

Some CMPS 242 Project Ideas 2006

Here are some project topics that I am interested in, and would be happy to collaborate on.

1. Learning when a source-code change is likely to be buggy. We have (through Jim Whitehead and his students) a lot of information on the changes made to several open-source systems. By analyzing the content of change log messages associated with code commits to software configuration management systems, it is possible to characterize changes as either bug fixes, which repair a software error, or non-bug-fixes, which introduce new functionality or adapt a software system to a changing environment. Working backwards through a project's revision history, it is possible to trace from a bug-fix change to the original change that injected a bug into the software system, a so-called bug-introducing change. Sung Kim has just finished a dissertation with encouraging results using Naive Bayes and basic SVM techniques to classify changes as buggy (or not buggy), but there still many open issues. This is an interesting problem, as high precision (being correct when you claim a change is buggy) is probably more important than raw accuracy.
 - (a) Sung used one model per open-source project, would building a different model for each programmer lead to better results?
 - (b) Can better training data be built from the available information, perhaps using co-training?
 - (c) The SVM methods Sung used are basically the Weka defaults, can one do better with other kernels and/or soft margin parameters?
 - (d) Can other learning methods (boosting, decision trees, logistic regression etc) do as well or better on this problem?
 - (e) Although the problem is inherently on-line (each change must be predicted on when it is committed) Sung treated the problem as a batch problem. How much loss in accuracy happens when algorithms must predict on changes using only the information available when they are committed?
 - (f) Do incremental on-line algorithms (such as Winnow) work well on this problem? There is a complicating issue in that changes are never (rarely) labeled "clean", and are only labeled "buggy" after they have been fixed.
 - (g) Are there ways to get better precision/accuracy tradeoffs?
2. Netflixs predict movie ratings contest
3. Identifying elephant seal dive types
4. Learning file lifetimes (systems group probably has data).
5. Learning adaptive cache management strategies (extending a paper of Manfred's).
6. Learning for power management in laptops (using Intel data from Theocharous).

Here are some other interesting project ideas:

- learning weights to more closely predict expert moves in Go
- Recursively use AdaBoost for segmentation: classify based on features and then on derived features which are the (previous iteration's) classifications of nearby points.
- Compare iterative vs. LP all-at-once boosting (see Manfred's papers)
- Finding the k in k-clustering - survey and evaluate the various approaches.
- Text data mining/classification problems (Yi Zhang has others also)
 - Classify reviews according to the prevailing opinion (favorable/unfavorable) (epinion review data)
 - Automatically sort mail into folders (enron email corpus)
 - Spam detection
 - topic detection and threading in streams of data: on-line text classification
- One of the TREC data mining challenges <http://trec.nist.gov/>
- handwritten digit classification
- Learning with non IID data - say the training set has two parts, one of which is known to have a little more noise than the other. How can you exploit this knowledge?