Learning to predict the quality of classifiers

Subset selection for multiple classifier systems based on a single set of features

A. Bram Neijt

December 2009

Master Thesis
Artificial Intelligence / Autonomous perceptive systems
Department of Artificial Intelligence
University of Groningen, Groningen, The Netherlands

Supervisors:

prof. dr. L.R.B. Schomaker (Artificial Intelligence, University of Groningen)
drs. T. van der Zant (Artificial Intelligence, University of Groningen)

Contents

Chapter 1

Introduction

Since the invention of the computer, people have been working on getting as much data into them as possible. All means are put to the task: keyboards, mice, pen tablets, bar-code readers and even sound cards can be seen as input devices. Once the information is in the computer, the computer allows us to quickly search, sort and change the information. Getting the information out of a computer is astonishingly simple: send an image to a printer. Printing something out is much easier than putting printed things in, which is a problem if you have printed data for which there is no computer readable equivalent.

This problem was already known in 1929, when G. Tauchneck filed a patent [43] for a Reading Machine, shown in figure 1.1. This reading machine uses a photoelectric cell to test a character against a set of templates (seen on the main wheel in figure 1.1). However, the optical character recognition (OCR) problem is not solved yet, which is best illustrated by the fact that the Google OCR system today files this patent under *Beading Machine*, instead of *Reading Machine* [38].

The field of character recognition has made significant progress and has since seen a large growth [10]. Despite this growth, text recognition has not been solved yet: if automatically reading the bar-code with lasers fails, the cash register can not automatically switch to a camera to read the numbers below it. It is still the task of the cashier to type these digits in. Looking at the problem of handwritten as opposed to printed text, the amount of variation possible [6] makes interpretation even more difficult. There have been a lot of different methods to solve these problems, but none of them have been able to solve the recognition problems as good as humans do. Maybe research will never find a single approach, only a collection of specialised approaches combined in the right way: a multiple classifier system (MCS).

In an MCS, different methods are combined into one system. While

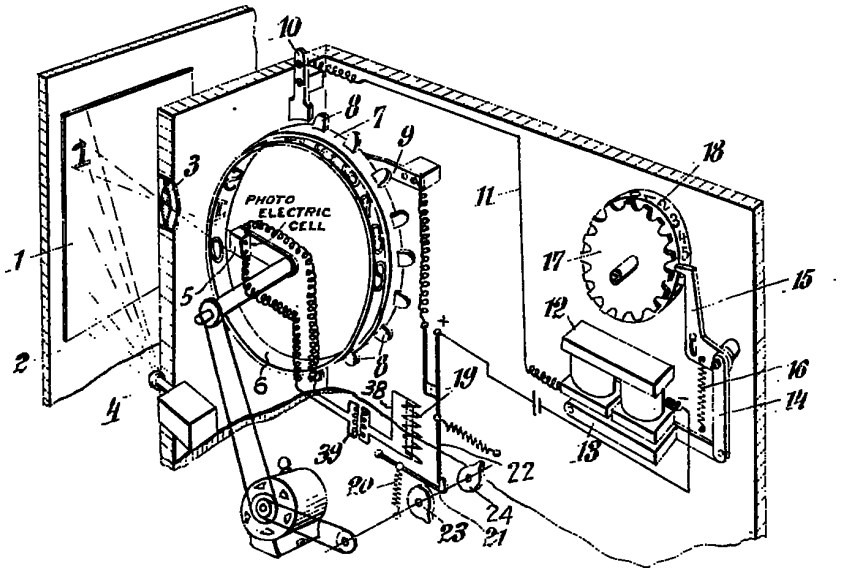


Figure 1.1: The Reading Machine as depicted in The 1929 filed patent by G. Tauchneck [43].

each method is applied separately, the final answer depends on the result of multiple methods. This thesis aims to investigate the ability of an MCS to automatically select the optimal set of classifiers for a given problem. This would allow the complete system to grow as newer and better classifiers are found. This way, the system may stay flexible enough to handle various datasets while staying computationally tractable.

The following chapters will introduce the basis of text recognition, focussing on off-line handwritten text recognition. Chapter 3 describes a flexible and distributable MCS implementation. Chapter 4 describes preliminary experiments which form the basis of three large experiments described in chapters 5, 6 and 7. Finally chapter 8 discusses the overall results and possible future research based on these experiments.

Chapter 2

Theory

In 1950 D.H. Shepard filed a patent titled "Apparatus for reading", describing a machine which would do optical character reading with the intention of connecting the output of the machine to a computer. Due to the flexibility of the computer, research today focusses on using solely the computer. But using a computer or not, optical character recognition can generally be split up into separate processing stages [32]. This chapter describes what these data processing stages are and how they allow for each stage to be implemented using a common interface. Using these common interfaces in turn allows for combining a set of different implementations to create a large array of different OCR approaches. These approaches, in turn, allows us to build a large parallel MCS.

2.1 Handwritten text recognition

Automated handwriting recognition can be split up into two major fields: on-line and off-line handwritten text recognition [31].

On-line recognition is performed on information captured on a special tablet, which not only records where the pen has been but also when [24]. This information allows the computer to trace the path and know which parts are connected and which ones are not.

Off-line handwritten text recognition is performed on images of written text, an input that lacks most of the information on how the shape has been created. Author specific traits [4] is an example of information which is still available. Not knowing which movement created the stroke makes off-line handwritten text recognition generally more difficult then on-line handwritten text recognition [37].

Generally off-line handwritten text recognition systems consist of three

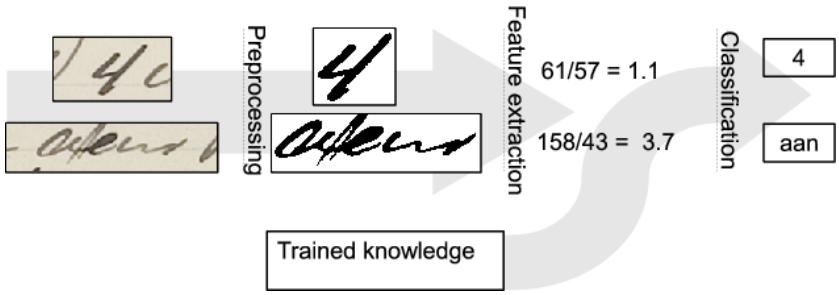


Figure 2.1: A schematic model of the data flow while classifying the number "4" and the Dutch word "aan". After preprocessing, which includes filtering and cropping the input image, the ratio of width and hight is used as a feature. This feature is combined with earlier trained knowledge to create a classification.

stages: preprocessing, feature extraction and classification. An example is given in figure 2.1, where these three stages are shown for handwritten input that denotes "4" and "aan". The intermediate results are also shown to illustrate the effect of the stage.

At the first stage, the input image is preprocessed. In the example this means: conversion from a grey-scale image to a black and white image, where only the ink of the text is left. After this all the white borders (non-ink) of the image are removed, also known as "cropping". This results in a cut-out of only the written text itself.

The second stage, called feature extraction, focusses on extracting only the important data from the image. In the example given in figure 2.1, the feature is the width-height ratio. For the example image of the number 4, this ratio is 1.1, for the image of the word "aan" this is 3.7. This stage is performed by the feature extractor and the resulting information is called the feature.

The final stage is the classification, where prior knowledge is needed. Generally this prior knowledge can be extracted from examples in a training phase. In our example, prior knowledge dictates that if the ratio is above 2, then it is the word "aan", otherwise it must be the number 4. Prior knowledge such as this is automatically extracted from the training examples. These training examples are presented to the classifier as feature-classification pairs during the training stage (not depicted in figure 2.1). The final stage is performed by the classifier and the result is called the classification.

Usually the difference between these steps is blurred into one system, where everything is said to be the result of the classifier. When designing

an MCS it makes more sense to keep these stages separated. The example portrayed in figure 2.1 and described above is trivial, but illustrates the basis of automated handwritten text classification: the computer extracts the information it considers useful from training examples, then when new (unclassified) instances are shown, it checks which class it resembles most.

2.2 Classifiers

2.2.1 k Nearest Neighbour

One of the simplest classification algorithms is k Nearest Neighbour (kNN) [12]. During training the kNN algorithm will store all training instances. Each training instance is interpreted as a vector in n dimensional space. During classification, the distance between all these vectors and the unclassified instance is calculated [40]. When using one nearest neighbour, the class of the training example with the minimal distance is considered the class of the unclassified instance.

kNN is a generalisation of one nearest neighbour (1NN). Instead of one, k closest neighbours are taken into account and the most prevalent class of these k nearest neighbours is considered the correct one. Increasing k will make the decision boundaries depend on more instances, making the relative density of a class within the region an important factor. Doing so may increase the performance.

To determine which instance is closest in the n dimensional space, a distance measure needs to be defined. This distance measure makes it possible to sort all training instances from closest to furthest, relative to our unclassified instance. The closest k determine which class the unclassified vector belongs to. Even though only a sorting is needed, generally a distance measure is applied according to $(\Re^n, \Re^n) \rightarrow \Re$, where the result is then used for sorting.

A commonly used measure is the Euclidean distance, equation 2.3, which is the square root of the sum of the squared per dimension difference. The Manhattan distance, equation 2.2, is the sum of the absolute difference per dimension. Both the Manhattan distance and the Euclidean distance can be expressed by a particular order of the Minkowski distance, equation 2.4. A Minkowski distance of order 2 equals to the Euclidean distance, while an order of 1 equals the Manhattan distance.

The Hamming distance, equation 2.1, determines the number of dimensions which have different value in both vectors. For example, the Hamming distance between the vectors $(1, 2, 3)$ and $(1, 2, 4)$ equals to 2 as the first two

dimensions have of these vectors contain the same value.

The distance does not have to refer to an actual distance, it may also be a similarity measure like the negative correlation, equation 2.5. The negative correlation of two vectors is then defined by the fraction between the negated fraction of the covariance of the two vectors and the product of their individual standard deviations.

$$\text{Hamming distance} \quad d(\mathbf{X},\mathbf{Y}) = \sum_{i=1}^{n} |x_i ! = y_i| \tag{2.1}$$

$$\text{Manhattan distance} \quad d(\mathbf{X},\mathbf{Y}) = \sum_{i=1}^{n} |x_i - y_i| \tag{2.2}$$

$$\text{Euclidean distance} \quad d(\mathbf{X},\mathbf{Y}) = \sum_{i=1}^{n} (x_i - y_i)^2 \tag{2.3}$$

$$\text{Minkowski distance of order } o \quad d(\mathbf{X},\mathbf{Y}) = \sqrt[o]{\sum_{i=1}^{n} (x_i - y_i)^o} \tag{2.4}$$

$$\text{Negative correlation} \quad d(\mathbf{X},\mathbf{Y}) = -\frac{E((\mathbf{X}-\mu_{\mathbf{X}})(\mathbf{Y}-\mu_{\mathbf{Y}}))}{\sigma_{\mathbf{X}}\sigma_{\mathbf{Y}}} \tag{2.5}$$

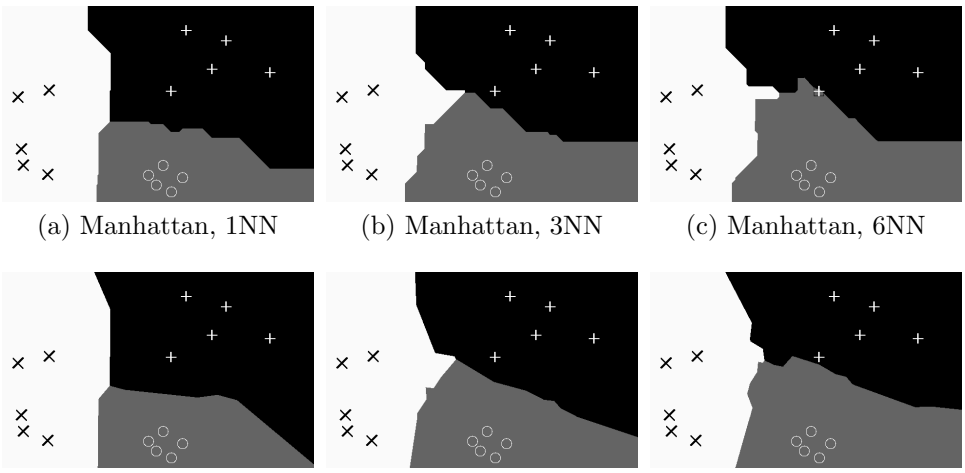
Figure 2.2: Examples of different kNN distance measures. Each equation defines a distance between two vectors X and Y.

The decision boundary created by kNN classification has a complexity which is defined by both the distance measure and the number of training examples. With more examples, it is possible to create a more complex decision boundary. Figure 2.3 shows a 2 dimensional, 3 class, classification problem solved using the kNN neighbour algorithm. The training examples are given in table A.1. At each training example a cross in the opposite colour is drawn, with a box in the class colour. Because the colour of the box is the same as the correct classification colour, it will not be visible if the classification at that point is correct.

Although it is a straightforward and robust method, it is often said that the kNN algorithm is not able to learn. Because plain kNN will store all training examples and use them during classification, the algorithm does not infer the distributions to generalise towards a simplified set of solutions. By not generalising over the data kNN does not really learn, it merely remembers. That said, the power and efficiency of the kNN algorithm [42] gives it mayor role in the field of handwritten text classification.

2.2.2 The support vector machine

Linear discriminants where first described by R. Fisher in 1936 [20] and later evolved into the Rosenblatt Perceptron [39]. The Rosenblatt Perceptron



(a) Manhattan, 1NN (b) Manhattan, 3NN (c) Manhattan, 6NN

(d) Euclidean, 1NN (e) Euclidean, 3NN (f) Euclidean, 6NN

Figure 2.3: kNN classification examples using a 3 class problem. Each point in this 2D space is classified by the kNN algorithm by assigning it the classes colour. The class colours for x, o and $+$ are silver, grey and black respectively. The differing boundaries of these images display the influence of using a different number of nearest neighbours and how these influences can differ depending on the distance measure used.

is a linear separator describing a hyper-plane which separates two different classes. In two dimensions, this hyper-plane describes a line, as shown in figure 2.4. The line is defined by an basis vector, b and a plane normal vector w.

To find the correct values for b and w, the Rosenblatt Perceptron algorithm applies a gradient descent on the training examples. The gradient descent minimised the error by working through the examples till either the total error has fallen below a threshold, or a maximum number of passes has been reached. There has to be an upper limit on the number of passes the algorithm is allowed to make, as the Rosenblatt Perceptron learning algorithm is not guaranteed to converge.

With the work of Vapnik [46], published in 1982, the basis of the support vector machine (SVM) was introduced. Using statistical learning and the work of A. Chervonenkis [3], the Vapnik Chervonenkis theory was introduced (VC theory). The VC theory minimises the error of a classification hyper-plane by maximising its margin. By rewriting the classification problem using the margin, it becomes a quadratic programming problem. The quadratic programming problem is a special type of mathematical optimisation problem for which there is a numerical approach available, making it possible to approximate a solution using a computer algorithm.

The margin of a classification solution is given by the shortest distance from the hyper-plane to a negative or positive training example, as shown in figure 2.4. A larger margin will mean that the classification vector w can be rotated more without changing the classification of any of the examples. This extra room for rotation, is what makes maximising the margin the same as minimising the error. The examples which determine the margin are called the support vectors.

This basic form will take care of linear classification, where a hyper-plane can solve the segmentation for the given input. However, it is not always clear that this is the case, for example when one class is enclosed by another class. To solve this problem, the input space can be transformed using a kernel function.

The kernel function maps the dot product of the input space vectors in feature space [14]. The feature space can be a high dimensional space where the input vectors are linearly separable [1]. When using a radial basis function, equation 2.5, an infinite dimensional space is used [11]. This is possible because the kernel function only depends on the definition of the dot product in feature space and not the actual transformation of the vectors into feature space. Using the right kernel function on non-linearly separable data will result in linearly separable data in feature space. Because it is linearly separable in feature space, margin optimisation can be performed in feature

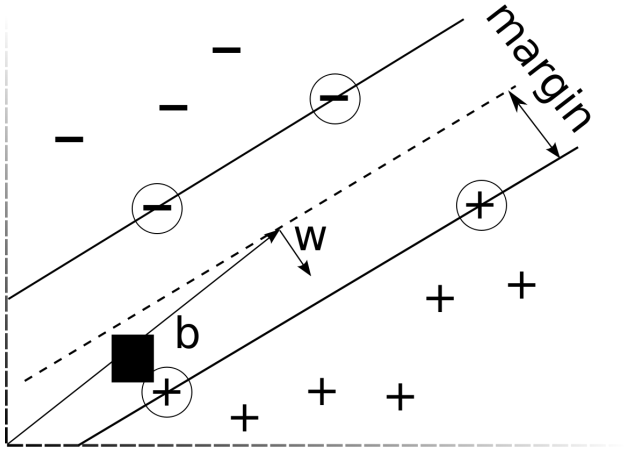


Figure 2.4: Example 2D classification plane. b denotes the basis vector, w the classification vector. The examples serving as support vectors have been circled.

space. Examples of kernel functions are given by the equations in figure 2.5.

Multi-class classification with SVM

The support vector machine is a binary classifier, only able to distinguish two different classes. There are various ways to have it distinguish between multiple classes. The most common technique is based on reducing a single multi-class problem into multiple binary problems. The simplest example is splitting the problem into sub problems which pit one class versus the rest or using special output codes [17]. More complex approaches include creating a decision tree [8].

It is also possible to move the multi-class problem into the support vector machine. In this approach, the classification problem is split up into multiple quadratic optimisation problems [13]. This results in a more correlated solution because the solution is not a collection of independent solutions, as is the case with a decision tree.

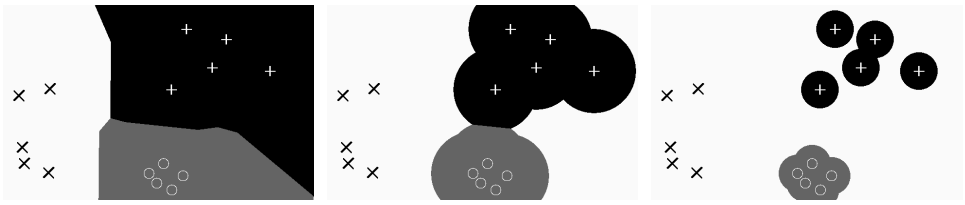
The multi-class support vector will still support all the different kernel functions described in figure 2.5 earlier.

2.2.3 Difference between kNN and SVM with radial basis function

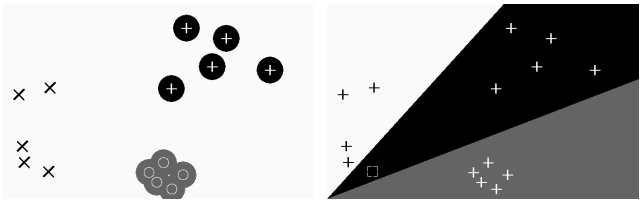
The previous section described the classification behaviour of both the kNN algorithm and the SVM. The effect of using different kernel functions on

$$
\begin{aligned}
\text{Linear:} \quad & k(x_i, x_j) = (x_i \cdot x_j + b) \\
\text{Polynomial:} \quad & k(x_i, x_j) = (x_i \cdot x_j + b)^d \\
\text{Radial basis:} \quad & k(x_i, x_j) = e^{-\gamma ||x_i - x_j||^2} \\
\text{Sigmoid (hyperbolic tangent):} \quad & k(x_i, x_j) = \tanh(x_i \cdot x_j + b)
\end{aligned}
$$

Figure 2.5: Kernel function examples, each kernel function defines the dot product between two feature vectors for the associated feature space. The characteristics of the feature space that is implied by these kernel functions influence the separability of the dataset in feature space.

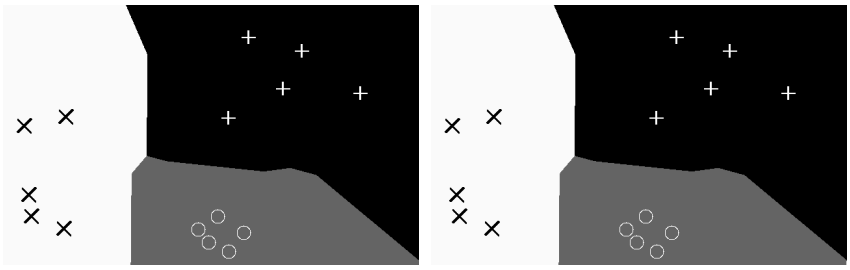


(a) Radial kernel, $\gamma = 0.01$ (b) Radial kernel, $\gamma = 0.1$ (c) Radial kernel, $\gamma = 0.5$



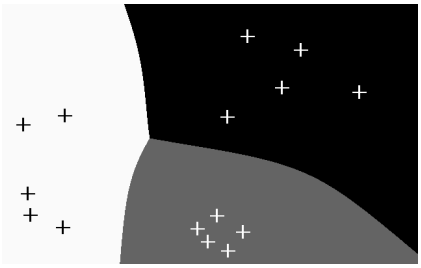
(d) Radial kernel, $\gamma = 1$ (e) Linear kernel

Figure 2.6: Multiple class SVM classification examples. Each point in this 2D example space is classified using a different kernel function configuration. The class colours for x, o and $+$ are silver, grey and black respectively. These examples show both the influence of the different kernel functions on classification boundaries and the influence of γ for the radial basis function.



(a) 1NN with euclidean distance

(b) Radial kernel, $\gamma = 0.01$



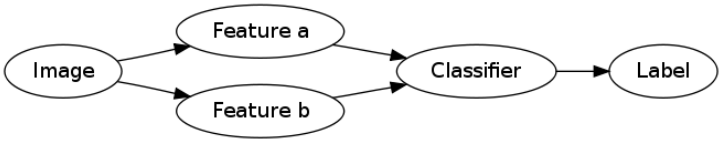
(c) Radial kernel, $\gamma = 0.0001$

Figure 2.7: The difference between a multi-class support vector machine with radial kernel and 1NN. Each point in this 2D space is classified using the algorithm mentioned below, assigning it the classes colour. The class colours for x, o and $+$ are silver, grey and black respectively.

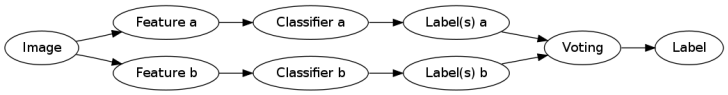
the classification has been shown by plotting the classification boundaries produced for a set of examples vectors. Comparing the classification results of 1NN using euclidean distance and a support vector machine with a radial kernel function, a resemblance becomes apparent.

Figures 2.7a and 2.7b show examples where the difference can hardly be noticed: 1NN euclidean distance and the support vector machine with a radial basis function using $\gamma = 0.01$. However, as soon as γ is decreased, it can be seen why the SVM is said to learn, while the NN algorithm does not. Figure 2.7c shows the radial basis function support vector machine with $\gamma = 0.0001$. As γ decreases, the SVM is forced to generalise further, making for smoother boundaries. The kNN algorithm allows us to do this for small datasets. This shows that the SVM algorithm will generalise, while the complexity of the decision boundary of the kNN algorithm is not influenced. For the NN algorithm, the boundary complexity is directly linked to the distribution and number of the examples near the boundary.

(a) Serialising features



(b) Merging features



(c) Merging labels

Figure 2.8: Examples of possible feature/classifier interconnections. Each connection type allows for a different MCS to be constructed.

2.3 Multiple classifier systems

All of the stages of handwritten text recognition described section 2.1 produce information. For most stages this information can be combined before it is transferred to the next stage. Using the possibility to combine and interchange elements, a large array of possibilities is created. For example, features can take either an image, or the output from another feature, serialising as shown in figure 2.8a. Multiple features can also be merged into one feature vector, as show in 2.8b. Finally multiple classifiers can be combined by voting 2.8c, where each classifier conveys its choice or a ranked list of choices. All of these three variations will be applied during this thesis, with the exception that feature a in the serialisation example, figure 2.8a, will be implemented as a preprocessing step. During the MCS construction in this thesis the following restrictions will apply: features can only accept feature vectors or images and classifiers accept feature vectors and output label(s).

The basic incentive for a system of networked features and classifiers is a possible gain in performance [22] [16]. Other benefits often considered are generalisation [19] [36], more robustness [25] and flexibility [25].

The performance of an MCS depends on its structure and which parts are used. Automatically creating combinations is the main purpose of algorithms like Boosting [18], AdaBoost [23] which is a popular variant of Boosting, Bagging [7] and Random Forests^{TM}[41]. Boosting creates a weighed combination of classifier based on the training data. Bagging, short for

bootstrap aggregating, will automatically reduce classifier over-fitting and Random Forests™will construct a decision tree out of multiple classifiers to increase classification performance. All of these methods lack the ability to dynamically change the set of methods used based on the input instance. Further more, all these methods assume a single classification task will be handled by the constructed system by increasing the performance for a single dataset.

Chapter 3

Implementation

3.1 Introduction

An MCS is a system with a flexible configuration of multiple modules: classifiers, features and datasets. Even though all modules can be seen as separate, they will eventually have to be combined into a single system. When a simple task as transforming a dataset is required multiple times, it becomes even more difficult to keep track of the data to ensure you are not doing the same calculation multiple times. As these calculations can take a lot of time, caching becomes a vital part or the MCS. The MCS framework should facilitate keeping track of results and ensure caching is properly handled.

At the time of writing there was no published MCS framework which will facilitate transparent caching and is accessible enough to wrap the various classifiers that have previously been implemented. This implies that a new framework will have to be implemented.

In its daily use an MCS framework should be able to cache data, share data caches between multiple machines, be platform independent and it should be portable enough for a scientist to run on their home computer. The ideal system would allow for scientists to take home a subset of their calculation, test new implementations and later transport their cache to their work computer or the computer of a colleague.

This led us to the following design requirements:

- Portable for various systems
- Calculation without the need for communication
- A distributable system handling
- Transparent caching

- Able to wrap previous implementations of classifiers, features and datasets
- Allow for generalised tasks to be performed easily
- Run multiple experiments on the same machine/file system
- Allow for quick development and deployment of an experiment

These requirements where met by creating the following three elements: a packaging system, a naming service, a set of standard fulfilment functions.

3.2 The packaging system

If a good cooperation between scientists is to be made possible, it is essential that modular parts of the framework are easily portable. This includes the modules, datasets and features. Each of these modules require configuration. To ensure that a configured module is transportable, it must be able to save its complete state to disk.

Once the module is on disk, it is easy to copy it to other places by packaging it up into an archive. Before this, the system must ensure that the state of the module is written to disk and that the module is correctly closed.

The framework ensures this through the packaging system. The packaging system is a class which handles the loading of all the packages and the closure of classes. To allow this to work, each package has to adhere to a very basic API.

Because the complete module state is stored on disk, opening a module with new set of parameters could change the state on disk. To ensure that default packages are not changed, the packaging system uses a working copy before opening and manipulating the modules. The packaging system needs to be initiated with both a working directory and a package directory. The use of the package directory will be described later.

The working directory contains the most recent copies of all opened modules. Opening a module therefore consist of the following steps:

- If the module is already open, an exception is raised.
- If the module is found in the working directory it is opened there.
- If the module is located in the path where the system was invoked, it will be copied to the configured working directory.

Every module is contained within a single directory. This directory contains, at least, a Python initialisation script which allows the packaging system to instantiate a class. The instance of the class is used as the module handle and as long as it exists, the module is considered to be open. To keep track of whether a module is open or not, the packaging system keeps a weak pointer[1] hash table of all open modules. These weak pointers allow it to check when the package is closed and is used to ensure only a single handle for every module is open.

By deleting a module's handle class it is closed, which will automatically sync everything to disk because it will destroy the class. It will not move the directory, so any opened and later closed class is still in the working directory.

For large datasets, the copy operation may seem cumbersome because it would require copying all the data. However, if the data is not altered, a POSIX [5] symbolic link[2] can be used to ensure there is only one global copy of the data. Because there is no restriction on the data handle implementation, it is also perfectly plausible to have the data not come from the directory but from a database server or any other on-line service.

After closing the module, its contents is still in the working directory, which is unique for the process as it must ensure the state on disk is correctly set. If a created module should be shared between multiple experiments or scientists, for example a converted dataset, the packaging system allows the scientist to close a module, archive its directory and then copy it into the package directory. The package directory is therefore a repository of previously calculated data and can be used by other scientists to gather cached data or specially transformed sets. This allows a scientist to publish a transformed dataset without the actual feature, making it possible for other scientists to verify classifier performance on that feature without obtaining its implementation.

Listing 3.1 shows an example of using a module via the packaging system. The example starts by setting up the packaging system with the right working directory and the right package directory. The configured package directory is not used in the example, as any package that is requested is looked up in the process working directory if it is not found in the configured working

[1] A weak-pointer is a shared pointer that does not imply ownership and will not be counted as a reference. The weak-pointer is therefore only valid as long as the underlying object has not been marked for garbage collection. The validity of the weak pointer makes it possible to check if other parts of the program are still using the object or not.

[2] A POSIX symbolic link is the file system equivalent of a weak pointer, normal access will be redirected to the file or directory it points to. If that underlying directory does not exists it is considered broken and will fail to open.

directory. The current process working directory is therefore effectively read-only, which is required for the experiments to be repeatable and ensures it is kept clean of any intermediate results.

Listing 3.1: An example of using the packaging system in Python

```
p = Packages(workingDirectory="/tmp/experiment_directory",
             packageDirectory="/home/scientists/packages")
#Load the mnist dataset
mnist = p.load('mnist')
#Print all instances on screen
for i in mnist:
  print i
del mnist #Close the dataset again, optional
```

The deletion of the module handle, `mnist` in listing 3.1, is not a necessary step if the module is not directly needed further along in the code. If this code would be part of a function, Python scoping ensures that the destructor is called on the handle, resulting in the same behaviour.

Listing 3.2 shows how a set can be transformed using a feature. Like the previous example, the packaging system is first initialised. Then a module is loaded using the packaging system. If the output set is already available, the load will open the output set and return its handle class, if not, load will return `None`. This behaviour makes it possible to check the existence of the dataset on disk, before the calculation.

Once it has been established that the output dataset is not available yet, the feature and the input dataset are opened. Instead of using the `load` function here, the `require` function is used. The `require` function will raise an exception if the module is not available. This ensures the module is available on the system and successfully opened after calling `require`.

With both the feature and the input set loaded, the output dataset is created. The generic `cpickle` set is used, which is a dataset using Python cPickle as a storage back-end. After successfully opening that set, it is saved under a different name and then closed. By saving it under a different name, any changes made to the set will not influence our original, empty, `cpickle` dataset. Again, closing the set using `del` is optional.

Finally all the modules are ready and the `convert` member of the feature handle is called to convert the input set into the output set using the feature. It is now left up to the end of scope to destroy and close all remaining modules.

Listing 3.2: An example of converting a dataset using a feature

```
p = Packages(workingDirectory="/tmp/experiment_directory",
             packageDirectory="/home/scientists/packages")

outSetName = 'the_output_set'
outSet = p.load(outSetName)
if outSet:
  #The output set already exists
  return 0

#Load feature and input set
feature = p.require(featureName)
inSet = p.require(setName)

#Create empty output set
emptySet = p.require('set/cpickle')
outSet = p.save(emptySet, outSetName)
del emptySet

feature.convert(inSet, outSet)
```

Because each module applies a special application programming interface the usage of the feature and the datasets is completely independent of the underlying implementation. The approach used in listing 3.2 is therefore also independent of the underlying implementation. When the packages are archived into a single file, they can easily be copied and accessed on other systems. This means that the packages are now portable across different systems. Because the portability does not rely on any of the running systems being connected during the packaging, the calculations can be performed without the need for communication.

Each intermediate result can be cached and distributed by storing a packaged version into a global repository of module output. There are three different mechanisms in place to protect these packaged modules from becoming corrupt. Firstly, the gzip compression used on the archive adds a validation checksum, ensuring the data is not corrupted in transit. Secondly, there is the option to taint a package. As soon as a package is tainted, it can no longer be loaded by the packaging system. This ensures that a package is in good condition when it is archived and in good condition if it is successfully loaded by the packaging system.

3.3 The naming service

The previous section described the packaging system which can load and save modules by name. By using the packaging system the on disk state of the module is protected from from multiple access. To ensure that the content of the package is reflected by its name, a standardised naming system for the packages is needed. Using this naming system will give the package names a coherent meaning, protects against naming collisions and allows one to predict the names of future results. This last feature is what makes the naming service essential for caching.

For each of the common operations performed on sets and packages, a naming convention is defined. Common operations used are:

parameterise If a package is loaded, it can be given parameters. These parameters are defining for future uses of the package, as they will be synced to disk as soon as the package is closed. This means that the parameters used are reflected in the name. For this, the parameters are sorted by name and placed as a comma separated list after the package's directory name.
As an example, loading the module `something` with parameters $a = 1$ and $b = 2$ results in the name `something,a=1,b=2`

converted set The name of a set that has been converted using a given feature. The resulting set will be placed in the `set` directory and has the name '`set name—feature name`'.

k-fold test results When a k-fold test has been run on a trained classifier using k different folds, the set is placed in the `set` directory and has the name `classifier name_`kfoldResults. An example is: `set/classifier_10foldResults`.

trained classifier When a classifiers is trained, its state on disk will change and therefore it is given a new name. After training the classifier with a given set, the classifier is named 'classifier/`classifier name—set name`'.

stripped set After testing a classifier on a set, the output results contains both the labels, the feature vectors and any other meta-data which has been added. This means that every test results in a set which is as large as its testing set with an added label entry. To ensure not all extra data is constantly copied to the test results, it is wise to strip the results of their original vector, leaving only the labelling and its correct

label. The stripped set is placed in the `set` directory and has the name `set name_stripped`.

test results The results of testing a classifier on a test set is stored under the `set` directory and has the name `classifier name?test set name`. If the set needs to be stripped, it has to be performed after the results have been calculated.

k-fold trained classifier Before doing a test on a classifier as part of the k-fold algorithm, it needs to be trained on everything but the testing fold. Because the internal state of the classifier changes, this classifier is copied to a separate module before hand. The classifier is stored in the `classifier` directory and has the name `classifier name!test fold name`.

shuffled set To properly test a set, the order must be shuffled. If the `seed` argument is defined, the random number generator will be seeded with that value. Using the seed makes the shuffle repeatable. The name of the shuffled set therefore includes the seed value and is '`set name_shuffledseed value`'. The shuffled set is stored in the `set` directory.

folded set When a set is split up into folds, each fold has a special name to signify that it is part of a given folded set. Each fold is stored as a set in the `set` directory and is named '`set name_foldfold iofk of k-folds`', where `fold i` is the index number of the fold and `k of k-folds` is the number of folds that are created.

merged set When two or more sets are merged together into one set, typically by concatenating the feature vectors, the combined set is stored in the `set` directory and given the name which results from concatenating the sorted list of set names using a colon (:).

The above naming rules are used for any operation on the basic feature, classifier and set modules and will ensure that the predictable nature of the names allows for transparent caching.

3.4 Standard task collection

In the section on the packaging system, listing 3.2 showed us how to load and convert a dataset using a feature. If this code is would use the naming service, the `outSetName` could be determined automatically based on the

input set name and the feature name. This allows the conversion to be performed knowing only the name of the feature and the name of the input set. All tasks for which the naming system described earlier defines a name, are automated into a standard task collection in a class called `Virtual`.

Because the naming service can determine the output name before it is generated, it also allows for transparent caching to be implemented. For this transparent caching, a rewrite of listing 3.2 into the more general form is shown in listing 3.3. This listing defines the `convertSet` member of the `Virtual` class, which takes the input set name and the feature name and then runs the conversion. By checking the existence of the output set, transparent caching is implemented which keeps our results from being calculated twice.

For added security, the general functions also perform some validity checks. The `convertSet` function, for instance, checks the number of instances in the output set. Because a feature, generally, should not change the number of instances in a set, this function will taint the output set if the number of instances is not equal to the number of instances in the input set. Once a module has been tainted, the packaging system will refuse to use it any longer making it impossible to be loaded or packaged. Tainting can be performed by using the `taint` member function of the package system and can be performed on both open and non-open modules. Because some experiments may use a feature to sift out bad example, the loss of instances in a set may not always be considered a problem. Therefore tainting is introduced as a separate function which allows the scientist to decide when data can no longer be trusted even if it is still readable.

Listing 3.3: An example of converting a dataset using a feature

```
def convertSet(self, setName, featureName):
  outSetName = self.naming.convertedSet(setName,
     featureName)
  outSet = self.packages.load(outSetName)
  if outSet:
    #The output set already exists, return cached
       instance
    return outSetName

  #Load feature and input set
  feature = self.packages.require(featureName)
  inSet = self.packages.require(setName)

  #Create an empty output set
  cp = self.packages.require('set/cpickle')
```

```
outSet = self.packages.save(cp, outSetName)
del cp

feature.convert(inSet, outSet)
#Close all modules
del inSet, outSet, feature

#Check set sizes the setSize member function
#will load the module and count the number of
    instances
inSize = self.setSize(setName)
outSize = self.setSize(outSetName)
if not inSize == outSize:
  self.packages.taint(outSetName)
  raise Exception('Converting resulted in fewer
     instances')
return outSetName
```

As can be seen in listing 3.3, all standard tasks will use names of sets for both input and output. This ensures the standard tasks will not leave any modules open, because at the end of the function all module handles in the scope will be destroyed.

3.5 Distributed parallel calculations and scalability

The system in itself is meant to be usable on both normal workstations and distributable over multiple workstations. For example, one workstation could convert a set for one feature, while the other does so for a second feature. This distribution of tasks does not require any communication and is often referred to as embarrassingly parallel computing [45].

Figure 3.1 shows how the data flow between multiple running experiments is shared through the package directory. These packages are protected by both the taint feature and corruption detection with checksums. This ensures that any data which is imported into the a working system from another system is always valid. The data flow between the working directory and the current process working directory is read only. Because a running experiment will copy any data to its working directory, the scientist is free to edit any modules already loaded by the experiment.

The package directory is presented by a networked directory in figure 3.1,

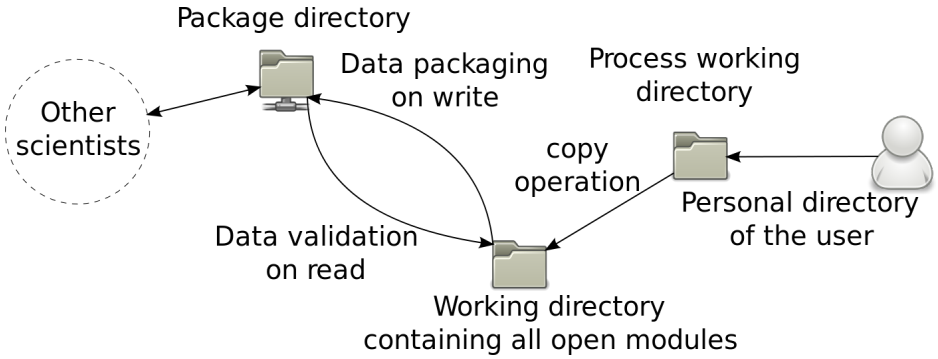


Figure 3.1: The data flow of multiple processes and/or scientists. The main elements are the package directory on the left, the working directory in the middle and the process working directory on the right. The tests created by the scientist are started in the process working directory and use the working directory as their on disk cache of all its information.

however there is no requirement for this. Because non of the directories are required to be network connected, a scientist is free to run the calculations anywhere and later upload or email the result packages by hand.

The naming service described in section 3.3 ensures the package names are valid across multiple experiments, making it possible for various experiments to share the same packaging directory even if they only share a single dataset transformation like shuffling a dataset. The data validation when a package is loaded ensures that a package has not been corrupted in transit. This validation also detects any concurrency problems that may occur when multiple processes are writing the same file, or when a slow writing operation results in only half a package being read by the experiment.

Chapter 4

Prediction by ten-fold test

In a multiple classifiers system, all classifiers play a role and therefore influence the overall error-rate. To decrease the overall error-rate of the MCS, only the best classification methods, a combination of classifier and feature, should be used. To test which methods are the best, a ten-fold test on the training data is performed. This should result in a good estimate on which classifiers to eventually use on the testing set. If using a ten-fold test is enough to determine which classifiers are the best, then using the best five classification methods may create a well performing MCS. By comparing the best five classifiers found in a ten-fold test with the best five according to a complete test, the validity of the approach is tested.

4.1 Method

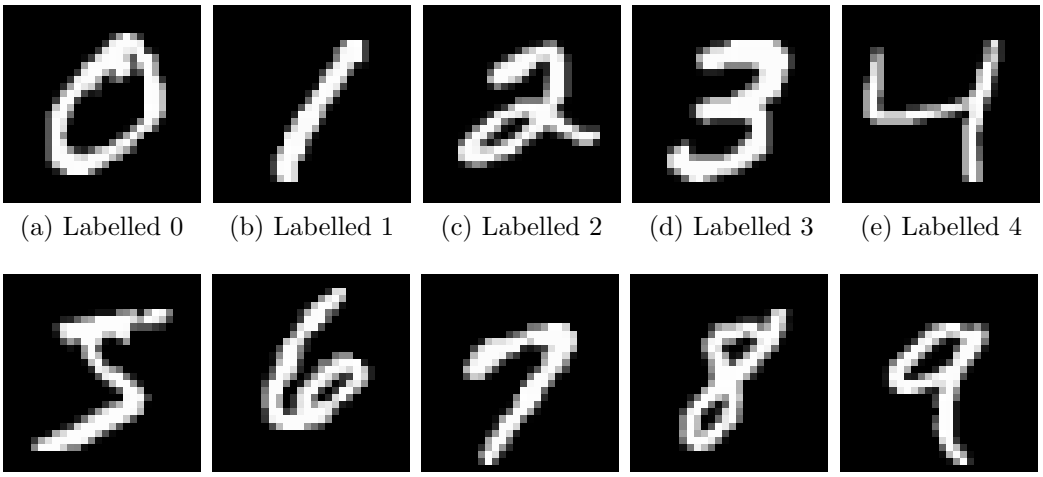
Using 12 configurations of the kNN classifier and a multi-class SVM as described in subsection 2.2.2 with linear and kernel functions.

The 12 kNN configurations are created using three distance measures: euclidean, hamming and negative correlation. All with four possible values of k: 1, 3, 6, 20, creating a varied selection of both distance measures and values of k.

The multi-class SVM implementation is based on SVM_struct, described in 4.1.3, with either a radial basis function with $\gamma = 0.001$ or a linear kernel.

Each of these classifiers was combined with the features described in section 4.1.2, each with various parameter settings.

Then a ten-fold test on the training set using all methods is performed to create an ordering. After this the methods are trained on the training set and tested on the testing set to create a second ordering. If these are comparable, then a ten-fold test on the training set should be enough to generate a good



(a) Labelled 0 (b) Labelled 1 (c) Labelled 2 (d) Labelled 3 (e) Labelled 4

(f) Labelled 5 (g) Labelled 6 (h) Labelled 7 (i) Labelled 8 (j) Labelled 9

Figure 4.1: MNIST instance examples as used in this thesis.

democratic MCS classifier from a sub selection of these best methods. This would make more complex selection and ordering criteria superfluous.

4.1.1 The MNIST dataset

The mnist dataset is maintained by Yann LeCun and Corinna Cortes [30]. It is a freely available set of 70000 images of segmented handwritten digits, of which 60000 training examples and 10000 testing instances. The dataset has been created by combining two NIST datasets, Special Database 3 [35] and Special Database 1 [34].

NIST Special Database 3 are handwritten digits by Census Bureau employees, NIST Special Database 1 are digits written by high-school students. These databases have been equally divided over the test and training sets. The handwritten digits have been segmented and their pixel based centre of mass has been placed at the centre of a 28 by 28 pixel sized 256 level grey-scale image. Figure 4.1 shows a random example of every class picked from the training set.

The MNIST dataset was first introduced by LeCun in 1998 [29]. One of the classification methods described in that paper had an error rate of 0.8%. In 2006 M.A. Ranzato, et. al. [33] showed a minimum error rate of 0.39%, which means that of the 10000 test images, only 39 images where wrongly classified.

4.1.2 The features

An image contains both relevant and irrelevant information. For example, if the goal is to determine which letter of the alphabet is displayed, the colour is not relevant information. Features attempt to extract as much relevant information, making it easier for a classifier to use it. The feature may transform the information, which may also help in classification. As described in section 2.2.2, the support vector machine relies on kernel functions to define its feature space where the data is, hopefully, more separable space.

For this implementation a large array of features have been designed, each with optional configuration parameters. Whether these features will perform well or not depends heavily on the dataset, which is why each feature is designed to be as general and flexible as possible.

In the following sections, all the features are described including their configuration parameters.

Angle

The angle feature determines the angle between the most prominent value peaks on a circle centred on each pixel. For every pixel it takes n evenly divided points on a circle, with radius r, centred on that pixel. Using the derivative of the sequence of n values, the position of the local maxima are determined. On these local maxima, connected components are replaced by a single value, setting everything else to zero. Between the top most two values left over, the minimal angle is calculated. This results in an angular value for every pixel in the image.

The value of any point which falls outside of the image, is considered to be the same as the value of the closest image-border pixel.

Optionally it is possible to calculate the histogram of encountered angles, using the `histogram` parameter described below.

The optional parameters are:

r The radius of the circle used (defaults to $r = 4$)

n The number of samples to take from the circle (defaults to $n = 4$)

histogram Whether or not to reduce the values to a histogram (defaults to $n =$ False)

Blur

The blur feature applies a blurring convolution kernel to the image using a 5 by 5 kernel as shown in table 4.1. Applying this kernel multiple times will

1	1	1	1	1
1	0	0	0	1
1	0	0	0	1
1	0	0	0	1
1	0	0	0	1
1	1	1	1	1

Table 4.1: Convolution kernel used by the blur feature

Class	Kernel(s)							
1	0	1	0	0	1	1	1	0
	1	1	0	1	1	0	0	0
2	0	0	0	1	1	1	1	0
	1	0	0	0	0	1	1	1
3	0	0	1	1				
	1	1	0	0				
4	0	1	1	0				
	0	1	1	0				

Table 4.2: Different contour classes

increase the amount of blurring.

The optional parameters are:

n The number of times to apply blur to the image (defaults to $n = 1$)

Contour direction

From [26], this features uses a 2 by 2 window on a binary version of the image. The windows are matched against 4 classes defined in table 4.2. The resulting $n - 1$ by $n - 1$ valued vector can then, optionally, be reduced to a histogram of the 4 classes by setting the `histogram` parameter (defaults to $histogram = \text{False}$).

Decomponent

The decomponent feature is based on connected components. Using pixel values it labels all connected components and removes any component that is not mentioned in its configuration. It leaves the largest component by default, except when you use the `skip` parameter (leading zeros of the bit-mask are ignored). If the shape is a single connected component, then using

decomponent with its default value will effectively remove all speckles from the image. Because all connected components that are detected are of the exact same colour value, it is wise to first reduce the number of colours used in the image.

Optional parameters are:

n The decimal value of a bit-mask used to select the top components (defaults to $n = 1$). All leading zeros of the bit-mask are ignored.

skip Skip the first number of components. This allows you to remove large components (defaults to $skip = 0$).

Enclosed

Enclosed will only leave the enclosed parts of the image using the ink colour. It uses the maximum value of the image as the ink colour, which means that images will need to be inverted if they use low values for ink. The image is then flood-filled from the edge pixels using the ink colour, after that the image is inverted. If the ink in the image encloses any part of the image, that part will be emphasised by the invert.

Any binary image without an enclosed region will become a solid colour. Using the enclosed feature on a binary image of a closed number 9, will leave the top circle of the nine as a solid blob.

This feature has no parameters.

Fnumpy

This feature applies one of the various supported vector transformations implemented in the Numeric Python [15] mathematical library. The following transformations are available:

sin Take the sine of every dimension.

cos Take the cosine of every dimension.

tan Take the tangent of every dimension.

diff Take the difference between the sequential dimensions. This results in a vector which is one element smaller then the source.

fft Apply a n-point, one-dimensional, discrete Fourier Transform on the vector where n is the length of the vector.

This feature supports one optional parameters called `op`. This parameter is used to describe which of the above transformations is applied by the feature(defaults to fft).

Hingefeat

The Hinge feature was introduced by M. Bulacu and L. Schomaker [9]. The hingefeat feature extracts a histogram of Hinge feature values.

The minimum hinge angle and leg length can be set. The feature vector is a histogram of all the angles found. The implementation was written in C by A. Brink, who is a Ph.D. student of Artificial Intelligence at the University of Groningen.

Optional parameters are:

leg (defaults to $leg = 4$)

angles (defaults to $angles = 32$)

Histograms

The histograms feature returns a value histogram of nc bins. The value histogram is an integer count of the number of times a bin value has been used.

Optional parameters are:

normalize If set this will normalize the resulting vector (defaults to $normalize =$ False)

nc Set the total number of bins (defaults to $nc = 0$). If this is zero, then the number of bins is automatically configured to: two if exactly two colours were found, 256 otherwise

Nccomponents

The nccomponents feature returns the number of connected components per horizontal scan-lines, vertical scan-lines, or both. The direction can be configured, if both are used, the resulting vector is a concatenation of the horizontal component count followed by the vertical. Only pixels with equal value are considered connected, which means that a gradient will produce a high number of connected components.

Optional parameters are:

direction Which direction to check for components, **h**orizontally, **v**ertically, or both. (defaults to $direction =$ hv)

normalize If true, the resulting vector will be normalised (defaults to $normalize =$ False)

Penetration depth

The penetration depth feature calculates the distance from the edge to the first different pixel. It can be configured to do this for the left, right, top, bottom, or any combination of sides.

Optional parameters are:

crop Whether to crop the result by making minimal penetration depth per side equal to 0 (defaults to *crop* = False)

sides Which sides to check, `l` for left, `r` for right, `t` for top and `b` for bottom (defaults to *sides* = lr)

normalize If set to true, the resulting vector will be normalised (defaults to *normalize* = False)

Pie

The pie feature splits the image into radial bins (or slices) relative to its centre of mass. The centre of mass is determined using the value of each pixel. For each bin the sum of pixel values falling in it is returned. Each pixel is assigned a bin using its angle to the centre of mass. To make this feature rotation independent it is possible to rotate the maximum value to the first position by setting *rotate* = False.

Optional parameters are:

nslices The number of slices to use (defaults to *nslices* = 100)

normalize If set to true, the resulting vector will be normalised (defaults to *normalize* = False)

rotate If set to true, the vector is rotated so that its maximum value is placed first in the vector (defaults to *rotate* = False)

Projection

Scale the image to a 20 by 20 pixels image and accumulate row and column values into vectors of 20 values. These summation vectors are normalised by default, to ensure the maximum and minimum values are between zero and one. If the summary parameter is set, the vector is extended with the maximum values in the row sums and column sums, followed by the minimum values of the row and column sums.

Optional parameters are:

normalize If set to true, the resulting vector will be normalised (defaults to *normalize* = True)

summary If set to true, a summary vector is concatenated to the vector (defaults to *normalize* = False)

Size

Various representations of the size of the image. Represented by a vector of four values: width in pixels, height in pixels, surface and the aspect ratio of the image.

Slant

For every pixel the tangent of the x difference and y difference of its closest neighbours is calculated. For a pixel at (x, y) the slant is given by equation 4.1. The edges of the image are ignored as they do not have well defined neighbours. The resulting vector is therefore 2 pixels less in width and in height.

$$slant(x, y) = \arctan\left(\frac{V(x, y-1) - V(x, y+1)}{V(x+1, y) - V(x-1, y)}\right) \tag{4.1}$$

If the optional parameter *histsize* is set, then a scaled histogram of *histsize* bins is created instead, containing the number of times each angle was found. The histogram bins will start at the smallest angle and end at the largest, during scaling the values can be divided over multiple bins, resulting in floating point values.

Speckles

A speckle is a pixel which has only other valued neighbours. The neighbours are defined by the 8 connected pixels next to the target (`e`).

For a matrix of pixels:

```
a b c
d e f
g h i
```

The pixel at e is considered a speckle if neither one of a, b, c, d, f, g, h, i is equal to e.

The feature vector is a single value representing the number of speckles, optionally divided by the surface half the surface if the `normalize` parameter is set to True (defaults to *normalize* = False).

4.1.3 The classifiers

Three classifiers for our implementation. These classifiers where chosen for both performance and to show the MCS implementation is general enough to handle these different sub programs.

kNN For the nearest neighbour implementation the `standclass` program is used. This program was developed at the faculty of Artificial Intelligence in Groningen by L. Schomaker.

SVM For the SVM implementation the SVMLight [28] program is used. It is a highly optimised SVM classifier written in C by T. Joachims. The version used in this thesis is 6.02.

Multi-class SVM The support vector machine described in chapter 2.2.2 has been implemented in the SVM_struct classifier [44]. The implementation used during this thesis was written in C by T. Joachims, who is also the author and maintainer for the SVMLight classifier. Version 3.10 is used.

4.2 Results

Not all feature classifier combinations could be run because of time constraints. From a large list of possible feature configurations, a subset of 57 could be completed within the limits of time and memory. This included 13 different configurations of the two previously introduced classifiers. Due to machine memory constraints, not all multi-class SVM results where calculable and thus not all tests on the multi-class SVM could be completed. Because more then half of all the radial basis function SVMs failed, they where left out completely. These two constraints led to 556 results out of the a theoretical set of 741 combinations.

Table 4.3 shows the error-rate for all the combinations with a successfully completed ten-fold test. This table shows that the lowest error-rate for the ten-fold evaluation results from using the original image as a feature vector and a kNN classifier. The best five classifiers using these results are shown in table 4.4.

Training the 741 classifier units (classifier/feature combinations) on all training data, allows for a selection of the best five classifiers for a complete test, shown in table 4.5.

Comparing the two tables, it is clear that none of the top five classifiers from the ten-fold test, show up in the actual top five. This leads us to

conclude that a straight-forward ten-fold test is not enough to create a sub-selection of classifiers for an MCS.

4.3 Conclusion

The results shows that the kfold-test does not find the best five methods accurately enough as these two best five had no feature in common. To be able to only use the best classifiers when doing a complete test, a better solution to classifier selection will have to be found. Using only the error-rate of the classification methods found with a ten-fold test will not result in a good MCS, while using all methods will make testing the MCS an impractically computationally intensive task.

mnist-train-60000_shuffled5263020	kNN euclid nn=1	kNN euclid nn=20	kNN euclid nn=3	kNN euclid nn=6	kNN hamm nn=1	kNN hamm nn=20	kNN hamm nn=3	kNN hamm nn=6	kNN -cor nn=1	kNN -cor nn=20	kNN -cor nn=3	kNN -cor nn=6	mSVM
angle n:10 r:4	5.11	6.70	5.01	5.42	4.02	5.53	4.08	4.52	4.87	6.04	4.64	5.00	?
angle n:10 r:6	4.91	5.51	4.58	4.70	4.06	4.94	3.91	4.11	4.74	5.33	4.47	4.64	?
angle n:10 r:8	5.96	6.96	5.73	6.11	4.65	5.71	4.52	4.93	5.64	6.46	5.46	5.71	?
angle n:100 r:4	11.94	14.39	11.37	11.91	7.70	9.31	7.23	7.67	8.27	7.80	7.18	6.89	?
angle n:100 r:6	11.86	12.18	10.89	10.77	7.34	8.06	6.93	7.00	8.99	8.35	8.03	7.61	?
angle n:100 r:8	12.84	13.37	12.11	12.02	8.14	8.47	7.51	7.67	10.56	9.84	9.65	9.31	?
angle n:50 r:4	11.52	14.13	11.01	11.55	7.09	8.44	6.72	6.94	8.20	8.10	7.32	7.21	?
angle n:50 r:6	9.78	10.60	9.11	9.26	5.96	6.41	5.43	5.61	7.86	7.42	6.95	6.92	?
angle n:50 r:8	10.14*	10.37	9.37	9.32	6.72	7.03	6.14	6.22	9.34	8.93	8.46	8.31	?
angle hist n:10 r:4	49.65	40.79	46.69	43.21	49.70	40.74	46.91	43.36	49.77	40.89	46.93	43.60	55.07
angle hist n:10 r:6	46.97	38.49	44.35	40.69	46.84	38.35	44.25*	40.57	47.13	39.31	44.78	41.16	55.27
angle hist n:10 r:8	49.48	40.96	46.69	43.32	49.56	40.66	46.76	43.34	50.28	41.91	47.37	44.20	53.90
angle hist n:100 r:4	48.37	39.92	45.90	42.15	40.30	30.64	36.76	32.85	49.70	42.48	47.13	44.17	38.52
angle hist n:100 r:6	47.14	39.89	45.43	41.73	38.20	29.97	35.65	32.07	50.08	43.84	48.38	45.27	41.12
angle hist n:100 r:8	44.10	37.66	42.32	39.31	37.15	29.04	33.23	30.17	46.49	42.49	45.72	43.18	42.08
angle hist n:50 r:4	49.64	40.84	47.40	43.35	44.48	34.43	41.03	36.93	51.62	43.83	49.13	45.93	43.97
angle hist n:50 r:6	41.47	33.81	39.07	35.64	36.29	27.96	33.57	30.09	43.49	36.37	41.09	37.78	41.68
angle hist n:50 r:8	39.79	33.69	38.08	35.28	34.94	28.32	32.73	29.90	40.99	35.30	39.03	36.49	46.21
blur n:1	3.36	4.54	3.23	3.63	3.54	4.73	3.36	3.82	3.10	4.10	3.00	3.29	9.07
blur n:2	5.66	6.73	5.37	5.74	5.65	6.76	5.41	5.73	4.72	5.75	4.56	4.88	10.97
blur n:3	8.45	9.08	8.02	8.09	8.51	9.01	7.89	7.96	6.86	7.29	6.37	6.49	?
contourdirection	12.82	16.73	12.83	13.49	11.81	15.67	11.69	12.34	10.00	10.53	9.42	9.48	?
contourdirection hist	64.93	57.55	63.33	60.31	64.87	57.65	63.17	60.42	68.55	72.03	69.58	69.34	70.89
decomponent n:127	29.21	26.76	28.06	26.68	28.61	26.44	27.41	26.11	19.10	15.36	17.17	15.82	21.85
decomponent n:3	66.74	64.83	66.67	66.51	66.72	64.84	66.66	66.48	65.70	64.77	65.35	65.16	66.82
decomponent n:7	39.86	38.56	38.09	38.12	39.92	38.86	38.31	38.17	40.13	37.92	38.62	38.00	44.53
enclosed	27.04	26.40	25.97	25.72	27.01	26.46	25.99	25.73	35.67	33.57	34.44	32.62	44.17
fnumpy op:cos	14.31	16.57	14.09	14.49	13.94	17.74	14.22	14.77	7.83	6.77	6.90	6.57	?
fnumpy op:diff	8.70	12.01	9.11	10.12	10.15	13.48	10.70	11.60	6.94	8.39	6.86	7.27	10.05
fnumpy op:fft	7.19	8.22	6.85*	7.09	8.43	9.79	8.19	8.28	7.38	8.55	6.98	7.24	?
fnumpy op:sin	71.88	79.54	74.32	75.68	39.37	48.01	41.76	42.54	29.24	24.44	27.41	25.56	53.08
fnumpy op:tan	72.52	76.78	73.46	74.27	62.81	69.02	64.23	65.09	63.95	63.85	64.12	63.04	?
hingefeat angles:32 leg:4	9.90	8.23	8.52	7.95	9.56	9.08	8.65	8.31	21.92	21.87	21.08	20.14	88.76
histograms	74.80	71.32	74.49	73.06	74.58	71.28	74.06	72.81	76.03	72.52	75.48	74.21	78.53
histograms norm	74.21	69.93	73.80	72.07	74.49	70.37	73.72	72.38	76.20	72.76	75.72	74.37	89.69
nccomponents	19.48	17.01	17.87	16.74	18.18	15.47	16.44	15.46	22.22	19.40	19.81	18.91	36.54
nccomponents norm	19.48	17.07	17.93	16.82	18.24	15.52	16.54	15.48	22.22	19.41	19.82	18.92	82.68
pass	2.73	3.82	2.68	2.94	3.35	4.58	3.37	3.62	2.39	3.21	2.33	2.53	7.76
pendepth sides:lrtb	12.06	15.43	12.08	12.75	9.43	11.21	9.09	9.58	12.05*	15.72	12.19	12.90	26.98
pendepth sides:ltb	13.80	16.48	13.70	14.21	11.05	12.43	10.62	11.00	13.91	16.61	13.85	14.30	29.77
pendepth sides:rtb	14.85	18.30	14.92	15.55	12.11	13.30	11.52	11.71	15.09	19.16	15.27	15.97	32.52
pendepth crop sides:lrtb	12.06	15.37	12.09	12.73	9.44	11.15	9.10	9.57	12.02	15.69	12.19	12.93	27.98
pendepth crop sides:ltb	13.80	16.46	13.70	14.20	11.05	12.41	10.62	10.98	13.90	16.60	13.85	14.31	30.71
pendepth crop sides:rtb	14.85	18.26	14.92	15.54	12.11	13.25	11.52	11.70	15.07	19.13	15.26	15.99	33.43
pie nslices:25	11.07	11.75	10.47	10.55	9.95	10.39	9.38	9.46	11.28	11.48	10.64	10.47	21.64
pie nslices:25 rotate	14.98	17.48	14.89	15.37	13.44	15.61	13.30	13.78	14.91	17.12	14.85	14.94	47.59
pie nslices:50	7.72	8.96	7.36	7.61	6.72	7.58	6.43	6.66	7.27	8.03	6.76	6.78	16.94
pie nslices:50 rotate	10.07	13.23	10.22	10.91	8.99	11.50	9.03	9.50	9.20	11.85	8.98	9.45	40.36
pie nslices:75	5.13	6.31	4.95	5.22	4.78	5.72	4.49	4.74	4.68	5.64	4.35	4.54	12.73
pie nslices:75 rotate	7.03	10.01	7.16	7.87	6.64*	8.94	6.57	7.10	6.15	8.82	6.08	6.65	32.40
projection	10.93	11.85	10.32	10.49	9.48	10.18	8.82	9.02	10.54	11.08	9.84	9.97	17.62
projection norm	75.90	71.53	75.58	74.08	69.17	66.77	70.27	68.58	37.60	48.99	39.85	40.89	52.69
slant	11.93	14.39	12.03	12.77	5.17	6.55	5.05	5.38	8.67	8.90	8.20	8.29	26.52
slant histsize:100	58.24	49.02	56.21	52.93	57.76	48.30	55.75	52.13	67.62	64.17	66.25	65.36	60.59
slant histsize:25	55.35	46.59	53.30	49.91	55.28	46.08	53.02	49.79	61.46	54.51	59.66	57.35	61.12
slant histsize:5	63.11	55.46	61.17	58.64	63.00	55.39	61.11	58.58	71.32	64.65	70.08	67.61	68.01
slant histsize:50	56.67	47.56	54.37	51.10	55.96	46.80	54.18	50.47	63.60	58.07	62.46	60.26	60.04

Table 4.3: The error rate in percentage for the ten-fold test of all classifier and feature combinations. The best five results have been underlined and are all acquired with the pass feature. The best five results for a complete training and testing using the test set are starred. The question marks indicate calculations which could not be completed due to either time or memory constraints.

Classifier	Feature	Error rate (%)
kNN negcor $k = 3$	pass	2.33
kNN negcor $k = 1$	pass	2.39
kNN negcor $k = 6$	pass	2.53
kNN euclid $k = 3$	pass	2.68
kNN euclid $k = 1$	pass	2.73

Table 4.4: Best five classifiers for the ten-fold test. The classifiers are kNN using the negative correlation distance measure (negcor), followed by the euclidean distance measures (euclid).

Classifier	Feature	Error rate (%)
kNN hamming $k = 3$	angle hist n=10 r=6	2.51
kNN euclid $k = 3$	fnumpy op=fft	2.74
kNN euclid $k = 1$	angle n=50 r=8	2.83
kNN negcor $k = 1$	pendepth sides=lrtb	2.91
kNN hamming $k = 1$	pie nslices=75 rotate	3.09

Table 4.5: Best five classifiers using a complete test. The classifiers include kNN using the hamming distance (hamming), euclidean distance (euclid) and negative correlation distance (negcor). After the name of each feature, the used parameters are given.

Chapter 5

Introducing a confidence classifier

Chapter 4 concluded that using only the ten-fold result as a measure of actual classifier performance would not help us boost the complete system performance. The k-fold test does not seem flexible enough to use for testing, which may be because it can not relate the type of image to the actual performance. One possible solution is the introduction of an extra classifier to make the selection here called a Classifier Confidence Classifier (CCC). This classifier is trained to perceive the performance of a classifier and guess its performance given a new classification task. The use of a separate classifier for this task was introduced by, among many others, Robert A. Jacobs [27].

The task of the CCC is to act as a gating network, allowing us to make the selection of the useful features for a given input. This specialisation based on the input allows us to keep all classifiers in the MCS, without making the MCS system calculable intractable.

Figure 5.1 shows how a CCC accompanies every classifier unit. Every CCC uses a subset of features, which means that at the time of testing, the features used by the classifier unit do not have to be calculated if the CCC predicts that the resulting classification would not be of any use.

Because the CCC correlates the mistakes made by the classifier to the example that is used, it should allow it to extract more performance information for use in the real test case. Using the result of the different classifiers, democratic voting can be used to create the answer for the MCS based on all, or a subset, of the classifiers. Democratic voting has the possibility of ending up in a tie situation, which is solved by randomly picking one from the solutions with the maximum number of votes.

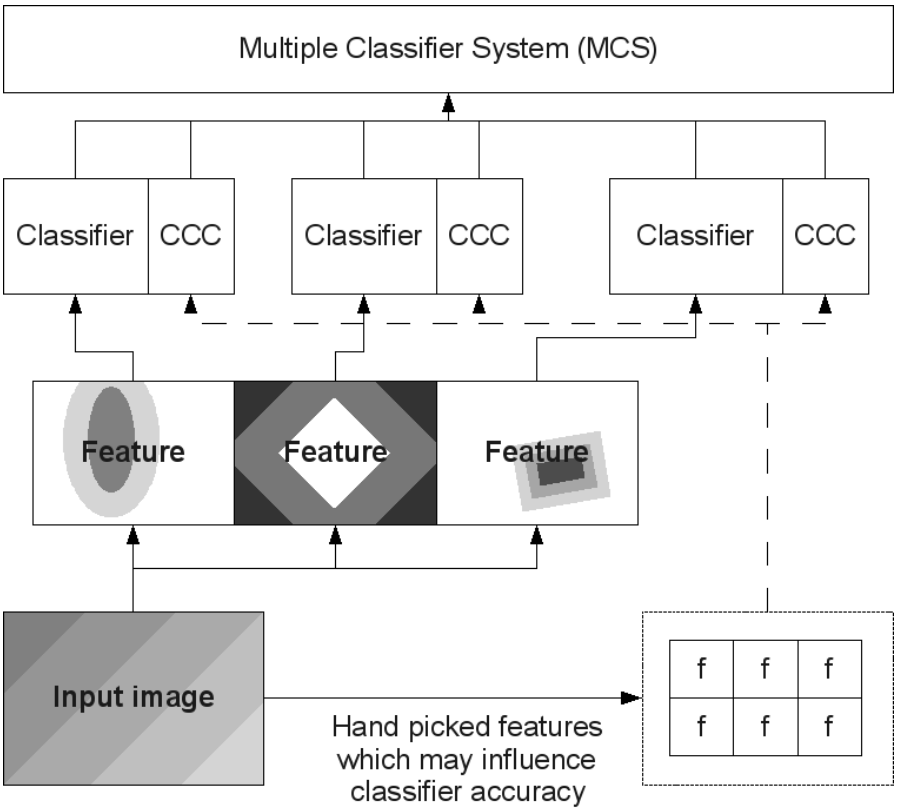


Figure 5.1: The MCS structure, illustrated with 3 classifiers, each with their own feature. CCC is the class confidence classifier used to classify the result of the classifier as confident or not. This classification uses concatenation of hand-crafted features of the original image, while each classifier uses the output of its feature. The classifications made by the classifiers and confidence the CCC has in this classification are combined into the final MCS answer.

5.1 Method

The previous section performed a ten-fold test on 741 classifiers. From these the worst performing methods are dropped to keep the calculation costs down. This sub selection results in the best 556 classification units, loosing a lot of the classifiers which performed at chance level. For each classification unit a ten-fold test on the MNIST training set is completed and these results are used to train a CCC for every classification unit.

The CCC uses a feature vector created by the concatenation of five earlier defined features: speckles (subsection 4.1.2), histograms (subsection 4.1.2), nccomponents (subsection 4.1.2), projection (subsection 4.1.2) and penetration depth (subsection 4.1.2) from all sides. All these feature vectors where independently normalised and then concatenated into one feature vector.

The classification method is are then tested by training it them on the complete training set and tested on the MNIST test set. This results in a performance measure for all of the classification methods.

After this all the CCCs are tested on the test set, which results in the confidence measure for the classification of each test instance.

5.2 Results

Using the complete test results, a ranking of the classifiers and their feature can be created. Figure 5.2 shows the performance of each classifier, sorted along the x-axes. Because the MNIST dataset has 10 classes, an error-rate of 90% can be expected when a classifier simply chooses at random. In the figure, the 90% error-rate threshold is marked by a dotted line, showing that the worst classifiers are performing at random error-rate.

Using this ranking, the error rate for the n best classifier units can be calculated. This result is shown in figure 5.3a. Focussing on the first 50 classifiers, figure 5.3b shows the performance of the best 5 classifiers in a democratic MCS is better than the performance of the best classifier on its own. This shows the main potential of an MCS. It is also interesting to note that, although the performance of the left-most additions is low, the overall MCS performance seems to stabilise.

For every instance, the CCC performs a linear classification predicting the usefulness of the classification unit vote. Taking the mean of all these classifications, using 1 as confident and 0 as not confident, results in figure 5.4. This figure shows the CCCs of the best 400 classifiers will except all classifications, after which the CCC starts rejecting classifications.

Using the knowledge of the CCC, it makes sense to look at the perfor-

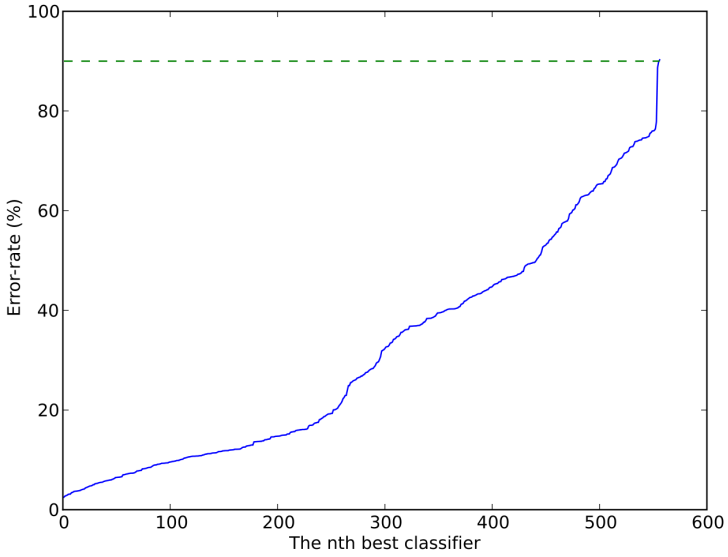
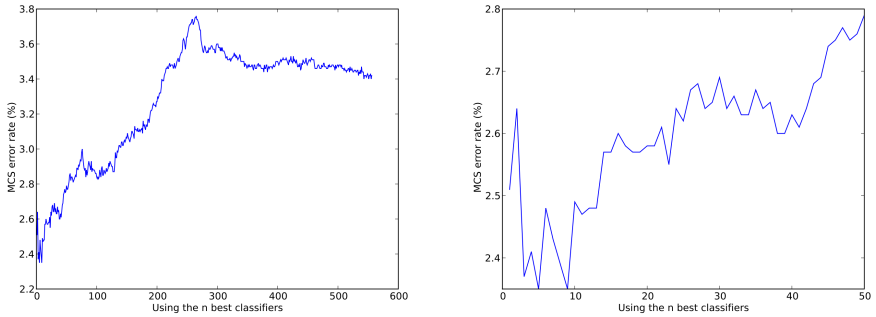


Figure 5.2: Error rate in percentage per classifiers. On the x axis are the classifiers ordered from best to worst. The dashed line is set at 90%, the chance level of the classification.



(a) For all classifiers (b) For the best 50 classifiers

Figure 5.3: Democratic MCS error rate of n best performing classifiers. On the x axis the classification methods from best to worst, on the y axis the error rate in $\%$ of the democratic MCS. The right figure zooms in on the first 50 classification methods of the left.

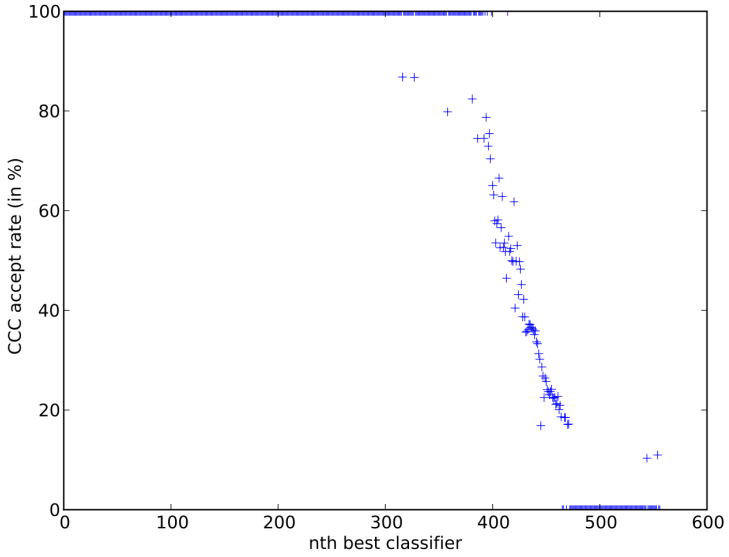


Figure 5.4: **The mean percentage of test instances the CCC is confident about, for each classification method ordered from best to worst. This figure displays the percentage of classifications the CCC will classify as good. As the figure shows, the first 300 classification methods are not influenced by the CCC as 100% of the answers are considered correct. The opposite holds true for methods ranked below the 500th place, where all classifications are considered incorrect. Between those two regions, the CCC selectively classifies parts of the classifications as correct and parts as incorrect.**

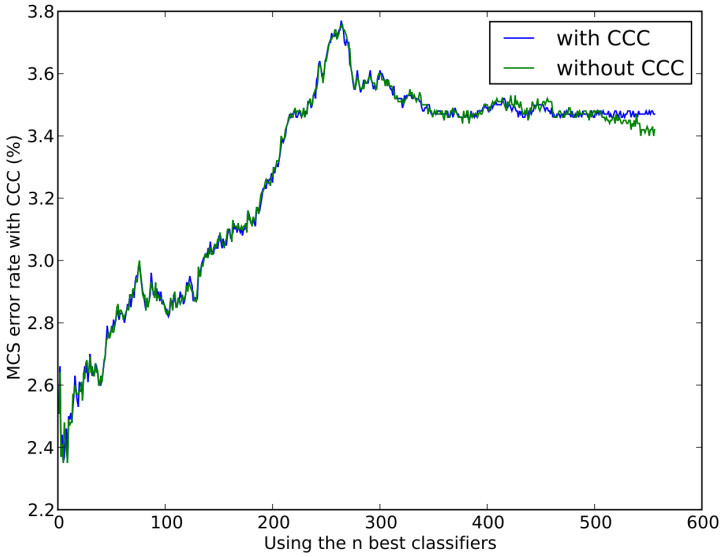


Figure 5.5: The MCS error rate in % for a democratic MCS with and without the use of the CCC, each error rate is calculated using the n best classifiers on the x axis. Both lines are not significantly different.

mance of the MCS with CCC and without. Figure 5.5 shows the error rate with and without the use of the CCC. The performance is not significantly impacted by the CCC at any point in this figure, even though the CCC does exclude some features. In figure 5.6, the correlation between the number of errors the CCC makes and the number of errors the classifier makes is plotted. This figure shows that when the classifier is good, the CCC will have the same performance as the classifier. The same holds true for when the classifier is bad. In the centre, at an error rate of about 50%, it is apparent that the CCC is fluctuating its answers as the error rate of the CCC and the error rate of the method are no longer the same.

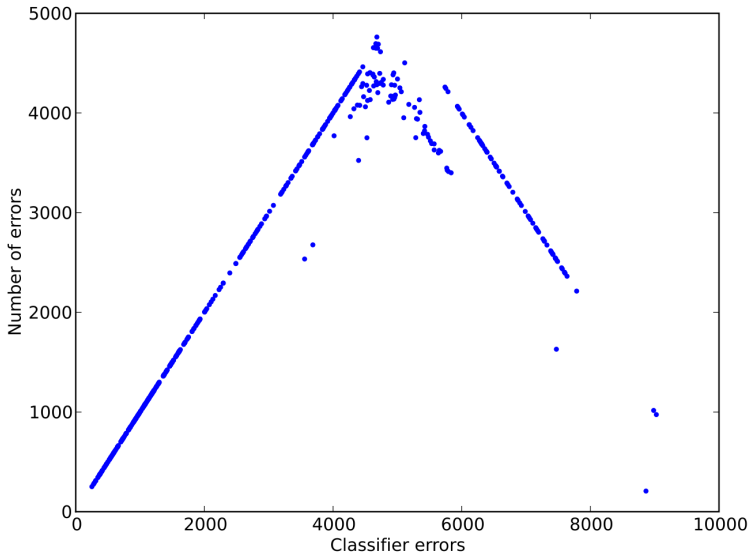


Figure 5.6: Scatter plot of the number of CCC errors (classifying a right answer as probably wrong) vs. the number of classifier errors (wrong classification). The linear correlation at the beginning and end are a result of the CCC either accepting or rejecting all classifications.

5.3 Conclusion

The results show that the CCC does not significantly improve the performance. This may be the result of the CCC not being able to distinguish between the incorrect and correct cases, because the variation in the cases is too large. This means that it may help to simplify the decision the CCC has to make by reducing the problem by specialising the classifier's task. Another potential explanation may be the relatively low number of incorrect classifications, which may inhibit the CCC from making an unbiased separation between correct and incorrect classifications.

Chapter 6

Per class confidence classifier

The previous experiment revealed that using the CCC per classifier could not significantly improve performance. The original MCS set up, as seen in 5.1, is therefore restructured by splitting up the CCC's classification task per class. As a result every class will introduce a separate classification unit, making the classifier's task more specialised. This may aid the CCC in creating a good decision boundary. Figure 6.1 reflects this new multi-class set up. Every classifier type is trained separately on one label, training it to distinguish between that label and the others. Figure 6.1 depicts the information flow for a three class problem with two features, each with one classifier. The results of each of these classifications are given to the MCS. Each classifier in the figure votes on whether it is its appointed class or not.

The MNIST training set contains ten classes which results in this set up requiring ten times the number of classifiers to be trained. Each classifier also has its own CCC. As a result the calculation costs are very high making it impossible test all 556 classifiers. To make this calculation feasible only the best 5 classifier found with the ten-fold test discussed in chapter 4 are used.

6.1 Method

The best five classifier/feature combinations found in a ten-fold test are each separately specialised on one of the ten different class labels. This results in 50 trained classification units, each with their own CCC. Each of these CCCs are trained using ten-fold test results of these 50 classifiers. Then the classifiers are trained on all the training data, after which they and their CCC are tested on the MNIST testing dataset.

On the votes generated by the MCS, the same democratic voting algo-

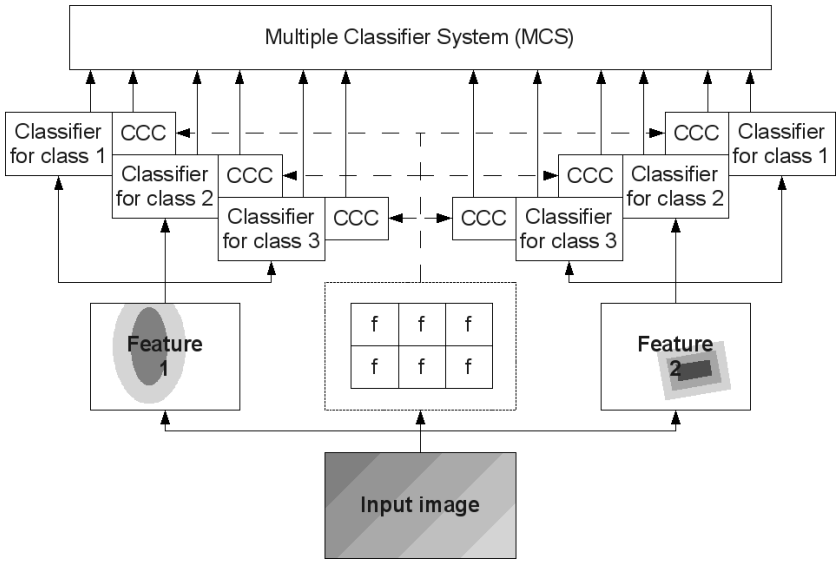


Figure 6.1: The MCS structure for a per class CCC approach. Each classifier is specialised for its own class, each with a CCC. This figure shows two classifier types and two features, given a three class problem.

rithm as described in chapter 5 is used. The outcome is the final classification of the MCS.

6.2 Results

After training the CCC, it was not able to reject any of the classifications made by the classification units during testing. This means that the CCC does not influence the classification in any way. The problem these results show is that the CCC was not able to find a boundary between the CCC features and the classification results.

The MCS has an error rate of 2.37 percent, which is higher then the minimal error rate of the best single classifier, which is 2.33 percent.

These results lead to the conclusion that the new strategy has not been able to outperform its best member. Moreover the added calculation costs did not impact the performance and should be avoided.

6.3 Conclusion

It has been shown that the CCC does not influence and therefore does not improve the performance. The reason may be that the CCC is not able to distinguish between the incorrect and correct cases because the dataset is to simple to be effectively classified using the CCC feature. Another problem may be that the performance of the classifiers is so high that the CCC can not be properly trained. A more difficult problem might allow the CCC to work more effectively.

Alternatively the CCC feature creation may be a problem. Because the CCC vector is a concatenation of multiple features, the influence of features with more dimensions is higher then the influence of smaller feature vectors, compensating for this may improve the overall effectiveness of the CCC.

Chapter 7

The "Kabinet der Koningin"

The previous experiment has shows that the performance of the CCC enables MCS was lower then the performance of its best classifier. This lead to the conclusion that there may be two problems keeping the CCC from having a beneficial effect on the MCS: the simplicity of the dataset and the construction of the CCC feature vector.

These conclusions led us to adapting the original experiment using a single CCC per classifier at two key points: change the dataset to include more classes and change the CCC feature vector to compensate for the dimensional differences between the various features.

The more difficult dataset is called the Kabinet der Koningin (KdK) (the Cabinet of the Dutch Queen [2]). This dataset consists of 336 handwritten words compared to the earlier used 10 handwritten digits.

7.1 Method

To ensure the CCC feature construction compensates for the difference in feature dimensionality, it is now created using a concatenation of copies of the same feature. The result is that the CCC features are now created using concatenated copies of the different hand picked features, ensuring that each feature has at least as many dimensions as the largest feature. The new set of hand-picked features are the normalised versions of: speckles, histograms, nccomponents, projection and penetrationdepth (for all sides, without cropping). Each of these vectors are normalised and then replicated to create vectors of at least the length of the feature with the highest dimensionality. The feature vectors are then concatenated into a feature with a total length of 1856 dimensions.

The implementation of the speckles feature has been altered to increase

the number of detected speckles. To make the number of detected speckles increase, the altered implementation uses 4 neighbouring pixels instead of the original 8. The implementation is still the same as described in subsection 4.1.2.

For the experiment 55 feature configurations were used, each configuration is described as a brace encapsulated list of variables per feature:

1. The `angle` feature using the parameters sets (n = 10, r = 4 and *histogram* = False), (n = 10, r = 6 and *histogram* = False), (n = 10, r = 8 and *histogram* = False), (n = 50, r = 4 and *histogram* = False), (n = 50, r = 6 and *histogram* = False) and (n = 50, r = 8 and *histogram* = False).

2. The `blur` feature using the parameters sets (n = 1), (n = 2) and (n = 3).

3. The `contourdirection` feature using the parameters *histogram* = False.

4. The `enclosed` feature using no parameters.

5. The `fnumpy` feature using the parameters sets (*op* = sin), (*op* = cos), (*op* = tan), (*op* = diff) and (*op* = fft).

6. The `hingefeat` feature using the parameters *angles* = 32 and *leg* = 4.

7. The `histograms` feature using the parameters sets (*normalize* = True and *nc* = 256) and (*normalize* = False and *nc* = 256).

8. The `nccomponents` feature using the parameters sets (*normalize* = True and *direction* = v), (*normalize* = False and *direction* = v), (*normalize* = True and *direction* = h), (*normalize* = False and *direction* = h), (*normalize* = True and *direction* = hv) and (*normalize* = False and *direction* = hv).

9. The `pass` feature using no parameters.

10. The `penetrationdepth` feature using the parameters sets (*normalize* = False, *sides* = l and *crop* = True), (*normalize* = False, *sides* = lrtb and *crop* = True), (*normalize* = False, *sides* = ltb and *crop* = True), (*normalize* = False, *sides* = lb and *crop* = True), (*normalize* = False, *sides* = r and *crop* = True), (*normalize* = False, *sides* = rtb and *crop* = True), (*normalize* = False, *sides* = rb and *crop* = True), (*normalize* = False, *sides* = t and *crop* = True), (*normalize* = False,

sides = tb and *crop* = True), (*normalize* = False, *sides* = b and *crop* = True), (*normalize* = False, *sides* = l and *crop* = False), (*normalize* = False, *sides* = lrtb and *crop* = False), (*normalize* = False, *sides* = ltb and *crop* = False), (*normalize* = False, *sides* = lb and *crop* = False), (*normalize* = False, *sides* = r and *crop* = False), (*normalize* = False, *sides* = rtb and *crop* = False), (*normalize* = False, *sides* = rb and *crop* = False), (*normalize* = False, *sides* = t and *crop* = False), (*normalize* = False, *sides* = tb and *crop* = False) and (*normalize* = False, *sides* = b and *crop* = False).

11. The `pie` feature using the parameters sets (*normalize* = False, *rotate* = True and *nslices* = 25), (*normalize* = False, *rotate* = False and *nslices* = 25), (*normalize* = False, *rotate* = True and *nslices* = 50), (*normalize* = False, *rotate* = False and *nslices* = 50), (*normalize* = False, *rotate* = True and *nslices* = 75) and (*normalize* = False, *rotate* = False and *nslices* = 75).

12. The `projection` feature using the parameters sets (*normalize* = True) and (*normalize* = False).

13. The `slant` feature using no parameters.

There features are combined with the following 17 classifiers configurations:

1. The `standclass` classifier, described earlier in section 4.1.3, using the parameters sets (*distype* = HAMMING and *nn* = 1), (*distype* = EUCLID and *nn* = 1), (*distype* = NEGCOR and *nn* = 1), (*distype* = HAMMING and *nn* = 3), (*distype* = EUCLID and *nn* = 3), (*distype* = NEGCOR and *nn* = 3), (*distype* = HAMMING and *nn* = 6), (*distype* = EUCLID and *nn* = 6), (*distype* = NEGCOR and *nn* = 6), (*distype* = HAMMING and *nn* = 20), (*distype* = EUCLID and *nn* = 20) and (*distype* = NEGCOR and *nn* = 20).

2. The `svmmulticlass` classifier, described earlier in section 4.1.3, using the parameters sets (*kernel* = radial and *gamma* = 0.01), (*kernel* = radial and *gamma* = 0.1), (*kernel* = radial and *gamma* = 0.5) and (*kernel* = radial and *gamma* = 1).

Combining the classifiers with the features created a total of 935 classification methods.

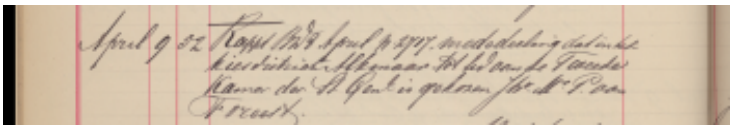


Figure 7.1: Part of a scanned KdK document page. After filtering and segmenting each word a binary image of each segmented word is used in the dataset.

(a) The word *School* (b) The word *welke*

Figure 7.2: Examples of the KdK segmented word dataset. The images in this state represent the dataset used in this thesis.

7.1.1 Dataset

The Kabinet der Koningin (KdK) (the Cabinet of the Dutch Queen [2]) is a large collection of handwritten documents. The text document important forms of government intervention in the Netherlands between 1798 and 1988. Parts of the document have been digitised and made available to the department of Artificial Intelligence at the University of Groningen, while the actual documents reside at the *Nationaal Archief* in The Hague, the Netherlands.

Large portions of the KdK are written by a single office clerk, however as shown in example figure 7.1, the document has a lot of overlapping text and is written in an old style cursive script.

From the document, a collection of 7257 word instances have been collected. These are whole words, segmented and converted to a binary image using a threshold. Examples of these binary images are given in figure 7.2. The dataset contains 336 different words.

The total set of 7257 words was split up into two sets: 5401 training examples and 1856 test instances, ensuring each set contained at least one instance of every of word. To make the data more manageable, the image sizes have been rescaled to 60 pixels wide and 30 pixels high bitmaps.

Classifier	Feature	Error rate (%)
kNN hamm nn=1	pie nslices:75	41.86
kNN hamm nn=1	slant	43.05
kNN euclid nn=1	pendepth sides:lrtb	43.16
kNN -cor nn=1	angle n:50 r:4	44.23
kNN -cor nn=1	angle n:50 r:6	45.8

Table 7.1: The best 5 classification methods for a full test on the KdK dataset.

7.2 Results

The 568 classification methods completed correctly and within the time limit. As with earlier experiments, the best 5 classifier where nearest neighbour based classifiers. Table 7.1 shows these best classifiers. Figure 7.3 shows the error rate per classifier for all 568 classifiers, ordered from best to worst. Because the number of classes is much larger, the error rate for random classification (the dashed line of figure 7.3) is much higher at 99.7%.

Because of the much larger set of classes, the influence of a wrong classifier is much lower: the possibility of the wrong classifiers picking the same wrong class is much lower. This can be seen in the results of the democratic MCS. Figure 7.4 shows that the overall error rate will drop when more and more classifiers are added, getting the error rate of the MCS below that of its best classifier. However, it is clear that the performance of the MCS with the help of the CCC is stuck at the performance of the best classifier. Figure 7.5, shows the mean CCC result and makes it clear that the CCC will ignore all the bad classifiers as their over-all measure drops below zero. The theoretical minimal error is the error rate where none of the classifiers gave a correct answer. Even if the combination and selection heuristic would work perfectly it would not be able to drop below this error rate.

Because of the low classifier performance, the CCC has trained to follow this performance. A good way to compensate for this is to use the actual value of the CCC to determine the voting power of each classifier. After thresholding the CCC value between -1 and 1, then adding 1, a range between 0 and 2 is created. Using these values to apply weighted democratic voting keeps the MCS performance from rising when the worst classifiers are added. Figure 7.6 shows the resulting MCS performance just below the democratic MCS with the name *thresholded CCC*. Using this method results in a better performance then using all the classifiers. This can be expected as the CCC results will keep the really bad classifiers from influencing the democratic process. This means that form the 110th classifier on, the per-

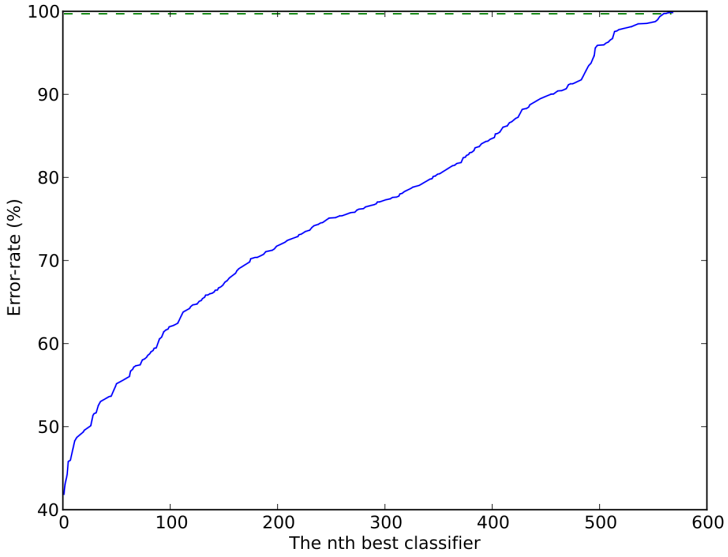


Figure 7.3: Error rate in % per classification method, for classification methods ordered from best to worst along the x axis. Chance level for this classification is denoted by a dashed line at 99.7%.

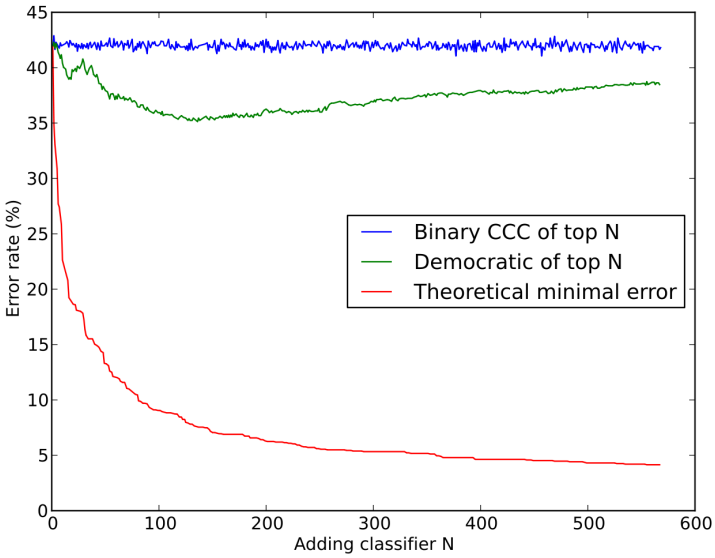


Figure 7.4: Various error rates in % for MCS combinations of the best n classifiers. The top most is a democratic MCS using only the answers the CCC is confident about, below that a democratic MCS using all answers of the top n classifiers. The theoretical error depicts the error rate if only the best answer of the top n classification methods is used.

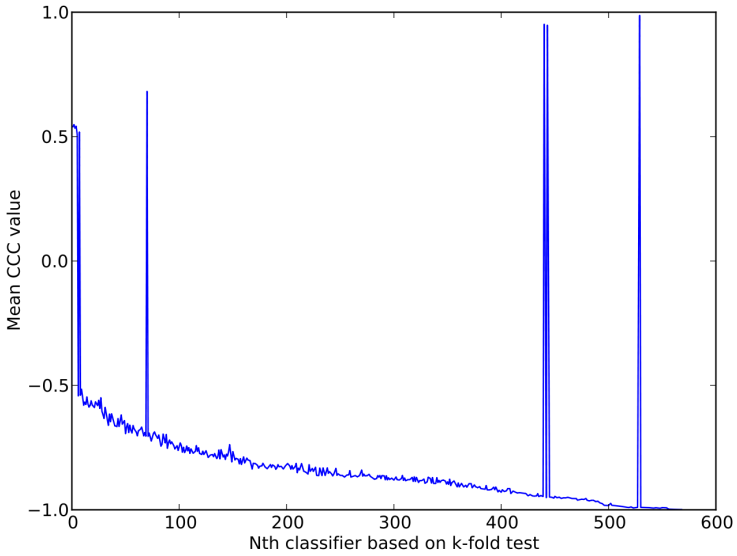


Figure 7.5: Mean value of the thresholded (between -1 and 1) CCC outcomes for the CCC of each classification method, ordered on the x axis from best to worst. This figure relates the mean CCC value to the real classification method performance, giving insight into how well the CCC is able to predict the quality of the method.

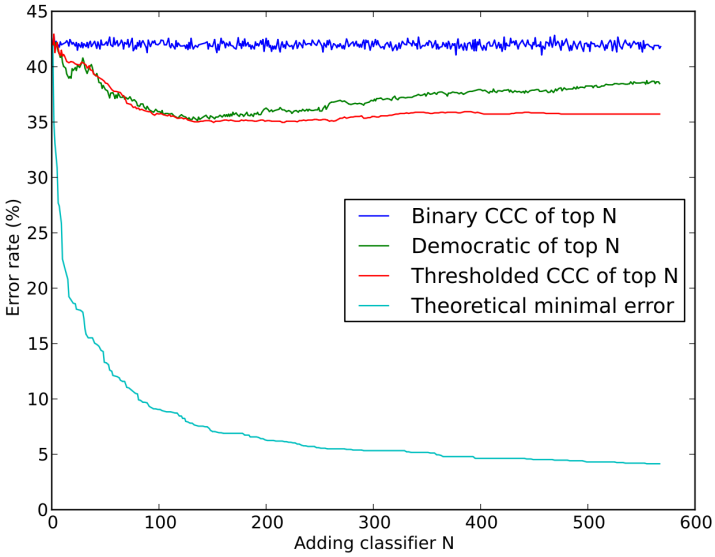


Figure 7.6: These lines depict the MCS with weighted voting using a thresholded CCC, MCS with a CCC, democratic MCS and theoretical minimal error rate per best n classification methods. The top two lines and the bottom line are the same as in figure 7.4. The thresholded CCC democratic MCS was added to this figure. The thresholded CCC democratic MCS uses the CCC outcome to apply weighted voting as described above.

formance is pretty much stable. Using this method does allow us to do some calculation costs saving, as there is no need to run a classifier for which the CCC result is less then -1.

7.3 Conclusion

The last experiment was not able to allow the CCC to make a good sub selection of answers. The CCC will however, make a sub selection of the best classification methods. Using the thresholded CCC values and a democratic voting rule keeps the performance almost as good as its best democratic sub selection.

The experiment does not show that the CCC is able to select the good answers of a bad classifier using a binary selection. This binary selection would be useful as it could keep the calculation cost down, making the MCS more scalable.

Chapter 8

Discussion and future research

Chapter 5 described how the introduction of an extra classifier did not help create a significant improvement in the performance of the MCS. Concluded from this was that the CCC may not have been specialised enough to make a proper judgement and decided to use more CCCs per feature by specialising the classifiers.

The results of chapter 6 did not support the hypothesis that the CCC needed to be specialised even more. As the performance did not improve when using a separate CCC and classification method for each label. These two results lead to the conclusion that the CCC outcome could not be effectively used as a binary classification, making it impossible to create a good sub selection of classification methods. This may be remedied by using a more complex feature with more dimensions as the CCC feature or by using a different classifier instead of the SVM used here.

If further research decides on using a more complex CCC feature, it should be noted that using a more complex feature may defeat the usefulness of the CCC when it comes to computational complexity.

Even though the approach does not guarantee a performance gain, the modular approach of an MCS proved to be very flexible. The approach allows or the structure of the MCS to be changed without changing any of the classifiers and features available. This added flexibility made it possible to try different MCS structures.

Another benefit of the system is its caching. Because the cache could be transported between systems, it was possible to extract parts of the calculation and try sub-experiments without influencing the flow of the main experiment. The caching system would also allow different scientists to share their results without sharing their implementation, making it easier to collaborate. Future research should also see if it is possible to create more collaboration using this feature.

Bibliography

[1] S. Amari and S. Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 12(6):783–789, 1999.

[2] Unknown author. *Archief van het Kabinet der Koningin*, volume NL-HaNa, chapter 2.02.14. Dutch National Archive, 1903.

[3] Yu Bakhtin, A. Danilov, A. Kantsel', and A. Chervonenkis. Stochastic systems- a method of restoration of conditional distributions from empirical data. *Automation and remote control*, 61(12):2003, 2000.

[4] A. Bensefia, A. Nosary, T. Paquet, and L. Heutte. Writer identification by writer's invariants. In *Frontiers in Handwriting Recognition, 2002. Proceedings. Eighth International Workshop on*, pages 274–279, 2002.

[5] IEEE-SA Standards Board. Standard for information technology-portable operating system interface (posix) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, 1 2008.

[6] V. Bouletreau, N. Vincent, R. Sabourin, and H. Emptoz. Handwriting and signature: one or two personality identifiers? In *Fourteenth International Conference on Pattern Recognition*, volume 2, pages 1758–1760, 1998.

[7] Leo Breiman. *Machine Learning*, chapter Bagging Predictors, pages 123–140. McGraw-Hill Science/Engineering/Math, 1996.

[8] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Chapman & Hall/CRC, January 1984.

[9] M. Bulacu and L. Schomaker. Text-independent writer identification and verification using textural and allographic features. *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*, 29(4):701–717, 2007.

[10] Horst Bunke. Recognition of cursive roman handwriting - past, present and future. In *7th Int. Conf. on Document Analysis and Recognition*, pages 448–459, 2003.

[11] Gustavo Camps-valls, Lorenzo Bruzzone, and Senior Member. Kernel-based methods for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 43:1351–1362, 2005.

[12] T. Cover and P. Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.

[13] Koby Crammer, Yoram Singer, Nello Cristianini, John Shawe-taylor, and Bob Williamson. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:2001, 2001.

[14] Nello Cristianini and John Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press, 1st edition, March 2000.

[15] NumPy Developers. Numpy: Python numeric library. http://numpy.scipy.org/.

[16] Thomas G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[17] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[18] H. Drucker, C. Cortes, L. D. Jackel, Y. LeCun, and V. Vapnik. Boosting and other ensemble methods. *Neural Computation*, 6(6), November 1994.

[19] John F. Elder IV. The generalization paradox of ensembles. *Journal of Computational and Graphical Statistics*, 4(12):853–864, December 2003.

[20] Ronald A. Fisher. The use of multiple measurements in taxonomic problems. *Annals Eugen.*, 7:179–188, 1936.

[21] Jr. Fred L. Drake and et. al. *Python/C API Reference Manual*, 2.5.2 edition, February 2008.

[22] Y Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and Systems Sciences*, 55:119–139, 1997.

[23] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *Thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.

[24] I. Guyon, L. Schomaker, R. Plamondon, M. Liberman, and S. Janet. Unipen project of on-line data exchange and recognizer benchmarks. In *Pattern Recognition - Conference B: Computer Vision & Image Processing., Proceedings of the 12th IAPR International.*, volume 2, pages 29–33, 1994.

[25] Tin Kam Ho, Jonathan J. Hull, and Sargur N Srihari. Decision combination in multiple classifier systems. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(1):66–75, 1994.

[26] S. Impedovo and Jean Claude Simon, editors. *From Pixels to Features III: Frontiers in Handwriting Recognition*. Elsevier Science Inc., New York, NY, USA, 1992.

[27] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffry E. Hinton. Adaptive mixture of local experts. *Neural Computation*, 3:79–87, 1991.

[28] T. Joachims. *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[30] Y. LeCun and C. Cortes. MNIST handwritten digit database. Online.

[31] M. Liwicki and H. Bunke. Combining on-line and off-line systems for handwriting recognition. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 1*, pages 372–376, Washington, DC, USA, 2007. IEEE Computer Society.

[32] J. Mantas. An overview of character recognition methodologies. *Pattern Recognition*, 19(6):425–430, 1986.

[33] Ranzato Marc'Aurelio, Christopher Poultney, Sumit Chopra, and Yann LeCun. Efficient learning of sparse representations with an energy-based model. In J. Platt et al. , editor, *Advances in Neural Information Processing Systems (NIPS 2006)*. MIT Press, 2006.

[34] National Institute of Standards and Technology. NIST Special Database 1. Online, 1997.

[35] National Institute of Standards and Technology. NIST Special Database 3. Online, 1997.

[36] David Opitz and Richard Maclin. Popular ensemble methods: an empirical study. *Journal of Artificial Intelligence Research*, 11:169–198, 1999.

[37] R. Plamondon and S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):63–84, Jan 2000.

[38] Automated Google result. Beading machine. Online.

[39] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Neurocomputing: foundations of research*, pages 89–114, 1988.

[40] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 20.4, pages 733–736. Prentice Hall, 2nd edition, December 2002.

[41] Leo Breiman Statistics and Leo Breiman. Random forests. In *Machine Learning*, pages 5–32, 2001.

[42] Songbo Tan. An effective refinement strategy for knn text classifier. *Expert Systems with Applications*, 30(2):290–298, 2006.

[43] G. Taunchek. Reading machine. *United States Patent 2026329*, 1929.

[44] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, New York, NY, USA, 2004. ACM Press.

[45] T. van der Zant, L. Schomaker, and E. Valentijn. Large scale parallel document image processing. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6815 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, January 2008.

[46] V. Vapnik. *Estimation of Dependences Based on Empirical Data*, chapter 5, pages 146–149. Springer, March 1982.

Appendix A

Additional data

A.1 Training examples used

Section 2.2.1 and 2.2.2 contains example images of classification of data points. These images where created using the training vectors in table A.1, with the origin at the lower left. The horizontal axis of these images runs from 0 to 640, the vertical axis from 0 to 400.

Class	$\mathbf{x}_0$	$\mathbf{x}_1$
white	96	172
white	93	344
white	39	292
white	43	325
white	32	186
gray	330	326
gray	370	351
gray	316	366
gray	300	346
gray	347	380
black	346	174
black	549	136
black	459	71
black	377	50
black	430	129

Table A.1: Training vectors used in classification figures

A.2 Caveats for MCS-implementations

When a classification system needs to be implemented, the following caveats should be noted.

ASCII data files ASCII data files are often used as any text editor will be able to open them. While true, ASCII is not a good format to store numbers. A 4 byte integer can contain the number 2147483648, which will create 11 bytes of ASCII data, almost 3 times as much. As a result, the number of bytes it takes to store a single number in ASCII format is not predictable. This will lead to problems when different types of classifiers and features are combined, as a simple division or multiplication by a feature may grow the data file size multiple times. Therefore all ASCII data representation of floating point numbers needs to be rounded to a smaller precision to keep these data files from blowing up, which makes it useless when the higher precision is needed.

`atof` Even though C is still widely used, it is often used incorrectly. One common mistake is the improper use of the `atof` function. This function does not check for errors, using it on a word will simply return 0. Any programmer using this function should first validate the input string first by checking the it against the locale character map for floating point numbers.

The Python GIL Multi-threading in Python has a big problem, the global interpreter lock (GIL) [21]. This means that any global variable is locked for multiple threads using a global lock that Python internally uses. This results in a lot of locking between processes using the same imported objects, like a math library. As a result multi-processing is preferred over multi-threading and is available through the Python 2.6 multiprocessing module.

List of Tables