Problem Set 2: Text classification

(*This problem set is graded out of 50 for students taking CS4650 and out of 53 for students taking CS7650. But like all problem sets, this will count towards 8% of your final grade*)

In this problem set, you will build a system for classifying book reviews on amazon as positive or negative. You will:

- Do some basic text processing, tokenizing your input and converting it into a bag-of-words representation
- Build a classifier based on sentiment word lists
- Build a machine learning classifier based on the generative model, using Naive Bayes
- Evaluate your classifiers and examine what they have learned
- Build a machine learning classifier based on the discriminative model, using Perceptron
- Build more stable discriminative classifier, using the averaged perceptron
- Build the logistic regression classifier (See instructions within the section)
- Implement techniques to improve your classifier
- Participate in a hopefully fun bakeoff competition.

To turn in this project, please submit on T-square:

- this notebook
- all files in the `gtnlplib` directory

0. Set up

In order to develop this assignment, you will have to install the following, if you don't have it already.

- python 2.7 (https://www.python.org/downloads/release/python-2710/) (and not Python 3, although if somebody wants to try that and tell us what goes wrong, that would be appreciated...)
- jupyter notebook (http://jupyter.readthedocs.org/en/latest/install.html)
- scipy (http://www.scipy.org/install.html)
- numpy (This will come if you install scipy like above, but if not install separately)
- nltk (http://www.nltk.org/install.html) (tested on NLTK 3.0.4)
- matplotlib (http://matplotlib.org/users/installing.html)
- nosetests (https://nose.readthedocs.org/en/latest/)

You also need to get the data, which is available here (https://github.com/jacobeisenstein/gt-nlp-class/releases/tag/amazon-fall-2015). Unzip this in the "data" directory of this project.

Notes

- The code with this assignment also contains test*.py files. These are used by nosetests which is one of python's framework to test your code. We will use these tests to help grade; you can run them too if you want. You can learn more about how to run them here (http://pythontesting.net/framework/nose/nose-introduction/).
- You are free to add more tests, but that is completely optional.
- Jupyter runs in a web browser. You want to be careful about the text output from your code; if you try to output a huge amount of text, the browser will require a lot of memory, and so the notebook will become very slow and hard to use.

You are also given some code to start with. As you progress, you will be writing the missing pieces in the code and/or implementing new code. All the code is in the **gtnlplib** directory which came as part of the assignment.

```
In [5]: import numpy as np
        from collections import defaultdict
        import gtnlplib.preproc
        import gtnlplib.preproc_metrics

        import gtnlplib.clf_base
        import gtnlplib.wordlist
        import gtnlplib.naivebayes
        import gtnlplib.perceptron
        import gtnlplib.avg_perceptron
        import gtnlplib.logreg

        import gtnlplib.scorer
        import gtnlplib.constants
        import gtnlplib.analysis

        # this enables you to create inline plots in the notebook
        %pylab inline

        Populating the interactive namespace from numpy and matplotlib
```

While developing the modules of the library, it is likely that you make mistakes. If that happens, you correct the mistake and reload whichever modules you edited, as shown below. This is part of the python development cycle. If reloading doesn't work, restart the kernel.

```
In [6]: reload(gtnlplib.preproc)
        reload(gtnlplib.preproc_metrics)

Out[6]: <module 'gtnlplib.preproc_metrics' from 'gtnlplib/preproc_metric
        s.pyc'>
```

1. Data Processing

(*Completing gtnlplib.preproc.docsToBOWs() - 3 pts, each question in Deliverable 1 is worth 1 pt*)

Your first step is to write code that can apply the following preprocessing steps. You will have to run this code fairly quickly on the test data when you receive it, so make sure it is modular and well-written.

- You will edit `gtnlplib.preproc.docsToBOWs` that takes as its argument a "key" document. It should produce a "BOW" (bag-of-words) document. Each line of the key document contains a filename and a label. Each line of the BOW document should contain a BOW representation of the corresponding file in the key document.
- A BOW representation looks like this: "word:count word:count word:count..." for every word that appears in the document. Do not print words that have zero count. Use space delimiters.
- Use NLTK's tokenization package (http://nltk.org/api/nltk.tokenize.html) function to divide each file into sentences, and each sentence into tokens.
- Downcase all tokens
- Only consider tokens that are completely alphabetic.

In [7]:
```
### TRAINKEY, DEVKEY and TESTKEY are defined in the gtnlplib.con
stants module

gtnlplib.preproc.docsToBOWs(gtnlplib.constants.TRAINKEY)
gtnlplib.preproc.docsToBOWs(gtnlplib.constants.DEVKEY)
## uncomment once you have the test data
#gtnlplib.preproc.docsToBOWs(gtnlplib.constants.TESTKEY)
```

The `gtnlplib.preproc` module defines a generator function (http://wiki.python.org/moin/Generators), called "dataIterator"

- This allows you to easily iterate through the dataset defined by a given keyfile.
- Each time you call "next" (possibly implicitly), it returns a dict containing features and counts for the next document in the sequence.
- In this case, the features include the words, and a special "offset" feature
- This is equivalent to $\mathbf{x}_i$ in the reading.
- You can see how this is used in the getAllCounts() function below, which takes a dataIterator as an argument.

Lines 7-8 of the code in the dataIterator function might look confusing if you are not a pythonista.

- This is a list comprehension (http://legacy.python.org/dev/peps/pep-0202/) nested inside a dict comprehension (http://legacy.python.org/dev/peps/pep-0274/).
- Here's an introduction (http://carlgroner.me/Python/2011/11/09/An-Introduction-to-List-Comprehensions-in-Python.html) with more examples.

Sanity check: How many unique words appear in the training set? (Types, not tokens.) In order to get this one correct you should pass the test "test_number_of_tokens" in testpreproc.py file.

In [8]:
```
reload(gtnlplib.preproc)
ac_train = gtnlplib.preproc.getAllCounts(gtnlplib.preproc.dataIt
erator(gtnlplib.constants.TRAINKEY))
ac_dev = gtnlplib.preproc.getAllCounts(gtnlplib.preproc.dataIter
ator(gtnlplib.constants.DEVKEY))
print "number of word types",len(ac_train.keys())-1
```

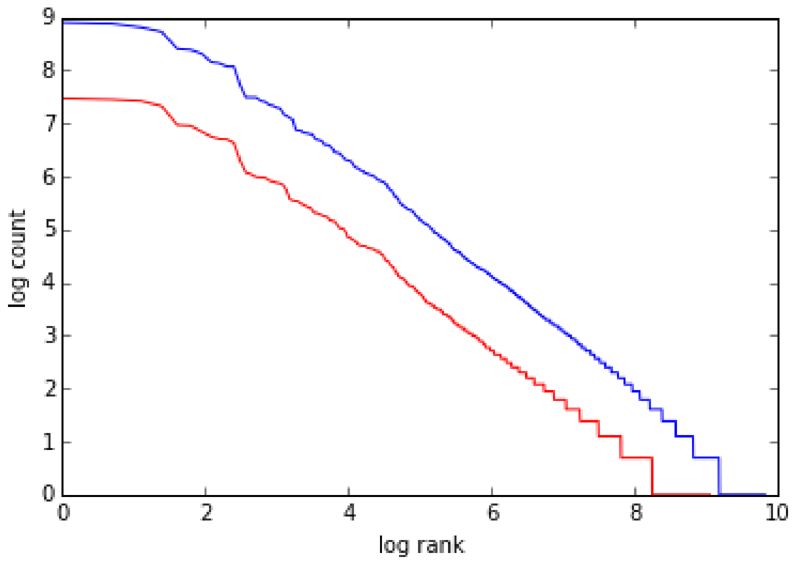
```
number of word types 18430
```

The following code makes a plot, with the log-rank (from 1 to the log of the total number of words) on the x-axis and the log count on the y-axis.

In [9]:
```
tr_logcounts = np.log(np.array(sorted(ac_train.values(),revers
e=True)))
plt.plot(np.log(range(len(tr_logcounts))),tr_logcounts)
dv_logcounts = np.log(np.array(sorted(ac_dev.values(),reverse=Tr
ue)))
plt.plot(np.log(range(len(dv_logcounts))),dv_logcounts,'r')
plt.xlabel('log rank')
plt.ylabel('log count')
```

```
/usr/local/lib/python2.7/dist-packages/ipykernel/__main__.py:2:
RuntimeWarning: divide by zero encountered in log
  from ipykernel import kernelapp as app
/usr/local/lib/python2.7/dist-packages/ipykernel/__main__.py:4:
RuntimeWarning: divide by zero encountered in log
```

Out[9]:
```
<matplotlib.text.Text at 0x7f5f0d10d310>
```



Deliverable 1

(1 point each)

- Explain what you see in the plot.
- Print the token/type ratio for the training data. You will have to implement `gtnlplib.preproc_metrics.get_token_type_ratio`
- Print the number of types which appear exactly once in the training data. These are called hapax-legomena (https://en.wikipedia.org/wiki/Hapax_legomenon). You will have to implement `gtnlplib.preproc_metrics.type_frequency`
- Print the number of types that appear in the dev data but not the training data (hint: use sets (https://docs.python.org/2/library/sets.html) for this). You will have to implement `gtnlplib.preproc_metrics.unseen_types`

Explain what you see in the plot:

1) The log(count) is appoximately linear with log(rank), i.e., log(count) ~ klog(rank), in other words, count ~ rank^k. 2) For the words with top ranks, the counts are similar. Because these words are typically some stopwords, like 'a', 'the', etc. 3) The figure in the tail have step-shaped curve. This means there are many words rarely appear.

It shows the word counts follows Zipf's law (https://en.wikipedia.org/wiki/Zipf%27s_law (https://en.wikipedia.org/wiki/Zipf%27s_law)), i.e., the word count follows the power law: a small fraction of words occupy the most amount of counts; while many other words only appear several times. The word count is long tail.

In [10]:
```
# You will have to implement this function
print 'tt-train', gtnlplib.preproc_metrics.get_token_type_rati
o(ac_train)
print 'tt-dev', gtnlplib.preproc_metrics.get_token_type_ratio (a
c_dev)
```

```
tt-train 13.1485540665
tt-dev 6.92344665885
```

In [11]:
```
# You will have to implement this function
print 'tr-hapax-legomena',gtnlplib.preproc_metrics.type_frequenc
y (ac_train, 1)
print 'de-hapax-legomena',gtnlplib.preproc_metrics.type_frequenc
y (ac_dev, 1)
```

```
tr-hapax-legomena 8758
de-hapax-legomena 4738
```

In [12]:
```
# You will have to implement this function
print 'unseen', gtnlplib.preproc_metrics.unseen_types (ac_train,
ac_dev)
```

```
unseen 2407
```

2. Basic classification

(*Completing predict() - 3 pts*)

To get started, we build a simple classifier, which labels all instances as positive. This is the "most common class" (MCC) baseline. Take a look at the implementation to see how the weights are stored and set.

In [13]:
```
weights_mcc = gtnlplib.wordlist.learnMCCWeights ()
print weights_mcc
```

```
defaultdict(<type 'int'>, {('NEU', '**OFFSET**'): 0, ('POS',
'**OFFSET**'): 1, ('NEG', '**OFFSET**'): 0})
```

To use these weights in a classifier, you need to complete `gtnlplib.clf_base.predict`, which represents the inner-product computation $\theta' \mathbf{f}(\mathbf{x}, y)$. It should have the following characteristics:

- **Input 1** an instance, represented as a dict (with features as keys and counts as values)
- **Input 2** a dictionary of weights, where keys are tuples of features and labels, and weights are the values. This corresponds to θ in the reading. See example below.
- **Input 3** a list of possible candidate class labels
- **Output 1** the highest-scoring label
- **Output 2** a dict with labels as keys and scores as values

Then you can call `gtnlplib.clf_base.evalclassifier` to compute accuracy. The relevant tests are testwlc.py.

In [14]:
```
outfile = 'all_pos.txt'
mat = gtnlplib.clf_base.evalClassifier(weights_mcc,outfile, gtnl
plib.constants.DEVKEY)
print gtnlplib.scorer.printScoreMessage(mat)
```

```
3 classes in key: set(['NEG', 'NEU', 'POS'])
1 classes in response: set(['POS'])
confusion matrix
key\response:   POS
NEG             111
NEU             135
POS             148
----------------
accuracy: 0.3756 = 148/394

None
```

Sanity check: You should get 37.56% accuracy just by classifying everything as positive.

- The printed output is a **confusion matrix**.
- The rows indicate the key and the columns indicate the response.
- In this case, the response is always "POS", so there is only one column.
- The cell NEG/POS tells you how often an example that was labeled "NEG" in the key was labeled "POS" in the system response.

3. Word list classification

(*setting weights - 2 pts, Deliverable 3 - 1 pt. Total 3 pts*)

- We will now build a sentiment analysis system based on word lists.
- The file "data/sentiment-vocab.tff" contains a sentiment lexicon from Wilson et al 2005 (http://people.cs.pitt.edu/~wiebe/pubs/papers/emnlp05polarity.pdf).
- The provided function `gtnlplib.wordlist.loadSentimentWords` reads the lexicon into memory, building sets of positive and negative words.

In [15]:
```
poswords, negwords = gtnlplib.wordlist.loadSentimentWords (gtnlp
lib.constants.SENTIMENT_FILE)
```

Now write a classifier that classifies each instance in a testfile. The classification rule is:

- 'POS' if the instance has more words from the positive list than the negative list
- 'NEG' if the instance has more words from the negative list than the positive list
- 'NEU' (neutral) if the instance has the same number of words from each list

Deliverable 3: run your classifier on dev.key, and use the following code to print the resulting confusion matrix. For this you will have to implement `gtnlplib.wlclf.learnWLCWeights` function based on the instructions given earlier in the section.

The confusion matrix should now have three columns, since the response should include every class at least once. The count of correct responses is found on the diagonal of the confusion matrix. What is the most frequent type of error?

In [ ]:
```
Answer:
    Many "NEU" comments are predicted as "POS". Our classifier a
re highly biased by the possitive words. I think
it is possible, since many sentences containing positive words a
re not saying positively, like "not good".
```

In [16]:
```
reload(gtnlplib.wordlist)
weights_wlc = gtnlplib.wordlist.learnWLCWeights (poswords, negwo
rds)
outfile = 'word_list.txt'
mat = gtnlplib.clf_base.evalClassifier(weights_wlc,outfile, gtnl
plib.constants.DEVKEY)
print gtnlplib.scorer.printScoreMessage(mat)
```

```
3 classes in key: set(['NEG', 'NEU', 'POS'])
3 classes in response: set(['NEG', 'NEU', 'POS'])
confusion matrix
key\response:   NEG     NEU     POS
NEG             36      14      61
NEU             22      14      99
POS             9       13      126
----------------
accuracy: 0.4467 = 176/394

None
```

4. Naive Bayes

(*Completing learnNBWeights() - 5 pts, Deliverable 4a - 1pt, 4b - 1 pt, explanation of plot output - 2pts. Total 8 points*)

Now you will implement a Naive Bayes classifier.

You already have the code for the decision function, "predict". So you just need to construct a set of weights that correspond to the classifier. These weights will contain two parameters:

- $\log \mu$ for the offset, which parametrizes the prior $\log P(y)$
- $\log \phi$ for the word counts, which parametrizes the likelihood $\log P(x|y)$

You should use maximum *a posteriori* estimation of the parameter ϕ,

$$\phi_{j,n} = P(w = n|y = j) = \frac{\sum_{i:y_i=j} x_{i,n} + \alpha}{\sum_{i:y_i=j} \sum_{n'} x_{i,n'} + V\alpha}$$

where

- $y_i = j$ indicates the class label j for instance i
- $w = n$ indicates word n
- α is the smoothing parameter
- V is the total number of words

For each class, normalize by the sum of counts of words **in that class**. In other words, $\sum_n \phi_{j,n} = 1$ for all j. You can estimate $\log \phi$ directly if you prefer.

For the prior $\log P(y)$, you can use relative frequency estimation.

Both probabilities should be estimated from the training data only. Please write this code yourself -- do not use other libraries, and try to do it without looking at other code online.

First call `gtnlplib.preproc.getCountsAndKeys`, which returns the following objects:

- word counts for every label in the training data.
- the count of instances with every label in the training data.
- a list of all word types that are observed in the training data

You want to do this once, because computing these counts is slow.

In [17]:
```
counts, class_counts,allkeys = gtnlplib.preproc.getCountsAndKey
s(gtnlplib.constants.TRAINKEY)
```

In [18]:
```
class_counts
```

Out[18]:
```
defaultdict(int, {'NEG': 416, 'NEU': 554, 'POS': 590})
```

You will first have to implement `gtnlplib.naivebayes.learnNBWeights`, and then run it to get the weights of the naive bayes classifier.

In [19]:
```
reload(gtnlplib.naivebayes)
weights_nb = gtnlplib.naivebayes.learnNBWeights (counts, class_c
ounts, allkeys, alpha=0.1)
```

Sanity check: the word probabilities for each class should sum to 1, or very close:

In [24]:
```
# sanity check!
print sum([np.exp(weights_nb[('NEU',basefeat)]) for basefeat in
allkeys if basefeat != gtnlplib.constants.OFFSET])
print sum([np.exp(weights_nb[('POS',basefeat)]) for basefeat in
allkeys if basefeat != gtnlplib.constants.OFFSET])
print sum([np.exp(weights_nb[('NEG',basefeat)]) for basefeat in
allkeys if basefeat != gtnlplib.constants.OFFSET])
```

```
1.0
1.0
1.0
```

Deliverable 4a Train your classifier from the training data, and apply it to the development data, with $\alpha = 0.1$. Report the confusion matrix and the accuracy.

In [25]:
```
outfile = 'nb.txt'
mat = gtnlplib.clf_base.evalClassifier(weights_nb,outfile, gtnlp
lib.constants.DEVKEY)
print gtnlplib.scorer.printScoreMessage(mat)
```

```
3 classes in key: set(['NEG', 'NEU', 'POS'])
3 classes in response: set(['NEG', 'NEU', 'POS'])
confusion matrix
key\response:   NEG     NEU     POS
NEG             33      62      16
NEU             24      82      29
POS             18      41      89
----------------
accuracy: 0.5178 = 204/394

None
```

Deliverable 4b Try at least seven different values of α. Plot the accuracy on both the dev and training sets for each value, using subplot (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.subplot) to show two plots in the same cell.The values of α should be chosen such that the max value is not at either endpoint.

In [26]:
```
alphas = [0.0001, 0.001, 0.01, 0.1, 0.5, 1, 1.3, 1.5, 1.8, 2, 5,
10] #your choice!
```

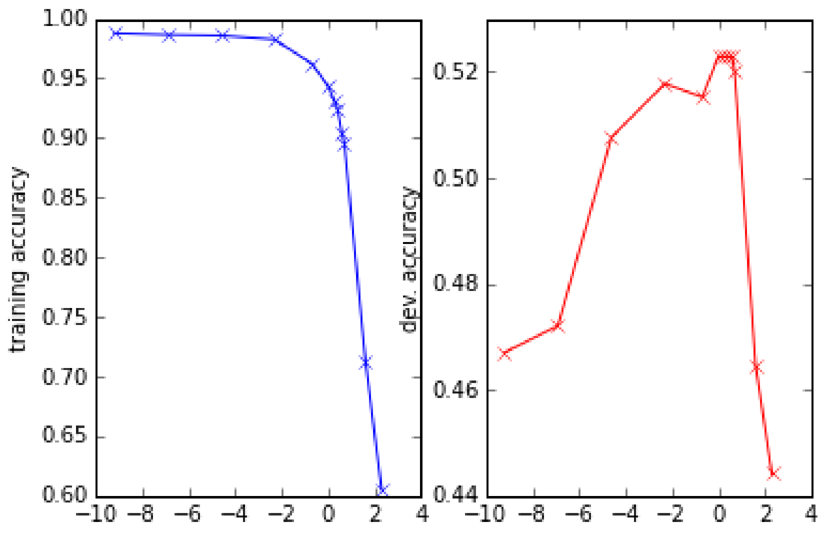
In [27]:
```
weights_nb_alphas, tr_accs, dv_accs = gtnlplib.naivebayes.regula
rization_using_grid_search (alphas,counts, class_counts, allkey
s)
```

In [28]:
```
for i,alpha in enumerate (alphas):
    print alpha, tr_accs[i], dv_accs[i]
```

```
0.0001 0.987820512821 0.467005076142
0.001 0.986538461538 0.472081218274
0.01 0.985897435897 0.507614213198
0.1 0.982692307692 0.517766497462
0.5 0.962179487179 0.515228426396
1 0.94358974359 0.522842639594
1.3 0.93141025641 0.522842639594
1.5 0.924358974359 0.522842639594
1.8 0.904487179487 0.522842639594
2 0.896153846154 0.520304568528
5 0.712820512821 0.464467005076
10 0.605128205128 0.444162436548
```

```
In [29]: # run this code to plot the accuracies
         subplot(1,2,1)
         plot(log(alphas),tr_accs,'bx-')
         ylabel('training accuracy')
         subplot(1,2,2)
         plot(log(alphas),dv_accs,'rx-')
         ylabel('dev. accuracy')

Out[29]: <matplotlib.text.Text at 0x7f5f0579b410>
```



(Use this cell to explain what you see in the plot above)

5. Feature Analysis

(*Completing getTopFeats() - 2 pts, Deliverable 5a - 1pt, 5b - 2 pts . Total 5 pts*)

Deliverable 5a What are the words that are most predictive of positive versus negative text? You can measure this by $\log \theta_{pos,n} - \log \theta_{neg,n}$ (which is similar to the likelihood ratio test (http://en.wikipedia.org/wiki/Likelihood-ratio_test)). Use $\alpha = 0.1$.

List the top five words and their counts for each class. Do the same for the top 5 words that predict negative versus positive.

Note

- You will have to implement `gtnlplib.analysis.getTopFeats`.
- You may need to sort dictionaries for getting the top features. Consider using operator.itemgetter() (http://docs.python.org/2.7/library/operator.html) to easily sort dictionaries by their values.

In [30]:
```
reload(gtnlplib.analysis)
print gtnlplib.analysis.getTopFeats(weights_nb_alphas[1e-1],'PO
S','NEG',allkeys)
print gtnlplib.analysis.getTopFeats(weights_nb_alphas[1e-1],'NE
G','POS',allkeys)
```

```
[('maizon', 5.2447540768676948), ('color', 5.0546097943925314),
('powell', 5.0008333976117267), ('informative', 4.88374073133036
47), ('jane', 4.819616203160825)]
[('dennett', 5.7231165727858819), ('ludemann', 5.339327190156758
7), ('duquette', 5.1464235240322669), ('poorly', 5.1464235240322
669), ('ok', 5.1464235240322669)]
```

Deliverable 5b Now do the same thing for $\alpha = 10$. Which words look better to you? Which gave better accuracy? Explain what you think is going on.

In [31]:
```
print gtnlplib.analysis.getTopFeats(weights_nb_alphas[10],'PO
S','NEG',allkeys)
print gtnlplib.analysis.getTopFeats(weights_nb_alphas[10],'NE
G','POS',allkeys)
```

```
[('war', 1.5395729748970055), ('heart', 1.3836921908319653), ('e
xcellent', 1.3763928883503525), ('wonderful', 1.355001698369035
2), ('favorite', 1.2035500755109414)]
[('waste', 1.4656052852615957), ('evidence', 1.425567912201758
6), ('savage', 1.324358010270247), ('dennett', 1.311454605434338
6), ('boring', 1.2482757038128067)]
```

\alpha = 10 looks better to me, because 1) it is more smooth; 2) the words are more informative and discriminative.

The main reason is: some rare words will typically have much higher ratio, especially when it only appears in one class. So applying larger \alpha will alleviate this problem (e.g., log(2) - log(1) is much larger than log(12) - log(11)).

6. Perceptron

(*6 points total*)

Implement a perceptron classifier. Using the feature-function representation, include features for each word-class pair, and also an **offset** feature for each class. Given a set of word counts $\vec{x}_i$, a true label y_i, and a guessed label $\hat{y}$, your update will be \begin{align*} \hat{y} & \leftarrow \text{argmax}_y \vec{\theta}' f(\vec{x}_i,y)\ \vec{\theta} & \leftarrow \vec{\theta} + f(\vec{x}_i, y_i) - f(\vec{x}_i, \hat{y}). \end{align*}*

Please write this yourself -- do not use any libraries, and try not to look at other code online.

Sanity check If you are not careful, learning can be slow. You may need to think a little about how to do this update efficiently.

- On my laptop, I can make 10 passes on the training data in roughly 30 seconds, including evaluating the accuracy on the dev and training sets.
- You can use the `%%timeit` cell magic to compute statistics like this.
- Your code doesn't have to be as fast as mine, but it needs to be written intelligently, and it needs to be fast enough for you to debug it properly.
- The `%%prun` cell magic is also useful for diagnosing speed

To begin with we will load all the training data: the training set and the development. This will increase the speed. `gtnlplib.preproc.loadInstances` implementation is provided.

```
In [32]: all_tr_insts,all_dev_insts= gtnlplib.preproc.loadInstances(gtnlp
         lib.constants.TRAINKEY, gtnlplib.constants.DEVKEY)
```

Deliverable 6a (5 points)

Implement the function `gtnlplib.perceptron.oneItPerceptron` that runs the perceptron for a single iteration (one pass through the training data). Its signature should be:

- **Input 1**: all training instances
- **Input 2**: a dictionary of weights, representing the current classifier at the time you call this function
- **Input 3**: a list of all possible labels
- **Output 1**: the weights after training
- **Output 2**: the number of training errors
- **Output 3**: the number of training instances

The second and third outputs allow you to compute the *training set accuracy*. This way, you can see whether you are overfitting or underfitting.

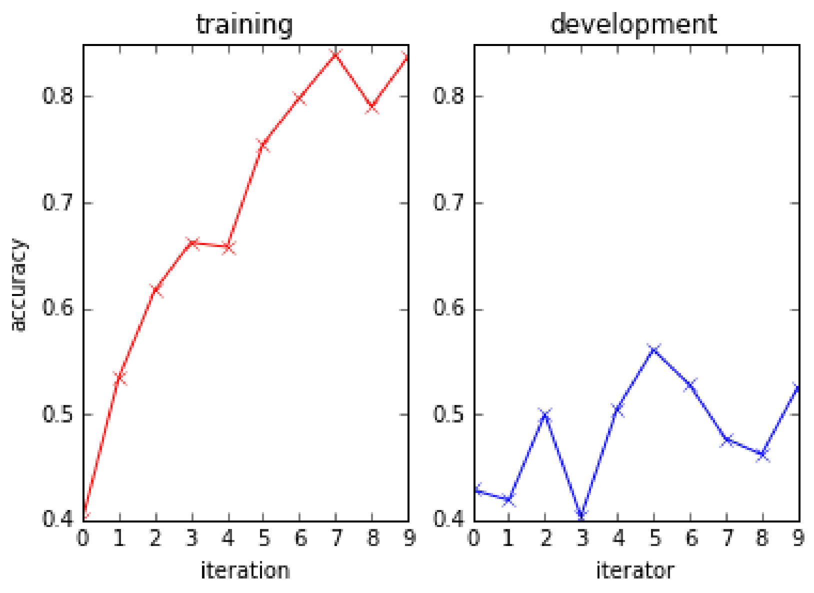
Deliverable 6b (1 point): Train your classifier on trainkey for ten iterations, and plot the output. For this you will have to implement `gtnlplib.perceptron.trainPerceptron` function making use of `gtnlplib.perceptron.oneItPerceptron`

In [33]:
```
reload(gtnlplib.perceptron)
outfile = "perc.txt"
w_perc,tr_acc_perc,dv_acc_perc = gtnlplib.perceptron.trainPercep
tron(10, all_tr_insts,gtnlplib.constants.ALL_LABELS, outfile, gt
nlplib.constants.DEVKEY)
```

```
0 dev:  0.428934010152 train:  0.400641025641
1 dev:  0.418781725888 train:  0.533974358974
2 dev:  0.5 train:  0.616666666667
3 dev:  0.403553299492 train:  0.662179487179
4 dev:  0.505076142132 train:  0.657692307692
5 dev:  0.560913705584 train:  0.755128205128
6 dev:  0.527918781726 train:  0.798076923077
7 dev:  0.477157360406 train:  0.839102564103
8 dev:  0.46192893401 train:  0.790384615385
9 dev:  0.52538071066 train:  0.836538461538
```

In [34]:
```
# this code makes plots of the training and development set accu
racy
def makePlots(tr_acc,dv_acc):
    ax1 = plt.subplot(1,2,1,xlabel='iteration',ylabel='accurac
y')
    plt.plot(tr_acc,'rx-')
    plt.title('training')
    plt.subplot(1,2,2,xlabel='iterator',sharey=ax1)
    plt.plot(dv_acc,'bx-')
    plt.title('development')
```

In [35]:
```
makePlots(tr_acc_perc,dv_acc_perc)
```



Sanity check Your training set accuracy should increase quickly, but your dev set accuracy might be disappointing.

7. Averaged Perceptron

Notice how the dev set performance of the perceptron was very unstable. Now you will try to improve it using averaging.

Conceptually, the idea is to keep a running total of the weights, and then divide at the end, after T updates:

$$\begin{aligned} \hat{y} &\leftarrow \text{argmax}_y \theta' f(\vec{x}_i, y) \\ \theta^t &\leftarrow \theta^{t-1} + f(\vec{x}_i, y_i) - f(\vec{x}_i, \hat{y}) \\ \overline{\theta} &= \frac{1}{T}\theta^T \end{aligned}$$

Then you can use $\overline{\theta}$ to make predictions.

But in practice, this is very inefficient. You can't store the weights after every update -- it's much too big. But you don't want to compute a running sum either. The reason is that the weight vector will quickly become dense, and this would require $O(\#F)$ operations at every update, where $\#F$ is the number of features. This is much more work than the standard perceptron update, which only involves the features that are active in the current instance. In a bag-of-words model, each document will typically have only a small fraction of the total vocabulary, and we would like each update to be linear in the number of features active in the document, not the total number of features.

An efficient solution was pointed out by Daume 2006 (http://hal3.name/docs/daume06thesis.pdf). Let δ_t indicate the update at time t. Then, assuming $\theta^0 = 0$, we have:

$$\begin{aligned} \theta^t &= \theta^{t-1} + \delta_t \\ &= \sum_{t' < t} \delta_{t'} \end{aligned}$$

We would like to compute the sum of the weight vectors, \begin{align} *\sumt^T \theta_t = & \sum_t^T \sum{t' \leq t} \delta_{t'} = T \delta_0 + (T-1) \delta_1 + (T - 2) \delta_2 + \ldots + \delta_T \ = & \sum_t^T (T - t) \delta_t\ = & T \sum_t^T \delta_t - \sum_t^T t \delta_t \ = & T \theta_t - \sum_t^T t \delta_t \ \frac{1}{T} \sum_t^T \theta_t = & \theta_T - \frac{1}{T} \sum_t^T t \delta_t \end{align*}

This means we need to keep another running sum, $\sum_t^T t\delta_t$, the sum of scaled updates. To compute the average, we divide by the number of updates T and subtract it from the current weight vector.

Deliverable 7a (5 points) Implement averaged perceptron, using two functions

- an outer loop, `gtnlplib.avg_perceptron.trainAvgPerceptron`, which should have the same inputs and outputs as `gtnlplib.perceptron.trainPerceptron`.
- an inner loop, `gtnlplib.avg_perceptron.oneItAvgPerceptron`, which makes a single pass through the training data. To do weight averaging, this function may have to take some additional arguments and offer some additional outputs.

In terms of implementation, your function `gtnlplib.avg_perceptron.oneItAvgPerceptron` should be similar to `gtnlplib.perceptron.oneItPerceptron`, but it needs to take additional arguments to keep track of the running sum of weights, and the total number if instances seen. It also needs to output this information.

Deliverable 7b (1 point): Train your classifier on trainkey for ten iterations, and plot the output, using the code in the cells below.

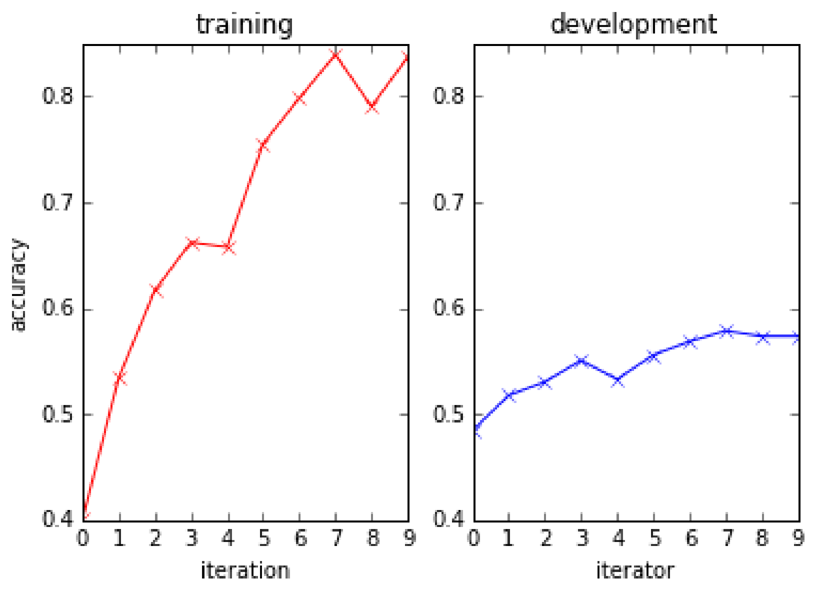
In [36]:
```
reload(gtnlplib.avg_perceptron)
```

Out[36]:
```
<module 'gtnlplib.avg_perceptron' from 'gtnlplib/avg_perceptro
n.pyc'>
```

In [37]:
```
# again, this takes roughly 30 seconds for me
outfile = "ap.txt"
w_ap,tr_acc_ap,dv_acc_ap = gtnlplib.avg_perceptron.trainAvgPerce
ptron(10,all_tr_insts,gtnlplib.constants.ALL_LABELS, outfile,gtn
lplib.constants.DEVKEY)
```

```
0 dev:  0.484771573604 train:  0.400641025641
1 dev:  0.517766497462 train:  0.533974358974
2 dev:  0.530456852792 train:  0.616666666667
3 dev:  0.55076142132 train:  0.662179487179
4 dev:  0.532994923858 train:  0.657692307692
5 dev:  0.555837563452 train:  0.755128205128
6 dev:  0.568527918782 train:  0.798076923077
7 dev:  0.578680203046 train:  0.839102564103
8 dev:  0.573604060914 train:  0.790384615385
9 dev:  0.573604060914 train:  0.836538461538
```

In [38]:
```
makePlots(tr_acc_ap,dv_acc_ap)
```



Sanity check the dev set performance should be much better than the non-averaged perceptron

Deliverable 7c (1 point) Use your getTopFeats function from pset 1a to compute the top ten features for positive and negative classes, by contrasting the weights $\theta_{pos,n} - \theta_{neg,n}$ and $\theta_{neg,n} - \theta_{pos,n}$

In [39]:
```
print gtnlplib.analysis.getTopFeats(w_ap,'POS','NEG',allkeys,
K=10)
print gtnlplib.analysis.getTopFeats(w_ap,'NEG','POS',allkeys,
K=10)
```

```
[('war', 116.04788154605474), ('great', 88.20614063201077), ('be
st', 87.6424588167425), ('both', 75.90346772642779), ('also', 7
2.72642779309018), ('favorite', 71.40843535670791), ('read', 6
6.21876802769053), ('well', 63.03608743029293), ('wonderful', 6
2.1106339337222), ('always', 62.08287930260881)]
[('author', 111.3349785270175), ('no', 105.10326261137106), ('d
o', 96.93237612973527), ('any', 93.12749182744696), ('evidence',
87.46497019421832), ('if', 78.43644638164221), ('believe', 75.75
270815973334), ('worst', 75.36734824690726), ('another', 75.1255
0477533492), ('boring', 71.81776809178899)]
```

8. Logistic regression

Now you will complete an implementation of logistic regression. We've provided a lot of scaffolding code, you just need to fill in some key parts.

Deliverable 8a (3 points): implement `gtnlplib.logreg.computeLabelProbs` to compute the normalized probability of each label.

- This function should have the same input arguments as your predict function
- It should output a dict, from labels to probabilities
- It will need to be fast. You may need to optimize this later. As always, `%%prun` and `%%timeit` are your friends.

sanity check: running the code below should give

- 'NEG': 0.0068674111043921151,
- 'NEU': 0.37494794181688146,
- 'POS': 0.61818464707872656

```
In [41]: reload(gtnlplib.logreg)
         weights = defaultdict(float)
         weights[('NEG','bad')] = 1
         weights[('NEG','best')] = -1
         weights[('POS','bad')] = -0.5
         weights[('POS','best')] = 2
         weights[('NEU',gtnlplib.constants.OFFSET)] = 3
         gtnlplib.logreg.computeLabelProbs({'bad':1,'best':2,gtnlplib.con
         stants.OFFSET:1},weights,gtnlplib.constants.ALL_LABELS)
```

```
Out[41]: defaultdict(float,
                     {'NEG': 0.0068674111043921151,
                      'NEU': 0.37494794181688146,
                      'POS': 0.61818464707872656})
```

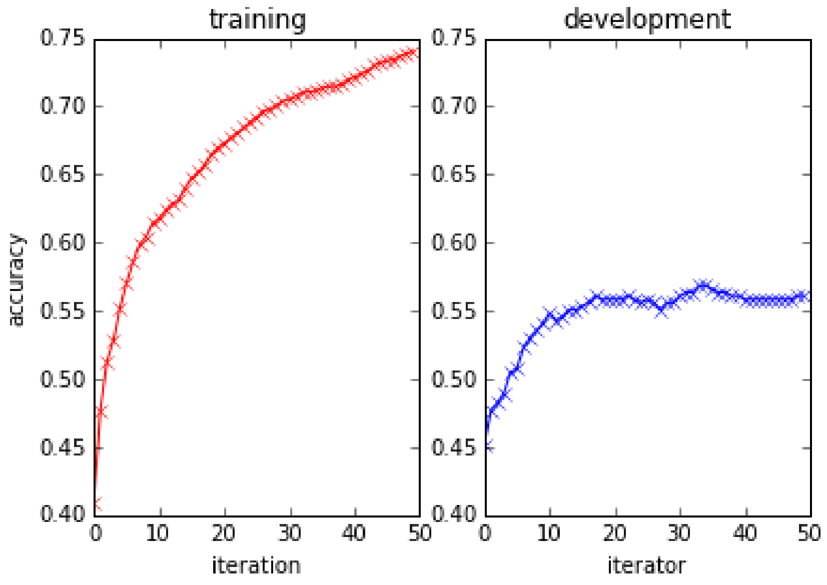
Deliverable 8b (3 points) Now complete the implementation of logistic regression, training by stochastic gradient descent.

- An outline of the code is provided in `gtnlplib.logreg.trainLRbySGD`, including the regularization
- You need to provide the code that computes the update for each instance
- My implementation takes around 3 seconds per iteration over the training set
- Unlike the perceptron code, you can do everything within the single function `gtnlplib.logreg.trainLRbySGD`
- For a reminder about how SGD works, see my notes

In [42]:
```
reload(gtnlplib.logreg)
outfile = "sgd.txt"
w_sgd,tr_acc_sgd,dv_acc_sgd = gtnlplib.logreg.trainLRbySGD(50,al
l_tr_insts, outfile, gtnlplib.constants.DEVKEY, regularizer=1e-
1)
```

```
0 dev: 0.451776649746 train: 0.409236690186
1 dev: 0.477157360406 train: 0.477228992944
2 dev: 0.482233502538 train: 0.513149454779
3 dev: 0.489847715736 train: 0.527902501604
4 dev: 0.505076142132 train: 0.551635663887
5 dev: 0.507614213198 train: 0.570237331623
6 dev: 0.522842639594 train: 0.585631815266
7 dev: 0.530456852792 train: 0.598460551636
8 dev: 0.535532994924 train: 0.604233483002
9 dev: 0.540609137056 train: 0.613855035279
10 dev: 0.548223350254 train: 0.61770365619
11 dev: 0.543147208122 train: 0.623476587556
12 dev: 0.545685279188 train: 0.628608082104
13 dev: 0.55076142132 train: 0.631173829378
14 dev: 0.55076142132 train: 0.640153944836
15 dev: 0.553299492386 train: 0.64720974984
16 dev: 0.555837563452 train: 0.651699807569
17 dev: 0.560913705584 train: 0.656831302117
18 dev: 0.558375634518 train: 0.664528543938
19 dev: 0.558375634518 train: 0.669660038486
20 dev: 0.558375634518 train: 0.672867222579
21 dev: 0.558375634518 train: 0.676715843489
22 dev: 0.560913705584 train: 0.6805644644
23 dev: 0.558375634518 train: 0.684413085311
24 dev: 0.555837563452 train: 0.688261706222
25 dev: 0.558375634518 train: 0.692110327133
26 dev: 0.555837563452 train: 0.696600384862
27 dev: 0.55076142132 train: 0.697883258499
28 dev: 0.555837563452 train: 0.700449005773
29 dev: 0.555837563452 train: 0.703656189865
30 dev: 0.560913705584 train: 0.704939063502
31 dev: 0.56345177665 train: 0.707504810776
32 dev: 0.56345177665 train: 0.71007055805
33 dev: 0.568527918782 train: 0.710711994869
34 dev: 0.568527918782 train: 0.711353431687
35 dev: 0.565989847716 train: 0.713277742142
36 dev: 0.56345177665 train: 0.714560615779
37 dev: 0.56345177665 train: 0.714560615779
38 dev: 0.560913705584 train: 0.715843489416
39 dev: 0.560913705584 train: 0.719692110327
40 dev: 0.558375634518 train: 0.721616420783
41 dev: 0.558375634518 train: 0.723540731238
42 dev: 0.558375634518 train: 0.725465041693
43 dev: 0.558375634518 train: 0.729955099423
44 dev: 0.558375634518 train: 0.731879409878
45 dev: 0.558375634518 train: 0.733162283515
46 dev: 0.558375634518 train: 0.734445157152
47 dev: 0.558375634518 train: 0.736369467607
48 dev: 0.560913705584 train: 0.738935214881
49 dev: 0.560913705584 train: 0.740218088518
```

In [43]:
```
makePlots(tr_acc_sgd,dv_acc_sgd)
```



9. Making it better

There are two general paths for improving these classifiers: data and algorithms.

- Data-oriented approaches relate to the features. For example, you could try to use bigrams, remove stopwords, lemmatize (using wordnet), etc.
- Algorithm-oriented approaches relate to the learning itself. For example, you could implement Passive-Aggressive, AdaGrad (described in my notes), feature hashing (see this paper (http://alex.smola.org/papers/2009/Weinbergeretal09.pdf)), alternative regularizers, or various improvements to naive bayes (see this paper (http://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf)). Note that not all these approaches will improve accuracy; some will improve speed.
- Students in 4650 should try one improvement of either type. Students in 7650 should try one improvement of each type, and for at least one of the improvements, they should cite a specific research paper that motivated their choice. The paper should be from ACL, NAACL, EMNLP, ICML, NIPS, AAAI, or a similar journal.

Deliverable 9 (3 points for 4650; 6 points for 7650): Clearly explain what you did, and why you thought it would work. Do an experiment to test whether it works. Creativity and thoughtfulness counts more than raw performance here.

Deliverable 9

Data-oriented

Data augmentation

I found that, all the algorithms typically get overfitting, and the regularization term is not so useful. So another way to handle the overfitting issue is to augment the dataset. Since I only have 1500 documents for training, I need to do some trick on it. The data augmentation is widely used in Computer Vision and other domains. Here is a related paper for the motivation: *Paulin, Mattis, et al. "Transformation pursuit for image classification." Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on. IEEE, 2014.*

- sentence level training I decompose each document into sentences. The idea is that, if a document is pos/neg/neu, so are the sentences in that document. It gives me around 11000 training instances.
- random crop the document (review) If I randomly cut out some words from a review, it is still simple for human to judge whether it is pos/neg/neu. So based on this, I randomly cropped each document many times, and constructed a large dataset.
- randomc concatenate the sentences If I concatenate two sentences from different reviews with same label (pos/neg/neu), the resulting document should still have same label. So based on this, I can also perform data augmentation.

But unfortunately, all of these methods doesn't contribute to the dev-accuracy. Here are some reasons: 1) The model is linear, and decompose the features in a linear way might not be helpful. 2) I found the training accuracy increases very fast,

So I didn't use that in my submission.

stopwords

I removed the stopwords from the corpus. The stopwords are comming from the NLTK. It only contains tens of words, which means that it is not so aggressive.

The result is not too much different from the origin. I think since the stopwords doesn't carry out some information, the training algorithms might already ignored them.

sentiment words

Besides the bag-of-words feature, I also created another two dimensions, which counting the number of positive/negative words in current review.

It improves my SVM classifier for 1% accuracy on the dev-data. So it is useful. I adopted that.

I tried two different representations, both works similarly.

- 3 x V representation: the feature is like [pos_words, neg_words, neu_words], still a bag-of-words representation.
- V + 2 representation: add two additional dimensions, as described above.

word2vec representation

Besides the bow feature, I also tried the distributed vector representations for words. Since the corpus is really small, I used the pre-trained word representations from here: https://code.google.com/p/word2vec/ (https://code.google.com/p/word2vec/) The relavent paper is from NIPS: *Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.*

The feature is combinded with the CNN model. See the algorithm-oriented section for more information.

Algoritm-oriented

classifier ensemble

This gave me the best performance on the dev-dataset. The idea is to avarage the classifier weights, to make a more robust classifier.

I simply used the classifiers trained on this problem set, they are:

```
    * WCL classifier, which counts the positive/negative words;
    * naive bayes
    * avg perceptron
    * logistic regression
```

Although the avg perceptron and logistic regression performs well, the weights assigned to them are small. The best weight I tuned on the dev dataset is:

[(weights_wlc, 7.0), (weights_nb, 3.0), (w_ap, 0.4), (w_sgd, 1)]

I should probably use more advanced techniques to automatically learn the weights. But currently I only tuned it via grid search. It might have the posibility of overfitting the dev-data.

CNN sentence

Here I tried the algorithm from EMNLP 2014, *Kim, Yoon. "Convolutional neural networks for sentence classification." arXiv preprint arXiv:1408.5882 (2014).* I modified the code from https://github.com/yoonkim/CNN_sentence (https://github.com/yoonkim/CNN_sentence)

The idea is to concatenate the vector representation of words to form an 'image', and then do convolution like usual. This model can take advantages of both word2vec, convolutional feature and the time sequence information (since we maintain the order of the words).

I decomposed the reviews into sentences, and train the model to classify the sentences. The performance is like: training 99% accuracy, dev 58% accuracy.

The main reason is that, these deep-learning approaches require large amount of the data. It will easilty get overfitting on the small dataset. So finally I gave up this idea.

Support Vector Machine

I tried the libsvm. It produces quite good results (around 62.00% on the dev dataset, which approaches the ensemble results). The max-margin intuition makes it generalize well on the dev-dataset. So I mainly use this model for evaluating my features.

I also tried kernels, including RBF kernel, polynomial kernel, etc. RBF kernel makes no difference, while polynomial kernel hurts the performance. The reason is that, since the feature is already in a very high dimension, which makes it linear separatable already (and we can already get ~100% training accuracy). There seems no need to use the kernel methods.

Since it is quite robust, I also made a submission for it.

xgboost

Since xgboost (https://github.com/dmlc/xgboost (https://github.com/dmlc/xgboost)) is one of the most popular model in kaggle competitions, I algo tried that. The parameters I tried are:

- gb linear model, with different L1/L2 regularization term
- gb tree model, with different depth/L1/L2

I should use shallow trees and large l1/l2 coefficients, otherwise it overfits the training data severely. The best score I can get on dev-data is 58%, which is not so good.

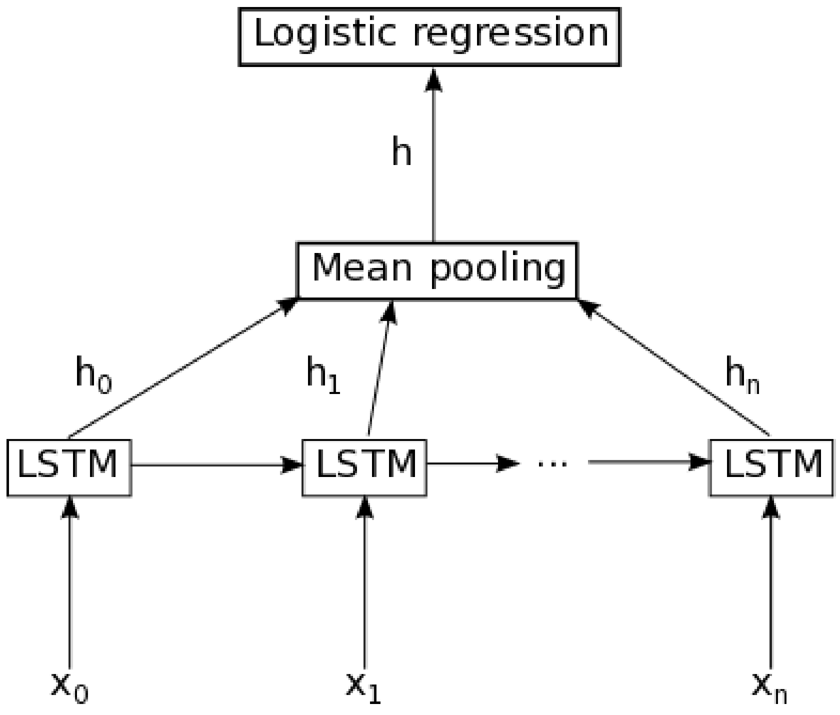
LSTM

The long short-term memory is quite popular nowadays. Probably I should cite the earlist one here *Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.*.

Here I also tried to use this model. The reasons are:

- BOW feature doesn't have the order information for words.
- LSTM is said to be better than RNN. It can take long term memory into consideration.

I modified the code from here: http://deeplearning.net/tutorial/lstm.html (http://deeplearning.net/tutorial/lstm.html), and the model averages the hidden states of each words, and do classification on top of that.



And in my experiment, it failed to fit the training data. Probably there are some bugs on my code. But I guess it also requires large amount of training data to get a good performance, which might not suitable for our case.

You can find all my codes under the "better-ps2" directory.

10. Bakeoff!

48 hours before the assignment is due, I will send you unlabeled test data. Your job is to produce a response file, and submit it to our Kaggle bakeoff (link here (https://inclass.kaggle.com/c/gt-book-review-sentiment-analysis)). The Kaggle contest compares your classifier's results on the dev data to generate a class-visible leaderboard, and compares your classifier's results on the unlabeled test data for the bakeoff. You can use the dev data results as a sanity check, to make sure you submit the correct file.

I'll present the results in class and give the best scorers a chance to explain what they did.

Deliverable 10 (3 points) Run your best system from any part of the assignment on the test data using the `generateKaggleSubmission()` function. Submit your response file to the class Kaggle bakeoff (https://inclass.kaggle.com/c/gt-book-review-sentiment-analysis). Also submit your Kaggle response file to T-Square as 'lastname-firstname.response'. The top scores will be announced in class.

In [ ]:
```
yourBestWeights = weights_mcc # Change this to your best model
gtnlplib.clf_base.generateKaggleSubmission(yourBestWeights, 'Da
i-Hanjun.response')
```