Artificial Intelligence	**Assignment 6**
comp440/comp557 Fall 2016	**31 October 2016**

This assignment is due November 10 at 8 pm on Canvas. Download `assignment6.zip` *from Canvas. There are four problems worth a total of 90 regular points + 20 extra credit points for comp440 and 110 regular points for comp557. Problems 1, 2 and 4 require written work only, Problem 3 requires Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included at the labeled points in* `submission.py`*. For Problem 3, please run the autograder using the command line* `python grader.py` *and report the results in your writeup. Upload* `submission.py` *and* `writeup.pdf` **as two separate files** *on Canvas by the due date/time.*

1 Hidden Markov Models (20 points)

A professor wants to know if students are getting enough sleep. Each day, the professor observes whether the students sleep in class, and whether they have red eyes. The professor believes the following.

- The prior probability of getting enough sleep, with no observations, is 0.7.
- The probability of getting enough sleep on night t is 0.8 given that the student got enough sleep the previous night $t-1$, and 0.3 if not.
- The probability of having red eyes is 0.2, if the student got enough sleep, and 0.7 if not.
- The probability of sleeping in class is 0.1 if the student got enough sleep, and 0.3 if not.

- (5 points) Formulate this problem as a Hidden Markov model with a single hidden state and a single observation variable. Specify all the required probabilities for the model.
- (5 points) Given the evidence variables e_1 =(not red eyes, not sleeping in class); e_2 =(red eyes, not sleeping in class); e_3 =(red eyes, sleeping in class); compute P(EnoughSleep$_t|e_{1:t}$) for $t = 1, 2, 3$.
- (5 points) Given the evidence variables e_1 =(not red eyes, not sleeping in class); e_2 =(red eyes, not sleeping in class); e_3 =(red eyes, sleeping in class); compute P(EnoughSleep$_t|e_{1:3}$) for $t = 1, 2, 3$.
- (5 points) Compare the filtered probability P(EnoughSleep$_t|e_{1:t}$) with the smoothed probability P(EnoughSleep$_t|e_{1:3}$) for t=1,2,3.

2 Understanding human emotions (15 points)

A major milestone for building a strong AI agent – one whose intelligence is on par with human beings – is to teach a program to recognize emotion and personality. In the first part of the problem,

we would like to track a single human's emotion given an audio stream of the human talking. Then, given how their emotions change, we would like to guess at their personality type.

At each time slice, we observe a segment of sound from which we can extract the pitch contour — how the speakers pitch changes over the sound segment. Let the pitch contour at time t be C_t. The domain of pitch contour is {angular, glideup, descending, flat, irregular}. We would like to know the emotion at each time slice E_t. The domain of emotions is {sadness, surprise, joy, disgust, anger and fear}.

- (5 points) Formalize the task of tracking emotion using an HMM.
  - What are the hidden variables?
  - What are the observed variables?
  - What are the dimensions of the state transition conditional probability table?
  - What are the dimensions of the emission conditional probability table?
  - What is a reasonable probability distribution for the hidden variables at time = 0?
- (5 points) We want to compute the likelihood of a sequence of observations given our HMM. Write a formula to calculate the probability of seeing a particular sequence of observations over n time slices. Use probabilities from the HMM description. You can use a normalization constant.
- (5 points) Our agent knows about two different personality types represented by the random variable R with values x and y. The prior probabilities of being a specific personality type $P(R = x)$ and $P(R = y)$ are ϕ_x and ϕ_y respectively. For each personality type, we can calculate the probability of a sequence of n observations using the formula derived in the previous part. Call that probability Θ_x for personality type x and Θ_y for personality type y. Compute the probability of the person's personality trait being x given a particular sequence of observations.

3 Conditional random fields and named entity recognition (40 points + 20 EC points for comp440/required points for comp557)

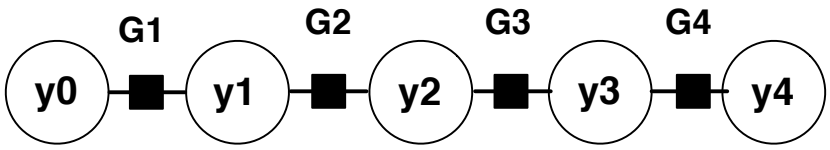
One of the principal aims of natural language processing is to build a system that can automatically read a piece of text and determine who is doing what to whom. A first step towards that goal is named-entity recognition, the task of taking a piece of text and tagging each word as either person, organization, location, or none of the above. Here is an example.

```
          --PER--            --LOC--
In 1971, Obama returned to Honolulu to live with his maternal grandparents,
 --PER--   --PER--  --PER--
Madelyn and Stanley Dunham, and with the aid of a scholarship attended
 --ORG-- --ORG--
Punahou School, a private college preparatory school, from fifth grade until
his graduation from high school in 1979.
```

In this assignment, we will build a named-entity recognition system using factor graphs representing conditional random fields (CRF). We will start with a chain-structured factor graph called a linear-chain conditional random field, which admits exact inference via variable elimination. Then, for extra credit for comp440, and required for comp557, we will develop a more sophisticated factor graph to capture long-range dependencies which are common in natural language. For this model, we will use Gibbs sampling to perform approximate inference.

3.1 Linear-chain conditional random fields

Let $\mathbf{x} = (x_1, \ldots, x_T)$ be a sequence of words and let $\mathbf{y} = (y_1, \ldots, y_T)$ be a sequence of tags. We will model the Named Entity Recognition (NER) task with the following factor graph. Here the



tags are the variables, and the words $\mathbf{x}$ only affect the potentials G_t (denoted by the black boxes). The probability of a tag sequence $\mathbf{y}$ given $\mathbf{x}$ is

$$\begin{aligned} p(\mathbf{y}|\mathbf{x};\theta) &= \frac{1}{Z(\mathbf{x};\theta)} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta) \\ Z(\mathbf{x};\theta) &= \sum_{\mathbf{y}} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta) \end{aligned}$$

where y_0 = `-BEGIN-`, a special tag indicating the beginning of a sentence, and $Z(\mathbf{x};\theta)$ is the normalization constant. The potentials G_t are

$$G_t(y_{t-1}, y_t; \mathbf{x}, \theta) = exp(\theta.\phi_{local}(t, y_{t-1}, y_t, \mathbf{x}))$$

where ϕ_{local} is the local feature function and $\theta \in \Re^d$ is the parameter vector. $\theta.\phi_{local}(t, y_{t-1}, y_t, \mathbf{x})$ stands for the dot product of the parameter vector θ with the feature vector ϕ_{local}. ϕ_{local} can depend arbitrarily on the input $\mathbf{x}$, and will generally access the words around position t (i.e., x_{t-1}, x_t, x_{t+1}).

We have provided you with the function `LinearChainCRF.G(t,y_,y,xs)` to compute the value of $G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$, where `y_` is y_{t-1}, `y` is y_t and `xs` is $\mathbf{x}$. In mathematics, indexing starts from 1, so y_1 is the first tag, but in Python, indexing starts from 0 (i.e., `ys[0]` is the first tag). To get the value of $G_3(y_2, y_3; \mathbf{x}, \theta)$, call `LinearChainCRF.G(2,ys[1],ys[2],xs)` where `ys` is the tag sequence $\mathbf{y}$ and `xs` is the observation sequence $\mathbf{x}$. For y_0 =`-BEGIN-`, use the provided constant `BEGIN_TAG`. For example, $G_1(-$ `BEGIN`$-, y_1; \mathbf{x}, theta)$ is `LinearChainCRF.G(0,BEGIN_TAG,ys[0],xs)`.

3.2 Problem 3.1: Inference in linear chain CRFs (30 points)

- (10 points) Our first task is to compute the best tag sequence $\mathbf{y}*$ given a sentence $\mathbf{x}$ and a fixed parameter vector θ.

$$\begin{aligned} \mathbf{y}* &= argmax_{\mathbf{y}} P(\mathbf{y}|\mathbf{x};\theta) \\ &= argmax_{\mathbf{y}} \prod_{t=1}^{T} G_t(y_{t-1}, y_t; \mathbf{x}, \theta) \end{aligned}$$

The Viterbi algorithm eliminates the variables $y_1, \ldots, y_T$ in the order y_1 to y_T, producing a sequence of forward messages:

$$\begin{aligned} Viterbi_1(y_1) &= G_1(-\texttt{BEGIN}-, y_1; \mathbf{x}, \theta) \\ Viterbi_2(y_2) &= max_{y_1} Viterbi_1(y_1) G_2(y_1, y_2; \mathbf{x}, \theta) \end{aligned}$$

Repeating this process gives us the following algorithm:

1. Initialize $Viterbi_0(y_0) = 1$.
2. For $t = 1, \ldots, T$, compute $Viterbi_t(y_t) = max_{y_{t-1}} Viterbi_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$
3. Return $max_{y_T} Viterbi_T(y_T)$.

To recover the actual sequence $\mathbf{y}*$, we work backwards from the value of y_T that maximizes $Viterbi_T$, back to the optimal assignment of y_1.

1. Compute $y_T^* = argmax_{y_T} Viterbi_T(y_T)$
2. For $t = T, \ldots, 2$, compute $y_{t-1}^* = argmax_{y_{t-1}} Viterbi_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t^*; \mathbf{x}, \theta)$

Implement the computation of $Viterbi_t$ and the recovery of the best sequence in the function `computeViterbi` in `submission.py`. Once you have implemented `computeViterbi`, you can run the following command to get an interactive shell to play around with your CRF.

```
>> python run.py shell --parameters data/english.binary.crf --featureFunction binary
>> viterbi Mark had worked at Reuters
          -PER- -O- -O- -O- -ORG
```

- (10 points) Next, let us compute forward and backward messages. Here is the algorithm for computing forward messages. To prevent numerical underflow/overflow errors, we normalize $Forward_t$ and keep track of the log normalization constant A.

1. Initialize $Forward_0(y_0) = 1$ and $A = 0$.
2. For $t = 1, \ldots, T$ do
   - Compute $Forward_t(y_t) = \sum_{y_{t-1}} Forward_{t-1}(y_{t-1}) G_t(y_{t-1}, y_t; \mathbf{x}, \theta)$
   - Update $A \leftarrow A + log(\sum_{y_t} Forward_t(y_t))$
   - Normalize: $Forward_t(y_t) \leftarrow \frac{Forward_t(y_t)}{\sum_{y_t'} Forward_t(y_t')}$
3. Return A, which equals the log of the normalization constant $Z(\mathbf{x}; \theta)$.

Implement `computeForward`, which returns the log normalization constant A as well as the normalized forward messages $[Forward_1(y), \ldots, Forward_T(y)]$. We have provided `computeBackward` which produces a sequence of normalized backward messages $[Backward_1(y), \ldots, Backward_T(y)]$.

- (10 points) Given the forward and backward messages, we can combine them to compute marginal probabilities:

$$P(y_{t-1}, y_t|\mathbf{x}, \theta) = \frac{Forward_{t-1}(y_{t-1})G_t(y_{t-1}, y_t; \mathbf{x}, \theta)Backward_t(y_t)}{Z(\mathbf{x}; \theta)}$$

Implement `computeEdgeMarginals` that will compute $P(y_{t-1}, y_t|\mathbf{x}, \theta)$. You should use `computeForward` and `computeBackward`.

We have implemented the learning algorithm that uses these marginals to compute a gradient. You can train the CRF (with standard features explained in the next subsection) now by running:

```
>> python run.py train --featureFunction binary --output-path my.crf
```

It could take up to 10-20 minutes to train, so only do this after you are confident that your code works. The program will write the parameters of the trained CRF to `my.crf`. You should get a dev F1 score of around 56.7%, which is quite poor. In the next section, we will design better features that will substantially improve the accuracy.

3.3 Problem 3.2: Named-entity recognition (10 points)

In the previous part, we developed all the algorithms required to train and use a linear-chain CRF. Now we turn to designing better features. We are using a subset of the CoNLL 2003 dataset consisting of 2,000 sentences with the following NER tags: `-PER-` (person), `-LOC-` (location), `-ORG-` (organization), `-MISC-` (miscellaneous), and `-O-`(other). We have provided two very simple feature functions. We will describe the feature functions using the example three-word sentence: `xs = ["Beautiful", "2", "bedroom"]`.

- `unaryFeatureFunction(t, y_, y, xs)` introduces an indicator feature for the current tag `y` and the current word `xs[t]`. For example, `unaryFeatureFunction(2, "-FEAT-", "-SIZE-", xs)` would return { `("-SIZE-", "bedroom") : 1.0` }. Note that `"-SIZE-"` is `y` and `x[2]` is `"bedroom"`.
- `binaryFeatureFunction(t, y_, y, xs)` includes all the features from `unaryFeatureFunction` and introduces another indicator feature for the previous tag `y_` and the current tag `y`. For example, `binaryFeatureFunction(2, "-FEAT-", "-SIZE-", xs)` would return, { `("-FEAT-", "-SIZE-") : 1.0, ("-SIZE-", "bedroom") : 1.0` }. Note that `"-FEAT-"` is `y_` and `"-SIZE-"` is `y`.

To train a model, use the following command,

```
>> python run.py train --featureFunction [unary|binary] --output-path my.crf
```

Use one of the featureFunctions `unary` or `binary` to do the training. As the model trains, it will print the likelihood of the training data at each iteration as well as a confusion matrix and F1 score for the NER tags. If you specify an output path, you can interact with the CRF you trained by providing the path as an argument to the shell,

```
>> python run.py shell --parameters my.crf --featureFunction [unary|binary]
```

Remember to use the same feature function as the one you used to train!

A common problem in NLP is making accurate predictions for words that we've never seen during training (e.g., Punahou). The reason why our accuracy is so low is that all our features thus far are defined on entire words, whereas we would really like to generalize. Fortunately, a CRF allows us to us to choose arbitrary feature functions $\phi_{local}(t, y_{t-1}, y_t, \mathbf{x})$. Next, we will define features based on capitalization or suffixes, which allow us to generalize to unseen words, as well as features that look at the current tag and the previous and next words, to capture more context.

Implement `nerFeatureFunction(t, y_, y, xs)` with the features below. Again, we will illustrate the expected output using the sentence: `xs = ["Beautiful", "2", "bedroom"]`. For convenience, think of the sentence as being padded with special begin/end words: `xs` is `-BEGIN-` at position -1 and `-END` at position `len(xs)`.

- All the features from `binaryFeatureFunction`.
- An indicator feature on the current tag `y` and the capitalization of the current word `xs[t]`. For example, `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would include the following features, {`("-FEAT-", "-CAPITALIZED-"):1.0`}. On the other hand, `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would not add any features because `"bedroom"` is not capitalized.
- An indicator feature on the current tag `y` and the previous word `xs[t-1]`. For example, `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would add: {`("-SIZE-", "PREV:2"):1.0`}. This is because `2` occurs before `"bedroom"` which is `xs[2]`. And `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would add {`("-FEAT-", "PREV:-BEGIN-"):1.0`}.
- A similar indicator feature for the current tag `y` and next word `xs[t+1]`; For example, `nerFeatureFunction(0, "-BEGIN-", "-FEAT-", xs)` would include, {`("-FEAT-", "NEXT:2"):1.0`}, and `nerFeatureFunction(2, "-SIZE-", "-SIZE-", xs)` would include, {`("-SIZE-", "NEXT:-END-"):1.0`}.
- Repeat the above two features except using capitalization instead of the actual word; consider `-BEGIN-` and `-END-` to be uncapitalized. An an example, `nerFeatureFunction(1, "-SIZE-", "-SIZE-", ["Beautiful", "2", "Bedroom"])` would include {`("-SIZE-", "-PRE-CAPITALIZED-"):1.0, ("-SIZE-", "-POST-CAPITALIZED-"):1.0`}.

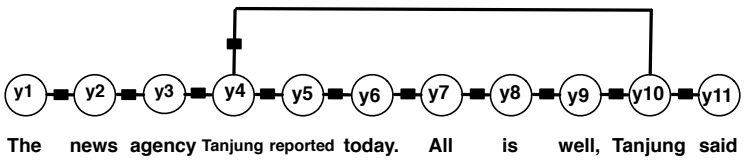
After you verify that your features are working using grader.py, train your CRF:

```
>> python run.py train --featureFunction ner --output-path ner.crf
```

Your dev F1 score should be around 71%.

3.4 Problem 3.3: Handling long-range dependencies (20 points) EC for comp440/required for comp557

Consider the following example,



It is clear that in the first occurrence of the word, "Tanjung" is an organization. Given only the second sentence though, it is ambiguous whether "Tanjung" is a person or organization. In fact, the CRF we previously trained predicts "Tanjung" to be a person here. To address this, we would like to add a constraint that all occurrences of a word are tagged the same. The catch is that such a constraint introduces long-range dependencies between the tag variables, complicating inference.

Gibbs sampling for linear chain CRFs (5 points)

Before we look at a Gibbs sampler to handle long-range dependencies, let us revisit the linear-chain CRF that we studied in the first part of this problem. Recall that Gibbs sampling updates y_t by sampling from its conditional distribution, given the values of the rest of the variables, $y_{-t} = (y_1, \ldots, _{t-1}, y_{t+1}, \ldots, y_T)$. Write an expression for the conditional distribution $P(y_t|y_{-t}, \mathbf{x}; \theta)$ for the linear-chain CRF in terms of the potentials G_t. Place your expression in `writeup.pdf`.

Implementing Gibbs sampling for linear chain CRFs (15 points)

We have provided you a function `gibbsRun()` in `submission.py` which provides the framework for the Gibbs sampler. Read the documentation for this function first. Now implement the following.

- (5 points) Implement `chooseGibbsCRF` that samples a value for y_t based on its conditional distribution you derived above, and reassigns y_t to that value. Note that you should only use the potential between y_t and its Markov blanket.
- (5 points) Implement `computeGibbsProbabilities` that estimates the probability for each output sequence based on the samples of the Gibbs sampler.
- (5 points) Implement `computeGibbsBestSequence` that estimates the most likely sequence (the interface is similar to `computeViterbi`).

Once you have implemented these functions, you can run the following command(s) to look at the output of Gibbs.

```
$ python run.py shell --parameters data/english.binary.crf --featureFunction binary
>> gibbs_best Mark had worked at Reuters
```

```
-PER--O--O--O--ORG-
>> gibbs_dist Mark had worked at Reuters
0.622  -PER--O--O--O--ORG-
0.1902  -PER--O--PER--O--ORG-
0.124  -PER--O--ORG--O--ORG-
0.0321  -ORG--O--O--O--ORG-
0.0084  -ORG--O--PER--O--ORG-
```

Your numbers will not match exactly due to the randomness in sampling.

4 Decision networks (15 points)

Consider a student who has the choice to buy or not buy a textbook for a course. We will model this decision problem with one boolean decision node B, indicating whether the student chooses to buy the book, and two Boolean nodes M, indicating whether the student has mastered the material in the book, and P indicating whether the student passes the course. There is a utility node U in the network. A certain student, Sam, has an additive utility function: 0 for not buying the book and -$100 for buying it; and $2000 for passing the course and 0 for not passing it. Sam's conditional probability estimates are:

$$\begin{array}{ll} P(p|b,m) = 0.9 & P(p|b,\neg m) = 0.5 \\ P(p|\neg b,m) = 0.8 & P(p|\neg b,\neg m) = 0.3 \\ P(m|b) = 0.9 & P(m|\neg b) = 0.7 \end{array}$$

You might think that P would be independent of B given M. But, this course has an open-book final – so having the book helps.

- (5 points) Draw the decision network for this problem.
- (5 points) Compute the expected utility of buying the book and not buying the book.
- (5 points) What is the optimal decision for Sam?