# Single-character OCR using Support Vector Machines

Olli Jarva & Jarno Rantanen

Aalto University School of Science

olli@jarva.fi & jarno@jrw.fi

**Abstract**

This paper describes a solution to an optical character recognition problem for bitmap characters using Support Vector Machines with an RBF kernel, including a description of RBF parameter search and bitmap normalization. Classification performance of 90.8% was achieved against a given training set of 40000 correctly labelled samples [**?**].

KEYWORDS: SVM, Support Vector Machine, RBF, OCR, Character Recognition

## 1 Data set description

The data set against which our solution was developed consisted of a provided set of 42152 black-and-white bitmaps, depicting hand-written instances of characters from the English alphabet (that is, the task was to classify the bitmaps into 26 distinct categories). Each bitmap was given as a 16-by-8 image, in the form of a binary vector of length 128 (meaning an array of 128 ones or zeroes). The vectors were delivered in a text file, with the correct label associated with each vector.

As an optional extra task, we were provided with the opportunity of participating in a competition amongst different solutions to the same classification problem. The competition was organized by first providing only a subset of the training data (10000 vectors), using that to train a classifier, and then calculating an error rate against the rest of the training data (which was not yet made available at that time). Our participation in the competition is discussed further in Section 8.

The same data set was used for both training our classifiers, and testing them afterwards. The data was split using k-fold cross-validation to minimize overfitting, while making the most of the available data. This testing technique is discussed in further detail in Section 6.

## 2 Method selection

- What other options were there

- Why we chose SVM?
- A nod to why we think we chose correctly

# 3 Support Vector Machines

Support Vector Machines (SVM's) are a method for supervised machine learning, meaning they solve a classification problem by creating a classifier function from a set of existing, labeled data points. This function can then be used to classify (or *label*) subsequent data points *without* supervision. In contrast to *regression methods* which produce continuous output, the output of a classifier function is always exactly one class into which the input data point (likely) belongs.

In its most basic implementation, an SVM is trained with data labeled into exactly two separate groups. The resulting classifier is a *binary one*, meaning it will classify subsequent input into those same two categories. If the input data points belong to a two-dimensional space, this can be intuitively thought of separating the data points into two clusters. To minimize the potential for generalization error, the clusters should be separated by as wide a band as possible (or in simpler terms, by a line, with as much space between the line and the nearest data points as possible).

SVM's generalize nicely into higher dimensional spaces. In an n-dimensional input space, the two classes are linearly separated by an (n-1)-dimensional hyperplane instead of a line. They also generalize into working with >2 classes by way of reducing the multi-class classification problem into a set of binary classification problems. This can be done, for example, by chaining the binary classifiers so that the first classifies the input data as either belonging to class "A" or "other", the second one to "B" or other, and so on.

The above works off the assumption that the sets being discriminated are linearly separable in the input space. With realistic data sets, however, this is often not the case. To keep the sets linearly separable (and thus the SVM approach applicable), the input space can be mapped into a much higher-dimensional space. The assumption is that with this added sparsity, a linearly separating hyperplane can be found, even for problematic input data sets.

Such mappings can be achieved using what are known as *kernel functions*. Kernel functions have special properties that, in addition to helping the linear separability, reduce the computational load bearable. Numerous kernel functions have been proposed in literature, and new ones are being researched. The performance characteristics of the functions are highly dependent on the type of classification problem at hand, and the properties of the input space, and not all kernels work for all problems. There is, however, little theoretical base on how to *choose* a suitable kernel for a given data set, and thus it is in fact common to simply rely on empirical methods and compare the performance some common kernels, choosing the one that best fits the specific data set.

# 4 Kernel selection

The Radial Basis Function (RBF) kernel is a common first-choice kernel suggested by literature [?, ?]. It has several desireable properties. Firstly, it is numerically robust in comparison to some other kernels. Secondly, another common alternative - the linear kernel - is simply a special case of the RBF one. Thirdly, and perhaps most importantly, the SVM kernel is configured with only two parameters (as opposed to the 4 parameters of the polynomial kernel, for example). As the parameter search is in essence a local optimization problem, having a 2-dimensional search space makes this problem more manageable than, say, searching in 4 dimensions. The parameter search is discussed further in Section 6.

# 5 Character preprocessing

An SVM operates on input data represented as vectors of real numbers. Scaling the input data is very important with SVM's; features of the input data with large numerical ranges can easily dominate ones with smaller ranges, even though the width of their range has no real correlation with their importance in the actual classification problem at hand. It is thus suggested to always scale the data into a normalized range of [-1,1], or even [0,1].

Much of the preprocessing for the input data in our experiment was already done for us; the image data was nicely encoded into a set of binary vectors, so no bitmap processing was required. Also, since the vectors were binary, no data scaling was required.

The single preprocessing technique we opted for was basic noise reduction, by moving each image to the bottom left corner. That is, making sure images otherwise identical except for their padding would still look identical to the algorithm. We found this to increase our initial classification performance by 0.5%.

# 6 RBF kernel parameter search

Optimal parameters for the SVM (and the kernel function) present a local optimization problem in a 2-dimensional search space. The dimensions being explored are $\gamma$ (the RBF kernel parameter) and $C$ (the SVM penalty parameter). Since an exhaustive search in this space isn't possible, a basic grid search was employed.

In the first step of the search, the space was divided into a grid of 6-by-6 (totaling 36 cells), covering the range [-10,0] for $\gamma$, and [-1,9] for $C$. At the origin point in each cell, the classification performance was then measured. Out of the explored cells, the best one was chosen for the next step. In the second step, the candidate cell was further explored with a finer grid. This grid was X-by-Y around the origin point, covering the area within a few percent of the origin.

The aforementioned classification performance was measured using a technique known as *k-fold cross-validation*. In a naive implementation of classifier training, the input data would be divided into two different sets: a training set, which is used to train the classifier, and a validation set, that is used to measure the performance of the classifier (as both sets contain the correct labels for data points). This approach is not optimal, however: the classifier may suffer from *overfitting*, meaning it ends up modeling more the training set, instead of the actual classification problem being solved. This is due to the classifier being trained only against a specific part of the training data, and validated against another. How those sets are chosen (that is, picked from the entire set of labelled data available) can greatly affect the resulting classifier. Should the validation data set be chosen with a bit of bad luck, for example, it may end up containing a very specific subset of the entire data, skewing the resulting error rates.

K-fold cross-validation avoids these issues by dividing the labelled data into k-segments of equal size, and then using each as the validation set in turn. The rest of the segments are then used to train the classifier being validated. Once each segment has been validated against, the error rate is calculated over the entire set of validations. This both makes use of the entire data set for training, and avoids overfitting, all parts of the data are (at one point) used for validation.

# 7 Classifier chaining

# 8 Results and performance

- k-fold cross validation: k=20, error rate 11%
- k-fold cross validation: k=5, error rate 11.5%
- One iteration with training set n=40000 and validation set n=2152 about 17 min with 2.1GHz Xeon (single thread)
- about 300MB of memory for training set n=42152
- Predicting one character: about 2 milliseconds

# 9 Quick comparison to other algorithms

- kNN (+PCA/LDA)
- ...?

[1]

# References

[1] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello. mlpy: Machine learning python, 2012.