Classification of Academic Papers based on Abstract Content

Deepanjan Roy, Ian Forbes, Muntasir Chowdhury

October 12, 2014

1 Abstract

We explore various machine learning methods for text classification. Using a set of approximately 100 000 abstracts, labelled into one of four topic, we try to accurately predict the topic of unseen abstract, solely based on their textual content.

2 Introduction

3 Related Work

4 Algorithms

4.1 Naive Bayes

We analysed two variations of the Naive Bayes classifier, Bernoulli and Multinomial. We implemented the Bernoulli classifier from scratch and used the Multinomial classier included in scikit-learn for comparison.

4.1.1 Results

4.2 K Nearest Neighbour

The K Nearest Neighbour algorithm (kNN) is a *lazy* machine learning algorithm that can be used for both classification and regression tasks. For classification task, such as text classification, it classifies inputs by simply finding the nearest neighbour of the input in the training set based on some metric function $d(x_i, x_j)$ and outputs the label of that nearest neighbour. One of the algorithm's main parameters is K, the number of neighbours to consider. In the most simplistic version of the algorithm the majority label of the K nearest neighbours is selected. More complex methods use weighting functions based on the distance of the nearest neighbours, or their relative position in terms of distances, i.e. the first nearest neighbour is weight differently than *ith* nearest neighbour based on i.

4.2.1 Features

We used binary valued vectors, as features. We tried different vector lengths from 2^{10} upto 2^{14}.

4.2.2 Implementation

For our implementation we selected $d(x_i, x_j)$ to be number of element 2 given vectors have in common. In particular we took the bitwise and of the 2 given vectors the counted the number of bits that were set. The metric corresponds to the number of words that 2 given abstracts have in common.

Implemented in C++ the code for computing the distances looks similar to the following. In order to store the binary feature vectors we used bitsets.

```
// declare the length of the feature vector
const size_t N = 2048;
// create the bitsets
std::bitset<N> a,b;
// read bitstrings from stdin
std::cin >> a >> b;
uint_t distance = (a & b).count();
```

4.2.3 Optimizations

Using bitsets to represent feature vectors allows for various optimizations in terms of CPU and memory usage. Compared to the naive approach of using byte vectors, or integer vectors, we can reduce the amount memory needed to store the training and test set by factors of 8 and 32 respectively. This meant our algorithm never consumed more than 300MB of memory.

Also the contiguous nature of bitsets meant the that all of the training sets could be in one large chunk of memory. This allows for allows for large speed up on newer machines with hardware prefetching. In particular forward iteration over contiguous memory on machines with prefetching gives the illusion that the CPU has an infinite amount of cache.

Another advantage of using bitsets is that the bitwise AND and the POPCNT instructions are implemented directly in hardware. In particular the POPCNT (population count) instruction which counts the number bits that are set to 1 in 64 bit integer (unsigned long) can replace a loop with bit shifting.

4.2.4 Results

We tested a large

4.3 Support Vector Machine

4.3.1 Results

5 Discussion