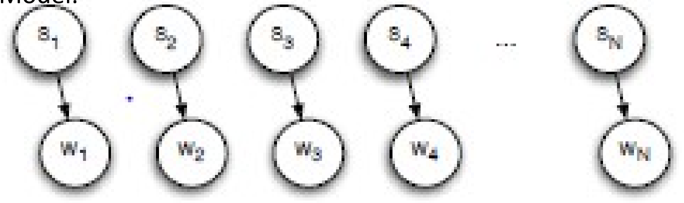
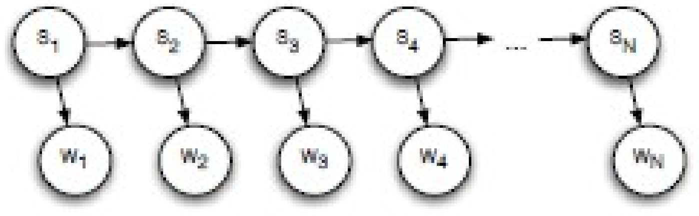
To tag parts of speech I have used three models and compared their sentence and words accuracy. We use the Viterbi Algorithm to predict pos tags using HMM and Markov Chain Marco Polo (MCMC) Algorithm to predict pos tags using the complex models.

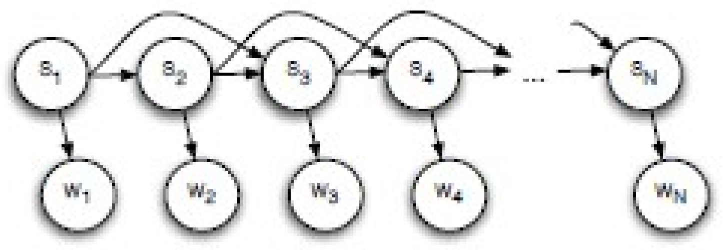
Simplified Model:



Hidden Markov Model:



Complex Model:



To calculate the various probabilities, we made the following dictionaries:-

1. For P(W=word|S=tag)
We filtered the training words first on a given tag, for eg: filtered all words with tag=noun
Then in this result we counted how many times a word appears with tag=noun and stored it a in nested dictionary.
The structure of the dictionary looks like:

```
self.tag_to_word_dict = {
                    'noun':{W1:20,W2:30....Wn:100},
                    'verb':{W1:20,W2:30....Wn:100},
                   }
```

Thus P(W=word|S=tag) = self.tag_to_word_dict[tag][word]/sum(self.tag_to_word_dict[tag].values())

2. For P(W=word)
For this we created a simple dictionary with word as its key and the number of times the word appears in the training data as its value.
The structure of the dictionary looks like:
```
  self.word_dict = {
                 'w1':34,
                 'w2':56,
                 .
                 .
                 'wn':100
               }
```
Thus P(W=word) = self.word_dict[word]/sum(self.word_dict.values())

3. For P(S=tag)
For this we created a simple dictionary with tag as its key and the number of times the tag appears in the training data as its value.
The structure of the dictionary looks like:
```
  self.tag_dict = {
                 't1':34,
                 't2':56,
                 .
                 .
                 'tn':100
               }
```
Thus P(W=word) = self.tag_dict[tag]/sum(self.tag_dict.values())

4. For P(S0=tag): For storing initial probabilities
For this we created a dictionary with tag as its key and the number of times that tag starts a sentence as its value.
The structure of the dictionary looks like:
```
  self.initial_prob_tag_dict = {
                          't1':44,
                          't2':50,
                          .
                          .
                          'tn':100
                        }
```
Thus P(S0=tag) = self.initial_prob_tag_dict[tag]/sum(self.initial_prob_tag_dict.values())

5. For P(S2=curr_tag|S1=prev_tag):transition probabilities
For this we created a dictionary with prev_tag as the first key, curr_tag as the second nested key and the number of times such transition happens in training data as its value.
The structure of the dictionary looks like:
```
  self.transition_dict = {
                     'prev_tag_1':{'curr_tag_1':32,'curr_tag_2':34...'curr_tag_n':100},
                     'prev_tag_2':{'curr_tag_1':32,'curr_tag_2':34...'curr_tag_n':100},
                     .
                     .
                     'prev_tag_n':{'curr_tag_1':32,'curr_tag_2':34...'curr_tag_n':100}
                   }
```
Thus P(S2=curr_tag|S1=prev_tag) = self.transition_dict[prev_tag][curr_tag]\

```
                    /sum(self.transition_dict[prev_tag].values())
```

6. For P(Si|Sj,Sw):transition probabilities for complex model.
For this we created a 3 level nested dictionary with Sw as the first key, Sj as the second nested key, Si as teh third nested key and the number of times such transition happens in training data as its value.
The structure of the dictionary looks like:

```
  self.transition_2_dict = {
                    'Sw_1':{'sj_1':{'s1_1':23,'si_2':43....'si_n':34},
                        'sj_2':{'s1_1':23,'si_2':43....'si_n':34},
                        .
                        .
                        .
                        'sj_n':{'s1_1':23,'si_2':43....'si_n':34}},

                }
```

Thus P(Si|Sj,Sw) = self.transition_2_dict[Sw][Sj][Si]\

```
                    /sum(self.transition_2_dict[Sw][Sj].values())
```

The following describes how each model was implemented:
1. Simple Model:
For this model we simply calculated prob = P(Wi|si)*P(Si) for each of the 12 tags for each word in the sentence.
The tag getting the maximum prob is returned as the predicted tag for that word.
The posterior of the same is calculated as sum(log(P(Si|Wi)*P(Si))) over each predicted tag

2. Hidden Markov Model:
We use the viterbi algorithm to predict tags using the HMM model.
For this we calculate:
  a. Initial_probability : P(S0=tag) given by self.initial_prob_tag_dict
  b. Emmision_probability : P(W-word|S=tag) given by self.tag_to_word_dict
  c. Transition_probability : P(S=curr_tag|S=prev_tag) given by self.transition_dict

To implement viterbi, we first created a dictionary tags = {tag:0 for each of the 12 tags}.
This would store the prob returned by viterbi in the current iteration for each tag. This dictionary is initialized at the start of each iteration and appended to a Viterbi_list at the end of each iteration.
At each iteration :

```
  for curr_tag in tags:
    for prev_tag in viterbi_list[-1]:
      if iteration == 0:
         tags[tag] =  Emmission_probability*Initial_probability
      else:
        tags[tag] =  Emmission_probability*max(viterbi_list[-
1][prev_tag]*transition_prob_prev_to_curr)
        tags[prev_key] = prev_key
  We fill this dictionary and append it to the viterbi_list.
```

Once the viterbi list is filled for every word in the sentence, we look at the last word and select the entry having the maximum probability. For this word, the key having the maximum probability becomes its
predicted tag.

In the dictionary tags, apart from storing just the probability returned by viterbi, I also store the tag from which I got my current value. Thus I only need tho look at the key : 'prev_key' to backtrack. Keep
doing this till we reach the first sentence.
The posterior for HMM is calculated as P(S0)P(W0|S0)P(S1|S0)......P(Sn|Sn-1)P(Wn|Sn). The log of each factor is
calculated a sum over all factors is returned.

3. Complex Model:
We use the MCMC algorithm to predict tags using the complex model. We have created a seperate function
called calc_prob_distribution(self, sample_tags, sentence, sampling_pos) which takes the current sample
the sentence for which we are sampling and the index of the sample on which we are sampling on. It then returns the probability distribution of that sample being one among the the 12 tags. I use this distribution for Gibbs sampling.

```
MCMC ALgorithm:
   a. Store a random sample = sample.
   b. Fix limit for warm up period.
   c. While iter < n_iters:
          For i in len(sentence):
              Call calc_prob_distribution to get the probability distribution.
              Do the coin flip and change the sample value at index i biased on the above
prob_distribution.
              if iter>warm_up period:
                  Append the above sample in sample_list
   d. Based on the sample_list, find the most commonly occuring tag and return it as its predicted tag.
```

```
calc_prob_distribution(sample_tags, sentence, sample_pos):
    This function implements the following model:
       if sample_pos == 0:
          Sample on S0 for P(S0)P(S1|S0)P(S2|S1,S0)P(W0|S0)
       if sample_pos == 1:
          Sample on S1 for P(S1|S0)P(W1|S1)P(S2|S1,S0)P(S3|S2,S1)
       if sample_pos > 1:
          Sample on Si for P(Si|Si-1,S1-2)P(Wi|Si)P(Si+1|Si,Si-1)P(Si+2|Si+1,Si)
       if sample_pos == len(sentence)-1:
          Sample on Si for P(Si|Si-1,Si-2)P(Wi|Si)
       Append the samples over each tag in a list
       Calculate the sum of the list
       Divide each entry of the list by its sum
       Return the list
```

The posterior for Complex model is calculated as
P(S0)P(W0|S0)P(S1|S0)P(S2|S1,S0)P(W2|S2)......P(Sn|Sn-1)P(Wn|Sn)P(Sn|Sn-2,Sn-1).
The log of each factor is calculated a sum over all factors is returned.

We have checked the training data, and found out that if a word that appears in testing set does not appear in the training set then its highly likely to be a noun. Thus we have taken this assumption throughout this program for all the three models.