**NP Chunking (programming)**

Construct an NP chunker for English web text. By the time your chunker sees the data, you can expect it to be tokenized and POS tagged. Your system should then infer the NP chunk brackets. Walk through the process of creating it and describe the decisions you make along the way. What assumptions are you making about how it's going to be used? Characterize its performance and describe what you would do and what resources you would need to improve it.

In order to build NP chunker. I apply following steps:

* Text is sentence-tokenized.
* Each sentence then is word-tokenized.
* Tokens then are tagged using nltk.pos\_tag()
* Recover phrases using regular expression. There is another approach using NLTK and its CoLL 2000 corpus.
* To run the code, install nltk package, and run following command: python NPchunker.py

How would the system perform if the input consisted of something different from what it was trained on, like transcripts of spoken language? (Since you don't have annotated data for this, you don't need to actually test this.)

The down side of applying Regex for chunking problem is the more rules we add in the lower precision is. The rules might work well on this type of text but might have lower performance on the other.

How would the performance of the chunker be affected if the POS tags were only 85% accurate? Can you minimize this effect? (Measure the error, but just describe what you would do to mitigate it.)

As far as I know there is another approach to chunk parser using CoNLL 2000 Corpus. ChunkParse scores: Precision: 82.5%, Recall 86.8%. Above accuracy is very impressive already?

How easily could your system be adapted to Russian? What resources would you need? Would there be significant changes to the code? (Again, no need to test this.)

I don’t know about Russian. But I assume that we could do chunking similarly if we have a “Penn Treebank” of Russian. There might not be any change to the code.

**Tweet Classification (experimental design)**

Consider the task of classifying tweets into those supporting or opposing a particular piece of legislation. This happens offline, so you can use whatever complicated processing you want. The constraint is that acquiring annotated data is expensive. Once you've decided on a particular classification approach, how do you know you have enough training data? What strategy would you use to decide what tweets you want annotated and when to stop?

IMO, the best way to know when we have enough training data is splitting the data into training set and test (or cross validation) set (70-30%). I will plot two learning curves for the training set and test set. When the gap between these two curves close I know I have enough data.

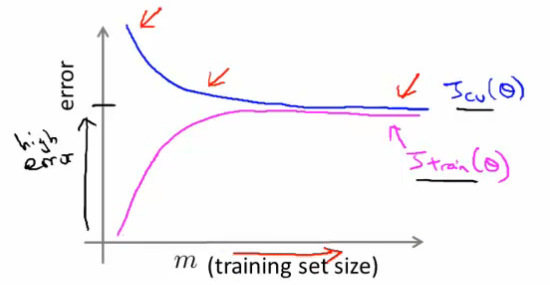


Figure : Adapted from Andrew Ng's lectures

Over time, the conversation will evolve and the classifier might degrade in performance (since whatever features it was relying on might have changed distribution.) How can you measure this drift? How can you mitigate it? You can tag more data, but which and how much? What do you do with the old tagged data?

With the same size of test set, but is shifted to more current time, the gap of above learning curve will open. Therefore the gap size might be used to measure the drift. In order to mitigate this degradation, I will get more data. But this data is like a window of data shifting to the right along with the evolution of tweets. The size of data depends on the learning curve above.

**Train/Test Split (machine learning)**

Describe the motivation for train/test splits. Why would you use a train/development/test split?

The motivation for train/test splits is to evaluate the performance.

Describe training a classifier using supervised learning in terms of optimizing a function. What value are you optimizing? How do you calculate that value?

For a classifier, the value used to define the performance is cost function. Cost function is derived from the hypothesis outputs and real outcomes. For example in logistic regression models, the cost function is a convex logistic regression cost function.

**Compression (information theory)**

A text file containing the slice of the Brown corpus found in NLTK is 6127074 bytes. (Sentences separated by newlines, tokens separated by spaces.) You can create a new file of the same size by randomizing the order of the tokens in each sentence.

The gzipped sizes of these two files are 2260992 and 2511610 bytes, respectively. What could cause the difference? How would you try to find out? Do you think the effect would be different in a language like Russian?

As far as I know the text compression works by creating on-the-fly dictionaries. So tokens are replace by an index to that dictionary. Therefore by randomizing the order of the tokens in each sentence, the order of tokens will change. As the result, the gzipped sizes change respectively.

I am not sure about this question: How would you try to find out? But in order to compare two files in unix, a command: diff file\_1 file\_2 could tell us the difference of two files.

I don’t know Russian, but if above compression theory is applied to Russian. I think that the gzipped size will change.

**POS Tagger (linguistics)**

Annotate the parts of speech of the words in the following sentence, using the Penn Treebank tagset: "Mary purchases breakfast." Many POS taggers (including the one in NLTK) get this wrong. Why might that be?  http://textanalysisonline.com/nltk-pos-tagging (Web interface to the NLTK POS tagger.)

Mary|NNP purchases|VBZ breakfast|NNP .|.

Reason POS taggers might give low performance is probably due to lexicon gap. The word “purchases” occurred many times in the corpus but as a noun. Tagger also relies on the contextual features, in this case “breakfast” occurs as a verb with high probability.