Presentations next week:

Hannah, Lara, (Guoyan), Lucien, (John), Rebecca

Final project due: Wednesday, May 13 @ 5:00pm

Turn in a paper explaining what you did, how well it worked, why it worked as well as it did but not better, etc., plus any programs you wrote

The project grade will be based on the paper, which should look like a conference paper (~8 pages, references)

---

Are chunks within the same clause part of the same argument?

Feature vector:

NP, NP, Britain, NNP, ’s, POS, yes.
NP, VP, industry, NN, is, VBZ, no.

Use Timbl for classification

Best result so far: 81.27% accuracy

Largest source of error is PPs

---

Much of the power of SVMs comes from the use of kernel functions and derived feature spaces

Linear kernels allow efficient processing of the very large feature vectors that come with a bag of words model

Polynomial kernels capture dependencies between features

Special purpose kernels reflect the structure of a particular problem

Combinations of kernels are also kernels
**String kernels**

- *String subsequence kernels* represent as string as a bag of (possibly discontinuous) n-grams
- The feature set is very large, but dot products can be computed efficiently
- Dynamic programming and suffix trees
- For text classification SSKs give a small improvement over n-gram kernels for small training sets

**Tree kernels**

- We can use similar tricks to compare trees by comparing common subtrees
- Given trees $T_1$ and $T_2$, with nodes $N_1$ and $N_2$, define:
  $$I_i(n) = \begin{cases} 1 & \text{if subtree } i \text{ is rooted at } n \\ 0 & \text{otherwise} \end{cases}$$
- The kernel function is:
  $$K(T_1, T_2) = h(T_1) \cdot h(T_2)$$
  where
  $$h_i(T_1) = \sum_{n_1 \in N_1} I_i(n_1)$$
or the number of times subtree $i$ occurs in tree $T_1$.

---

**Tree kernels**

- The feature vector $h(T_i)$ will have as many dimensions as there are possible subtrees (which will be astronomical)
- But, the dot product $h(T_1) \cdot h(T_2)$ can only depend on dimensions for subtrees which occur in both $T_1$ and $T_2$
- Let $C(n_1, n_2)$ be the number of common subtrees rooted at $n_1$ and $n_2$
- The kernel function is:
  $$K(T_1, T_2) = h(T_1) \cdot h(T_2)$$
  $$= \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_i I_i(n_1) I_i(n_2)$$
  $$= \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} C(n_1, n_2)$$

- We can compute efficiently compute $C(n_1, n_2)$ by recursion
- If the rules applied at $n_1$ and $n_2$ are different, then there are no common subtrees and $C(n_1, n_2) = 0$
- If the rules are the same and $n_1$ and $n_2$ are preterminals, then $C(n_1, n_2) = 1$
- Otherwise:
  $$C(n_1, n_2) = \prod_{i=1}^{nc(n_1)} (1 + C(ch(n_1, i), ch(n_2, i)))$$

- Worst case, $K$ can be computed in $O(|N_1| |N_2|)$ time, but in practice $C(n_1, n_2) = 0$ for most $n_1, n_2$ and the computation is much cheaper
Tree kernels

- The number of common subtrees depends on the total number of subtrees.

- To correct for this we can normalize this (or any) kernel:

\[
\hat{K}(x_i, x_j) = \frac{K(x_i, x_j)}{\sqrt{K(x_i, x_i)K(x_j, x_j)}}
\]

- Like the string kernel, we can add a decay term \( \lambda \) to give larger subtrees less weight:

\[
C(n_1, n_2) = \prod_{i=1}^{n_c(n_1)} (1 + \lambda C(ch(n_1, i), ch(n_2, i)))
\]

Kernel functions

- Implementation available at http://www.isi.edu/~hdaume/SVMsequel/

- Similar kernels have been proposed for dependency structures, tag sequences, etc.

- A general divide and conquer strategy for designing kernels

- Locality-improved kernels have been used in image processing and bioinformatics

- Specific domain knowledge may also be built into the kernel function (codon-improved kernels in bioinformatics)

- The benefits of kernel methods have only begun to be explored in linguistics

Tree kernels

- Some parsing results (Collins and Duffy 2002):

<table>
<thead>
<tr>
<th></th>
<th>LR</th>
<th>LP</th>
<th>CBs 0</th>
<th>CBs 0</th>
<th>CBs 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤40 Words (2245 sentences)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO99</td>
<td>88.5%</td>
<td>88.7%</td>
<td>0.92</td>
<td>66.7%</td>
<td>87.1%</td>
</tr>
<tr>
<td>VP</td>
<td>89.1%</td>
<td>89.4%</td>
<td>0.85</td>
<td>69.3%</td>
<td>88.2%</td>
</tr>
<tr>
<td>≤100 Words (2416 sentences)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO99</td>
<td>88.1%</td>
<td>88.3%</td>
<td>1.06</td>
<td>64.0%</td>
<td>85.1%</td>
</tr>
<tr>
<td>VP</td>
<td>88.6%</td>
<td>88.9%</td>
<td>0.99</td>
<td>66.5%</td>
<td>86.3%</td>
</tr>
</tbody>
</table>

- And named entity recognition:

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>R</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxEnt</td>
<td>84.4%</td>
<td>86.3%</td>
<td>85.3%</td>
</tr>
<tr>
<td>VP</td>
<td>86.1%</td>
<td>89.1%</td>
<td>87.6%</td>
</tr>
</tbody>
</table>

Support vector machines

- Support Vector Machines come from separating hyperplanes + margin error bound + kernel functions + optimization

- The math is daunting, but there are a number of SVM libraries that are reasonably efficient and easy to use

- Many, many model variants and training methods

- Generally give very high performance (as in accuracy), but don’t scale to large training sets well

- Designing appropriate kernel function for linguistic problems
Digital computers can't represent real numbers:

```bash
bulba% python
Python 2.2.1 (#1, Aug 30 2002, 12:15:30)
[GCC 3.2 20020822 (Red Hat Linux Rawhide 3.2-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 3.3
3.2999999999999998
```}

- Financial calculations use integers
- Scientific calculations use approximations, which vary in their accuracy
- Standard for floating point calculations: IEEE 754

Floating point numbers are stored as a *mantissa* and an *exponent*

IEEE floating point formats:

<table>
<thead>
<tr>
<th>precision</th>
<th>min</th>
<th>max</th>
<th>eps</th>
<th>digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>$1.2 \times 10^{-38}$</td>
<td>$3.4 \times 10^{38}$</td>
<td>$1.2 \times 10^{-7}$</td>
<td>7</td>
</tr>
<tr>
<td>double</td>
<td>$2.2 \times 10^{-308}$</td>
<td>$1.8 \times 10^{308}$</td>
<td>$2.2 \times 10^{-16}$</td>
<td>16</td>
</tr>
</tbody>
</table>

Just because you can represent $10^{300}$ doesn't mean you get 300 significant digits!

Default in python and perl is double precision

Don't use single precision (float) unless you have a good reason

It's easy to lose precision:

$$1000.2 - 1000.0 = 0.200012$$

Things to watch out for:
- subtractions of numbers that are nearly equal,
- additions of numbers whose magnitudes are nearly equal, but whose signs are opposite
- additions and subtractions of numbers that differ greatly in magnitude

Exact comparisons between floating point numbers can be misleading

The same operations performed in a different order or on different hardware may given different results

We've come a long way, from flipping coins to Support Vector Machines

Non-parametric methods:
- decision trees
- instance-based learning
- transformation-based learning
- perceptron
- support vector machines

Parametric methods:
- naive Bayes
- maximum entropy
A look back

- One theme that runs through machine learning research is the way we characterize *generalization*
- curse of dimensionality
- bias vs. variance
- overtraining
- simplicity
- capacity

A look ahead

- Some current directions in machine learning for NLP:
  - getting at ‘deep’ structures
  - task-specific representations (remember, there’s no free lunch!)
  - scaling methods to deal with huge datasets
- Data mining uses machine learning to find patterns in unstructured data collections...
- . . .which we’ll be looking at in more detail in the fall