SOFT COMPUTING

PROJECT

SEMANTIC CLASSIFICATION OF SENTENCES

Professor:
Đorđe Obradović, PhD
Teaching assistant:
Marko Jocić, MSc

Students:
Nenad Todorović RA 78/2012
Dragan Vidaković RA 134/2012

Novi Sad, February 2016

Table of Contents

1. Introduction

1.1. Motivation

Amount of textual data being created on the Internet is growing exponentially, and that data has a lot of information hidden underneath. There are a lot of reasons why someone would like to reveal that hidden information, for example, to study social behavior, public opinion, to be able to focus advertisement, to gather domain specific knowledge, etc. We, as authors of this project, wanted to learn how to classify and reveal some semantics hidden in textual data.

The domain of the problem we've chosen - classifying sentences from academic papers, is a good starting point for involving in this field of Machine Learning, because of the way those papers are structured. They always contain several topics: past work that provides the basis for the work in the article, comparison or critique of the past work, the specific research goal of the paper, authors own work (methods, results, and conclusions), etc. Structured data is easier for beginners to comprehend, and allowed us to try out various methods for solving this problem.

1.2. Problem Definition

Generally, the goal of sentence categorization is the classification of sentence into a fixed number of predefined categories. Each sentence can be in exactly one, multiple, or no category at all. Using Machine Learning, the objective is to learn classifiers from examples which perform the category assignments automatically. This is a supervised learning problem. Since categories may overlap, each category is treated as a separate binary classification.

Our problem presents one type of sentence categorization, and represents automatic assigning of input sentences from academic papers to a set of categories. Sentence assigning process is based on its semantic. Categories are defined with indicator words, and there are five of them:

1. Aim - The specific research goal of the paper
2. Own - The author's own work (e.g. methods, results, conclusions)
3. Contrast - Contrast, comparison or critique of past work
4. Basis - Past work that provides the basis for the work in the article
5. Miscellaneous - Any other sentences.

2. Background

In the next three sections we'll take a closer look at three Machine Learning methods that we used to automatically build classification models: Naïve Bayes Classifier, Decision Trees and Support Vector Machines. We'll take a closer look at how these learning methods select models based on the data in training set. An understanding of these methods guided our selection of appropriate features.

2.1. Naïve Bayes Classifier

In Naïve Bayes Classifiers, every feature gets a say in determining which label should be assigned to a given input value. To choose a label for an input value, the Naïve Bayes Classifier begins by calculating the prior probability of each label, which is determined by checking frequency of each label in the training set. The contribution from each feature is then combined with this prior probability, to arrive at a likelihood estimate for each label.

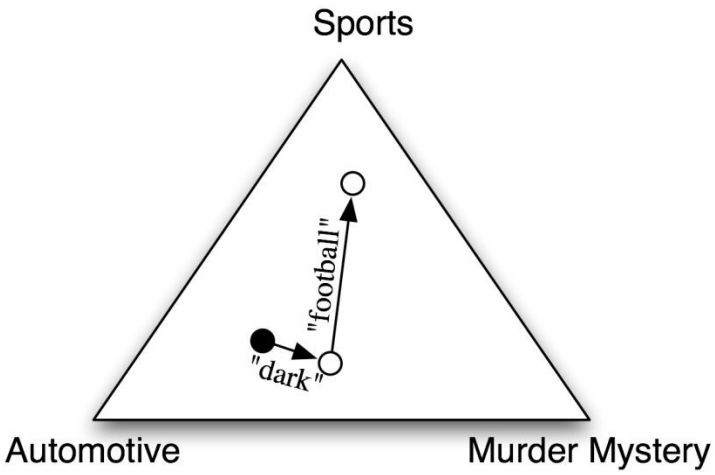


Figure 1: An abstract illustration of the procedure used by the Naïve Bayes Classifier to choose the topic for a document. In the training corpus, most documents are automotive, so the classifier starts out at a point closer to the "automotive" label. But it then considers the effect of each feature. In this example, the input document contains the word "dark," which is a weak indicator for murder mysteries, but it also contains the word "football," which is a strong indicator for sports documents. After every feature has made its contribution, the classifier checks which label it is closest to, and assigns that label to the input.

Another way of understanding the Naïve Bayes Classifier is that it chooses the most likely label for an input, under the assumption that every input value is generated by first choosing a class label for that input value, and then generating each feature, entirely independent of every other feature. Of course, this assumption is unrealistic; features are often highly dependent on one another. We'll return to some of the consequences of this assumption at the end of this section. This simplifying assumption, known as the Naïve Bayes assumption (or independence assumption) makes it much

easier to combine the contributions of the different features, since we don't need to worry about how they should interact with one another.

The reason that Naïve Bayes Classifiers are called "naive" is that it's unreasonable to assume that all features are independent of one another (given the label). In particular, almost all real-world problems contain features with varying degrees of dependence on one another. If we had to avoid any features that were dependent on one another, it would be very difficult to construct good feature sets that provide the required information to the machine learning algorithm.

2.2. Decision Tree Classifier

A Decision Tree is a simple flowchart that selects labels for input values. This flowchart consists of decision nodes, which check feature values, and leaf nodes, which assign labels. To choose the label for an input value, we begin at the flowcharts initial decision node, known as its root node. This node contains a condition that checks one of the input value's features, and selects a branch based on that features value. Following the branch that describes our input value, we arrive at a new decision node, with a new condition on the input value's features. We continue following the branch selected by each node's condition, until we arrive at a leaf node which provides a label for the input value.

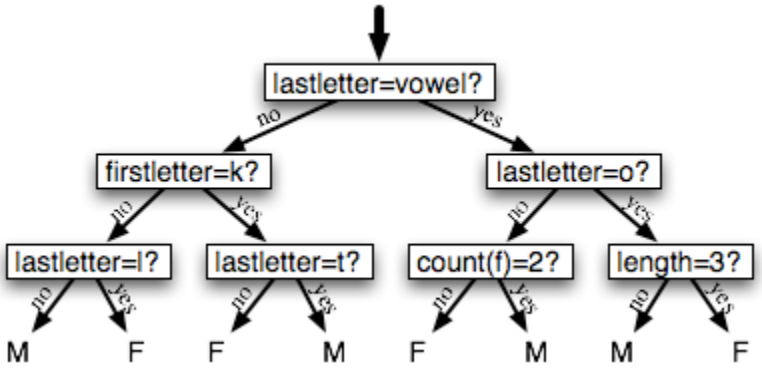


Figure 2: Decision Tree model for the name gender task

Once we have a Decision Tree, it's straightforward to use it to assign labels to new input values. What's less straightforward is how we can build Decision Tree that models a given training set. Before building Decision Trees, we should pick the best "Decision Stump" for a corpus. A Decision Stump is a Decision Tree with a single node that decides how to classify inputs based on a single feature. It contains one leaf for each possible feature value, specifying the class label that should be assigned to inputs whose features have that value. In order to build a Decision Stump, we must first decide which feature should be used. The simplest method is to just build a Decision Stump for each possible feature, and see which one achieves the highest accuracy on the training data, although there are other alternatives that we will discuss below. Once we've picked a feature, we can build the decision stump by assigning a label to each leaf based on the most frequent label for the selected examples in the training set (i.e., the examples where the selected feature has that value). Given the

algorithm for choosing Decision Stumps, the algorithm for growing larger Decision Trees is straightforward. We begin by selecting the overall best Decision Stump for the classification task. We then check the accuracy of each of the leaves on the training set. Leaves that do not achieve sufficient accuracy are then replaced by new Decision Stumps, trained on the subset of the training corpus that is selected by the path to the leaf. For example, we could grow the decision tree in Figure 2 by replacing the leftmost leaf with a new decision stump, trained on the subset of the training set names that do not start with a "k" or end with a vowel or an "l."

As was mentioned before, there are several methods for identifying the most informative feature for a Decision Stump. One popular alternative, called information gain, measures how much more organized the input values become when we divide them up using a given feature. To measure how disorganized the original set of input values are, we calculate entropy of their labels, which will be high if the input values have highly varied labels, and low if many input values all have the same label.

Once we have calculated the entropy of the original set of input values' labels, we can determine how much more organized the labels become once we apply the Decision Stump. To do so, we calculate the entropy for each of the Decision Stumps leaves, and take the average of those leaf entropy values (weighted by the number of samples in each leaf). The information gain is then equal to the original entropy minus this new, reduced entropy. The higher the information gain, the better job the Decision Stump does of dividing the input values into coherent groups, so we can build Decision Trees by selecting the Decision Stumps with the highest information gain.

Another consideration for Decision Trees is efficiency. The simple algorithm for selecting Decision Stumps described above must construct a candidate Decision Stump for every possible feature, and this process must be repeated for every node in the constructed Decision Tree. A number of algorithms have been developed to cut down on the training time by storing and reusing information about previously evaluated examples.

Decision Trees have a number of useful qualities. To begin with, they're simple to understand, and easy to interpret. This is especially true near the top of the Decision Tree, where it is usually possible for the learning algorithm to find very useful features. Decision Trees are especially well suited to cases where many hierarchical categorical distinctions can be made. For example, Decision Trees can be very effective at capturing phylogeny trees.

However, Decision Trees also have a few disadvantages. One problem is that, since each branch in the Decision Tree splits the training data, the amount of training data available to train nodes lower in the tree can become quite small. As a result, these lower decision nodes may overfit the training set, learning patterns that reflect idiosyncrasies of the training set rather than linguistically significant patterns in the underlying problem. One solution to this problem is to stop dividing nodes once the amount of training data becomes too small. Another solution is to grow a full Decision Tree, but then to prune decision nodes that do not improve performance on a dev-test.

A second problem with Decision Trees is that they force features to be checked in a specific order, even when features may act relatively independently of one another. For example, when classifying documents into topics (such as sports, automotive, or murder mystery), features such as hasword(football) are highly indicative of a specific label, regardless of what other the feature values are. Since there is limited space near the top of the Decision Tree, most of these features will need to

be repeated on many different branches in the tree. And since the number of branches increases exponentially as we go down the tree, the amount of repetition can be very large.

A related problem is that Decision Trees are not good at making use of features that are weak predictors of the correct label. Since these features make relatively small incremental improvements, they tend to occur very low in the Decision Tree. But by the time the decision tree learner has descended far enough to use these features, there is not enough training data left to reliably determine what effect they should have. If we could instead look at the effect of these features across the entire training set, then we might be able to make some conclusions about how they should affect the choice of label.

The fact that Decision Trees require that features be checked in a specific order limits their ability to exploit features that are relatively independent of one another.

2.3. Support Vector Machines

Support Vector Machines are based on the Structural Risk Minimization principle from computational learning theory. The idea of Structural Risk Minimization is to find a hypothesis *h* for which we can guarantee the lowest true error. The true error of *h* is the probability that *h* will make an error on an unseen and randomly selected test example. An upper bound can be used to connect the true error of a hypothesis *h* with the error of *h* on the training set and the complexity of *H* (measured by Vapnik–Chervonenkis dimension), the hypothesis space containing *h*. Support Vector Machines find the hypothesis *h* which (approximately) minimizes this bound on the true error by effectively and efficiently controlling the Vapnik–Chervonenkis dimension *H*.

SVMs are very universal learners. In their basic form, SVMs learn linear threshold function. Nevertheless, by a simple "plug-in" of an appropriate kernel function, they can be used to learn polynomial classifiers, radial basic function (RBF) networks, and three-layer sigmoid neural nets.

One remarkable property of SVMs is that their ability to learn can be independent of the dimensionality of the feature space. SVMs measure the complexity of hypotheses base on the margin with which they separate the data, not the number of features. This means that we can generalize even in the presence of very many features, if our data is separable with a wide margin using functions from the hypothesis space.

The same margin argument also suggest a heuristic for selecting good parameter setting for the learner. The best parameter setting is the one which produces the hypothesis with the lowest Vapnik-Chervonenkis dimension. This allows fully automatic parameter tuning without expensive cross-validation.

When learning text classifiers, one has to deal with very many features. Since SVMs use overfitting protection, which does not necessarily depend on the number of features, they have the potential to handle these large feature spaces.

SVMs acknowledge the particular properties of text:

a. high dimensional feature spaces
b. few irrelevant features
c. sparse instance vectors

3. Data Set and Environment

3.1. Data Set

We used "Sentence Classification Data Set" downloaded from Machine Learning Repository of Center for Machine Learning and Intelligent Systems – University of California, Irvine (http://cml.ics.uci.edu/). Data Set contains data for classification of sentences from scientific articles into categories:

1. **AIMX** – The specific research goal of the paper
2. **OWNX** – The author's own work (methods, results, conclusions…)
3. **CONT** – Contrast, comparison or critique of past work
4. **BASE** – Past work that provides the basis for the work in the article
5. **MISC** – Any other sentences.

Data Set contains sentences from the abstract and introduction of 90 scientific article from three different domains:

1. PloS Computational Biology (PLOS)
2. The Machine Learning repository on arXiv (ARXIV)
3. The psychology journal Judgment and Decision Making (JDM).

Training Set contains 72 articles with 25000 sentences:

- 24 PLOS articles with 893 sentences
- 24 ARXIV articles with 793 sentences
- 24 JDM articles with 814 sentences

Test Set contains 18 articles with 617 sentences:

- 6 PLOS articles with 220 sentences
- 6 ARXIV articles with 197 sentences
- 6 JDM articles with 200 sentences

3.2. Tools and Environment

For project realization we used Python 3.4 and PyCharm IDE. We relied on Python libraries for natural language processing to successfully complete this project:

1. *NLTK 3.0 (Natural Language ToolKit)*

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries. Thanks to a hands-on-guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK has been called "a wonderful tool for teaching, and

working in, computational linguistics using Python" and "an amazing library to play with natural language."

2. *TextBlob v 0.11.1*

TextBlob is a Python library for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more. TextBlob stands on the giant shoulders of NLTK and pattern, and plays nicely with both.

4. Results

We measured performance on Intel Core i5 CPU @ 2.7 GHz and 8 GB of DDR3 RAM memory.

4.1. Initial Results

The initial results of the testing were very good. All three methods took pretty much the same time to train, around 3300 seconds. Testing showed that the most accurate method was Support Vector Machine method, with accuracy of 87.68%! SVM also took the least time to complete testing. Decision tree was good, but it was the worst of the three, with worst accuracy, and much longer test time. The table beneath shows all of the results:

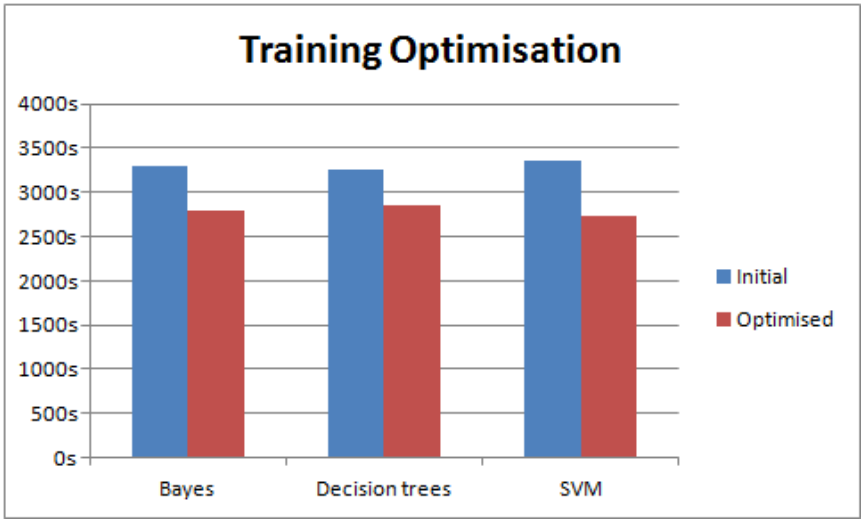
	Training Time	Test Time	Accuracy
Naïve Bayes	3293.93 s	1006.28 s	85.08 %
Decision Tree	3256.33 s	3382.86 s	84.76 %
SVM	3351.35 s	820.52 s	87.68 %

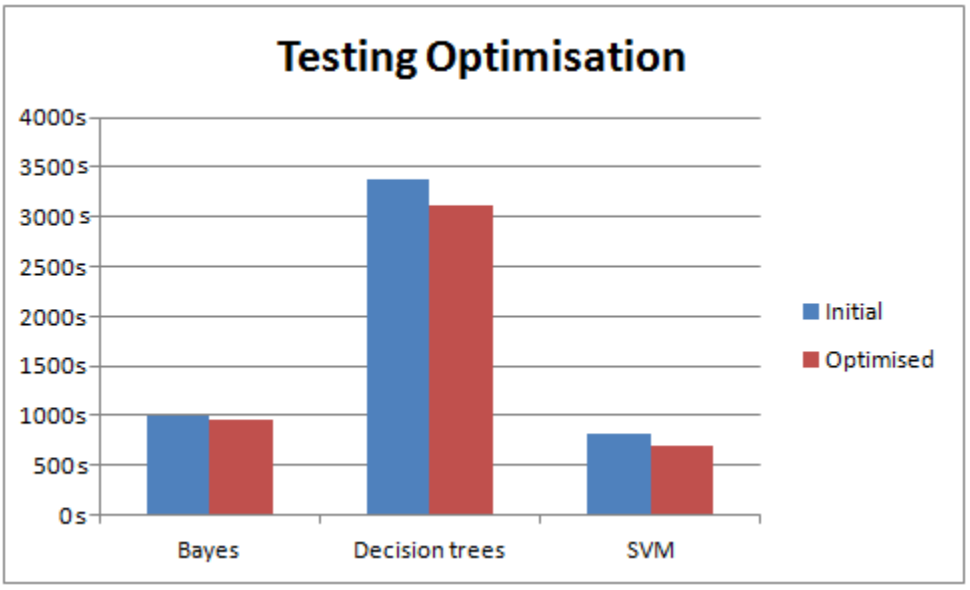
Table 1: Initial Results

4.2. Optimization

Optimization of the task was based on excluding some of the most common English words (as, the, that, a, etc.), that have little semantics that could contribute to solving this problem. We've tried using two groups of stopwords, and results showed us that the shorter group gave us better results.

Results of using cleansed data set were better than with data used initially. All three of the methods showed better accuracy, but what changed the most was the time it took to train all three methods. SVM training time was more than 500 seconds shorter than initial training time! Graphs beneath show comparisons of the methods:





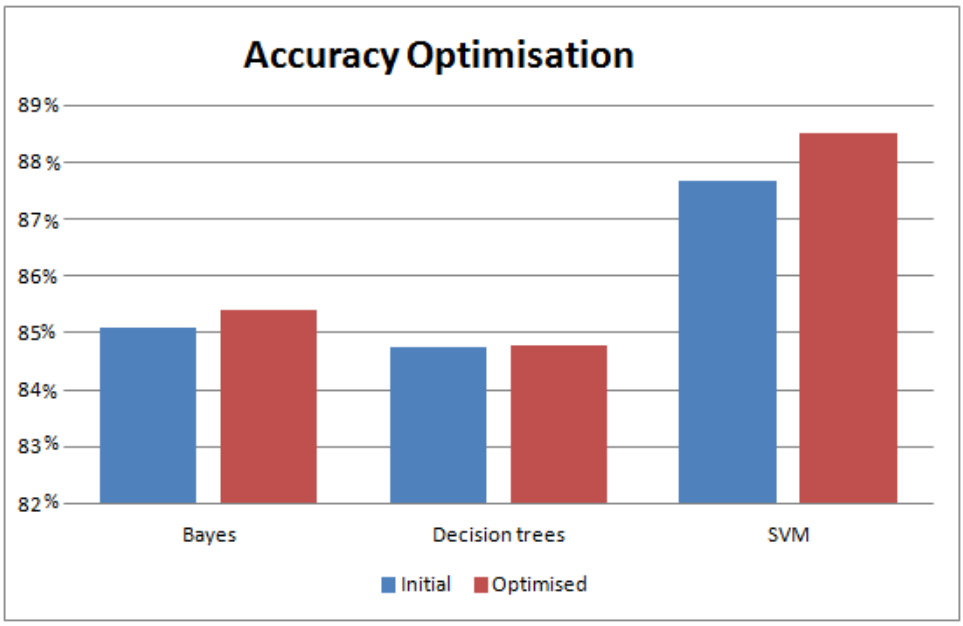


Figure 3: Graphs of optimized performance: training, testing and accuracy

5. Summary

Results show that SVMs consistently achieve good performance of text categorization with accuracy of 88.5 %, outperforming other used methods substantially and significantly (Naïve Bayes 85.41 %, Decision Tree 84.77 %). With their ability to generalize well in high dimensional feature spaces, SVMs eliminate the need for feature selection, making the application of text categorization considerably easier. SVMs do not require any parameter tuning, since they can find good parameter settings automatically. All this makes SVMs very promising and easy-to-use method for learning text classifiers from examples.

Literature

[1] Soft Computing 2015/16 course
[2] Computational Intelligence Fundamentals 2014/15 course
[3] www.nltk.org
[4] textblob.readthedocs.org
[5] Thorsten Joachims, *Text Categorization with Support Vector Machines: Learning with Many Relevant Features,* Univeristaet Dortmund