CS 585 Problem Set 4: Perceptron and Coreference

Brendan O'Connor and David Belanger

November 16, 2014

Submission Instructions

Your submission will be a mix of a text writeup and code. Unlike last problem set, we you will be submitting this one on moodle. Upload a file with your text (.doc or .pdf) and also submit the python files coref.py and perc.py. It is due at 2:15pm on Tuesday Nov 25.

In your text writeup, include your name and collaborators list at the top. (See course policy here.)

Overview

This problem set has two parts.

First, you will implement a binary classification perceptron, and also the averaging form of training. This will be done for sentiment analysis (with the same documents as you used for Naive Bayes in PS2); the averaged perceptron will be implemented with the averaging strategy you analyzed in PS3.

Second, you will implement and analyze a coreference resolution system. You'll make a simple rule-based system, then second, make a system based on the perceptron classifier you implemented in Part 1. You can still do this section if you don't complete averaging; and some parts of the coref section don't require any perceptron at all.

There are 100 total points; 25 for Part 1 and the rest for the coref part.

1 Part 1: Perceptron Classifier

Get the starter code from `http://people.cs.umass.edu/~brenocon/inlp2014/ps/ps4.zip`.

Get reviews.zip from `http://people.cs.umass.edu/~brenocon/inlp2014/ps/reviews.zip` and unzip it in the same directory as the code where you're working. In this part you will only have to edit the starter code of *perc.py*. The most important function, *trainPerceptron*, will be used later for your coref system. But in this section, you will also call it from sentimentTest within perc.py, which gets executed if you run it from the command line (or run it in an interactive interpreter if you like).

trainPerceptron is currently incomplete. It loops over all the examples for as many iteration passes as specified, but it doesn't do any updates. There's no averaging either.

We have set up sentimentTest so that it loads the data, and then calls trainPerceptron on it. It also contains an evaluation function, which evaluates the accuracy of the current model on the test set. (If you like cool programming stuff, you'll see that it uses a Python closure passed as a callback to do this.) It's set up so that at every iteration of the perceptron trainer, it prints out the test set accuracy of the current model.

Note that labels are boolean objects, and feature vectors are represented as Python dictionaries, mapping from {name: value}.

Because this is a binary classification perceptron, we don't use the $f(x, y)$ mathematical notation, but instead just say a feature vector is $f(x)$, where f is the feature extraction function. We've pre-extracted feature vectors for you. They are just word counts from movie reviews. The labels y are Python boolean variables, and True if it's a positive review, otherwise False if it's negative. Since this is a binary classification perceptron, the prediction rule is: predict y=True if $\theta^{\mathsf{T}} f(x) > 0$, otherwise predict y=False.

Question 1.1. *[2 points]* Read evalClassifier. Explain in English what the line that updates num_correct is doing.

Question 1.2. *[2 points]* Run sentimentTest out of the box without changing anything. What accuracy does it get? Why it it so bad? (Hint: what is the model predicting?)

Question 1.3. *[8 points]* Implement the basic perceptron algorithm, inside of trainPerceptron. Use the stepSize parameter that's passed in to control the size of the update. This was called η in PS3. It's sometimes also called a "learning rate".

Run trainPerceptron with the parameters given in the starter code. It should produce the following result. Copy and paste the output you get. **[Updated 11/16:** updated the output to fix a small bug in our solution.]

```
Loading data
Num pos vs negative in test set: 407 pos, 401 neg, 808 total
Training perceptron
Training on 0'th pass through the training set
Classifier accuracy = 624/808 = 0.772277227723
Training on 1'th pass through the training set
Classifier accuracy = 659/808 = 0.815594059406
Training on 2'th pass through the training set
Classifier accuracy = 671/808 = 0.830445544554
Training on 3'th pass through the training set
Classifier accuracy = 673/808 = 0.832920792079
Training on 4'th pass through the training set
Classifier accuracy = 674/808 = 0.834158415842
Learned weights for 17987 features from 1596 examples
Accuracy of final model:
Classifier accuracy = 674/808 = 0.834158415842
```

```
17987 total features, showing highest and lowest weights
Top weights:
[(u'great', 24), (u'love', 18), (u'excellent', 18),
(u'favorite', 16), (u'best', 15), (u'loved', 15),
(u'amazing', 15), (u'also', 15), (u'you', 14), (u'subtle', 13)]
Bottom weights:
[(u'bad', -27), (u'worst', -25), (u'boring', -21),
(u'even', -20), (u'awful', -20), (u'wonder', -19),
(u'poor', -17), (u'?', -16), (u'nothing', -15), (u'looks', -15)]
```

Question 1.4. *[2 points]* Why are the weight values all integers?

Question 1.5. *[4 points]* Now play around with different settings of the numPasses parameter. How high do you have to make it in order for test set accuracy to converge? What does it converge to?

Question 1.6. *[10 points]* Implement the averaged perceptron. When useAveraging=True is passed in to trainPerceptron, it must return the averaged weight vector, instead of the last weight vector.

You can follow the pseudocode from PS3. This will require new code in several places in trainPerceptron: you have to initialize the cumulative sum vector and the t counter, and you have to update them at the perceptron updates, and use them atthe end to calculate the averaged weights.

Do not start saving weight values until after the first iteration. This is because the weights aren't very good at the very beginning.

Feel free to use a new function for this, but please maintain the API of trainPerceptron as we asked.

Recall that when training an averaged perceptron, you still use the current weight vector to make predictions during learning; the averaged version is never used during learning.

Averaging tends to give more stable answers. It is difficult to see in this sentiment example, because the test set is so small. But it will be very effective in Part 2.

Please copy and paste the output from one training run. The weight vectors should no longer be integers.

Question 1.7. EXTRA CREDIT: [Up to 4 points]

Turn on shuffling, which randomizes the order that the examples are iterated over. How much variability do you see with regards to solutions? Run some experiments to assess whether averaging improves stability and results, and report your findings.

You can do this a little bit with the sentiment dataset, but the effects are much clearer with the coreference problem in Part 2.

1.1 Features

Suppose that we have an input d (such as a document) and we want to represent it with a feature vector f (stored in a sparse manner as a dictionary from string to number). A standard technique in NLP is to implement feature extraction via a set of *feature templates*. Each feature template can be thought of as a pattern that instantiates many different features.

For example, suppose we want to train a classifier that takes a document d and classifies whether it is about movies or travel. We will extract features using the following code:

```
cities = set(['Paris','London','Rome','Berlin','Madrid'])
people = set(['Michael','John','Ellen','Rachel'])

def getFeatures(document):
    features = {}

    for word in document['tokens']:
        key = "CountForWord-" + word
        features[key] = features.get(key,0) + 1

    for word in document['tokens']:
        key = "CountForLowercasedWord-" + word.lower()
        features[key] = features.get(key,0) + 1

    for word in document['tokens']:
        if word in cities:
            key = "CountOfCityWords"
            features[key] = features.get(key,0) + 1

    for word in document['tokens']:
        if word in people:
            key = "CountOfPersonWords"
            features[key] = features.get(key,0) + 1

    return features
```

The above code can be thought of as four feature templates. The first adds a feature for every word in the document. The second adds a feature for the lowercased form of every word in the document. They are called "templates" because a single template produces many different features. The last two templates are simpler: they count the number of city words and person words mentioned in the document (in practice you would use much longer wordlists of cities and people).

Question 1.8. *[3 points]* Suppose that each document d has m tokens. (A) At most, how many nonzero elements are there in the feature vector output by getFeatures(d)? (B) Suppose that the training set has vocabulary size V (counting different cased words as separate words) and that we write the feature vector as an actual vector, rather than a dictionary that only includes nonzero entries. How long would the vector be?

2 Left-to-Right Pairwise Coreference

Here you will implement an antecedent selection system, similar to what was described in lecture. The algorithm loops through mentions in the document. For the nth mention, it makes an antecedent decision. There are n possible outcomes (n-way classification): either one of the $n-1$ previous mentions, or else it decides to not attach to anything mentioned previously.

Note that there is potentially more than one correct answer — a correct answer is, any of the previous mentions within the same entity cluster. If the mention is the first mention in the entity cluster, the correct answer is the null decision of not having an antecedent.

We'll first do a rule-based version and then a machine learning version.

3 Evaluating Coref Performance

There are a number of ways to evaluate the performance of a coreference system. The provided code computes 'pairwise' precision and recall, which doesn't look just at antecedent decisions, but instead looks at all pairs of mentions in the same cluster. You'll see there's only 1 gold-standard link in small.json (the he-him link). There's 55,753 gold links in the test set.

At times, we will want to compare the precision v.s. recall of a given approach. At other times, however, we'll want a single number to summarize performance. The code computes the 'F1' measure (`http://en.wikipedia.org/wiki/F1_score`) and prints this alongside precision and recall. Use this when assessing the overall impact of adding new features, etc.

4 Code

We have provided a substantial amount of helper code in both coref.py and corefutil.py. Everything you will submit will be in modifications to coref.py. Do not modify corefutil.py.

At the bottom of corefutil.py, we provide helper functions that may be useful when designing features and when debugging. To understand the document and mention datastructures, see how they are created in convertJsonDocIntoMyDoc, or see how they are used in coref.py. We also provided a script, mentiontest.py, which does not do any coreference, but instead just loops through the data structures and prints them out. This might help make it clear what these data structures look like. You can modify it to help debug your mention analysis functions. Do not submit your mentiontest.py.

coref.py is split into 4 parts. The first 3 are for code that you will complete in Sections 5, 6, and 7 below. The final part of coref.py provides a main function and driver code for running different kinds of coreference. Each of these will train a model (if necessary), perform coreference on the test set, and print an F1 score. We provide a commandline interface. Here are some examples of how to use it:

```
# Test the rule-based coref system on a small example file (for debugging)
python coref.py rule --test-file small.json

# Test the rule-based coref system
python coref.py rule --test-file corefdata/test.jsons

# Train and test the ml-based coref system
python coref.py ml --train-file corefdata/train.jsons --test-file corefdata/test.jsons
```

Finally, note that the main function defines a 'verbose' flag at the top level. Toggle this to print out lots of output. This is useful for debugging, error analysis, and feature engineering. You should always test things first on a very small example like small.json. Look at the file to see what's in it: it's just two sentences, and there's only one non-singleton entity. small.html contains an HTML version of this document, produced by view.py.

5 Mentions

We are using a dataset of coref-annotated documents from the CoNLL-2012 competition (`http://conll.cemantix.org/2012/`). This data defines mentions as phrases (token spans), and for each mention has an entity ID. It also contains POS tags, NER tags, and parses. We'll only use the POS tags to keep things simple.

Download the full dataset from the zip file posted on the Piazza Resources page at `https://piazza.com/umass/fall2014/cmpsci585/resources`. (Part of it comes from a copyrighted dataset that I'm guessing we are not allowed to post publicly.) train.json contains training data, one document per line. test.json contains test data.

I ran the first 5 documents through view.py to create an HTML version viewable here: `http://people.cs.umass.edu/~brenocon/inlp2014/ps/ps4/test_first_5.html`

5.1 Looking at the data

Question 5.1. *[4 points]* A good thing to do with a new dataset is read it a little bit.

In test_first_5.html, go to the document wsj_1504 and read the first several sentences of the Bob Stone story (and learn how you improve your own corporation's governance procedures). Explain in English what the distinction you think the annotators were making when they said e12 and e22 are different entities. (e22 first appears in sentence S2.) Do you agree or disagree, and why?

5.2 Implementation

The following code should be completed in coref.py.

Question 5.2. *[8 points]* Implement the isPronoun(m), isProper(m), and isPlural(m) mention attribute functions. A good way to do this is to use POS tags that came from a POS tagger. (Actually the tags in this data might be gold-standard, so that's perhaps overly optimistic.) Implementing these will require using the headTokenPOSTag() function in corefutil.py.

Note that the POS tags are in the Penn Treebank format. Find the PTB tagset documentation online (same as for when you did the POS exercise several weeks ago). You'll see which tags correspond to pronouns, proper nouns, and plural nouns. You might also have to hard-code some very small pronoun wordlists for plural pronouns (because the PTB tagset doesn't distinguish grammatical number for pronouns).

Please write the implementations of these functions within coref.py where they're specified. However, for debugging, we suggest you call them from mentiontest.py (see the comments in there). You can run it on the sample file with:

```
python mentiontest.py small.json
```

6 Rule-Based Coreference

In this section, you will implement and test a rule-based system. The algorithm is as follows:

- Assume a window size K, which means you will look at the last K mentions as antecedent candidates. For example, $K = 5$ is the default.
- For each of these candidate mentions, use a filter to accept or reject them.
- Of the accepted candidates, choose the closest one as the antecedent. If none were accepted, choose a null antecedent.

We've implemented this in doRuleCoref. It calls isAcceptableAntecedent for the accept/reject filter. You only need to implement isAcceptableAntecedent.

6.1 Implementation

The version of isAcceptableAntecedent that we provide always returns False. Therefore, it refuses to ever link a mention to any candidate. Run it on the test data (or the smaller example files) and confirm that it should predict 0 links, yielding 100% precision but 0 recall. You should get the following results out of the box.

```
% python coref.py rule --test-file small.json
Pairwise Prec = 1.000 (0/0), Rec = 0.000 (0/1), F1 = 0.000

% python coref.py rule --test-file corefdata/test.jsons
Pairwise Prec = 1.000 (0/0), Rec = 0.000 (0/55753), F1 = 0.000
```

Question 6.1. *[2 points]* doRuleCoref has a verbose mode. Turn it on and run on small.json or bobstone.json to see what it does.

Verbose mode is not fully implemented yet. Please implement it so that for every antecedent candidate, it prints diagnostic information about whether the system thinks the proposed link is acceptable or not. Fill in the skeleton code for this. So for "[Harry Potter] was a [wizard]. [Voldemort] said hi to [him]", when processing [him], it might print something like the following. We suggest using indentation like this to make it easier to read.

```
...
Processing mention: [him]
  Candidate = [Harry Potter]
    gold link = YES,  is acceptable? NO
  Candidate = [wizard]
    gold link = YES,  is acceptable? NO
  Candidate = [Voldemort]
    gold link = NO,  is acceptable? YES
```

Question 6.2. *[1 points]* In the above example, will the rule system correctly choose an antecedent for "[him]"?

Now play around with some different ways to implement isAcceptableAntecedent. For example, you could choose to: only resolve pronoun mentions; or only resolve pronouns to a candidate that is a non-pronoun; etc. For at least one of your versions, enforce some sort of grammatical agrement constraint: for example, only link mentions if their number (isPlural) attribute agrees.

Question 6.3. *[8 points]* Describe 3 different choices of rules you used for isAcceptableAntecedent and provide the precision and recall for each, when you run on the entire test set (corefdata/test.json). In coref.py, provide your best performing implementation. (Either choose the one with the highest F-score, or if you think that's not a good way to evaluate, explain your reasoning why the one you picked is best.)

Question 6.4. *[2 points]* The version of rule-based coreference that we used is quite restrictive. It always takes the first mention that satisfies $isAcceptableAntecedent(M_{\text{cur}}, M_{\text{cand}})$. Describe in words a different rule-based approach that you think would perform better.

7 Machine Learning-Based Coreference

We will use an ML approach similar to the one in class, though with a slight tweak. At prediction time, each antecedent candidate, a link from M_{cur} to M_{cand} as its antecedent, is scored with a linear scoring function. So for "[Harry Potter] was a [wizard]. [Voldemort] said hi to [him]", when resolving "[him]", there are 3 candidates. There is a scoring threshold t. If the scores of all candidates are below t, then the resolution is null. For example, in this case, "[him]" would be considered to be starting a new entity. Otherwise, the antecedent is chosen to be the highest scoring candidate.

(On Wednesday lecture I presented it slightly differently, with "null" being a candidate by itself. This might be a better way to do it actually, but it makes feature extraction slightly more complicated.)

If you run coref.py in 'ml' mode, it extracts features and calls the trainPerceptron function in perc.py, then runs new coreference predictions on the test set as explained above.

Question 7.1. *[1 points]* Changing the threshold t affects the precision-recall tradeoff. Explain how, if you were to increase or decrease t. You can answer without running code, and instead just thinking about how changing t will affect the outputs.

At training time, we take gold standard data and turn it into binary classification examples of each link. In the above example, if we had correct gold standard data, [him] that will generate 2 positive examples and 1 negative example, corresponding to the 3 candidate antecendents to its left.

7.1 Features

Every potential coreference link between M_{cur} and M_{cand} is associated with a feature vector, which we denote by the output of the function $g = f(M_{\text{cur}}, M_{\text{cand}})$. The scoring formula (for the linky-ness of the pair) for candidate M_{cand} is: $\theta^{\mathsf{T}} f(M_{\text{cur}}, M_{\text{cand}})$

We suggest developing features by concatenating various string representations for various mentions attributes. For example, for each mention, we can define a string for the output of isPronoun() on the mention, then we could define a feature for the pair by concatenating them. Say m1 is the current mention, and m2 is the candidate. We'd do:

```
key ="Pronoun:%s-%s" % (isPronoun(m1),isPronoun(m2))
features[key] = 1
```

This is one template, and it would yield 4 different features: "Pronoun:False-False" (meaning that neither the current nor candidate mentions are pronouns), "Pronoun:True-False" (meaning the current mention is pronoun, but the candidate is not), etc. For a given example, only one of these will have a nonzero value.

Note that it is very important to associate each feature template with a unique string identifier. Above, we use the prefix 'Pronoun:' Imagine we also used the template

```
key ="Plural:%s-%s" % (isPlural(m1),isPlural(m2))
features[key] = 1
```

If we didn't include the "Pronoun:" and "Plural:" strings, then these templates could return the same string representation, and thus overwrite each other.

We are providing only two features: the bias feature, and a binary feature whether the headwords match. This feature by itself isn't strong enough to do much. We suggest building features using the following concepts.

1. Mention attributes, like the plural/number/pronoun attributes you implemented already. It is also useful to make versions that conjoin them against the text of the head token for the other mention (using the headToken() function).

2. The string for the head token (using the headtoken() function in corefutil).

3. The POS tag for the head token (using the headTokenPOSTag() function).

4. Distance features: are the mentions in the same sentence? How many tokens are between them? (Use the sentenceIndexInDocument() and mentionPositionInDocument() functions).

You can use features from either the current or candidate mention, and you can conjoin features from both as well. Conjoining is important because it gets at agreement and other relationships between the mentions. Note that you can do both conjunction and individual versions. For example, if the candidate POS is NNP and the current mention POS is PRP, you can make three different features with value 1, like "CandidatePosIsNNP", "CurrentPosIsPRP", and "CandidatePosIsNNP_and_CurrentPosIsPRP" (and the names don't have to be cute, just have to be distinguishable). So that would be three different feature templates right there. There are tons of feature templates like this in coref and also dependency parsing papers.

You can also of course do additional conjunctions between different types of features. For example, the "head words are the same" feature that we gave you doesn't do much by itself. But if you're smart about how to conjoin it with other information, it's very powerful. (If you just print all the head word pairs, this might make apparent what the trick is — under what conditions does a string match imply coreference?)

7.2 Implementation

The following code should be completed in coref.py.

Question 7.2. *[4 points]* When using getTrainingExamples in verbose mode, it should print out some information about each training example, as described in comments in the code. Complete this. This will be helpful for developing and debugging features.

Question 7.3. *[12 points]* Implement getFeaturesForMentionPair(m1, m2). You must include at least 5 new feature templates, drawn from the above. (You don't have to implement from all four bullet points, since each one describes potentially several different templates you could make from it.)

Question 7.4. *[1 points]* In your text writeup, paste two feature vectors from the previous question. One for a positive-labeled training example and one from a negative one.

Question 7.5. *[2 points]* Describe in words what the 5 feature templates are that you implemented.

7.3 Engineering New Features

Question 7.6. *[8 points]* In addition to the 5 templates above, develop 2 new feature templates that you think should improve performance. They could be drawn from the above, or the suggested addition below, or anything else you want. For each, do the following:

- Create two examples, each consisting of one or two sentences, where coreference with the base set of features might make the wrong decision. You can make up the examples. They do not have to be from the training corpus. You do not need to run the code on these examples. Put them in your text writeup.
- Write a brief description of the template and why you think it will fix the particular errors you identified in the previous step.

Suggested addition: Gender attributes for pronouns (this can help a lot). PTB POS tags don't have this information; you have to write small word lists yourself. If you only have it for pronouns, that's good but it only helps with pronoun-pronoun matches. For names, you can find first name gender lists online from the U.S. Census or other sources. Common nouns are a lot harder/complex so don't worry about them. (if you want to, you can try this list: `http://www.clsp.jhu.edu/~sbergsma/Gender/`)

Question 7.7. *[5 points]* An important type of test (related to an "ablation test"), is to assess the effect of individual feature templates. Choose a base set of feature templates to use and compute the test set accuracy for using just these features. Then report the results from three other experiments: base plus one new template, and base plus two new templates, and then the whole new system (base + two new templates). (The easy way to do this code-wise is to toggle them with commenting/uncommenting. If you want to be fancy, you can add commandline flags for them.)

You will receive full credit even if the templates didn't actually improve F1. However, they need to be well-motivated by examples in the previous question. Provide a table with the following format:

Templates Used	Precision	Recall	F1
Base			
Base + template 1			
Base + template 2			
Base + both templates			

7.4 Trading Off Precision and Recall

Question 7.8. *[2 points]* Explain in words what precision and recall are.

Question 7.9. *[2 points]* Explain a context where you might prefer high-precision coreference decisions. When would you prefer high-recall coreference?

(Hint: think about different NLP applications like web search, machine translation, dialogue understanding, etc.)

Question 7.10. *[7 points]* In the top level main function, we have:

```
testSettings['thresh'] = 0
```

As discussed in Question 7.1, changing the threshold will impact the precision-recall tradeoff. Vary the threshold with a for loop and record the (precision,recall) values for each threshold. Plot a precision vs recall for at least 6 different values of the threshold. (Just use the points by themselves; there's no need to make an actual curve.) Include the graph in your text writeup. Say which setting of features you used for this experiment.