Homework 2, Intro to NLP, 2017

This is due at 11pm on Tuesday, October 10. Please see detailed submission instructions below. 100 points total.

How to do this problem set:

- What version of Python should I use? 2.7
- Most of these questions require writing Python code and computing results, and the rest of them have textual answers. To generate the answers, you will have to fill out supporting files, `vit_starter.py`,`classperc.py` and `structperc.py`.
- Write all the answers in this ipython notebook. Once you are finished (1) Generate a PDF via (File -> Download As -> PDF) and upload to Gradescope (2)Turn in `vit_starter.py`, `classperc.py`, `structperc.py` and `hw_2.ipynb` on Moodle.
- **Important:** Check your PDF before you turn it in to gradescope to make sure it exported correctly. If ipython notebook gets confused about your syntax it will sometimes terminate the PDF creation routine early. You are responsible for checking for these errors. If your whole PDF does not print, try running `$jupyter nbconvert --to pdf hw_1.ipynb` to identify and fix any syntax errors that might be causing problems.
- **Important:** When creating your final version of the PDF to hand in, please do a fresh restart and execute every cell in order. Then you'll be sure it's actually right. One convenient way to do this is by clicking `Cell -> Run All` in the notebook menu.

Academic honesty

- We will audit the Moodle code from a few dozen students, chosen at random. The audits will check that the code you wrote and turned on Moodle generates the answers you turn in on your Gradescope PDF. If you turn in correct answers on your PDF without code that actually generates those answers, we will consider this a potential case of cheating. See the course page for honesty policies.
- We will also run automatic checks of code on Moodle for plagiarism. Copying code from others is considered a serious case of cheating.

1. HMM (15 points)

Answer the following questions using the transition matrix T and emission probabilities E below. Below, Δ and $\square$ are two output variables, A and B are two hidden states; s_n refers to the n^{th} hidden state in the sequence and o_n refers to the n^{th} observation.

$T =$

	A	B	END
START	0.5	0.5	0.0
A	0.2	0.3	0.5
B	0.4	0.4	0.2

$E =$

	Δ	□
A	0.5	0.5
B	0.3	0.7

For all the questions in this section, write answer and show your work.

Question 1.1 (2 points)

Does $P(o_2 = \Delta | s_1 = B) = P(o_2 = \Delta | o_1 = \square)$?

Answer:

No. These values will be different. Markov model assumes output independence. Hence the nth observation depends only on the nth hidden state. Therefore o_2 will depend on s_2 and s_2 in turn is dependent on s_1.

s_1 is given to be B in the LHS equation. However, for RHS equation, $o_1 = \square$ can come from both $s_1 = A$ and $s_1 = B$. Thus, the answer is False.

Question 1.2 (2 points)

Does $P(s_2 = B | s_1 = A) = P(s_2 = B | s_1 = A, o_1 = \Delta)$?

Answer:

Yes. Using Markov's assumption, the proability of nth state depends only on the (n-1)th state. Therefore, probability of $s_2 = B$ depends only on s_1 which is A in both the cases. Therefore, the probabilities will be same. Also we have,

$$P(s_2 = B | s_1 = A, o_1 = \Delta) = \frac{P(s_2=B, s_1=A, o_1=\Delta)}{P(s_1=A, o_1=\Delta)}$$

$$= \frac{P(o_1=\Delta, s_2=B | s_1=A) * P(s_1=A)}{P(s_1=A, o_1=\Delta)}$$

$$= \frac{P(o_1=\Delta | s_1=A) * P(s_2=B | s_1=A) * P(s_1=A)}{P(s_1=A, o_1=\Delta)}$$

$$= \frac{P(s_1=A, o_1=\Delta) * P(s_2=B | s_1=A)}{P(s_1=A, o_1=\Delta)}$$

$$= P(s_2 = B | s_1 = A)$$

Hence Proved.

Question 1.3 (3 points)

Does $P(o_2 = \Delta | s_1 = A) = P(o_2 = \square | s_1 = A, s_3 = A)$

Answer:

No.

Solving LHS:

$$P(o_2 = \Delta | s_1 = A) = \sum_{s_2=A,B} P(o_2 = \Delta | s_2) * P(s_2 | s_1 = A)$$

$$= (0.5 * 0.2) + (0.3 * 0.3)$$

$$= 0.19$$

Solving RHS:

$$P(o_2 = \Box | s_1 = A, s_3 = A) = \frac{P(o_2 = \Box, s_1 = A, s_3 = A)}{P(s_1 = A, s_3 = A)}$$

$$= \frac{\sum_{s_2=A,B} P(o_2 = \Box | s_2) * P(s_1 = A | start) * P(s_3 = A | s_2) * P(s_2 | s_1 = A)}{\sum_{s_2=A,B} P(s_1 = A | start) * P(s_2 | s_1 = A) * P(s_3 = A | s_2)}$$

$$= \frac{(0.5*0.5*0.2*0.2)+(0.7*0.5*0.4*0.3)}{(0.5*0.2*0.2)+(0.5*0.3*0.4)}$$

$$= 0.65$$

Since,

$$0.19 \neq 0.65$$

Therefore, $P(o_2 = \Delta | s_1 = A) \neq P(o_2 = \Box | s_1 = A, s_3 = A)$

Question 1.4 (3 points)

Compute the probability of observing $\Box$ as the first emission of a sequence generated by an HMM with transition matrix T and emission probabilities E.

Answer:

$P(o_1=\Box)

$$= P(s_1 = A | START) * P(o_1 = A | s_1 = A) + P(s_1 = B | START) * P(o_1 = \Box | s_1 = B)$$

$$= (0.5 * 0.5) + (0.5 * 0.7)$$

$$= 0.60$$

Question 1.5 (5 points)

Compute the probability of the first state being A given that the last token in an observed sequence of length 2 was the token Δ.

Answer:

To calculate: $P(s_1 = A | o_2 = \Delta, s_3 = end)$

Using Bayes Rule,

$= \frac{P(o_2=\Delta, s_3=end|s_1=A)*P(s_1=A)}{P(o_2=\Delta, s_3=end)}$

Solving Numerator,

$P(o_2 = \Delta, s_3 = end|s_1 = A) * P(s_1 = A)$

$= \frac{P(o_2=\Delta, s_3=end, s_1=A)*P(s_1=A)}{P(s_1=A)}$

$= P(o_2 = \Delta, s_3 = end, s_1 = A)$

$= \sum_{s_2=A,B} P(o_2 = \Delta|s_2) * P(s_2|s_1 = A) * P(s_3 = end|s_2) * P(s_1 = A|start)$

$= (0.5 * 0.2 * 0.5 * 0.5) + (0.3 * 0.3 * 0.2 * 0.5)$

$= 0.034$

Solving Denominator,

$P(o_2 = \Delta, s_3 = end)$

$= \sum_{s_2=A,B} P(o_2 = \Delta|s_2) * P(s_3 = end|s_2)$

$= (0.5 * 0.5) + (0.3 * 0.2)$

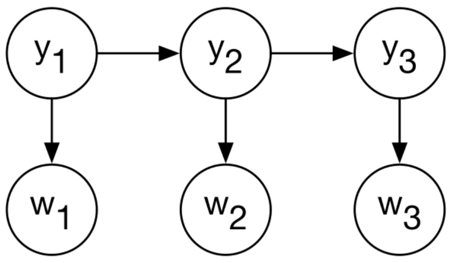
$= 0.31$

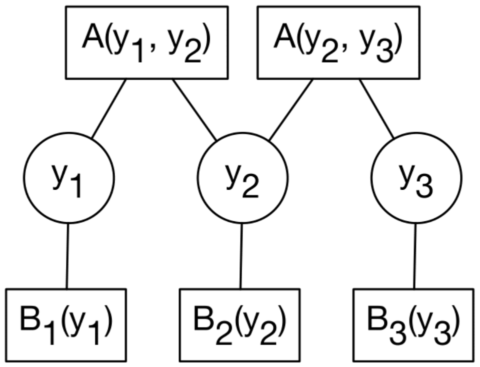
Finally,

$= \frac{0.034}{0.31}$

= 0.10967

2. Viterbi (log-additive form) (20 points)





One HMM chain is shown on the left. The corresponding **factor graph** version is shown on the right. This simply shows the structure of the A and B_t log-prob tables and which variables they express preferences over. A is the **transition factor** that has preferences for the two neighboring variables; for example, $A(y_1, y_2)$ shows how happy the model is with the transition from y_1 to y_2.

The same transition preference function is used at all positions $(t-1, t)$ for each $t = 2..T$. B_t is the **emission factor** that has preferences for the variable y_t. As a goodness function it is e.g. $B_1(y_1)$, $B_2(y_2)$, etc.

Let $\vec{y} = (y_1, y_2, \ldots, y_T)$ a proposed tag sequence for a T length sentence. The total goodness function for a solution $\vec{y}$ is

$$G(\vec{y}) = \sum_{t=1}^{T} B_t(y_t) + \sum_{t=2}^{T} A(y_{t-1}, y_t)$$

Question 2.1 (2 points)

Define A and B_t in terms of the HMM model, such that G is the same thing as $\log p(\vec{y}, \vec{w})$ under the HMM.

Answer:

$A(y_t, y_{t+1}) = \log p_{trans}(y_{t+1}|y_t)$

$B_t = \log p_{emit}(w_t|y_t)$

Question 2.2 (18 points)

Implement additive log-space Viterbi by completing the **viterbi()** function. It takes in tables that represent the A and B functions as input. We give you an implementation of $G()$ in **vit_starter**, you can check to make sure you understand the data structures, and also the exhaustive decoding algorithm too. Feel free to add debugging print statements as needed. The main code runs the exercise example by default.

When debugging, you should make new A and B examples that are very simple. This will test different code paths. Also you can try the **randomized_test()** from the starter code.

Look out for negative indexes as a bug. In python, if you use an index that's too high to be in the list, it throws an error. But it will silently accept a negative index ... it interprets that as indexing from the right.

In [103]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [128]:
```
# Implement the viterbi() function in vit_starter.py and then run this cell to sh

from vit_starter import *

if __name__=='__main__':
    A = {(0,0):3, (0,1):0, (1,0):0, (1,1):3}
    Bs= [ [0,1], [0,1], [30,0] ]
    # that's equivalent to: [ {0:0,1:1}, {0:0,1:1}, {0:30,1:0} ]

    y = exhaustive(A, Bs, set([0,1]))
    print "Exhaustive decoding:", y
    print "score:", goodness_score(y, A, Bs)
    y = viterbi(A, Bs, set([0,1]))
    print "Viterbi    decoding:", y
```

```
Exhaustive decoding: [0, 0, 0]
score: 36
Viterbi    decoding: [0, 0, 0]
```

Copy and paste the viterbi function that you implemented in `vit_starter.py`.

In [105]:
```
def viterbi(A_factor, B_factors, output_vocab):
    """
    A_factor: a dict of key:value pairs of the form
        {(curtag,nexttag): score}
    with keys for all K^2 possible neighboring combinations,
    and scores are numbers.  We assume they should be used ADDITIVELY, i.e. in lo
    higher scores mean MORE PREFERRED by the model.

    B_factors: a list where each entry is a dict {tag:score}, so like
    [ {Noun:-1.2, Adj:-3.4}, {Noun:-0.2, Adj:-7.1}, .... ]
    each entry in the list corresponds to each position in the input.

    output_vocab: a set of strings, which is the vocabulary of possible output
    symbols.

    RETURNS:
    the tag sequence yvec with the highest goodness score
    """

    N = len(B_factors)   # length of input sentence

    # viterbi log-prob tables
    V = [{tag:None for tag in output_vocab} for t in range(N)]
    # backpointer tables
    # back[0] could be left empty. it will never be used.
    back = [{tag:None for tag in output_vocab} for t in range(N)]

    # main viterbi implementaion
    #Initialization step - handling the t=0 step
    ## Setting V[0] and leaving back[0] as None
    for j in output_vocab:
        V[0][j]= B_factors[0][j]
    for t in range(1,N):
        for k in output_vocab:
            values = {}
            for j in output_vocab:
                values[j] = V[t-1][j] + A_factor[(j,k)] + B_factors[t][k]
            V[t][k] = max(values.values())
            back[t][k] = dict_argmax(values)

    decoded = list()
    # find the starting point for backpointer
    decoded.append(dict_argmax(V[N-1]))
    BackIndex = dict_argmax(V[N-1])
    for i in range(N-1,0,-1):
        decoded.append(back[i][BackIndex])
        BackIndex = back[i][BackIndex]
    # Returning the decoded values in correct order
    return list(reversed(decoded))
```

3. Averaged Perceptron (5 points)

We will be using the following definition of the perceptron, which is the multiclass or structured version of the perceptron. The training set is a bunch of input-output pairs (x_i, y_i). (For classification, y_i is a label, but for tagging, y_i is a sequence). The training algorithm is as follows:

For T iterations, iterate through each (x_i, y_i) pair in the dataset, and for each,

1. Predict $y^* := \arg\max_{y'} \theta^T f(x_i, y')$
2. If $y_i \neq y^*$: then update $\theta := \theta^{(old)} + rg$

where r is a fixed step size (e.g. $r = 1$) and g is the *gradient vector*, meaning a vector that will get added into θ for the update, specifically

$$g = \underbrace{f(x_i, y_i)}_{\text{feats of true output}} - \underbrace{f(x_i, y^*)}_{\text{feats of predicted output}}$$

Both in theory and in practice, the predictive accuracy of a model trained by the structured perceptron will be better if we use the average value of θ over the course of training, rather than the final value of θ. This is because θ wanders around and doesn't converge (typically), because it overfits to whatever data it saw most recently. After seeing t training examples, define the *averaged parameter vector* as

$$\bar{\theta}_t = \frac{1}{t} \sum_{t'=1}^{t} \theta_{t'} \tag{1}$$

where $\theta_{t'}$ is the weight vector after t' updates. (We are counting t by the number of training examples, not passes through the data. So if you had 1000 examples and made 10 passes through the data in order, the final time you see the final example is $t = 10000$.) For training, you still use the current θ parameter for predictions. But at the very end, you return the $\bar{\theta}$, not θ, as your final model parameters to use on test data.

Directly implementing equation (1) would be really slow. So here's a better algorithm. This is the same as in Hal Daume's CIML chapter on perceptrons, but adapted for the structured case (as opposed to Daume's algorithm, which assumes binary output). Define g_t to be the update vector g as described earlier. The perceptron update can be written

$$\theta_t = \theta_{t-1} + rg_t$$

Thus the averaged perceptron algorithm is, using a new 'weightsums' vector S,

1. Initialize $t = 1, \theta_0 = \vec{0}, S_0 = \vec{0}$
2. For each example i (iterating multiples times through dataset),
   - Predict $y^* = \arg\max_{y'} \theta^T f(x_i, y')$
   - Let $g_t = f(x_i, y_i) - f(x_i, y^*)$
   - Update $\theta_t = \theta_{t-1} + rg_t$
   - Update $S_t = S_{t-1} + (t-1)rg_t$
   - $t := t + 1$
3. Calculate $\bar{\theta}$ based on S

In an actual implementation, you don't keep old versions of S or θ around ... above we're using the t subscripts above just to make the mathematical analysis clearer.

Our proposed algorithm computes $\bar{\theta}_t$ as

$$\bar{\theta}_t = \theta_t - \frac{1}{t}S_t \quad (2)$$

For the following problems, feel free to set $r = 1$ just to simplify them.

For following questions write only math answers, no code required.

Question 3.1 (1 point)

What is the computational advantage of computing $\bar{\theta}$ using Equation (2) instead of directly implementing Equation (1)?

Answer:

Instead of storing all the intermediate values of θ, we only store the last θ observed i.e. θ_t. This will save a lot of space. Also, to calculate the average $\bar{\theta}_t$, we will not have to loop over t. Calculating the value of S_t along with θ_t, saves time in calculating $\bar{\theta}_t$, and reduces the complexity of $\bar{\theta}_t$ calculation from O(t) to O(1).

Now we'll show this works, at least for early iterations.

Question 3.2 (1 point)

What are $\bar{\theta}_1$, $\bar{\theta}_2$, $\bar{\theta}_3$, and $\bar{\theta}_4$? Please derive them from the Equation (1) definition, and state them in terms of g_1, g_2, g_3, and/or g_4.

Answer:

- $\bar{\theta}_1 = 1[\theta_1]$

  $= \theta_0 + rg_1$

  $= rg_1$

- $\bar{\theta}_2 = \frac{1}{2}[\theta_1 + \theta_2]$

  $= \frac{1}{2}[rg_1 + rg_1 + rg_2]$

  $= rg_1 + \frac{rg_2}{2}$

- $\bar{\theta}_3 = \frac{1}{3}[\theta_1 + \theta_2 + \theta_3]$

  $= \frac{1}{3}[rg_1 + rg_1 + rg_2 + rg_1 + rg_2 + rg_3]$

  $= rg_1 + \frac{2rg_2}{3} + \frac{rg_3}{3}$

- $\bar{\theta}_4 = \frac{1}{4}[\theta_1 + \theta_2 + \theta_3 + \theta_4]$

  $= \frac{1}{4}[rg_1 + rg_1 + rg_2 + rg_1 + rg_2 + rg_3 + rg_1 + rg_2 + rg_3 + rg_4]$

$= rg_1 + \frac{3rg_2}{4} + \frac{2rg_3}{4} + \frac{rg_4}{4}$

Question 3.3 (1 point)

What are S_1, S_2, S_3, and S_4? Please state them in terms of g_1, g_2, g_3, and/or g_4.

Answer:

- $S_1 = S_0 + 0$

  $= 0$

- $S_2 = S_1 + rg_2$

  $= rg_2$

- $S_3 = S_2 + 2rg_3$

  $= 2rg_3 + rg_2$

- $S_4 = S_3 + 3rg_4$

  $= rg_2 + 2rg_3 + 3rg_4$

Question 3.4 (2 points)

Show that Equation (2) correctly computes $\bar{\theta}_3$ and $\bar{\theta}_4$.

Answer:

- From question 3.2 we know that,

  $\bar{\theta}_3 = rg_1 + \frac{2rg_2}{3} + \frac{rg_3}{3}$

  Now, using equation(2) we have,

  $\bar{\theta}_3 = \theta_3 - \frac{1}{3}S_3$

  $= rg_1 + rg_2 + rg_3 - \frac{1}{3}[2rg_3 + rg_2]$

  $= rg_1 + rg_2 + rg_3 - \frac{2}{3}rg_3 - \frac{1}{3}rg_2$

  $= rg_1 + \frac{2rg_2}{3} + \frac{rg_3}{3}$

  Hence Proved.

- From question 3.2 we know that,

  $\bar{\theta}_4 = rg_1 + \frac{3rg_2}{4} + \frac{2rg_3}{4} + \frac{rg_4}{4}$

Now, using equation(2) we have,

$\bar{\theta}_4 = \theta_4 - \frac{1}{4}S_4$

$= rg_1 + rg_2 + rg_3 + rg_4 - \frac{1}{4}[rg_2 + 2rg_3 + 3rg_4]$

$= rg_1 + \frac{3rg_2}{4} + \frac{2rg_3}{4} + \frac{rg_4}{4}$

Hence Proved.

Question 3.5 (2 Extra Credit points)

Use proof by induction to show that this algorithm correctly computes $\bar{\theta}_t$ for any t.

Answer:

To prove, $\bar{\theta}_t = \theta_t - \frac{1}{t}S_t$

Given that,

$\theta_t = rg_1 + rg_2 + rg_3 + \ldots + rg_t$

$S_t = rg_2 + 2rg_3 + 3rg4 + \ldots + (t-1)rg_t$

- For t = 1:

  $\bar{\theta}_t = rg_1$

- For t = 2:

  $\bar{\theta}_t = rg_1 + \frac{1}{2}rg_2$

- Assume t = k holds:

  $\bar{\theta}_k = \theta_k - \frac{1}{t}S_k$

  $= rg_1 + rg_2 + \ldots + rg_k - \frac{1}{k}[rg_2 + 2rg_3 + \ldots + (k-1)rg_k]$

- To prove t = k+1 holds:

  $\bar{\theta}_{k+1} = rg_1 + rg_2 + .. + rg_k + rg_{k+1} - \frac{1}{k+1}[rg_2 + 2rg_3 + .. + (k-1)rg_k + krg_{k+1}]$

  Substituting the value of θ_k and S_k in above equation,

  $= \theta_k + rg_{k+1} - \frac{1}{k+1}[S_k + krg_{k+1}]$

  $= \theta_{k+1} - \frac{1}{k+1}[S_{k+1}]$

  Hence Proved.

4. Classifier Perceptron (20 points)

Implement the averaged perceptron for document classification, using the same sentiment analysis dataset as you used for HW1. On the first two questions, we're asking you to develop using only a subset of the data, since that makes debugging easier. On the third question, you'll run on the full dataset, and you should be able to achieve a higher accuracy compared to your previous Naive Bayes implementation. Starter code is provided in `classperc.py`.

Question 4.1 (8 points)

Implement the simple, non-averaged perceptron. Run your code on **the first 1000 training instances** for 10 passes through the training data. For each pass, report **the training and test set accuracies**.

```
(py27) D:\NLP\Homework 2>python classperc.py

[constructing dataset...]
        reading data from D:\NLP\Homework
2/large_movie_review_dataset\train\pos
        reading data from D:\NLP\Homework
2/large_movie_review_dataset\train\neg
[dataset constructed.]
[constructing dataset...]
        reading data from D:\NLP\Homework 2/large_movie_review_dataset\test\pos
        reading data from D:\NLP\Homework 2/large_movie_review_dataset\test\neg
[dataset constructed.]
[training...]

        Training iteration 0
TR RAW EVAL: 1631/2000 = 0.8155 accuracy
DEV RAW EVAL: 1472/2000 = 0.7360 accuracy
        Training iteration 1
TR RAW EVAL: 1705/2000 = 0.8525 accuracy
DEV RAW EVAL: 1521/2000 = 0.7605 accuracy
        Training iteration 2
TR RAW EVAL: 1656/2000 = 0.8280 accuracy
DEV RAW EVAL: 1409/2000 = 0.7045 accuracy
        Training iteration 3
TR RAW EVAL: 1864/2000 = 0.9320 accuracy
DEV RAW EVAL: 1555/2000 = 0.7775 accuracy
        Training iteration 4
TR RAW EVAL: 1743/2000 = 0.8715 accuracy
DEV RAW EVAL: 1419/2000 = 0.7095 accuracy
        Training iteration 5
TR RAW EVAL: 1919/2000 = 0.9595 accuracy
DEV RAW EVAL: 1591/2000 = 0.7955 accuracy
        Training iteration 6
TR RAW EVAL: 1785/2000 = 0.8925 accuracy
DEV RAW EVAL: 1425/2000 = 0.7125 accuracy
        Training iteration 7
TR RAW EVAL: 1969/2000 = 0.9845 accuracy
DEV RAW EVAL: 1566/2000 = 0.7830 accuracy
        Training iteration 8
TR RAW EVAL: 1985/2000 = 0.9925 accuracy
```

```
DEV RAW EVAL: 1591/2000 = 0.7955 accuracy
        Training iteration 9
TR RAW EVAL: 1993/2000 = 0.9965 accuracy
DEV RAW EVAL: 1621/2000 = 0.8105 accuracy

[learned weights for 77998 features from 2000 examples.]
```

Question 4.2 (8 points)

Implement the averaged perceptron. Run your code on **the first 1000 training instances** for 10 passes through the training data. For each pass, compute the $\bar{\theta}$ so far, and report its **test set accuracy**.

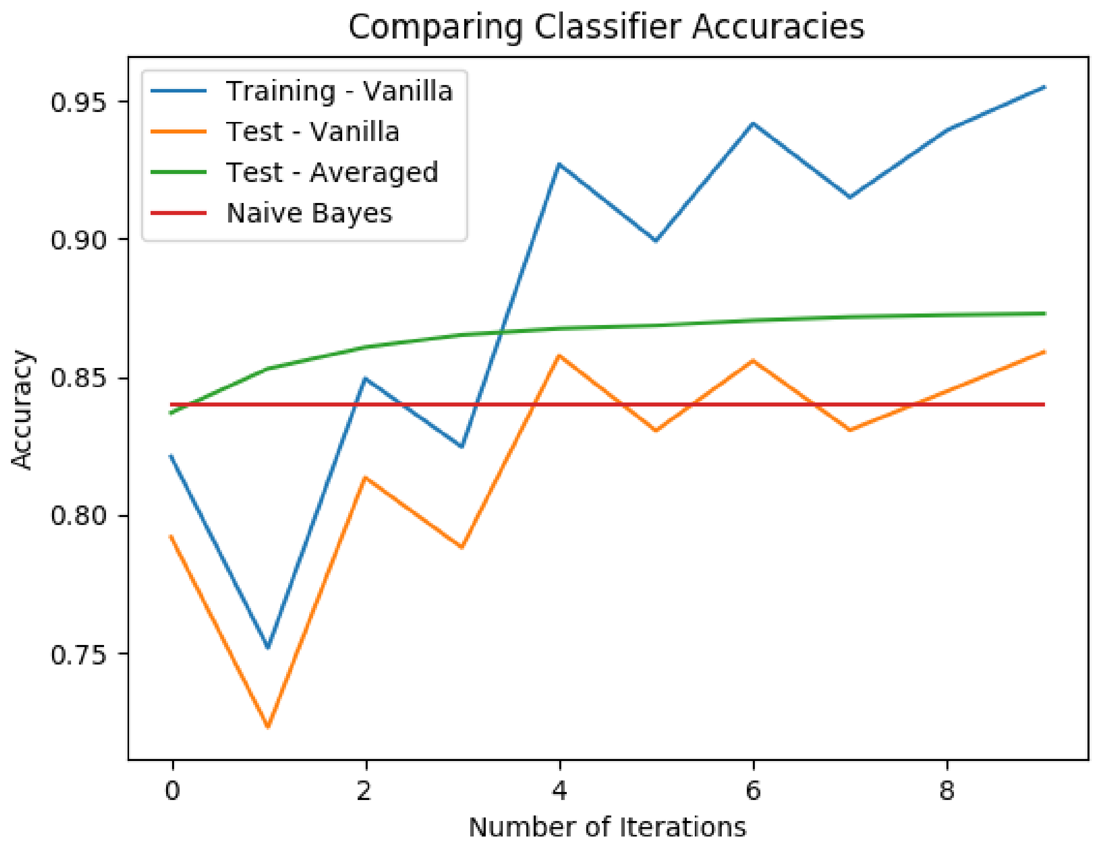
```
(py27) D:\NLP\Homework 2>python classperc.py

[constructing dataset...]
        reading data from D:\NLP\Homework
2/large_movie_review_dataset\train\pos
        reading data from D:\NLP\Homework
2/large_movie_review_dataset\train\neg
[dataset constructed.]
[constructing dataset...]
        reading data from D:\NLP\Homework 2/large_movie_review_dataset\test\pos
        reading data from D:\NLP\Homework 2/large_movie_review_dataset\test\neg
[dataset constructed.]
[training...]

        Training iteration 0
TR RAW EVAL: 1642/2000 = 0.8210 accuracy
DEV RAW EVAL: 1490/2000 = 0.7450 accuracy
DEV AVG EVAL: 1460/2000 = 0.7300 accuracy
        Training iteration 1
TR RAW EVAL: 1709/2000 = 0.8545 accuracy
DEV RAW EVAL: 1477/2000 = 0.7385 accuracy
DEV AVG EVAL: 1529/2000 = 0.7645 accuracy
        Training iteration 2
TR RAW EVAL: 1316/2000 = 0.6580 accuracy
DEV RAW EVAL: 1198/2000 = 0.5990 accuracy
DEV AVG EVAL: 1566/2000 = 0.7830 accuracy
        Training iteration 3
TR RAW EVAL: 1867/2000 = 0.9335 accuracy
DEV RAW EVAL: 1563/2000 = 0.7815 accuracy
DEV AVG EVAL: 1578/2000 = 0.7890 accuracy
        Training iteration 4
TR RAW EVAL: 1919/2000 = 0.9595 accuracy
DEV RAW EVAL: 1587/2000 = 0.7935 accuracy
DEV AVG EVAL: 1590/2000 = 0.7950 accuracy
        Training iteration 5
TR RAW EVAL: 1898/2000 = 0.9490 accuracy
DEV RAW EVAL: 1565/2000 = 0.7825 accuracy
DEV AVG EVAL: 1597/2000 = 0.7985 accuracy
        Training iteration 6
TR RAW EVAL: 1963/2000 = 0.9815 accuracy
DEV RAW EVAL: 1582/2000 = 0.7910 accuracy
DEV AVG EVAL: 1608/2000 = 0.8040 accuracy
        Training iteration 7
TR RAW EVAL: 1949/2000 = 0.9745 accuracy
DEV RAW EVAL: 1579/2000 = 0.7895 accuracy
DEV AVG EVAL: 1610/2000 = 0.8050 accuracy
        Training iteration 8
TR RAW EVAL: 1975/2000 = 0.9875 accuracy
DEV RAW EVAL: 1593/2000 = 0.7965 accuracy
DEV AVG EVAL: 1612/2000 = 0.8060 accuracy
        Training iteration 9
TR RAW EVAL: 1992/2000 = 0.9960 accuracy
DEV RAW EVAL: 1608/2000 = 0.8040 accuracy
DEV AVG EVAL: 1615/2000 = 0.8075 accuracy

[learned weights for 77146 features from 2000 examples.]
```

Question 4.3 (4 points)

Graph four curves on the same plot, using the **full dataset**:

- accuracy of the vanilla perceptron on the training set
- accuracy of the vanilla perceptron on the test set
- accuracy of the averaged perceptron on the test set
- accuracy of your Naive Bayes classifier from HW1 (you don't need to re-run it; just take the best accuracy from your previous results).

The x-axis of the plot should show the number of iterations through the training set and the y-axis should show the accuracy of the classifier. For this part of the HW run your code on **the entire dataset** (all instances). Since Naive Bayes doesn't require multiple passes through the data just produce a single horizontal line showing its overall accuracy. Make sure your plot has a title, a label on the x-axis, a label on the y-axis and a legend showing which line is which. Explain verbally what's happening in this plot.



The above graphs shows different accuracy values for Vanilla Perceptron, Averaged Perceptron and Naive Bayes Classifier. According to this graph, Average Perceptron gives the best results on the test data. The accuracy for Averaged Perceptron increases gradually and becomes stable with increasing number of iterations. The accuracy for Vanilla Perceptron keeps fluctuating and closely resembles the accuracy on training set. The performance of Naive Bayes Classifier is constant and is independent of the number of iterations.

5. Structured Perceptron with Viterbi (40 points)

In this problem, you will implement a part-of-speech tagger for Twitter, using the structured perceptron algorithm. Your system will be not too far off from state of the art performance, coding it all up yourself from scratch!

The dataset comes from http://www.ark.cs.cmu.edu/TweetNLP/ (http://www.ark.cs.cmu.edu/TweetNLP/) and is described in the papers listed there (Gimpel et al.~2011 and Owoputi et al.~2013). The Gimpel article describes the tagset; the annotation guidelines on that webpage describe it futher.

Your structured perceptron will use your Viterbi implementation from 2.2 as a subroutine. If that's buggy, this will cause many problems here---your perceptron will have really weird behavior. (This happened to us when designing your assignment!) If you have problems, try using the greedy decoding algorithm, which we provide in the starter code. Make sure to note which decoding algorithm you're using in your writeup.

The starter code is `structperc.py` and it assumes the two data files `oct27.train` and `oct27.dev` are in the same directory. (For simplicity we're just going to use this `dev` set as our test set.)

Question 5.1 (2 points)

First let's do a little data analysis to establish the **most common tag** baseline accuracy. Using a small script, load up the dev dataset (oct27.dev) using the function `structperc.read_tagging_file` (from `import structperc`). Calculate the following: What is the most common tag, and what would your accuracy be if you predicted it for all tags?

In [129]:
```
from structperc import *

ret = read_tagging_file("oct27.dev")
tag_count = {tag:0 for tag in OUTPUT_VOCAB}

for row in ret:
    for tag in row[1]:
        if tag in tag_count.keys():
            tag_count[tag] = tag_count[tag] + 1

total_count = sum(tag_count.values())

common_tag = max(tag_count, key=tag_count.get)
max_count = tag_count[common_tag]
print "The most common tag is: %s" % common_tag

accuracy = float(max_count)/float(total_count)
print "Accuracy: %.4f" % accuracy
```

```
The most common tag is: V
Accuracy: 0.1614
```

The structured perceptron algorithm works very similarly as the classification version you did in the previous question, except the prediction function uses Viterbi as a subroutine, which has to call feature extraction functions for local emissions and transition factors. There also has to be a large overall feature extraction function for an entire structure at once. The following parts will build up these pieces. First, we will focus on inference, not learning.

Question 5.2 (2 points)

We provide a barebones version of `local_emission_features`, which calculates the local features for a particular tag at a token position. You can run this function all by itself. Make up an example sentence, and call this function with it, giving it a particular index and candidate tag. Show the code for the function call you made and the function's return value, and explain what the features mean (just a sentence or two).

In [130]:
```
sentence = ["I",'love','dogs']
feats = local_emission_features(2, 'N', sentence)
print feats
```

```
{'tag=N_curword=dogs': 1, 'tag=N_biasterm': 1}
```

Explanation:

The function local_emission_features returns 2 features. The first feature 'tag=N_curword=dogs' assigns 1 to a particular word("dogs") of being of the type candidate tag("N"). The second feature 'tag=N_biasterm' adds a bias to the occurance of candidate tag("N").

Question 5.3 (2 points)

Implement `features_for_seq()`, which extracts the full feature vector $f(x, y)$, where $x$is a sentence and y is an entire tagging sequence for that sentence. This will add up the feature vectors from each local emissions features for every position, as well as transition features for every position (there are $N - 1$ of them, of course). Show the output on a very short example sentence and example proposed tagging, that's only 2 or 3 words long.

To define $f(x, y)$ a little more precisely: If $f^{(B)}(t, x, y)$ means the local emissions feature vector at position t (i.e. the `local_emission_features` function), and $f^{(A)}(y_{t-1}, y_t, y)$ is the transition feature function for positions $(t - 1, t)$ (which just returns a feature vector where everything is zero, except a single element is 1), then the full sequence feature vector will be the vector-sum of all those feature vectors:

$$f(x, y) = \sum_t^T f^{(B)}(t, x, y) + \sum_{t=2}^T f^{(A)}(y_{t-1}, y_t)$$

You implemented $f^{(B)}$ above. You probably don't need to bother implementing $f^{(A)}$ as a standalone function. You will have to decide on a particular convention to encode the name of a transition feature. For example, one way to do it is with string concatenation like this, `"trans_%s_%s" % (prevtag, curtag)`, where prevtag and curtag are strings. Or you could use a python tuple of strings, which works since tuples have the ability to be keys in a python dictionary.

In other words: the emissions and transition features will all be in the same vector, just as keys in the dictionary that represents the feature vector. The transition features are going to be the count of how many times a particular transition (tag bigram) happened. The emissions features are going to be the vector-sum of all the local emission features, as calculated from `local_emission_features`.

In [131]:
```
sentence = ["I",'love','dogs']
labels = ["V","V","N"]
print "The full feature vector is: \n"
full_feacture = features_for_seq(sentence, labels)
print full_feacture
```

```
The full feature vector is:

defaultdict(<type 'float'>, {'tag=V_curword=love': 1.0, 'trans_V_V': 1.0, 'tag=
V_curword=I': 1.0, 'tag=V_biasterm': 2.0, 'tag=N_biasterm': 1.0, 'tag=N_curword
=dogs': 1.0, 'trans_V_N': 1.0})
```

Question 5.4 (4 points)

Look at the starter code for `calc_factor_scores`, which calculates the A and B score functions that are going to be passed in to your Viterbi implementation from problem 2, in order to do a prediction. The only function it will need to call is `local_emission_features`. It should NOT call `features_for_seq`. Why not?

Answer:

Viterbi assumes that the n^{th} output/observation depends only on the n^{th} hidden state. To use Virterbi, we need to consider only the y_t and y_{t-1} tags at a time. In short, we need the local features and not the global ones. Hence we dont need the features_for_seq which is a global feature vector for the entire sequence of tags and tokens.

Question 5.5 (6 points)

Implement `calc_factor_scores`. Make up a simple example (2 or 3 words long), with a simple model with at least some nonzero features (you might want to use a `defaultdict(float)`, so you don't have to fill up a dict with dummy values for all possible transitions), and show your call to this function and the output.

In [132]:
```
# Possible tags = ['V', 'N']
sentence = ['I','love','dogs']

weights = defaultdict(float)
weights["trans_V_N"] = 5
weights["trans_N_N"] = 3
weights["tag=N_curword=I"] = 4
weights["tag=V_curword=love"] = 7
weights["tag=N_biasterm"] = 6
Ascores, Bscores = calc_factor_scores(sentence, weights)

print Ascores
print Bscores
```

```
{('V', 'N'): 5, ('V', 'V'): 0, ('N', 'N'): 3, ('N', 'V'): 0}
[{'V': 0, 'N': 10}, {'V': 7, 'N': 6}, {'V': 0, 'N': 6}]
```

Question 5.6 (4 points)

Implement `predict_seq()`, which predicts the tags for an input sentence, given a model. It will have to calculate the factor scores, then call Viterbi as a subroutine, then return the best sequence prediction. If your Viterbi implementation does not seem to be working, use the implementation of the greedy decoding algorithm that we provide (it uses the same inputs as `vit_starter.viterbi()`).

In [133]:
```
def predict_seq(tokens, weights):
    """
    takes tokens and weights, calls viterbi and returns the most likely
    sequence of tags
    """
    Ascores, Bscores = calc_factor_scores(tokens, weights)

    if (tokens == ['I', 'love', 'dogs']):
        OUTPUT_VOCAB = ['N', 'V']
    else:
        OUTPUT_VOCAB = set(""" ! # $ & , @ A D E G L M N O P R S T U V X Y Z ^ ""

    predlabels = vit_starter.viterbi(Ascores, Bscores, OUTPUT_VOCAB)

    return predlabels

example_seq = predict_seq(sentence, weights)
print example_seq
```

```
['N', 'V', 'N']
```

OK, you're done with the inference part. Time to put it all together into the parameter learning algorithm and see it go.

Question 5.7 (14 points)

Implement `train()`, which does structured perceptron training with the averaged perceptron algorithm. You should train on oct27.train, and evaluate on oct27.dev. You will want to first get it working without averaging, then add averaging to it. Run it for 10 iterations, and print the devset accuracy at each training iteration. Note that we provide evaluation code, which assumes `predict_seq()` and everything it depends on is working properly.

For us, here's the performance we get at the first and last iterations, using the features in the starter code (just the bias term and the current word feature, without case normalization).

```
Training iteration 0
DEV RAW EVAL: 2556/4823 = 0.5300 accuracy
DEV AVG EVAL: 2986/4823 = 0.6191 accuracy
...
Training iteration 9
DEV RAW EVAL: 3232/4823 = 0.6701 accuracy
DEV AVG EVAL: 3341/4823 = 0.6927 accuracy
Learned weights for 24361 features from 1000 examples
```

```
(py27) D:\NLP\Homework 2>python structperc.py

Training iteration 0
TR  RAW EVAL: 8523/14619 = 0.5830 accuracy
DEV RAW EVAL: 2556/4823 = 0.5300 accuracy
DEV AVG EVAL: 2989/4823 = 0.6197 accuracy
Training iteration 1
TR  RAW EVAL: 10643/14619 = 0.7280 accuracy
```

```
DEV RAW EVAL: 2956/4823 = 0.6129 accuracy
DEV AVG EVAL: 3170/4823 = 0.6573 accuracy
Training iteration 2
TR  RAW EVAL: 10148/14619 = 0.6942 accuracy
DEV RAW EVAL: 2692/4823 = 0.5582 accuracy
DEV AVG EVAL: 3272/4823 = 0.6784 accuracy
Training iteration 3
TR  RAW EVAL: 11681/14619 = 0.7990 accuracy
DEV RAW EVAL: 3150/4823 = 0.6531 accuracy
DEV AVG EVAL: 3304/4823 = 0.6851 accuracy
Training iteration 4
TR  RAW EVAL: 11722/14619 = 0.8018 accuracy
DEV RAW EVAL: 3028/4823 = 0.6278 accuracy
DEV AVG EVAL: 3309/4823 = 0.6861 accuracy
Training iteration 5
TR  RAW EVAL: 11799/14619 = 0.8071 accuracy
DEV RAW EVAL: 3003/4823 = 0.6226 accuracy
DEV AVG EVAL: 3327/4823 = 0.6898 accuracy
Training iteration 6
TR  RAW EVAL: 10765/14619 = 0.7364 accuracy
DEV RAW EVAL: 2839/4823 = 0.5886 accuracy
DEV AVG EVAL: 3333/4823 = 0.6911 accuracy
Training iteration 7
TR  RAW EVAL: 12355/14619 = 0.8451 accuracy
DEV RAW EVAL: 3076/4823 = 0.6378 accuracy
DEV AVG EVAL: 3339/4823 = 0.6923 accuracy
Training iteration 8
TR  RAW EVAL: 11224/14619 = 0.7678 accuracy
DEV RAW EVAL: 2947/4823 = 0.6110 accuracy
DEV AVG EVAL: 3334/4823 = 0.6913 accuracy
Training iteration 9
TR  RAW EVAL: 12581/14619 = 0.8606 accuracy
DEV RAW EVAL: 3232/4823 = 0.6701 accuracy
DEV AVG EVAL: 3336/4823 = 0.6917 accuracy

Learned weights for 24359 features from 1000 examples
```

Question 5.8 (6 points)

Print out a report of the accuracy rate for each tag in the development set. We provided a function to do this `fancy_eval`. Look at the two sentences in the dev data, and in your writeup show and compare the gold-standard tags versus your model's predictions for them. Consult the tagset description to understand what's going on. What types of things does your tagger get right and wrong?

To look at the examples, you may find it convenient to use `show_predictions` (or write up the equivalent manually). For example, after 1 iteration of training, we get this output from the first sentence in the devset. (After investigating TV shows that were popular in 2011 when the tweet was authored, we actually think some of the gold-standard tags in this example might be wrong.)

```
word                gold pred
----                ---- ----
@ciaranyree         @    @
it                  O    O
was                 V    V
on                  P    P
football            N    ^     *** Error
wives               N    N
,                   ,    ,
one                 $    $
of                  P    P
the                 D    D
players             N    N
and                 &    &
his                 D    D
wife                N    N
own                 V    V
smash               ^    D     *** Error
burger              ^    N     *** Error
```

To do this part, you may find it useful to save your model's weights with pickle.dumps (or json.dumps) and have a short analysis script that loads the model and devdata to do the reports. If you have to re-train each time you tweak your analysis code, it can be annoying.

In [134]:
```
import pickle

# Analysis Report
sentence_count = 1
final_weights = pickle.load( open( "final_weights.p", "rb" ) )
test_set = read_tagging_file("oct27.dev")

print "Accuracy rate for each tag in the development set: \n"
fancy_eval(test_set,final_weights)

print("\n\n")
for tokens,goldlabels in test_set:
    if sentence_count > 2:
        break
    predlabels = predict_seq(tokens, final_weights)
    show_predictions(tokens, goldlabels, predlabels)
    print("\n")
    sentence_count = sentence_count + 1
```

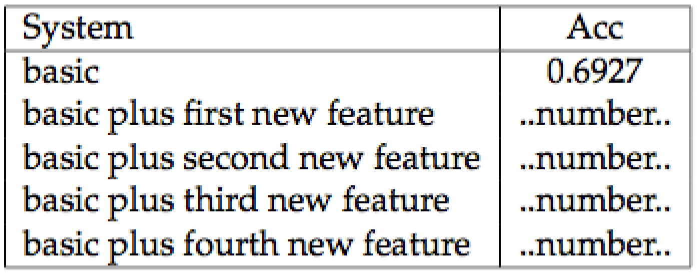
```
Accuracy rate for each tag in the development set:

gold O acc 0.9489 (316/333)
gold , acc 0.9460 (473/500)
gold D acc 0.9263 (289/312)
gold P acc 0.9250 (407/440)
gold & acc 0.9231 (84/91)
gold V acc 0.8296 (623/751)
gold L acc 0.7538 (49/65)
gold T acc 0.7500 (27/36)
gold N acc 0.6970 (460/660)
gold R acc 0.6507 (136/209)
gold A acc 0.5941 (142/239)
gold E acc 0.5577 (29/52)
gold ! acc 0.5152 (51/99)
gold @ acc 0.4280 (104/243)
gold $ acc 0.3605 (31/86)
gold U acc 0.2747 (25/91)
gold ^ acc 0.2379 (74/311)
gold G acc 0.2000 (13/65)
gold # acc 0.0577 (3/52)
gold S acc 0.0000 (0/5)
gold X acc 0.0000 (0/4)
gold Z acc 0.0000 (0/9)
gold ~ acc 0.0000 (0/170)



word                gold pred
----                ---- ----
@ciaranyree         @    @
it                  O    O
was                 V    V
on                  P    P
football            N    N
wives               N    V      *** Error
,                   ,    ,
one                     $    $
of                      P    P
the                     D    D
players                 N    N
and                     &    &
his                     D    D
wife                    N    N
own                     V    N     *** Error
smash                   ^    V     *** Error
burger                  ^    @     *** Error


word                    gold pred
----                    ---- ----
RT                      ~    A     *** Error
@TheRealQuailman        @    N     *** Error
:                       ~    ,     *** Error
Currently               R    A     *** Error
laughing                V    N     *** Error
at                      P    P
Laker                   ^    ^
haters                  N    N
.                       ,    ,
```

Analysis:

The tagger usually does not perform as well for the tags that are less common like S, X, Z, all of which occur less than 10 times in the dataset. As mentioned earlier, some of the gold-standard tags in this example appear to be wrong. This can be one of the reasons why the tagger is not able to identify them correctly (for ex, 'burger', 'smash'). The performance is really poor for ~ (indications of continuation of a message across multiple tweets) and inspite of 170 occurances, the tagger still fails to get it right. The tagger identifies the tokens like ":" as punctuations and not as discourse markers.

Question 5.9 (OPTIONAL: 4 Extra Credit points)

Improve the features of your tagger to improve accuracy on the development set. This will only require changes to `local_emission_features`. Implement at least 4 new types of features. Report your tagger's accuracy with these improvements. Please make a table that reports accuracy from adding different features. The first row should be the basic system, and the last row should be the fanciest system. Rows in between should report different combinations of features. One simple way to do this is, if you have 4 different feature types, to run 4 experiments where in each one, you add only one feature type to the basic system. For example:



System	Acc
basic	0.6927
basic plus first new feature	..number..
basic plus second new feature	..number..
basic plus third new feature	..number..
basic plus fourth new feature	..number..

Hint: if you make features about the first character of a word, that helps a lot for the # (hashtag) and @ (at-mention) tags. The URL tag is easy to get too with a similar form of character affix analysis. Character affixes help lots of other tags too. Also, if you have a feature that looks at the word at position t, you can make new versions of it that look to the left or right of the t^{th} position in question: for example, 'word_to_left=the'.

In [ ]: