Supervised Image Classification with Khoros - the Classify Toolbox Manual

Rafael Santos Takeshi Ohashi Takaichi Yoshida Toshiaki Ejima

May 5, 1997

Contents

List of Figures

List of Tables

Chapter 1

About this Document

Important Note: If you're reading this is because you got an alpha release of this software and document. The official release will not have this message on the manual. Please check the Ejima Lab Khoros Page for more information (see section 1.5 for its URL and other useful addresses).
The operators that are working in this release are listed in appendix A. See also a list of operators that are not working in this alpha release in chapter C.

1.1 What is this software and document ?

This document describes the *Classify Toolbox*, a software toolbox for the *Khoros* system, developed at the Ejima Laboratory at the Kyushu Institute of Technology.

The Classify toolbox contains programs and routines for basic supervised image classification with different techniques, report generation and other utilities. The programs were designed for use with Multispectral Digital Image Processing tasks, but with proper data preparation and formatting can probably be used with other applications of supervised classification too.

This document presents a simple introduction Supervised Image Classification and to some of its techniques and show how they can be applied for practical supervised image classification tasks. It describes the basics of each technique, the Khoros operators to prepare and classify data with that technique and some examples. This document doubles as a manual for the Classify Toolbox.

The latest version of this document can be obtained via the Ejima Laboratory Khoros Page or writing to the authors (see the section Useful Addresses below). Instructions on how to obtain the toolbox and install it are detailed on appendix B.

1.2 What do I need to use this software and document ?

The Classify Toolbox is a Khoros 2.1 toolbox - it was created from scratch so you'll need Khoros 2.1 installed and working in order to use it. It will not work on older versions of Khoros. You can get information on how to obtain Khoros 2.1 at the Khoral Research Home Page (the WWW address is listed in the Useful Addresses below).

In order to use this toolbox, at least a basic knowledge of Khoros is necessary, specially about Graphical User Interfaces (GUIs) usage and workspaces in Cantata. Knowledge about the Polymorphic Data Services is desirable but not essential.

You will also need to download and install the Classify toolbox - instructions are given on appendix B. The latest version of the Classify Toolbox can be obtained at the Ejima Laboratory Khoros Page (WWW address also listed in the Useful Addresses below). New versions of the toolbox will be released to fix bugs, enhance the features or to keep pace with the development of Khoros itself.

As described above, this document serves also as a manual for the Classify Toolbox and as an introduction to the classification, pre- and post-processing techniques covered by the toolbox. It is written to be more of practical use than theoretical, so mathematical and statistical details of the techniques will be presented superficially, but some familiarity with digital image processing is expected. Some of the concepts presented are not precisely defined in order to keep this document simple, whenever necessary, pointers to more complete or detailed references are given.

1.3 What's on this document

This document is organized as follows:

Chapter 2 - **Introduction to Supervised Image Classification** present basic concepts about supervised image classification that will serve as basis for comprehending the classifiers described in chapters 3 to 9.

Chapter 3 - **Parallelepiped Classification** describes one simple classifier which uses only the bounds for each feature to classify or reject pixels.

Chapter 4 - **K-Nearest Neighbors Classification** describes a classifier that relies on the number of neighbors of an unknown pixel to determine the class of that pixel, since usually pixels that belongs to the same class are grouped together in the feature space.

Chapter 5 - **Minimum Distance Classification** describes a classifier that uses the average of all samples in a class as a signature for that class and classify unknown pixels based on their distance to that average.

Chapter 6 - **Maximum Likelihood Classification** describes a classifier similar to the Minimum Distance classifier but that uses also information about the distribution of the pixels in the feature space to classify unknown pixels.

Chapter 7 - **Histogram Overlay Classification** describes a classifier that uses the values distributions (frequencies) over sample areas to classify unknown areas.

Chapter 8 - **Back-Propagation Neural Network Classification** describes a simple multilayer back-propagation neural network adapted for image classification, but which can be used for other classification tasks.

Chapter 9 - **Table Look-up Classification** describes another simple classifier that can be applied when the feature space discrete size is small.

Chapter 10 - **Pre-Classification Techniques** describes operators that are related with pre-classification techniques: ROI extraction and compression, signature appending and labeling, etc.

Chapter 11 - **Post-Classification Techniques** describes operators that are related with post-classification techniques: report and thematic map generation, inspection and retouch of classification results, etc.

Chapter 12 - **Other Utilities in the Classify Toolbox** describes other classification-related utilities in this toolbox.

Chapter 13 - **File Formats for the Classify Toolbox** describes the file formats for a-priori probability information, thematic map and report generation and other file formats used in this toolbox.

Appendix A - **Operators and Sample Files on the Classify Toolbox** contains tables with basic information for all operators in the toolbox and an index to the sections of this document that describes these operators. All sample files and workspaces are also listed.

Appendix B - **Installing the Classify Toolbox** shows instructions on how to obtain and install the latest versions of this document and toolbox.

Appendix C - **Future Work** shows what is still to be done in this toolbox and some planned improvements.

Khoros operators for all the techniques in this document are explained, with usage examples.

1.4 Acknowledgments & Thanks

The toolbox described in this document was developed as a tool for image classification by Rafael Santos, under the orientation of professors Takeshi Ohashi, Takaichi Yoshida and Toshiaki Ejima of the Ejima Laboratory, Department of Artificial Intelligence, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Japan.

The Back-Propagation Neural Network Classification (chapter 8) is based on programs by Prof. Takeshi Ohashi.

The authors would like to thanks Khoral Research for the great work they had developing Khoros and explaining several issues related to toolbox and software objects developing.

Thanks also to Professors T. Yamakawa and S. Ogata of the Kyushu Institute of Technology for explanations about several related points.

Thanks also to N.S. Environmental Science Consultants, Co. Ltd., Japan., Mr. Adriano Venturieri of Embrapa, Brazil and Mr. Ademir Morelli of Univap, Brazil for some sample images used in this document and toolbox. The Land Use map images are copyright © The Japanese Environment Agency.

This work was supported by a scholarship of the Japanese Ministry of Education. Rafael Santos was also supported by the University of Paraíba Valley (Univap), São José dos Campos, São Paulo, Brazil.

1.5 Useful Addresses

This section lists some useful addresses for more information:

- All information about Khoros (including requirements for the software, how to obtain Khoros, how to join the Khoros mailing list and more) can be obtained at the Khoral Research Home Page at `http://www.khoral.com/`
- Information about the Classify Toolbox and other Khoros-related works (including a Khoros Programming Tutorial) can be found at the Ejima Laboratory Khoros Page at `http://www.mickey.ai.kyutech.ac.jp/~khoros21/` (in case of change of location there will be a pointer to the new location on the address above)
- The Ejima Laboratory address is

  Ejima Laboratory - Department of Artificial Intelligence
  Faculty of Computer Science and Systems Engineering
  Kyushu Institute of Technology
  Kawazu 680-4, Iizuka-shi, Fukuoka-ken 820 JAPAN

- Rafael Santos can be reached by e-mail at `santos@mickey.ai.kyutech.ac.jp` until March 1998 and (probably) at `rafael@univap.br` from April 1998.

Chapter 2

Introduction to Supervised Image Classification

In this section we'll explain some basic terms about supervised image classification in general. Mathematical and statistical details will be covered only superficially, for more details please refer to one of the references. Even if some explanations on this section seems simplistic or incomplete, the purpose is explain the basic concepts about supervised image classification in a practical way. The classification steps and explanations on how to implement these steps in Khoros (Cantata) will also be shown.

2.1 What is Image Classification ?

A digital image is composed by pixels or points[1], and these points usually represent values in a multidimensional space. Each point can be represented as

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

where x_i is the value of pixel x in the *band* or *feature* i (the term feature is more used since band is more related with spectral bands). The vector x is also called the *feature vector* or *measurement vector*. *Feature Space* is the set of all possible feature vectors.

Usually the value of a pixel x in a band i is the brightness or gray level for that pixel, but for some classification tasks one may want to use other features, for example, texture measures, etc. In this case it would be necessary to normalize the values in the feature space so all feature space dimensions will be the same.

Classification is a method by which unique labels are assigned to pixels based on the values of the vector x. This decision is made by applying a *discriminant function* $g_i(x)$ associated with class ω_i to vector x, and choosing the largest $g_i(x)$. In other words: for all classes $\omega_1, \omega_2, \ldots \omega_M$ in a classification task the pixel x is classified as class ω_i if its $g_i(x)$ is the largest value for all $i \in M$, or:

$$x \in \omega_i \quad \text{if} \quad g_i(x) > g_j(x) \quad \text{for all} \quad i \neq j \tag{2.1}$$

The total number of classes in supervised classification is determined by the nature of the problem and by user's decision. The discriminant function depends on the chosen classification method. Discriminant functions used in the different classifiers in this toolbox will be presented.

Classification can also be considered as the partition of the feature space in mutually exclusive parts. Pixels are assigned to classes based on this partition. Depending on the classification

[1] Both terms are used with the same meaning in this document

method a threshold can be selected so some pixels for which $g_i(x)$ are not large enough will be *rejected*.

Some of these concepts can be illustrated with figures. Figure 2.1 shows one simple 2-

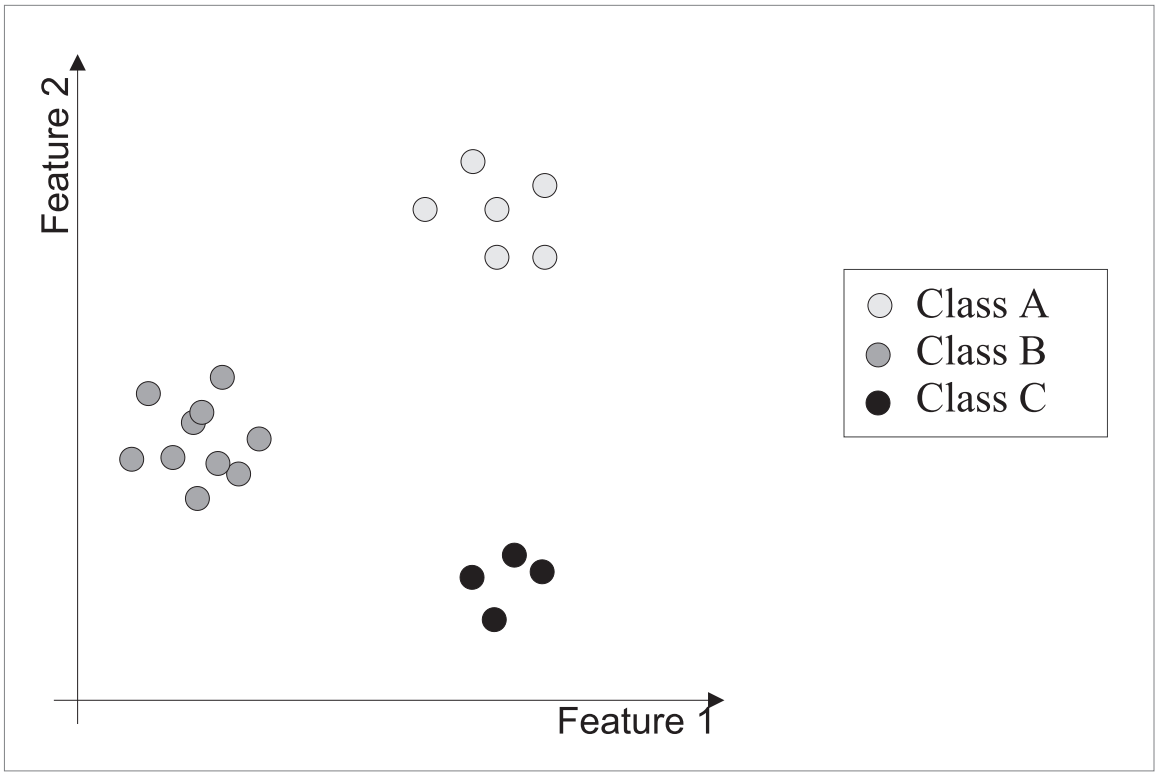

Figure 2.1: A 2-Dimensional Feature Space

dimensional feature space with some points divided in three classes, represented by different gray levels. The class assignment here was made just to exemplify one of the characteristics we can expect in the points distribution: points with similar feature values are usually grouped together and usually belong to the same class.

Figure 2.2 shows the same 2-dimensional feature space but with some unknown pixels plotted on it. The generic classification problem is decide to which class the unknown pixels belongs to. Considering our discriminant function to be based on the proximity from the unknown pixel to the center of the samples' cluster, we could easily decide that the unknown pixel 1 belongs to class A, but it is more difficult to decide whether unknown pixel 2 belongs to class B or class A. For the unknown pixel 3 the classification problem is even more difficult: it seems to be at the same distance from classes A and C but at the same time it is too far from both, it could probably be rejected depending on the classification method and parameters chosen.

Figure 2.3 shows the three classes in the 2-dimensional feature space with a possible partition in three areas, each one corresponding to a class. A pixel will be classified to a class X if its coordinates are inside the region determined by that linear decision boundary (two hyperplanes which separates the three classes). There is no rejection of any pixel in that feature space, even pixels with extreme values for one or both features will be classified. Note also that there are more than one set of hyperplanes that can separate the three classes (using the partition presented above all unknown pixels in figure 2.2 would be assigned to class A !). This kind of separation can be obtained by a Minimum Distance to Class Mean (chapter 5) when no thresholds for minimum distance are considered.

Figure 2.4 shows the same example with another possible partition of the feature space. In this case instead of hyperplanes we have hyperspheres that encloses regions of the feature space. A pixel is classified to a class X if its coordinates are inside this hypersphere. The region R that corresponds to the feature space minus the union of the regions A, B and C is the reject

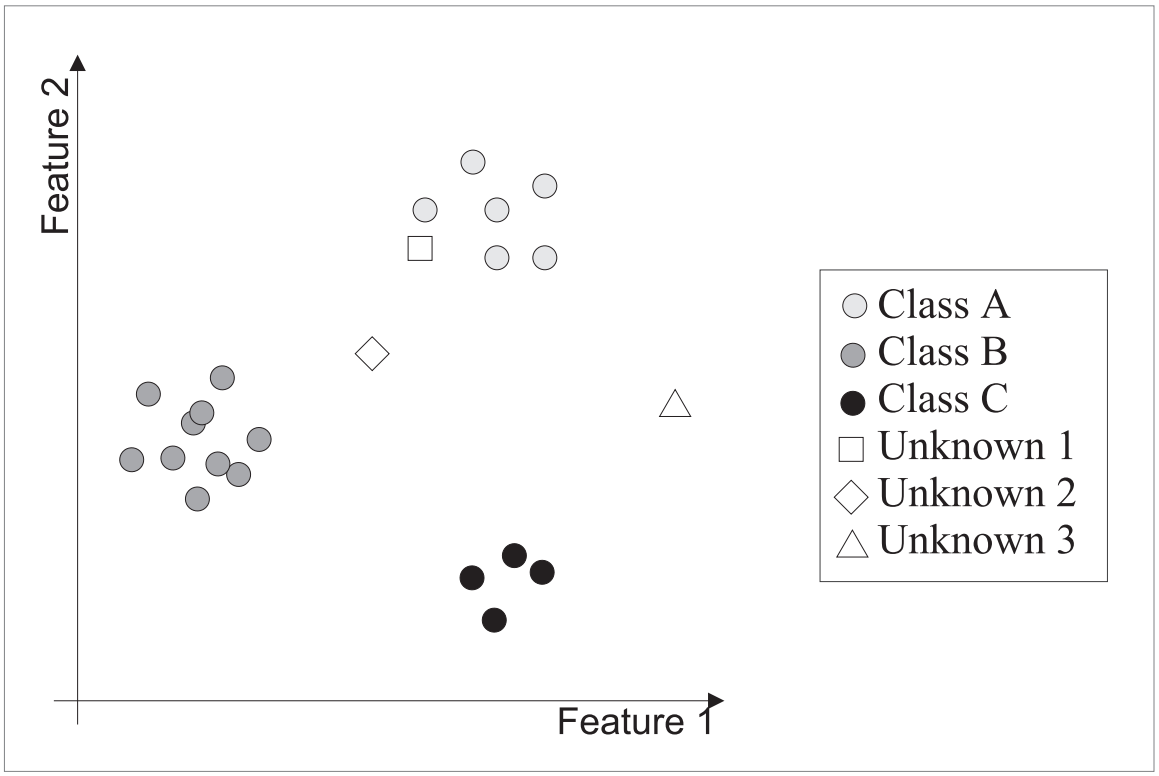

Figure 2.2: Unknown points in the 2-Dimensional Feature Space

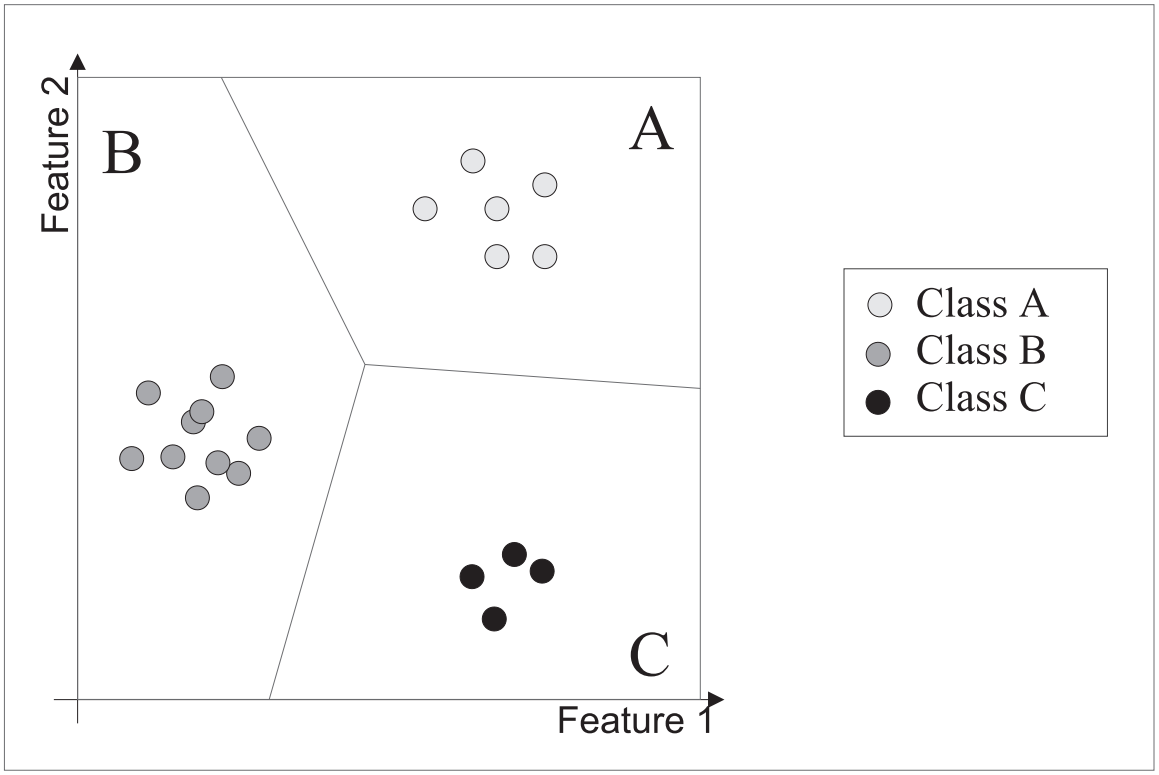

Figure 2.3: A possible partition for the 2-Dimensional Feature Space

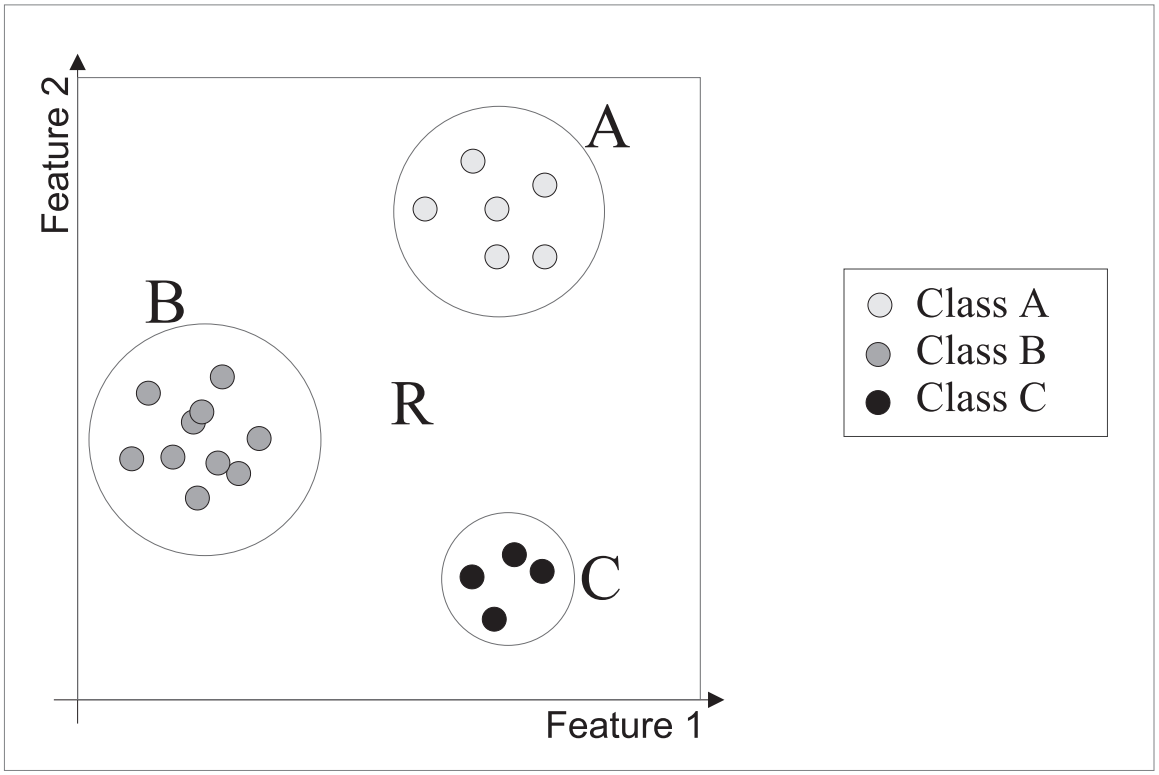

Figure 2.4: Another possible partition for the 2-Dimensional Feature Space

region - any pixel which coordinates is inside this region is therefore outside all other regions (remember that the partition is mutually exclusive) so it is assigned to the reject class.

2.1.1 Bayes' Classification

Here is a little more formal definition of the above, which is known as *Bayes' Classification*:

The classes in a classification task can be denoted by

$$\omega_i, \quad i = 1, \ldots M$$

where M is the total number of classes. The probability that the correct class for x is ω_i is given by

$$p(\omega_i|x), \quad i = 1, \ldots M$$

where $p(\omega_i|x)$ is called the *a-posteriori probability*. To decide which class i is the best (or has the least classification error) for the pixel x we should select the largest $p(\omega_i|x)$ - on other words, select from between the probabilities that the correct class for pixel x is ω_i for all i the highest one, or:

$$x \in \omega_i \quad \text{if} \quad p(\omega_i|x) > p(\omega_j|x) \quad \text{for all} \quad i \neq j \tag{2.2}$$

The problem is that these $p(\omega_i|x)$ we need to determine the class for pixel x are unknown. If we have enough samples from all classes we can estimate the probability for finding a pixel from class ω_i in position x, denoted by $p(x|\omega_i)$. If there are M classes, there will be also M values for $p(x|\omega_i)$ denoting the relative probabilities that the pixel x belongs to the class i. The relation between $p(\omega_i|x)$ and $p(x|\omega_i)$ is given by Bayes' theorem:

$$p(\omega_i|x) = \frac{p(x|\omega_i)p(\omega_i)}{p(x)}$$

where $p(\omega_i)$ is the probability that the class ω_i occurs in the image, also called *a-priori probability* (described in section 2.1.3) and $p(x)$ is the probability of finding a pixel from any class at position x. Using this definition and removing $p(x)$ since it is a common factor we can change equation 2.2 to:

$$x \in \omega_i \quad \text{if} \quad p(x|\omega_i)p(\omega_i) > p(x|\omega_j)p(\omega_j) \quad \text{for all} \quad i \neq j \tag{2.3}$$

And substituting

$$\begin{aligned} g_i(x) &= \ln\left(p(x|\omega_i)p(\omega_i)\right) \\ &= \ln p(x|\omega_i) + \ln p(\omega_i) \end{aligned}$$

we get

$$x \in \omega_i \quad \text{if} \quad g_i(x) > g_j(x) \quad \text{for all} \quad i \neq j$$

which is the equation 2.1, where $g_i(x)$ is the discriminant function, and is calculated differently for the different classification schemes. When $p(\omega_i)$ is not available it is considered as equal to 1 for all classes.

2.1.2 Signatures

In our examples above we had some knowledge about the classes before trying to classify the unknown pixels. This knowledge is obtained by examining the feature vectors for pixels that we know to which class they belong. Practically speaking, in an image to be classified, we select some *sample pixels* or *samples* and assign them to classes. These pixels' vectors' information will be used to create a *signature* that will be used with the discriminant function to determine the class for the unknown pixels.

For some classifiers, the signature will be composed of some parameters obtained from the data - these classifiers are commonly referred as *parametric classifiers*. For other classifiers, the signature will be composed of the set of points itself - these classifiers are called *non-parametric classifiers*.

The term signature usually refer to the set of parameters extracted from the sample for parametric classifiers, in this document we will consider that signature is the data used to represent the classes for classification, parametric or non-parametric. For example, if we assume that the classification method for the examples in the figures above is something like the Minimum Distance to Class Mean (chapter 5), the signature for each class will be the mean vector $\bar{x}_i$ obtained from all $x \in \omega_i$ and the pixel will be classified to the class which center is closest. If we decide to use the K-Nearest Neighbors Classifier (chapter 4) the signatures will be the set of sampled points.

Signatures must be created for all classes in a classification task, and usually are created in a two-step process: First the user select sample areas (usually by extracting *ROIs*, regions-of-interest) in an image that are homogeneous in respect to that class, and later these ROIs are used to create the signatures that are specific for the classification method.

2.1.3 A-Priori Probabilities

For some classification problems we can use the *a-priori probability* of a class ω_i occurring on the image to be classified (denoted by $p(\omega_i)$). This factor (or its logarithm) can be used as a bias by some classifiers - for example, if we know that 10% of the pixels in an image belong to class A we could say that the a-priori probability for class A is 0.1 and use this value as a bias. When the a-priori probability cannot be estimated for a classification task, it will be considered as equal to 1 for all classes.

A-priori probabilities can be either estimated by knowledge of the classes' distribution on the image or by sampling the image and calculating the occurrence of each class. This process must be done carefully, making sure that all present classes are sampled so none will have $p(\omega_i) = 0$.

2.1.4 References

Several of the terms presented in this section are defined in [1]. For a more detailed description of the a-priori and a-posteriori probabilities and Bayes' classification please see chapter 8 of *Remote Sensing Digital Image Analysis - An Introduction* [2]. Chapter 2 of *Neural Computing: An Introduction* [3] also gives an introductory overview of this topic. Another good, introductory reading is *Adaptive Pattern Recognition and Neural Networks* [4]. For more detailed discussions please refer to *Applications of Pattern Recognition* [5] or to *Pattern Classification and Scene Analysis* [6].

2.2 Steps on Supervised Image Classification

The basic steps on supervised image classification are:

1. *Selection of a classification method*: There are different classification methods that can yield different results depending on the nature of the data to be classified. Some of them are faster than others, some are more useful with data which pixels' distribution is normal or close to normal. This choice depends on several factors.

2. *Extraction of Samples*: For each class in the image a significant number of samples must be chosen. The exact or even approximate number of pixels for the samples is hard to determine, but insufficient sampling can easily lead to poor classification.

3. *Creation of Signatures*: The samples that were extracted can be used to create signatures that will characterize the classes. For some classifiers the signatures are statistical information extracted from the samples, for others it is more based on the samples' values themselves.

4. *Classification*: All pixels x in the image are used with the discriminant function to determine which ω_i for $i \in 1..M$ is the largest. If necessary and required, rejection schemes can be applied. Classification can be performed with different images than the one that originated the samples if the images are considered to have approximately the same classes distribution.

5. *Post-filtering and Evaluation*: Some basic post-filtering techniques can be applied to eliminate areas that are too small in the classification result. Results can also be evaluated by comparing them with an expected result image (ground truth).

6. *Report and Thematic Map Creation*: In this step reports with the quantitative results of the classification can be created to show numerical information about the classified image. A visual representation of the classification (thematic map) where classes are assigned to colors for ease of visualization can also be created.

Steps 2 to 6 are shown in figure 2.5. The image to be classified is used to create the classes samples, which will be used to create the signatures. The set of signatures and the input image (not necessarily the same as the one which generated the samples, as explained before) will serve as input for the classifier which will create a classified image. This classified image can be used to create reports, tables or thematic maps.

The same steps as they should be implemented in a Cantata workspace are shown in figure 2.6.

In figure 2.6, the ROI for each class could be created either with the interactive ROI extractor (`extractor`) or with specifying coordinates for rectangular regions on the image (single or multiple) and using the `cROIfromcoords` operator. ROIs for use in signature extraction will be usually either a non-masked region of an image where all pixels belong to a class or a region with a mask segment where only the pixels masked as valid belongs to a class (making possible the selection of non-contiguous ROIs in an image).

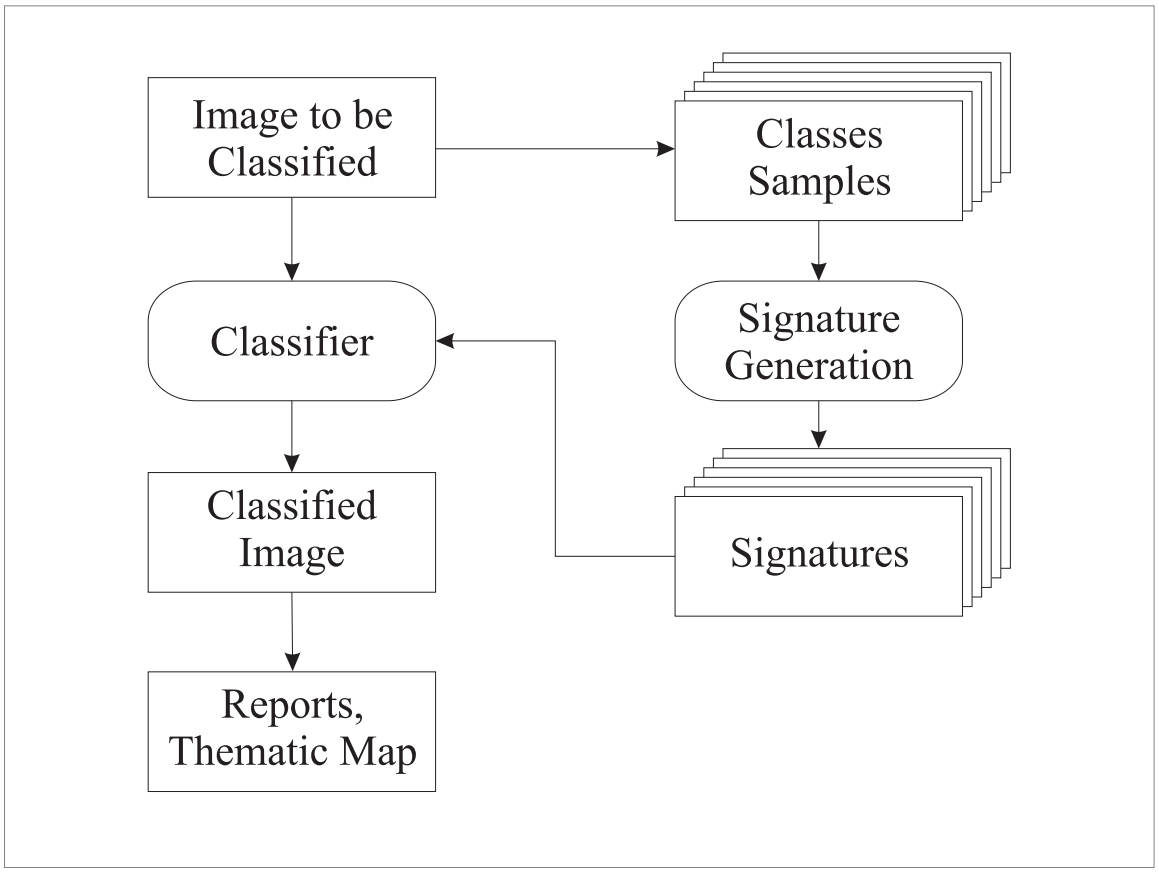

Figure 2.5: Basic Classification Steps

After ROI extraction the signatures will be created for each class. The algorithm, stored data, file dimensions and formats are specific for each classifier and will be discussed in details in the classifiers' respective sections in this document.

In this toolbox, signatures will be created separately for each class and must be joined together later, usually with the `kappend` operator when the signatures are for use with parametric classifiers or with the `csigappend` (section 10.5.1) operator. The appended block of signatures will serve as input to the classifiers. There are two reasons for this: with making the input to the classifiers a block we can have as much classes as we want and if we have a classification task which can be applied to more than one image (or if we want to test the classifier over an small area before trying it on a larger one) we can separate the signature extraction and classification parts shown in figure 2.6. Details on appending the signatures for the classifiers are shown in section 2.2.1.

The classifier accepts as input an image to be classified, which can have more than one time and depth dimensions but must have the same number of elements as the one that generated the signatures, and a block of appended signatures. Some classifiers will accept an a-priori information file too, which format is shown in section 13.4.

As output, all classifiers will create a single-element image with the same W, H, D, T dimensions as the input image, and each pixel in the output image will be the index of the appended signature, starting with 1 and reserving 0 for unclassified pixels. Classification is usually done by choosing the best probability or likelihood but some classifiers can also get the second, third, etc, best likelihoods.

Most classifiers are also able do create a *Output Probabilities* file with the relative probabilities for each class and pixel, which can be used to inspect the classification result with `cxinspector` (section 11.7.1) or filtering with `cprobmode` (section 11.2). Pixel-by-pixel classification results can also be created, so the user can check the classification results while and after

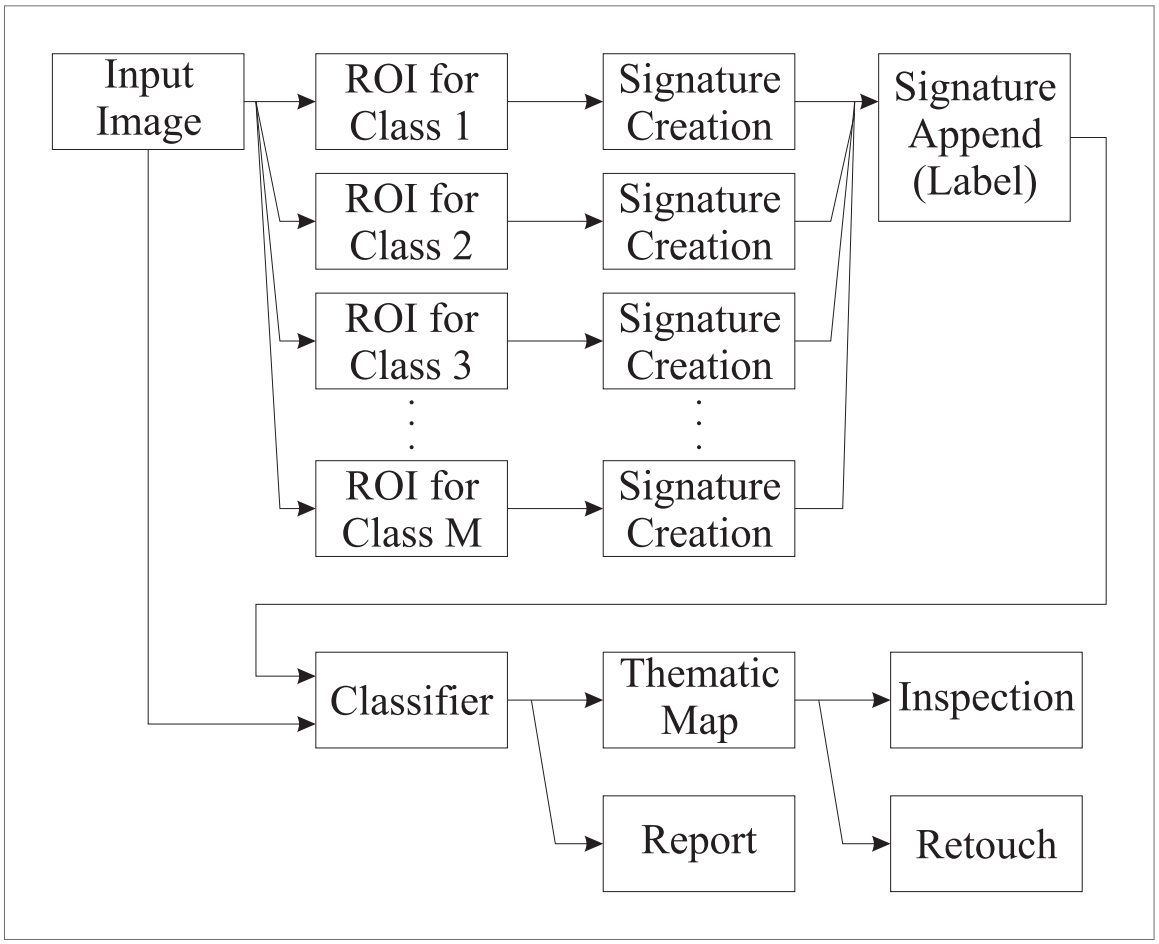

Figure 2.6: Basic Classification Steps for a Cantata workspace

classification, but this slows down the classification (see appendix C for future work on this).

The expected and created dimensions for the polymorphical data objects involved are:

- **Input Image**: $W_i \times H_i \times D_i \times T_i \times E_i$, can have mask, masked pixels will not be classified.
- **Input Signatures**: Will vary according to the classification method used. Must be generated with the signature extraction and one of the appending methods mentioned above.
- **Output Image**: will be created with size $W_i \times H_i \times D_i \times T_i \times 1$, with each pixel corresponding to either 0 (non-classified) or an index from 1 to M where M is the number of classes (equal to the number of appended signatures). If the classifier rejects any point the point will be masked too.
- **Output Probabilities**: will be created with size $W_i \times H_i \times D_i \times T_i \times M$

2.2.1 Appending the classes' signatures

As shown in figure 2.6, the signatures are created separately for each class and must be joined prior to classification. When the classifier use parameters as signatures, all signatures have the same dimensions and can be joined together with the `kappend` operator along the time dimension, creating an implicit labeling which will be used by the classifier - note that it does not mean that the signatures have any temporal characteristic, they're just indexed by the time element. The signatures appending for parametric classifiers is shown in figure 2.7. The Parallelepiped classifier (chapter 3), the Minimum Distance classifier (chapter 5), the Maximum

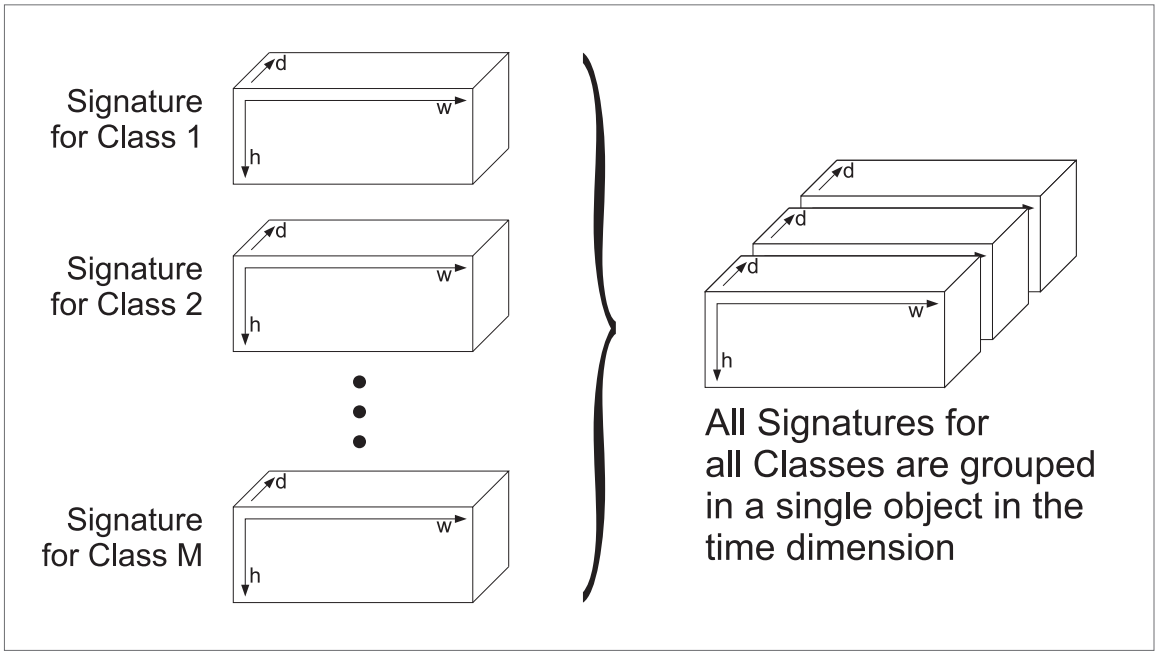

Figure 2.7: Appending classes' signatures (constant-sized signatures)

Likelihood classifier (chapter 6) and the Histogram Overlay classifier (chapter 7) uses constant-sized signatures which can be appended with `kappend`.

For non-parametric classifiers, the signatures for the classes are the pixel values (feature vectors), which can vary in size depending on the size of the samples (ROIs) for each class and there is no way to use an implicit label for the classes. In this cases we must label the samples before joining them. This can be done by the `csigappend` operator, described in section 10.5.1. The signatures appending for non-parametric classifiers is shown in figure 2.8. The K-Nearest Neighbors classifier (chapter 4) and the Table Look-up classifier (chapter 9) uses variable-sized signatures which must be appended with `csigappend`.

Another classifier in this toolbox, the Back-Propagation Neural Network classifier (chapter 8) does not use signatures in the same sense. Please refer to chapter 8 for more information and classification examples.

2.3 Unsupervised Image Classification

Unsupervised image classification techniques are not covered by this toolbox at the present moment, but will be presented briefly for completeness and comparison with the supervised classification approach.

In unsupervised image classification there is no specification of signatures or samples, instead of giving these parameters the classifier try to obtain parameters by analyzing the distribution of the pixels in the image. Since pixels that belongs to the same spectral classes are usually close in the feature space, they form groups (or clusters) that can be detected.

The most common approach to unsupervised image classification is clustering of the data. Initially a large enough number of initial clusters is chosen, and a clustering algorithm is applied. From the result, depending on the algorithm parameters, clusters can be merged or separated, and the process is repeated until the algorithm decides it reached a stable state. The resulting clusters are the classes and the pixels assigned to these clusters are considered classified. There are several variations upon this basic algorithm.

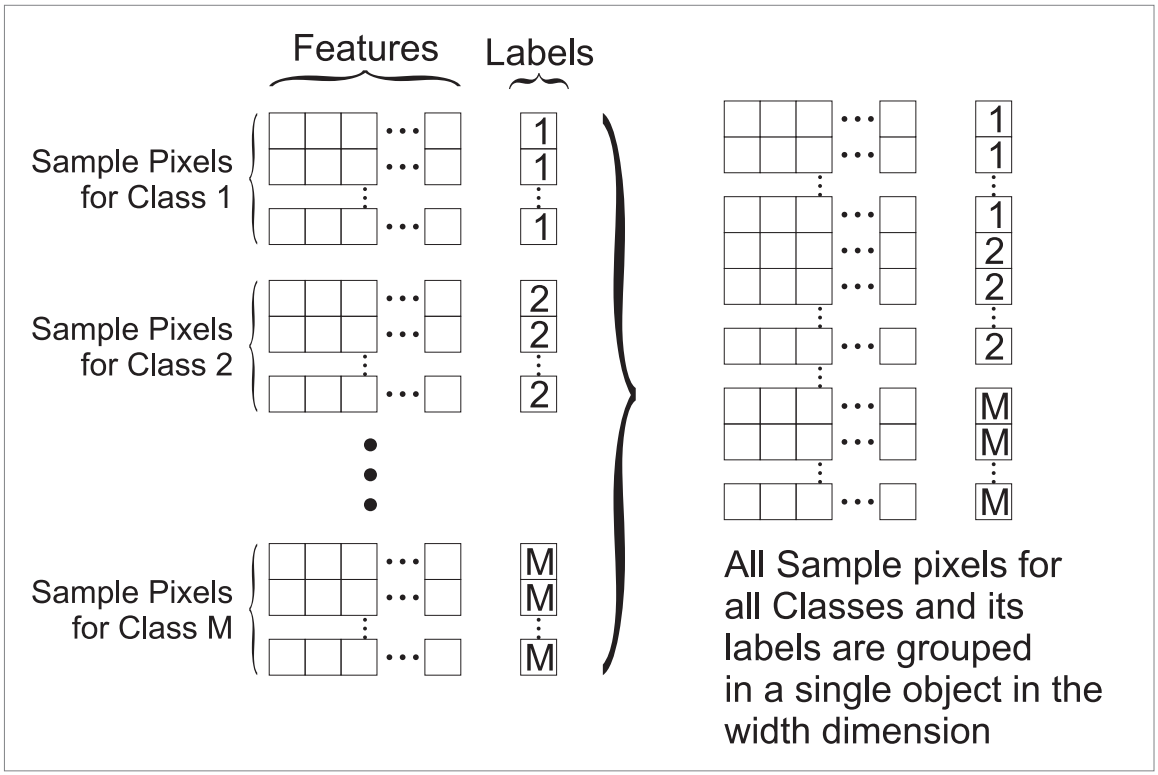

Figure 2.8: Appending classes' signatures (variable-sized signatures)

Chapter 3

Parallelepiped Classification

3.1 Introduction

The Parallelepiped Classifier is a parametric classification method in which we assume that all pixels that belongs to a class have feature values inside bounds defined by that class samples. A pixel x is classified to class ω if all values of the vector x are inside the bounds for the class ω. Figure 3.1 shows one example of classification with the Parallelepiped classifier in the 2-dimensional feature space.

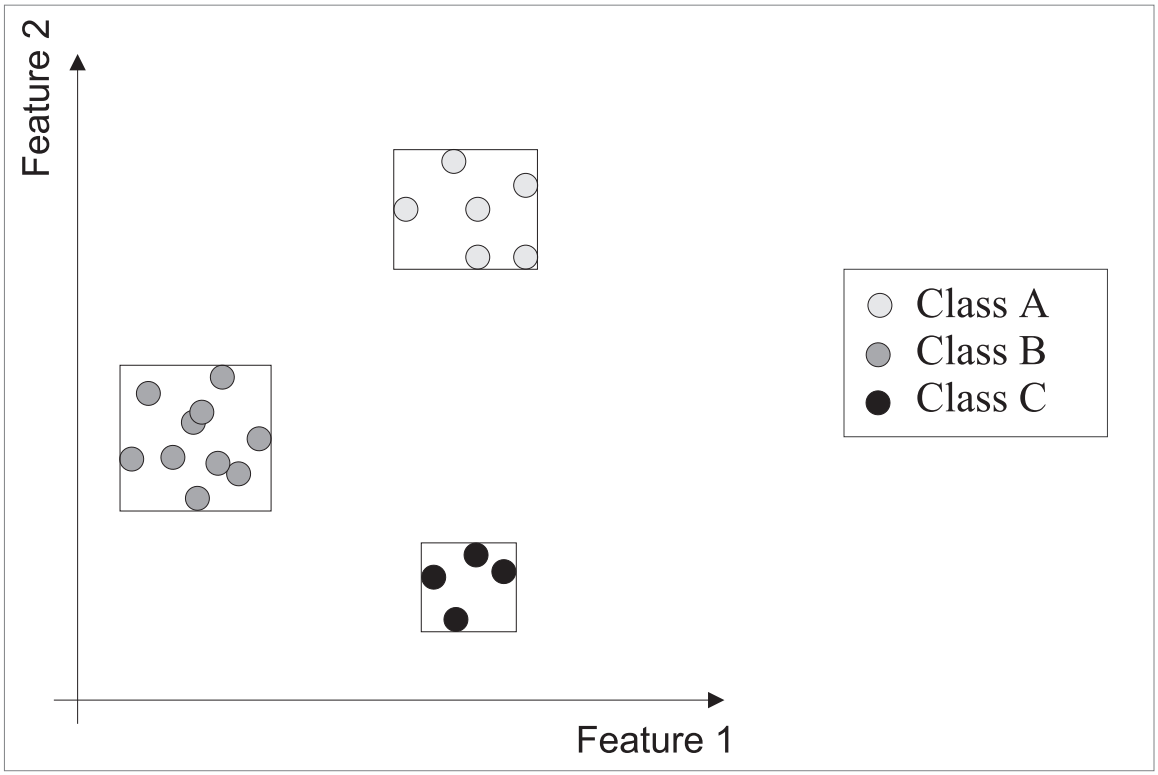

Figure 3.1: Parallelepiped Classification example in the 2-Dimensional Feature Space

Classification is done by checking whether the pixel is inside or outside the bounds for each feature and ANDing these true/false results. If any of the inside/outside checks is false, the pixel will be left unclassified. This is shown in figure 3.2, where in each axis representing a feature in the 2-dimensional space, a pixel value can be assigned to class A, B, C or ? (unknown). If any feature value for a pixel is outside the bounds, that pixel will be left unclassified (or classified as rejected).

The Parallelepiped classifier can be used when we know that there will be no overlap of the

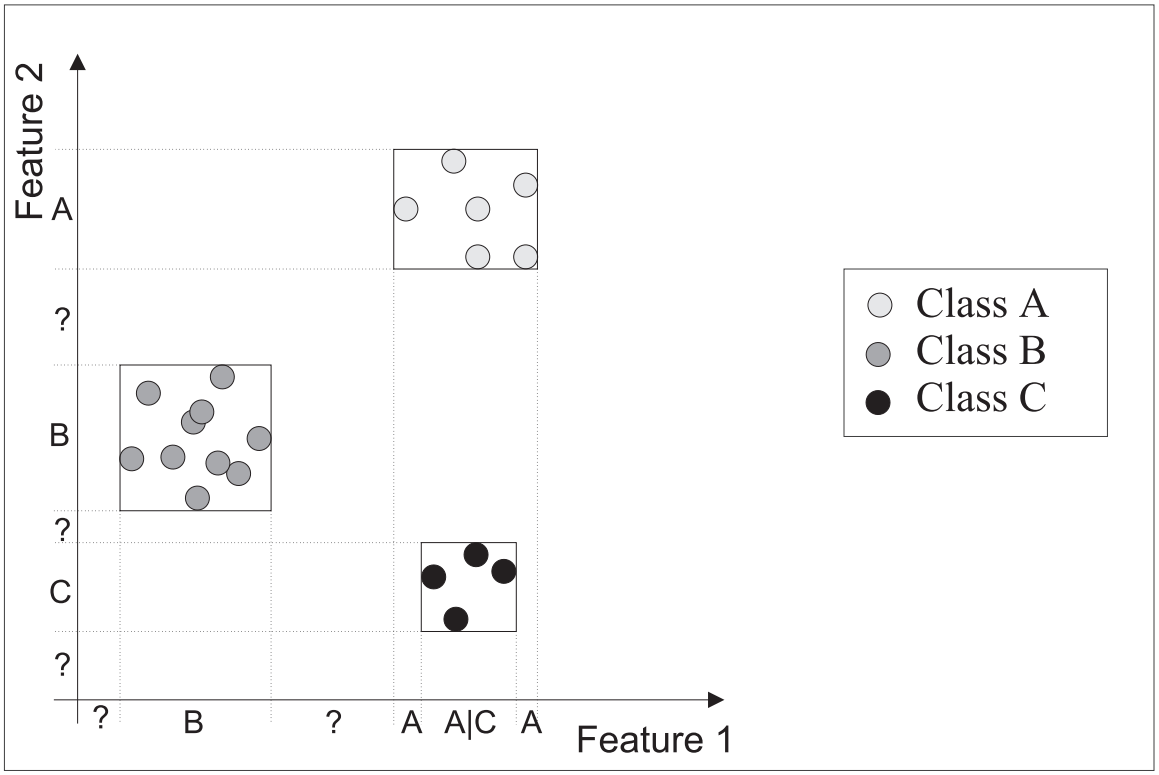

Figure 3.2: Parallelepiped Classification example - Bounds Identification Example

classes in all features in the multidimensional feature space. When there is overlap for a feature as it is shown in figure 3.2 for classes A and C for feature 1, other features can be used to solve this tie. When there is overlap on all features, there is no simple way to determine which class should be assigned. An example of this overlap is shown in figure 3.3 - the area in gray is the overlap area between classes A and B, and pixels inside this area cannot be classified (in this implementation of the classifier pixels on overlap regions will be rejected).

Another problem with the Parallelepiped classifier is the determination of the lower and upper bounds for each class. If the distribution of the values of a feature for a class is sparse, the lower and upper bounds will be separated, increasing the chances of overlap that for that feature and class. This can be avoided by selecting a threshold for the bounds in a way that only values above that threshold will be considered for the bounds. Figure 3.4 shows a more or less sparse distribution of values for a feature and the determination of the lower and upper bounds using the data as is, using a threshold T and using the bounds specified by a distance $\pm D$ from the mean μ of the samples where $D = N \times \sigma$ and σ is the standard deviation for that feature and class.

An advantage of the Parallelepiped classifier is it simplicity and speed, since classification is a matter of comparing the pixel values with the bounds. Also, if we want to classify an image in some few classes and have a general reject class, the Parallelepiped classifier is also useful since all points outside the bounds for the known classes will be assigned to a catch-all reject class.

3.2 Signatures for the Parallelepiped Classifier

Signatures for the Parallelepiped classifier are simply the values of the upper and lower bounds for each feature and can be generated with the operator `cparallel_signature` for each class. In this toolbox approach to classification the signatures for each class will be created separately and joined together (figures 2.5 and 2.6). For the Parallelepiped classifier, the signatures

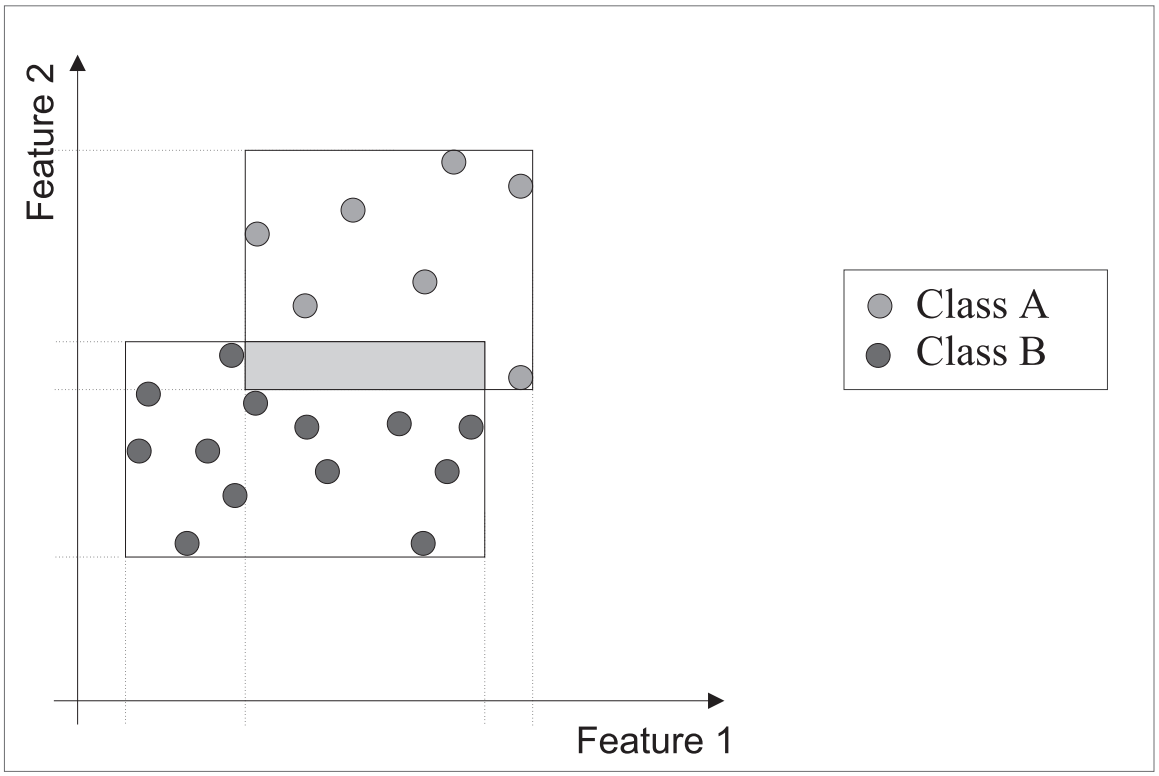

Figure 3.3: Parallelepiped Classification example - Classes Overlap

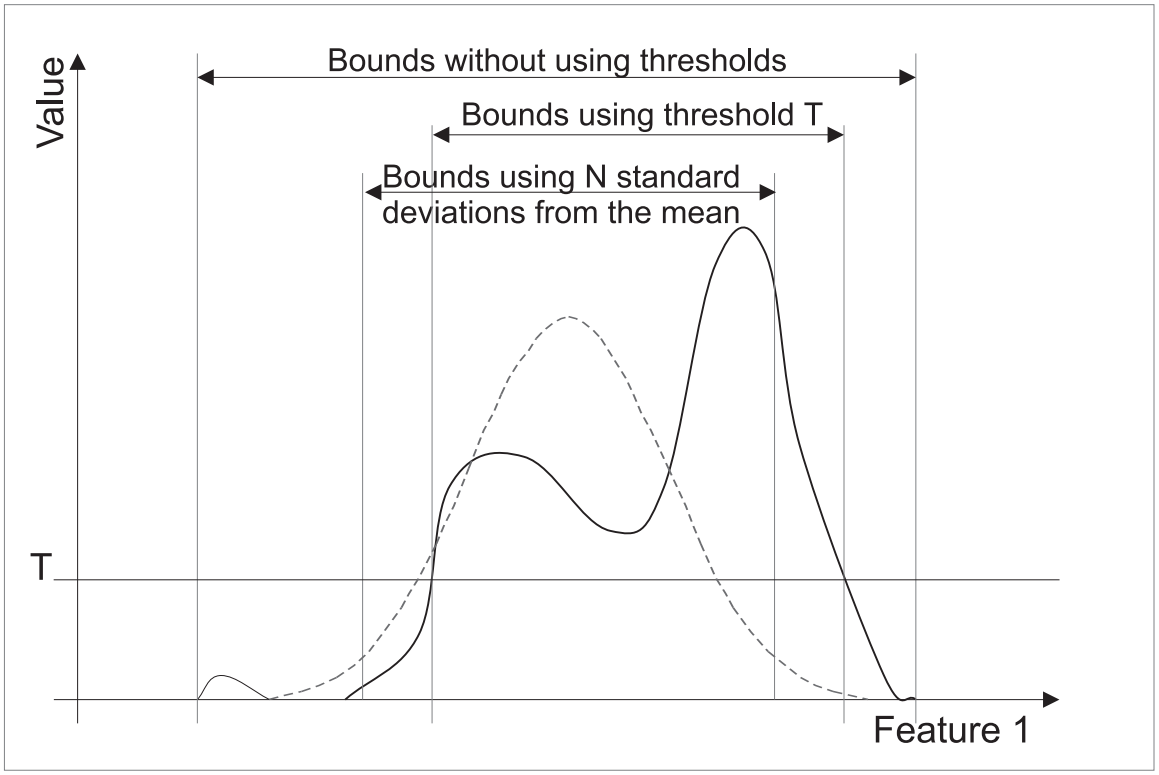

Figure 3.4: Parallelepiped Classification example - Bounds Identification Details

can be joined with the `kappend` operator along the time dimension for classification with the `cparallel_classify` operator.

3.2.1 The `cparallel_signature` kroutine

Object name: cparallel_signature
Icon name: PARAL. Signature
Category: Image Classification
Subcategory: Parallelepiped

3.2.2 Parameters

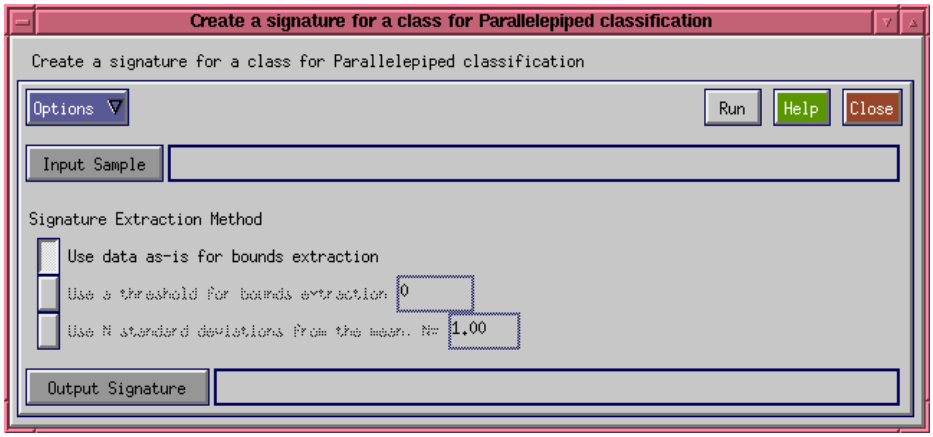

Figure 3.5: The `cparallel_signature` kroutine GUI

The parameters for the `cparallel_signature` kroutine are shown in its GUI pane (Figure 3.5):

- **Input Sample**: The input sample file name. The file can be in any format used by Khoros. The `cparallel_signature` kroutine will not work with objects which depth or time dimensions are different than one or which data type is complex, and the features should be represented along the elements dimension. In other words, for an image of size $W \times H$ with F features the data should have dimensions $(W \times H \times 1 \times 1 \times F)$. Mask information is used, only valid masked points will be considered for the signatures.
- **Signature Extraction Method**: There are three methods for signature extraction: if **Use data as-is for bounds extraction** is selected default), the values for each feature will be scanned for the minimum and maximum values, and the values will be used without further processing. If **Use a threshold for bounds extraction** is selected, a threshold will be used to search for the minimum and maximum values for the feature that are larger than the specified value. This method requires some knowledge of the distribution of the pixels for each feature. If **Use N standard deviations from the mean, N=** is selected, instead of using the values from the samples as they are, the signature will be extracted by calculating the mean value μ of the feature and using $\mu - N \times \sigma$ as the lower bound and $\mu + N \times \sigma$ where σ is the standard deviation for the feature.
- **Output Signature**: The output signature file name. The output file will contain the signature for that sample. The samples object data dimension will be $(F \times 2 \times 1 \times 1 \times 1)$ for F features.

3.2.3 Preparing the Signatures for Classification

As mentioned in section 2.2.1, the signatures must be joined in a single object before classification. For the Parallelepiped classifier, the signatures are of constant size and can be joined

by the `kappend` operator. Please refer to figure 2.7 for more information on the signature appending.

3.3 Image Classification with the Parallelepiped Classifier

With the signatures ready for use, we can classify the image with the Parallelepiped classifier. Classification is done with the `cparallel_classify` kroutine, described in section 3.3.1.

Classification is done by comparing each pixel in the input image with the bounds for each class and feature. A pixel is assigned to a class if all values in the vector x are inside the feature bounds for that class. Pixels will be unclassified if any of the values of the vector is outside the bounds or if more than one class is chosen (see figure 3.3 for details).

3.3.1 The `cparallel_classify` kroutine

Object name: cparallel_classify
Icon name: PARAL. Classify
Category: Image Classification
Subcategory: Parallelepiped

3.3.2 Parameters

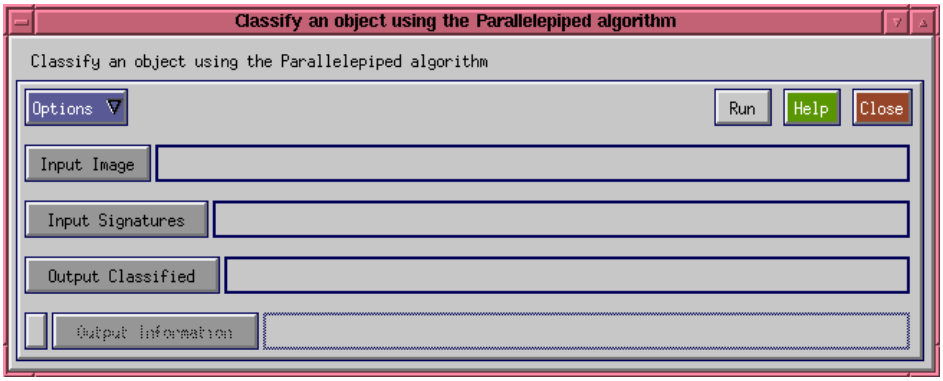

Figure 3.6: The `cparallel_classify` kroutine GUI

The parameters for the `cparallel_classify` kroutine are shown in its GUI pane (Figure 3.6):

- **Input Image**: The file name of the image to be classified, in any format supported by Khoros. It does not needs to be the same file that generated the signatures, although it is common to use the same image for signatures and classification. The `cparallel_classify` operator can process objects with dimensions depth and time > 1. The input image must have the same element dimension as the used for the signatures. In other words, the input image should have dimensions $(W \times H \times D \times T \times E)$ where E should be the same as the input used for the `cparallel_signature` operator.

- **Input Signatures**: The file name of the group of classes signatures. This file is usually obtained from the output of the `kappend` operator or (if a single class is used) from the `cparallel_signature` operator. The input signatures should have dimensions $(F \times 2 \times 1 \times C \times 1)$ where F is the number of features (must be the same as E in the **Input Image** above) and C is the number of classes.

- **Output Classified**: The result of the classification. If there were any masked points in the input, these points will be masked in the output. If any rejection was done, the rejected pixels will also be masked. The output classified result will have dimensions

$(W \times H \times D \times T \times 1)$ and each element will be the index of the class as appended by the `kappend` operator.

- **Output Information**: If chosen, pixel-to-pixel information about the classification process will be send out to this file. For big images, this file can get huge and it slows down the classification process.

3.4 Utilities

3.4.1 Printing the Parallelepiped Signature

The contents of a Parallelepiped Signature (created by the `cparallel_signature` operator, described in section 3.2.1) can be printed for inspection or use in reports. Pure text, LaTeX [12] table, HTML [13] table and CSV formats are supported.

3.4.1.1 The `cparallel_printsig` kroutine

Object name: cparallel_printsig
Icon name: PARAL. Print Signature
Category: Image Classification
Subcategory: Parallelepiped

3.4.1.2 Parameters

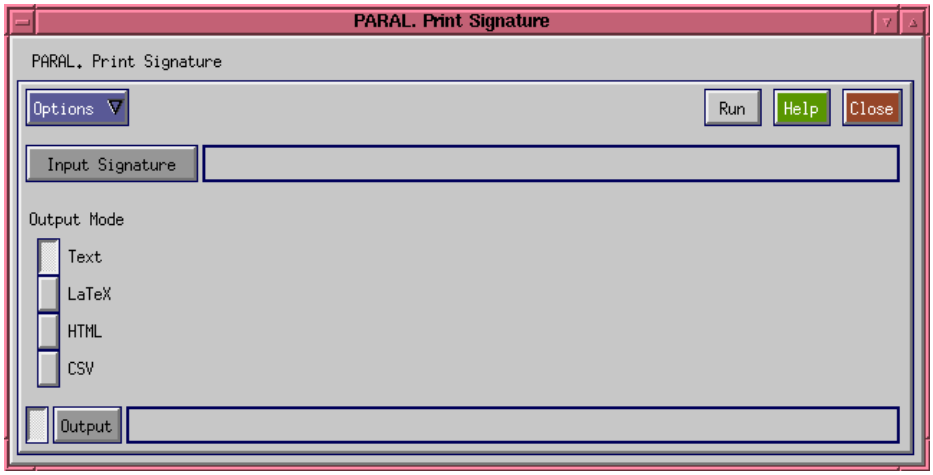

Figure 3.7: The `cparallel_printsig` kroutine GUI

The parameters for the `cparallel_printsig` kroutine are shown in its GUI pane (Figure 3.7):

- **Input**: The input file, generated by `cparallel_signature` or by joining several signatures with `kappend` - both single and multiple signatures are supported and will be printed in a single file. In other words, the input signatures should have dimensions $(F \times 2 \times 1 \times C \times 1)$ where F is the number of features and C is the number of classes.

- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.

- **Output**: The output file. If selected and set to a file, will save the contents of the signature in that file, if unselected will display it on the console.

3.5 Classification Example

To exemplify the Parallelepiped Classification process, we will classify the urban aerial image SJC in three different classes, corresponding to pools, trees and house roofs. Since the Parallelepiped classifier rejects anything outside the feature bounds for each classes, we can expect that a great part of the image will be left unclassified, and will correspond to other classes than the three mentioned above. This is not a very practical classification example but will serve as an example of the features and problems of the Parallelepiped classifier.

3.5.1 Creating a Cantata Workspace

In this example, we will create a Cantata workspace that will contain both parts of the classification process: the Signature Extraction and the Classification itself (see figure 2.6). Remember that we can use the same set of signatures to classify more than one image if the images have the same characteristics, if it is the case it would be better to separate the signature extraction from the classification process.

The workspace to classify the urban aerial image with the Parallelepiped Classifier is shown in figure 3.8.

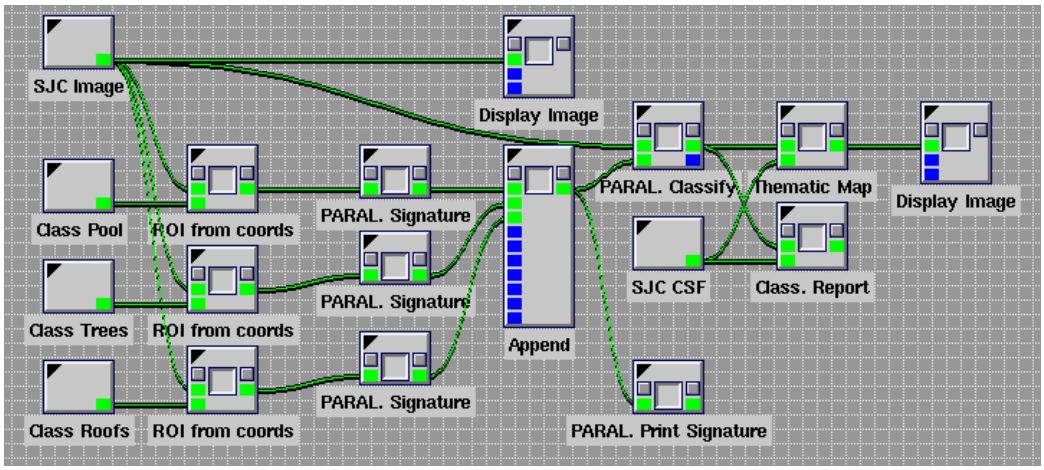

Figure 3.8: A Cantata Workspace for classification with the Parallelepiped Classifier

From the leftmost column in the workspace, we have four `User defined` glyphs (with the glyph names changed), the first containing the Urban Aerial Image SJC and the other three containing the Coords file for the three classes we want to identify in the image (named Class Pool, Class Trees and Class Roofs). These glyphs serve as input to the `ROI from coords` glyphs (for the `cROIfromcoords` kroutine, described in section 10.1.1), which will mark the points inside the coordinates as valid and then input this result to the `PARAL. Signature` glyph (for the `cparallel_signature` kroutine, described in section 3.2.1). In this example, we will use the values of the samples as they are (alternatives are using a threshold or a number of standard deviations from the mean).

All signatures are joined together by the `Append` glyph, which output will serve as input to the `PARAL. Classify` glyph (section 3.3.1). The output of the classifier will serve as input to both a `Thematic Map` glyph (section 11.3) and a `Class.Report` glyph (section 11.4) which will create respectively a thematic map for visual evaluation and a table with the classification results, shown in the next section.

3.5.2 Results for the Classification

Signatures for each of the three classes used in this example are shown in tables 3.1, 3.2 and 3.3. The original image is shown in figure 3.9 and the thematic map result for the classification

workspace above is shown in figure 3.10, with red points representing roof pixels, green points representing tree pixels and blue points representing pool pixels. Areas in white are non-classified or rejected. The report with the pixel count is shown in table 3.4 with the pixel counts for the three classes and for the reject class.

Lower Bounds		
82.000	161.000	165.000
Upper Bounds		
184.000	226.000	224.000

Table 3.1: Parallelepiped Signature for Parallelepiped classification of the urban aerial image (Class 1 of 3)

Lower Bounds		
5.000	5.000	12.000
Upper Bounds		
139.000	133.000	133.000

Table 3.2: Parallelepiped Signature for Parallelepiped classification of the urban aerial image (Class 2 of 3)

Lower Bounds		
204.000	154.000	114.000
Upper Bounds		
237.000	197.000	174.000

Table 3.3: Parallelepiped Signature for Parallelepiped classification of the urban aerial image (Class 3 of 3)

Comparing the classification result in figure 3.10 with the original image we can see that there are matches between the original image and the classification result and the most noticeable classification mistakes are several patches of bare ground streets that got confused with the roofs and dark regions (mostly shadows of houses' walls) that got confused with trees. The pool regions weren't clearly defined, but some of the pixels on it were, and this classification result could serve as a hint for regions which could belong to pools.

As mentioned before, this classifier is not particularly powerful, and classification errors would increase if we had more mutually exclusive classes due to classes overlap in the multidimensional feature space.

The classification results could be enhanced by a simple mode filter (section 11.1.1) or a probabilistic mode filter (section 11.2).

Figure 3.9: Urban Aerial Image

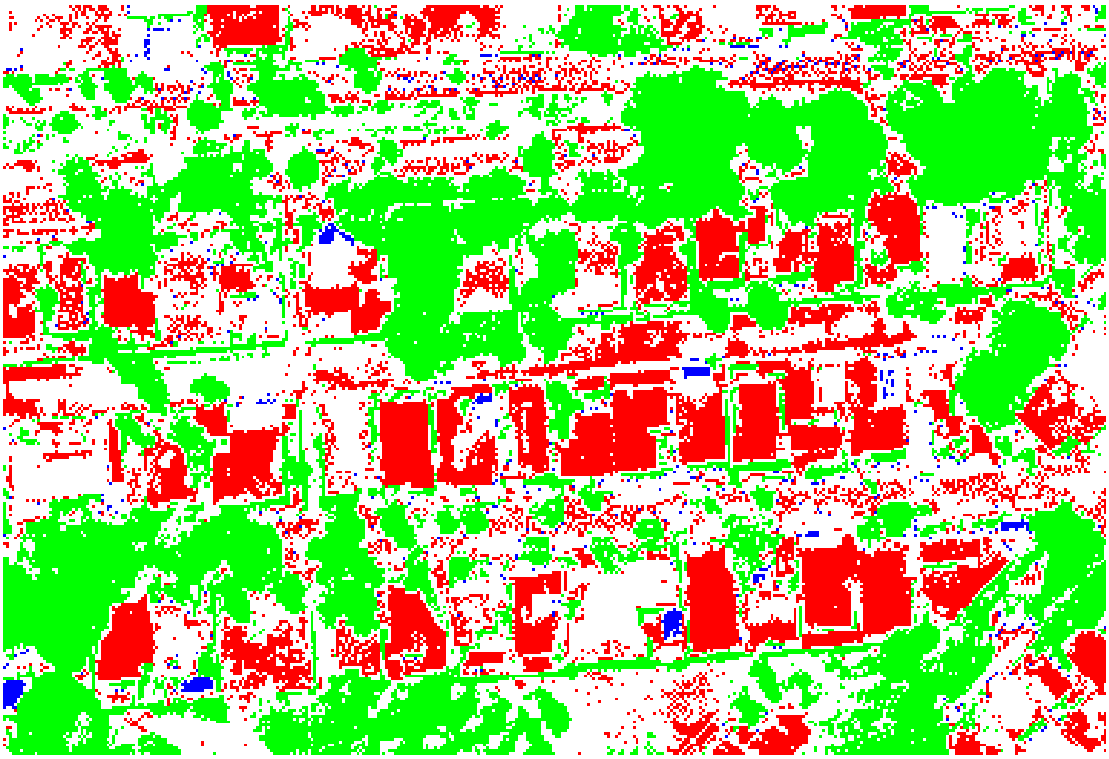

Figure 3.10: Classification of the Urban Aerial Image with the Parallelepiped Classifier

Class	Pixels	%
Rejected or non-classified	44040	43.94
Pool	920	0.92
Trees	33797	33.72
Roofs	21467	21.42
TOTAL	100224	100.00

Table 3.4: Classification report for the Urban Aerial Image with the Parallelepiped Classifier

Chapter 4

K-Nearest Neighbors Classification

4.1 Introduction

The K-Nearest Neighbors Classification is a non-parametric classification method in which pixels are assigned to classes based on the number of neighbor pixels that belongs to the classes. Usually the classification algorithm select the K pixels with known classes which are closest to the pixel x being investigated and selects the class which have more pixels in this neighborhood. K should be odd to avoid ties in the class decision.

Unlike parametric classifiers where we want to estimate the $p(x|\omega_i)$ we can estimate directly the a-posteriori probability $p(\omega_i|x)$ by examining the known pixels' classes that are close to the unknown pixel.

Figure 4.1 shows an example of K-Nearest Neighborhood classification in the 2-dimensional feature space. We have three classes with their samples and we want to determine the classes of unknown pixels 1 and 2. The classification idea is simple and instinctive: to classify a pixel, first we get the K-nearest neighbors of it and inside of this set we choose the class that is most represented. In figure 4.1, the 7-nearest neighbors of each unknown pixel are shown by the lines that connect them. In this example, the unknown pixel 1 will be classified as class B and the unknown pixel 2 will be classified as class A.

One problem with the basic K-Nearest Neighbors algorithm is that for each point to be classified, its K-Nearest neighbors must be found, which is time-consuming. A variation of the basic algorithm is shown in the next section.

4.1.1 A Variation of the K-Nearest Neighbors Classification

The algorithm for the K-Nearest Neighbors Classification must calculate the distance from all signature pixels to the pixel being classified, determine the minimum distance that includes K samples and count the pixels for each class that are inside of this set of size K. For a large number of sample pixels it is very time-consuming.

One variation of this algorithm that is a little faster consist of selecting pixels that are inside a hypersphere of radius R and counting the pixels for each class that are inside of this hypersphere. The drawback is that R must be decided empirically by the user, and for some cases ties and hyperspheres without any points can occur (while this can be used for rejection of pixels).

Figure 4.2 shows the same classification example of figure 4.1 with the modified algorithm. In this case the points which will be considered as the neighbors are the pixels inside (or in this example, touched by) the circle centered in the pixel being classified (with multiple features the circle becomes a hypersphere). For the unknown pixel 1 there are 5 points inside or partially inside the circle, and it should be classified as class B. For the unknown pixel 2 there is only one pixel from class A so it will be assigned to class A. This approach also allows a simple scheme of rejection of pixels: if there isn't any sample inside the hypersphere of radius R, we can reject

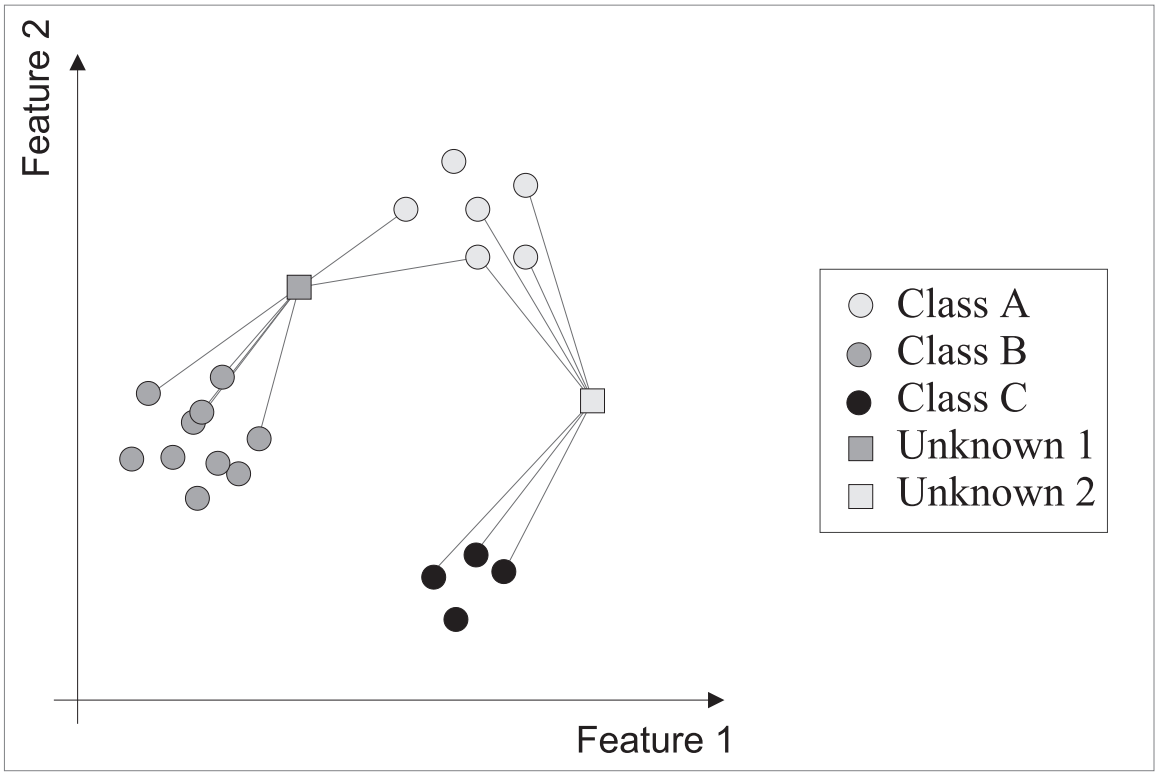

Figure 4.1: K-Nearest Neighbors Classification example in the 2-Dimensional Feature Space

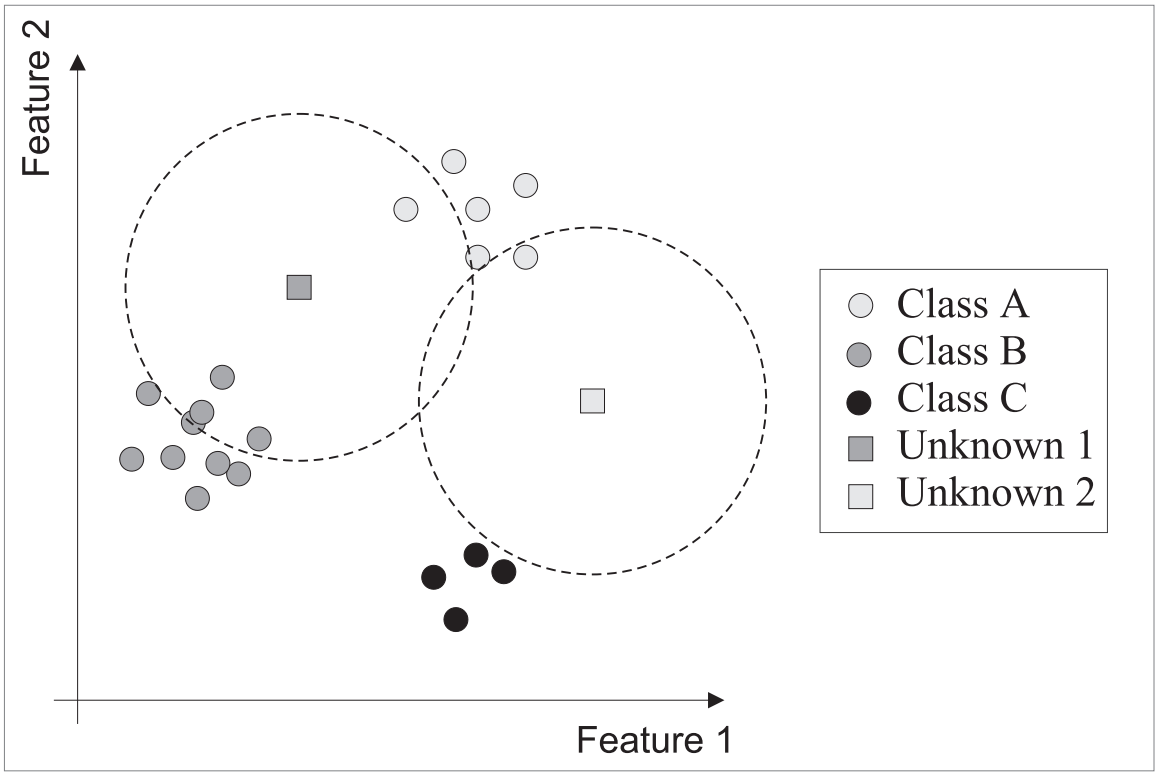

Figure 4.2: K-Nearest Neighbors Classification example in the 2-Dimensional Feature Space - Variation

that pixel (just consider the example above with a smaller radius, the unknown pixel 2 will probably be rejected).

4.2 Signatures for the K-Nearest Neighbors Classifier

The K-Nearest Neighbors Classification is a non-parametric classifier - it uses the values extracted from the samples directly instead of a calculated representation of it. The signatures for each class are the pixels that were selected as samples for that class.

For many classes or classes with many samples the number of points for comparison on the classification step can grow very large, and this can influence the classification time. One way to avoid this is explained in section 4.2.1.

4.2.1 Reducing the dimension of the Signatures

As mentioned above, the dimension of the signature for each class depends on the number of samples for that class. While a large number of samples is desirable, it will slow down the classification process since for classify one point we must first determine which are the k-nearest neighbors of it.

If we also have more samples for one class than for another, some biasing may occur. For example, if the number of points in the feature space that are related to class A is larger than the number of points that is related to class B, it is possible that some biasing for class A over B occur if points for class A are very spread over the feature space.

For those reasons sometimes it is desirable to reduce the number of samples inside one signature for the K-Nearest Neighbors classifier. A simple way to reduce the number of points but preserve their ability to represent the class is by clustering the points and selecting a number of cluster centers. This will work good enough for most classes, if their values are not too spread over the feature space and if the clustering process is iterated enough times.

The operator `cknn_signature` can perform this reduction with some user-defined parameters. It can also create the signature with the pixels values without reduction.

4.2.2 The `cknn_signature` kroutine

Object name: cknn_signature
Icon name: KNN Signature
Category: Image Classification
Subcategory: K-Nearest Neighbors

4.2.3 Parameters

The parameters for the `cknn_signature` kroutine are shown in its GUI pane (Figure 4.3):

- **Input Sample**: The input sample file name. The file can be in any format used by Khoros. The `cknn_signature` kroutine will not work with objects which depth or time dimensions are different than one or which data type is complex, and the features should be represented along the elements dimension. In other words, for an image of size $W \times H$ with F features the data should have dimensions $(W \times H \times 1 \times 1 \times F)$. Mask information is used, only valid masked points will be considered for the signatures.

- **Signature Extraction Method**: The method for signature extraction. If **Use points as they are** is chosen, each point in the input image that is not masked as nonvalid will be used to create the signature. If **Reduce number of points by clustering** is chosen, prior to signature generation the data will be clustered with the k-means algorithm so only some selected points will be used for the signature (the reduction method is described with more detail in section 4.2.1).

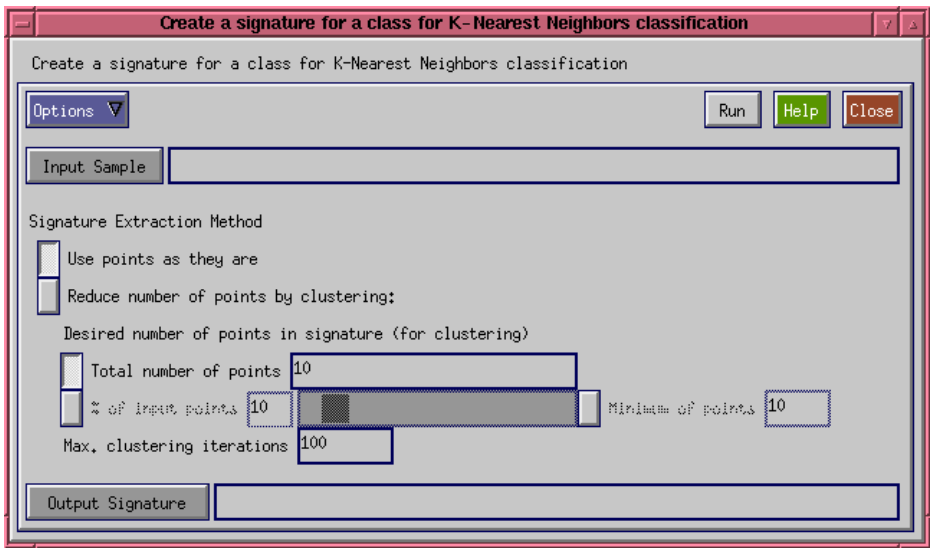

Figure 4.3: The `cknn_signature` kroutine GUI

- **Desired number of points in signature (for clustering)**: If **Reduce number of points by clustering** is chosen above, this value will determine the number of clusters (samples) to be used for the k-means algorithm, which corresponds to the number of signatures created for that class. It can be selected as an absolute value (by selecting **Total number of points** and entering a value for it) or as a value proportional to the number of points in the sample (by selecting **% of input points** and entering a value from 1% to 99%). To avoid subrepresentation of classes when the option **% of input points** is chosen, a minimum number of points can be specified in the **Minimum of points** field.
- **Max.Iterations**: If **Reduce number of points by clustering** is chosen above, this value will determine the maximum number of iterations for the k-means algorithm.
- **Output Signature**: The output signature file name. The output file will contain the signature for that sample. The samples object data dimension will be $(N \times 1 \times 1 \times 1 \times F)$ for F features and N samples.

4.2.4 Preparing the Signatures for Classification

The input to the classification algorithm is the image with the unknown pixels and a set of signatures. In the approach used in this toolbox, for flexibility all signatures should be stored in a single data structure to make this process easier. To correctly identify to which class a point belongs, a *label* must be assigned to this point. Please refer to section 2.2.1 and figure 2.8 for more details.

Labeling and appending signatures for the K-Nearest Neighbors Classifier are done with the operator `csigappend`, described in section 10.5.1. This operator will accept up to ten ROI (regions-of-interest) files and append and label these to create the signature. Multiple copies of this operator can be called when more than 10 classes are to be used. Please refer to section 10.5.1 for more information and usage.

4.3 Image Classification with the K-Nearest Neighbors Classifier

With the signatures ready for use, we can classify the image with the K-Nearest Neighbors classifier. Classification is done with the `cknn_classify` kroutine, described in section 4.3.1.

Basically the classifier counts the number of points in the set of K-Nearest Neighbors (or inside the hypersphere of radius R) and selects the class which is more frequent. Rejection is done only if the hypersphere method is chosen (see section 4.1.1) and the unknown point is too far from any point in the sample set.

4.3.1 The `cknn_classify` kroutine

Object name: cknn_classify
Icon name: KNN Classify
Category: Image Classification
Subcategory: K-Nearest Neighbors

4.3.2 Parameters

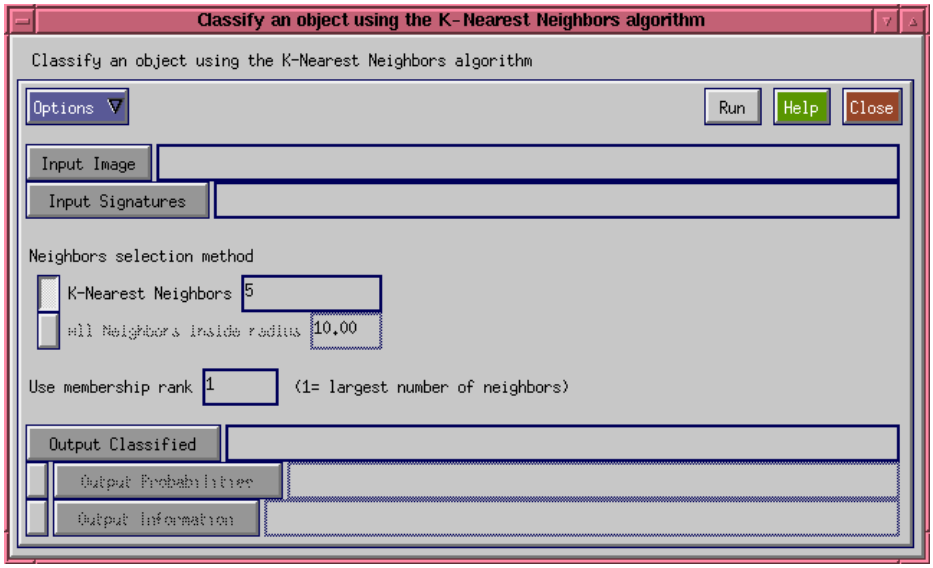

Figure 4.4: The `cknn_classify` kroutine GUI

The parameters for the `cknn_classify` kroutine are shown in its GUI pane (Figure 4.4):

- **Input Image**: The file name of the image to be classified, in any format supported by Khoros. It does not needs to be the same file that generated the signatures, although it is common to use the same image for signatures and classification. The `cknn_classify` operator can process objects with dimensions depth and time > 1, i.e., the input image can have dimensions $(W \times H \times D \times T \times E)$ where E must be the same as the used for input for the `cknn_signature` operator.

- **Input Signatures**: The file name of the group of classes signatures. This file is usually obtained from the output of the `csigappend` operator (described in section 10.5.1), see also section 4.2.4. Its dimensions should be $(N \times 2 \times 1 \times 1 \times E)$ where N is the total number of samples for all classes, joined by the `csigappend` operator.

- **Neighbors selection method**: As described above, we can use the traditional method for classification which search for the K-Nearest Neighbors in the signatures (by selecting **K-Nearest Neighbors** and entering a value for K) or we can select all points inside a hypersphere of radius R (by selecting **All Neighbors Inside radius** and entering a value for R).

- **Use membership rank**: Usually we will be interested only in classifying the pixel to the class with most neighbors, but for evaluation purposes sometimes it is useful to have

the classification with the *second* class with more neighbors and so on. A value larger or equal to 1 and smaller or equal to the number of classes can be provided here to obtain the n-rank classification result. The default is 1 (for most neighbors).

- **Output Classified**: The result of the classification. If there were any masked points in the input, these points will be masked in the output. If any rejection was done, the rejected pixels will also be masked. Depends on the parameter **Use membership rank**. Its dimensions will be $(W \times H \times D \times T \times 1)$ and each element will be the index of the class as appended by the `kappend` operator.
- **Output Probabilities**: If chosen, the neighbor counts for all classes will be saved to this file name, without any ordering. This is useful for post-classification inspection of the membership values (see section 11.7.1 for more information). Its dimensions will be $(W \times H \times D \times T \times C)$ where C is the number of classes.
- **Output Information**: If chosen, pixel-to-pixel information about the classification process will be send out to this file. For big images, this file can get huge and it slows down the classification process.

4.4 Utilities

4.4.1 Printing the K-Nearest Neighbors Signature

The contents of a K-Nearest Neighbor Signature (created by the `cknn_signature` operator, described in section 4.2.2) can be printed for inspection or use in reports. Pure text, LaTeX [12] table and HTML table formats are supported.

4.4.1.1 The `cknn_printsig` kroutine

Object name: cknn_printsig
Icon name: KNN Print Signature
Category: Image Classification
Subcategory: K-Nearest Neighbors

4.4.1.2 Parameters

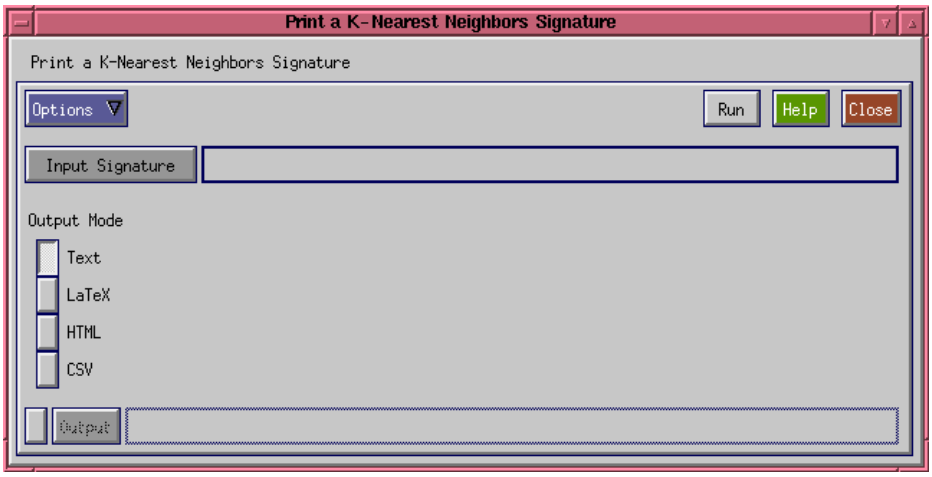

Figure 4.5: The `cknn_printsig` kroutine GUI

The parameters for the `cknn_printsig` kroutine are shown in its GUI pane (Figure 4.5):

- **Input**: The input file, generated by `cknn_signature` or by joining several signatures with `csigappend` - both single and multiple signatures are supported and will be printed in a single file. The input file should have dimensions $(N \times 1 \times 1 \times 1 \times E)$ for non-labeled signatures (label will be printed as -1) or $(N \times 2 \times 1 \times 1 \times E)$ for labeled signatures. See the usage information for the `cknn_signature` operator in section 4.2.2 for more details.

- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.

- **Output**: The output file. If selected and set to a file, will save the contents of the signature in that file, if unselected will display it on the console.

One caveat is necessary: when creating a LaTeX table output, if the number of samples on that signature is large the table that will be generated will be too big to fit in a single page. In this case the user will need to manually split the table in the generated file.

4.5 Classification Example

To exemplify the K-Nearest Neighbors Classification process, we will classify the land use map in six different classes, corresponding to the classes 03, 34, 51, 55, 60 and 63. Variations on the classification process will be presented.

4.5.1 Creating a Cantata Workspace

In this example, we will create a Cantata workspace that will contain both parts of the classification process: the Signature Extraction and the Classification itself. Remember that we can use the same set of signatures to classify more than one image if the images have the same set of classes, if it is the case it would be better to separate the signature extraction from the classification process.

The workspace to classify the land use map image with the K-Nearest Neighbors Classifier is shown in figure 4.6.

From the leftmost column in the workspace, we have seven `User defined` glyphs, the first containing the Land Use Map Image and the other six containing the Coords file for the six classes in the image (named Class 03, Class 34, Class 51, Class 55, Class 60 and Class 63). These glyphs serve as input to the `ROI from coords` glyphs (for the `cROIfromcoords` kroutine, described in section 10.1.1), which will mark the points inside the coordinates as valid and then input this result to the `KNN Signature` glyph (for the `cknn_signature` kroutine, described in section 4.2.2). In this example, instead of using all points in the samples to create the signatures, we chose to cluster the samples using 10% of the number of points.

All signatures are joined together by the `Signature Append` glyph (section 10.5.1). In this example we have less than 10 classes but two `Signature Append` glyphs are used to exemplify the usage for the cases with more than 10 classes. The output to the last `Signature Append` glyph serve as input to the `KNN Classify` glyphs (section 4.3.1). The glyph on the top will use the K-Nearest Neighbors with $K = 5$, while the glyph on the bottom will use all neighbors within a sphere of 10-unit radius. Both classifiers will create results for rank = 1, meaning that the more frequent classes within the neighborhood will be selected.

After classification both Thematic Maps and Confusion Matrices will be generated with the classification results and the "ground truth" for the Land Use Map image. Before using the ground truth data its values must be unmapped with the `Unmap data` glyph (section 12.2.1). Thematic Maps are created with the `cthematicmap` operator, described in section 11.3.1 and Confusion Matrices with the `ccompare` operator, described in section 11.5.1.

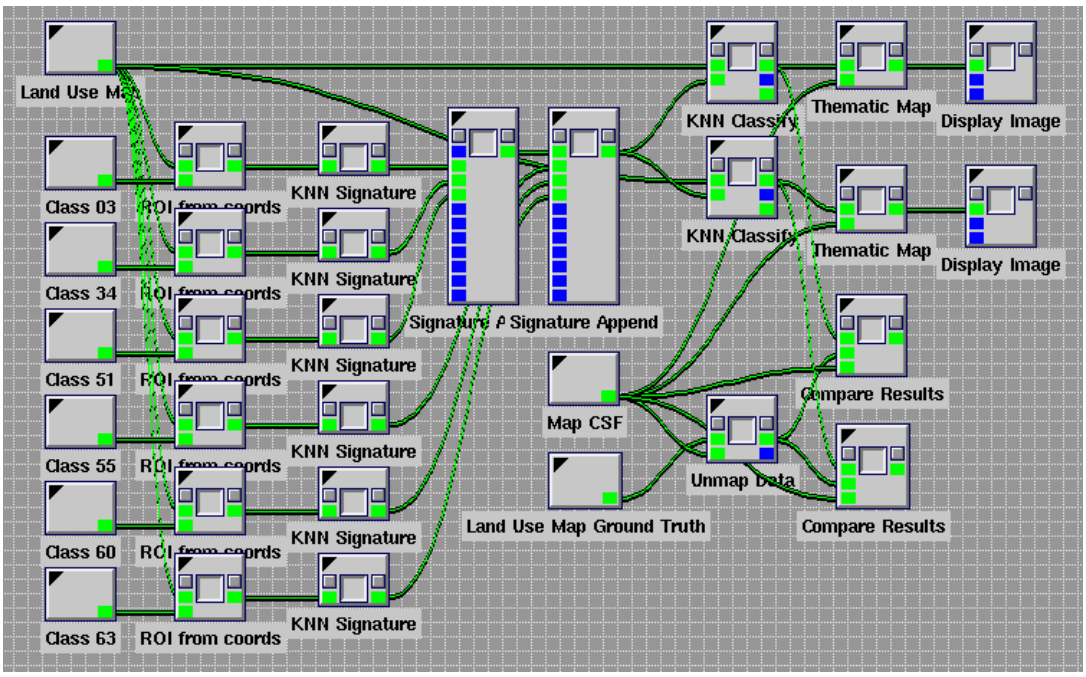

Figure 4.6: A Cantata Workspace for classification with the K-Nearest Neighbors Classifier

4.5.2 Results for the Classification

The results for the classification glyphs above using the image shown in figure 4.7 are shown in figures 4.8 (using the 5-Nearest Neighbors) and 4.9 (using all neighbors inside the hypersphere with 10-unit radius). The confusion matrices for both results are shown in tables 4.1 and 4.2 for the result in figure 4.8 and in tables 4.3 and 4.4 for the result in figure 4.9 (The legend *N/C* on the tables shows the results for non-classified or rejected pixels).

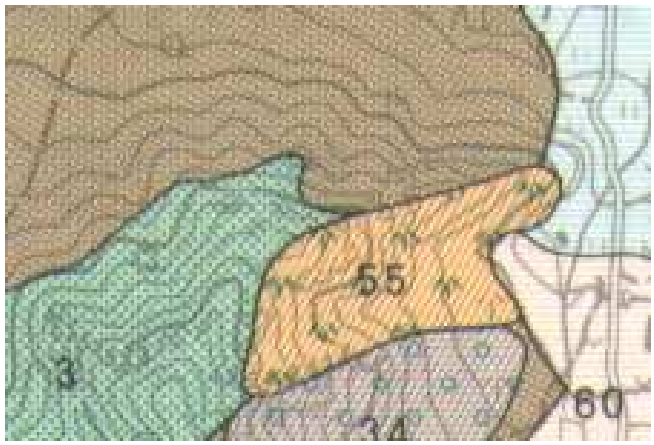

Figure 4.7: Land Use Map

We can see in these classification results that several points are either misclassified (for the classification using the 5-Nearest Neighbors) or misclassified and rejected (for the classification using all neighbors inside the hypersphere with 10-unit radius). Areas in the original image that were misclassified or rejected are the borders, isolines and marks. Spatial information or pre-classification smoothing should improve the classification.

The classification results could be enhanced by a simple mode filter (section 11.1.1) or a probabilistic mode filter (section 11.2).

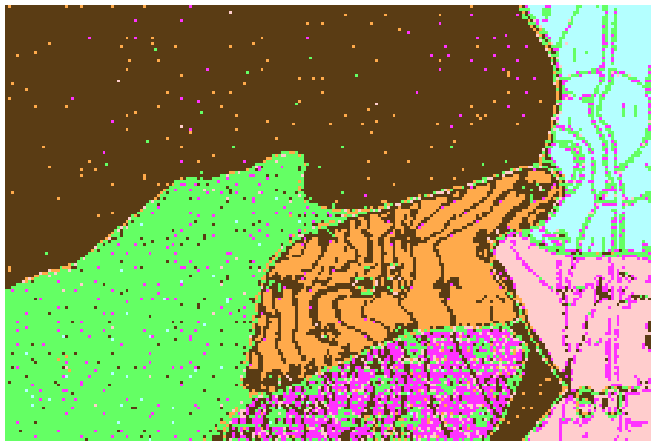

Figure 4.8: Classification of the Land Use Map image with the K-Nearest Neighbors Classifier ($K = 5$)

	Classified as							
Actual Class	0	1	2	3	4	5	6	TOTAL
0. N/C	0	0	0	0	0	0	0	0
1. Map Class 03	0	6583	162	166	66	38	34	7049
2. Map Class 34	0	627	1279	577	79	281	10	2853
3. Map Class 51	0	190	47	13165	211	54	0	13667
4. Map Class 55	0	187	44	1555	2866	74	0	4726
5. Map Class 60	0	61	412	209	32	2007	0	2721
6. Map Class 63	0	659	267	69	22	50	2117	3184
TOTAL	0	8307	2211	15741	3276	2504	2161	34200

Table 4.1: Confusion Matrix for K-Nearest Neighbors Classification with $K = 5$

Class	# Points	G.Truth	Omission	Commission	Correct
N/C	0	0	0.00	0.00	100.00
Map Class 03	8307	7049	6.61	24.46	93.39
Map Class 34	2211	2853	55.17	32.67	44.83
Map Class 51	15741	13667	3.67	18.85	96.33
Map Class 55	3276	4726	39.36	8.68	60.64
Map Class 60	2504	2721	26.24	18.27	73.76
Map Class 63	2161	3184	33.51	1.38	66.49
Overall Accuracy					81.92

Table 4.2: Omission and Commision errors and Accuracy for K-Nearest Neighbors Classification with $K = 5$

	Classified as							
Actual Class	0	1	2	3	4	5	6	TOTAL
0. N/C	0	0	0	0	0	0	0	0
1. Map Class 03	680	6007	177	93	5	36	51	7049
2. Map Class 34	238	519	1340	497	53	195	11	2853
3. Map Class 51	421	47	96	12948	121	34	0	13667
4. Map Class 55	473	15	82	1366	2750	40	0	4726
5. Map Class 60	122	23	448	178	18	1931	1	2721
6. Map Class 63	190	582	218	46	8	26	2114	3184
TOTAL	2124	7193	2361	15128	2955	2262	2177	34200

Table 4.3: Confusion Matrix for K-Nearest Neighbors Classification with $R = 10$

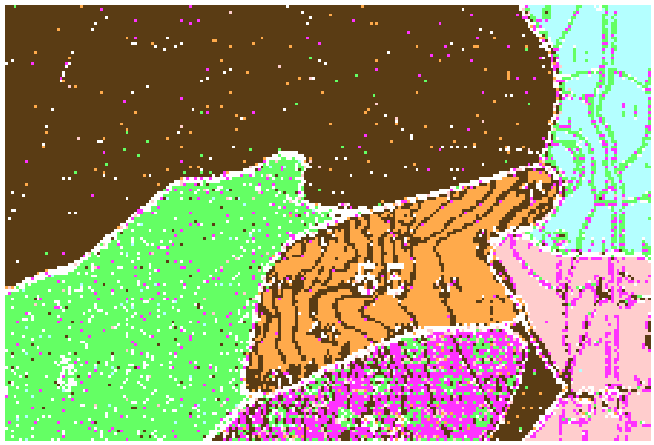

Figure 4.9: Classification of the Land Use Map image with the K-Nearest Neighbors Classifier ($R = 10$)

Points in white are points which were rejected by the classifier and masked.

Class	# Points	G.Truth	Omission	Commission	Correct
N/C	2124	0	0.00	0.00	100.00
Map Class 03	7193	7049	14.78	16.83	85.22
Map Class 34	2361	2853	53.03	35.79	46.97
Map Class 51	15128	13667	5.26	15.95	94.74
Map Class 55	2955	4726	41.81	4.34	58.19
Map Class 60	2262	2721	29.03	12.16	70.97
Map Class 63	2177	3184	33.61	1.98	66.39
Overall Accuracy					79.21

Table 4.4: Omission and Commision errors and Accuracy for K-Nearest Neighbors Classification with $R = 10$

Chapter 5

Minimum Distance Classification

5.1 Introduction

The Minimum Distance Classifier (actually Minimum Distance to Class Means Classifier [2]) is a simple classifier that relies only on the Euclidean distance from the center (or mean, or average) of the samples to the unknown points on the n-feature space. The point is assigned to the class of the sample which center is nearest to it. Figure 5.1 shows a simple classification of pixels in the 2-dim space. In figure 5.1 there are three different classes, and for each one a big

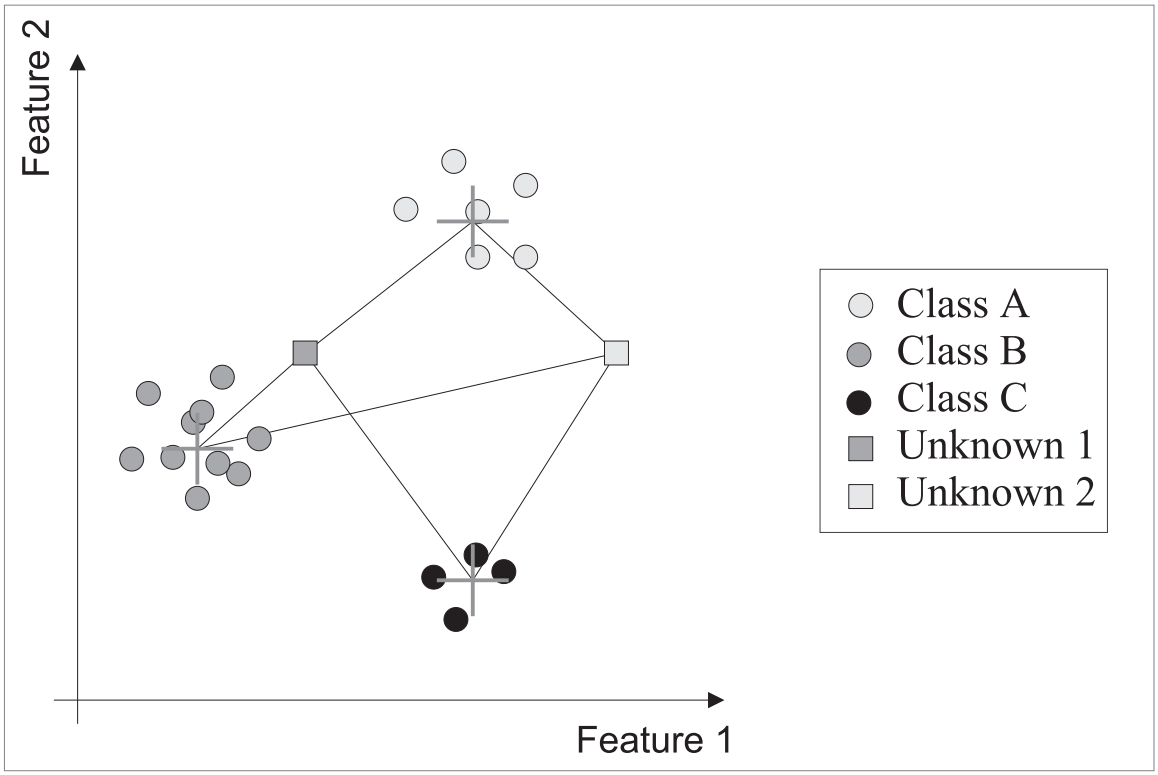

Figure 5.1: Minimum Distance Classification of a pixel in the 2-feature space

gray + sign indicates the center of the classes (the mean value for its samples). An unknown pixel is said to belong to the class which center is nearest. The distances from class center to the unknown pixels in this example can be measured by the length of the lines that connects the pixels to the center of the classes. In other words, the shortest line connects the unknown pixel to the class it belongs.
The main advantages of the Minimum Distance Classifier is its simplicity and speed - both calculation of its signature and distances can be done quickly. Its disadvantage is that it is not

as flexible as other classification methods, since it uses only the means of classes information. Suppose a classification case like the one shown in figure 5.2. The distance from the unknown pixel to the center of class A is smallest than the distance to the center of class B, making the unknown pixel assigned to A where it should maybe be assigned to B since the cluster B is more elongated. Other classifiers such as the Maximum Likelihood Classifier (chapter 6) or the K-Nearest Neighbors Classifier (chapter 4) use different approaches and can avoid this kind of problem.

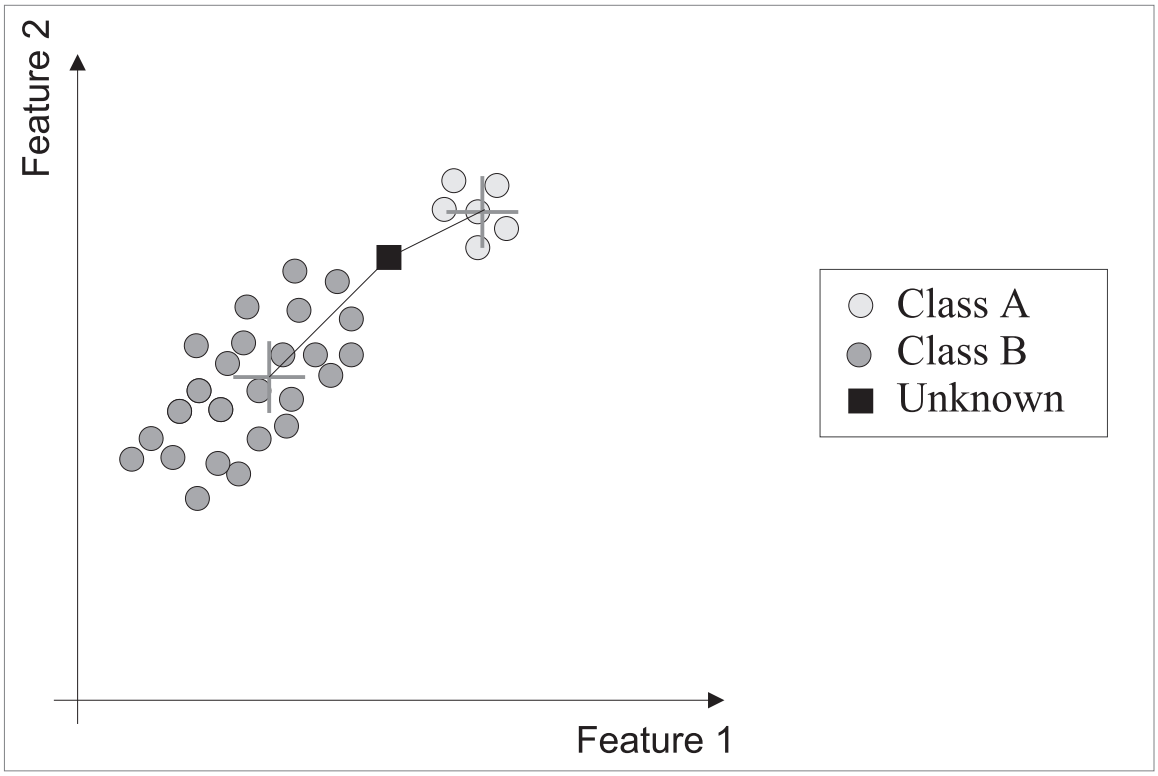

Figure 5.2: Bad Decision Example for the Minimum Distance Classification

5.2 Signatures for the Minimum Distance Classifier

Signatures for the Minimum Distance Classification can be generated from images with the `cmindist_signature` kroutine (described in section 5.2.1) which extracts signatures from the classes samples.

For a Minimum Distance Classifier signature we need only the classes' mean vector - the central point of all sample pixels for that class. The mean vector is a $n-$dimension vector, where n is the number of features (or image bands). The mean vector μ for class i is calculated for each feature as:

$$\mu_i = \frac{1}{N} \sum_{j=1}^{N} x_j \tag{5.1}$$

where N is the number of samples and x_j is th jth sample.

When classifying we will also have the option to reject some points based on their distance to this central point, for this the standard deviations for each feature in the signature will be used. The standard deviation is also a $n-$dimension vector, where n is the number of features (or image bands). The standard deviation σ for class i is calculated for each feature as:

$$\sigma_i = \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(x_j - \mu_i)} \tag{5.2}$$

where N is the number of pixels for the sample.

5.2.1 The `cmindist_signature` kroutine

Object name: cmindist_signature
Icon name: MINDIST Signature
Category: Image Classification
Subcategory: Minimum Distance

5.2.2 Parameters

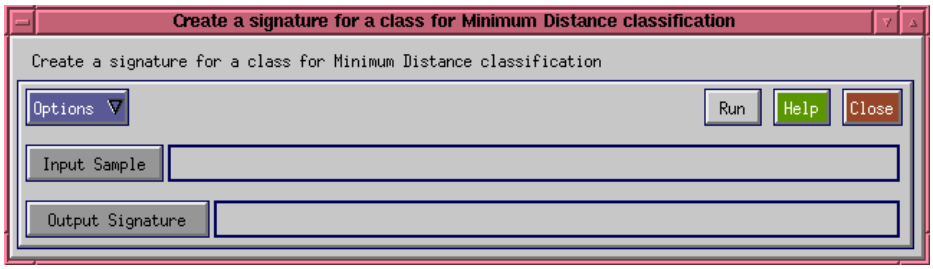

Figure 5.3: The `cmindist_signature` kroutine GUI

The parameters for the `cmindist_signature` kroutine are shown in its GUI pane (Figure 5.3):

- **Input Sample**: The input sample file name. The file can be in any format used by Khoros. The `cmindist_signature` kroutine will not work with objects which depth or time dimensions are different than one or which data type is complex, and the features should be represented along the elements dimension. In other words, for an image of size $W \times H$ with F features the data should have dimensions $(W \times H \times 1 \times 1 \times F)$. Mask information is used, only valid masked points will be considered for the signatures.
- **Output Signature**: The output signature file name. The output file will contain the signature for that sample. The samples object data dimension will be $(F \times 2 \times 1 \times 1 \times 1)$ for F features.

5.3 Image Classification with the Minimum Distance Classifier

After extracting the samples from the classes and its signatures, we can classify the whole image or part of it with the Minimum Distance Classifier. Classification is done with the `cmindist_classify` kroutine, described in section 5.3.1.

Classification is performed on the minimum distance from each pixel to the signatures centers. Basically the classifier assign class c_i to pixel x if:

$$d(x,\mu_i)^2 < d(x,\mu_j)^2 \qquad \text{for all} \qquad j \neq i \tag{5.3}$$

where d is the Euclidean Distance between the vector x and the classes' means.

For some classification tasks rejection of some points which aren't close to any class mean can be desirable. Rejection is done based on a threshold which can be either absolute or based on a number of standard deviations from the classes means. If the distance between a class i

and the pixel being considered is larger than a factor multiplied by σ (eq. 5.2) then the pixel will be rejected if assigned to that class.

5.3.1 The `cmindist_classify` kroutine

Object name: cmindist_classify
Icon name: MINDIST Classify
Category: Image Classification
Subcategory: Minimum Distance

5.3.2 Parameters

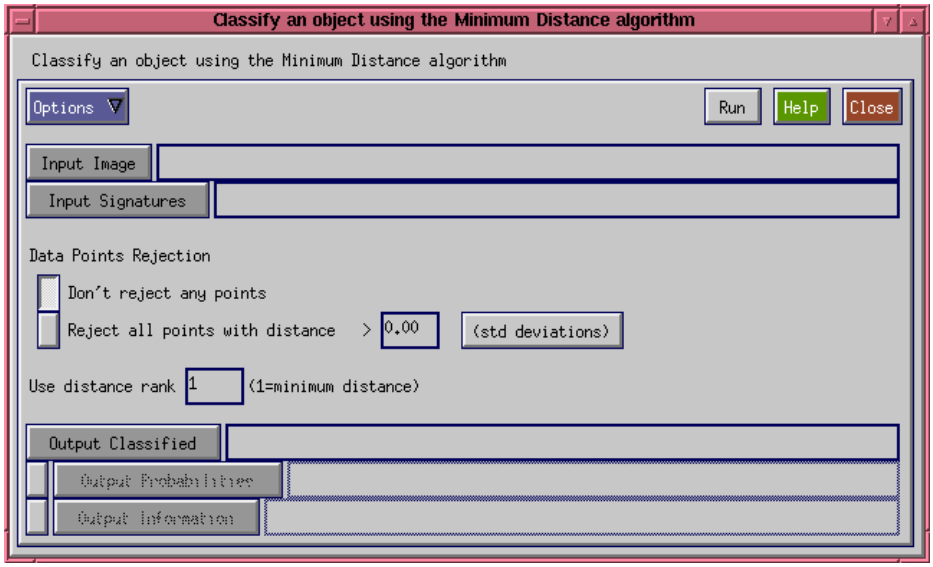

Figure 5.4: The `cmindist_classify` kroutine GUI

The parameters for the `cmindist_classify` kroutine are shown in its GUI pane (Figure 5.4):

- **Input Image**: The file name of the image to be classified. It does not needs to be the same file that generated the signatures, although it is common to use the same image for signatures and classification. The `cmindist_classify` operator can process objects with dimensions depth and time > 1. The input image must have the same element dimension as the used for the signatures. In other words, the input image should have dimensions $(W \times H \times D \times T \times E)$ where E should be the same as the input used for the `cmindist_signature` operator.

- **Input Signatures file**: The file name of the group of classes signatures. This file is usually obtained from the output of the `kappend` operator, please see section 2.2.1 for more information. Its dimensions should be $(F \times 2 \times 1 \times C \times 1)$ for F features and C classes.

- **Data Points Rejection**: If the **Don't reject any points** option is chosen, all pixels will be classified regardless of its distance from the classes centers. If the **Reject all points with distance** is chosen, the user must provide a value > 0.0 for the distance threshold (default is 0.0) and also choose if this threshold should be used as-is (**absolute val**) or if it means the number of standard deviations from each class center **std deviations**. Rejection is done by masking the rejected pixels in the output.

- **Use distance rank**: Usually we will be interested only in classifying the pixel to the class with the minimum distance, but for evaluation purposes sometimes it is useful to

have the classification with the *second* minimum distance and so on. A value larger or equal to 1 and smaller or equal to the number of classes can be provided here to obtain the n-rank minimum classification result. The default is 1 (for minimum distance).

- **Output Classified**: The result of the classification. If there were any masked points in the input, these points will be masked in the output. If any rejection was done, the rejected pixels will also be masked. Depends on the parameter **Use distance rank**. Its dimensions will be $(W \times H \times D \times T \times 1)$.
- **Output Distances**: If chosen, the output distances for all classes will be saved to this file name, without any ordering. This is useful for post-classification inspection of the distance values with the `cxinspector` operator. Its dimensions will be $(W \times H \times D \times T \times C)$ where C is the number of classes.
- **Output Information**: If chosen, pixel-to-pixel information about the classification process will be send out to this file. For big images, this file can get huge and it slows down the classification process, so avoid using it except for tests.

5.4 Utilities

5.4.1 Printing the Minimum Distance Signature

The contents of a Minimum Distance Signature (created by the `cmindist_signature` operator, described in section 5.2.1) can be printed for inspection or use in reports. Pure text, LaTeX [12] table, HTML table and CSV (comma-delimited) formats are supported.

5.4.1.1 The `cmindist_printsig` kroutine

Object name: cmindist_printsig
Icon name: MINDIST Print Signature
Category: Image Classification
Subcategory: Minimum Distance

5.4.1.2 Parameters

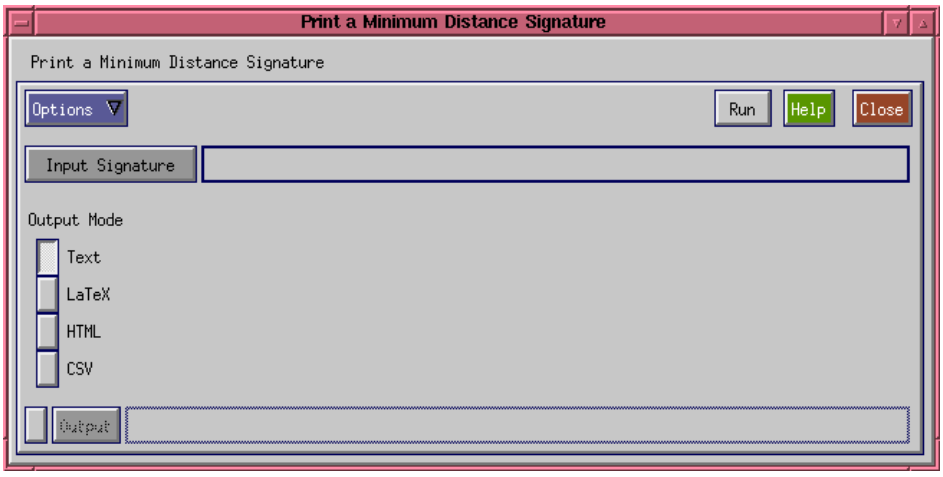

Figure 5.5: The `cmindist_printsig` kroutine GUI

The parameters for the `cmindist_printsig` kroutine are shown in its GUI pane (Figure 5.5):

- **Input**: The input file, generated by `cmindist_signature` or by joining several signatures with `kappend` - both single and multiple signatures are supported and will be printed in a single file. In other words, the input signatures should have dimensions $(F \times 2 \times 1 \times C \times 1)$ where F is the number of features and C is the number of classes.
- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.
- **Output**: The output file. If selected and set to a file, will save the contents of the signature in that file, if unselected will display it on the console.

5.5 Classification Example

To exemplify the Minimum Distance Classification process, we will classify the jelly beans image in five different classes, corresponding to the jelly beans colored red, yellow, orange, purple and green. While it is not a very practical example (unless you hate yellow jelly beans) it will serve as a good example of several points on minimum distance classification.

5.5.1 Creating a Cantata Workspace

In this example, we will create a Cantata workspace that will contain both parts of the classification process: the Signature Extraction and the Classification itself. Remember that we can use the same set of signatures to classify more than one image if the images have the same set of classes, if it is the case it would be better to separate the signature extraction from the classification process.

The workspace to classify the jelly beans image with the Minimum Distance Classifier is shown in figure 5.6.

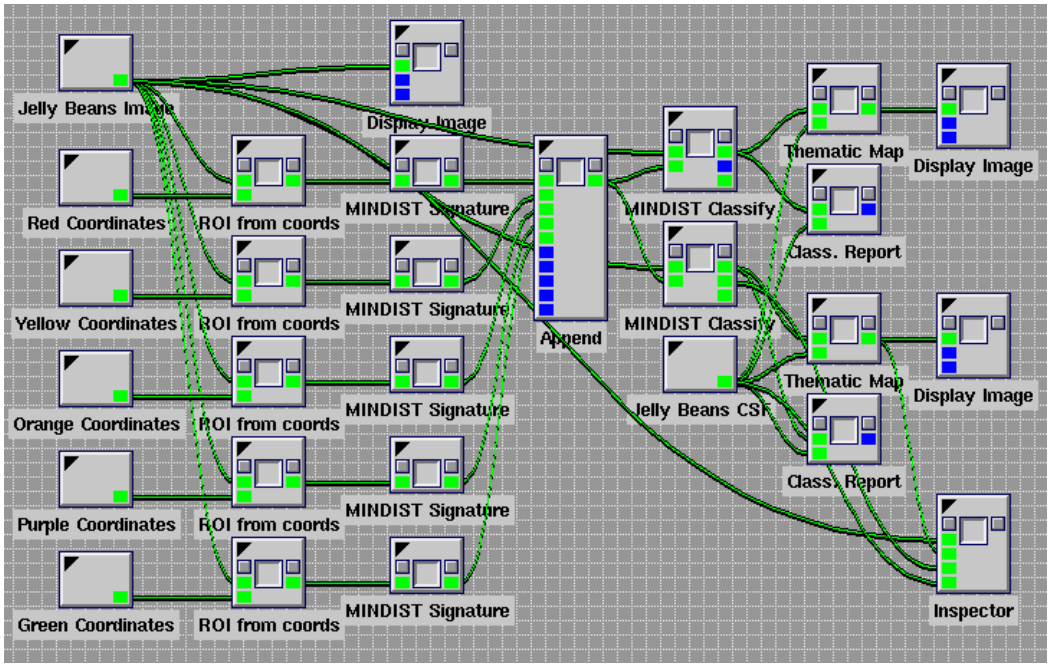

Figure 5.6: A Cantata Workspace for classification with the Minimum Distance Classifier

In the first column at the left, we have six `User defined` glyphs with contains the jelly beans image file name (for the first) and the coordinates for samples for the five classes (Red,

Yellow, Orange, Purple, Green). These glyphs serves as input to the `ROI from coords` glyphs (for the `cROIfromcoords` kroutine, described in section 10.1.1), which will mark the points inside the coordinates as valid and then input this result to the `MINDIST Signature` (for the `cmindist_signature` kroutine, described in section 5.2.1). All signatures are then joined together by the `Append` glyph (`kappend` operator), in the time dimension. The file created by this procedure contains the signatures for all classes (which in this case are the R,G and B means and standard deviations) and are summarized on tables 5.1 (Red), 5.2 (Yellow), 5.3 (Orange), 5.4 (Purple) and 5.5 (Green).

Mean Values		
141.739	15.004	66.530
Standard Deviations		
21.295	35.193	24.782

Table 5.1: Minimum Distance Signature for the Jelly Beans image (class Red)

Mean Values		
169.277	155.718	78.305
Standard Deviations		
11.845	15.255	22.282

Table 5.2: Minimum Distance Signature for the Jelly Beans image (class Yellow)

Mean Values		
154.474	71.211	56.865
Standard Deviations		
6.910	6.389	4.737

Table 5.3: Minimum Distance Signature for the Jelly Beans image (class Orange)

After the signatures are joined together we can use the resulting file and the original image file for classification. Classification is performed by the `MINDIST Classify` glyph (for the `cmindist_classify` kroutine, described in section 5.3.1). In this sample workspace there are two classifier glyphs, the one on the top will classify the whole image without rejection, and the one on the bottom will reject any pixel which distance to any class is larger than 2.0 standard deviations.

After the classification we will create thematic maps and reports for the images with the `cthematicmap` and `cclassreport` operators, described respectively in sections 11.3.1 and 11.4.1. We will also be able to inspect interactively the classification results with the `cxinspector` operator, described in section 11.7.1.

Results for these classifications using the image shown in figure 5.7 are presented as thematic maps in figure 5.8 (without rejected pixels) and figure 5.9 (with pixels rejected if their distances to the signatures were > 2.0). We can see in figure 5.8 that the classifier assigned the class Purple to every pixel which is more or less dark or blue (the majority of the background) while in figure 5.9 most of the background, blue pixels and very light pixels (bright spots in the jelly beans) were rejected.

Reports for the classification result shown in figure 5.8 and figure 5.9 are shown in tables 5.6 and 5.7.

5.5.2 Classification using the Second-minimum Distance

As mentioned in section 5.3.1, it is possible to obtain the Distance Classification results for ranks other than one. We will classify the jelly beans image and obtain the rank 2 classification, i.e.

Mean Values		
81.569	57.948	83.478
Standard Deviations		
8.168	9.825	9.271

Table 5.4: Minimum Distance Signature for the Jelly Beans image (class Purple)

Mean Values		
104.136	140.869	80.759
Standard Deviations		
23.555	20.690	26.499

Table 5.5: Minimum Distance Signature for the Jelly Beans image (class Green)

Figure 5.7: The Jelly Beans Image

Figure 5.8: The Jelly Beans Image classified with the Minimum Distance Classifier - no rejections

Figure 5.9: The Jelly Beans Image classified with the Minimum Distance Classifier - with rejections
(The rejected areas are shown in white, all samples which distances from the signatures were > 2.0 standard deviations were rejected)

Class	Pixels	%
Rejected or non-classified	0	0.00
Red Jelly Beans	2395	12.14
Yellow Jelly Beans	2671	13.54
Orange Jelly Beans	1492	7.56
Purple Jelly Beans	10716	54.33
Green Jelly Beans	2449	12.42
TOTAL	19723	100.00

Table 5.6: Classification report for the Jelly Beans image (no rejection)

Class	Pixels	%
Rejected or non-classified	8964	45.45
Red Jelly Beans	2235	11.33
Yellow Jelly Beans	2067	10.48
Orange Jelly Beans	865	4.39
Purple Jelly Beans	3245	16.45
Green Jelly Beans	2347	11.90
TOTAL	19723	100.00

Table 5.7: Classification report for the Jelly Beans image (with rejection)

the second least distance classification result. The results could be interpreted as the second most likely class for the pixels.

Figure 5.10 corresponds to figure 5.8 but using rank 2. Note that while some areas make sense for the second guess (e.g. the second nearest class for green pixels is yellow and vice-versa), others are very different from the expected (e.g. some areas where the second guess for purple were green), considering the nature of the samples (spectral distance).

Similarly, figure 5.11 corresponds to the second least distance for figure 5.9. Note that in this case there are considerably fewer classified pixels since the distances to the signatures is larger than for the pixels in figure 5.9.

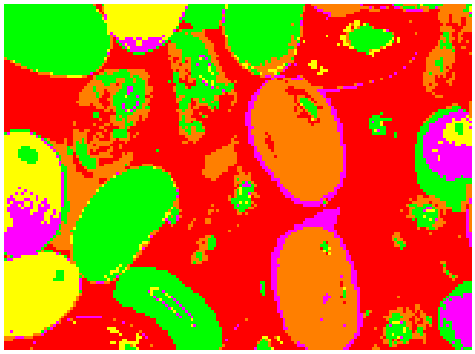

Figure 5.10: The Jelly Beans Image classified with the Minimum Distance Classifier - Rank 2, no rejections

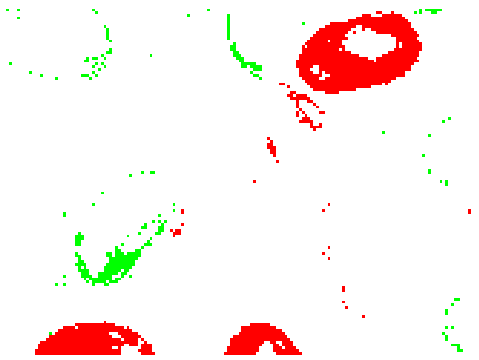

Figure 5.11: The Jelly Beans Image classified with the Minimum Distance Classifier - Rank 2, with rejections
(The rejected areas are shown in white, all samples which distances from the signatures were > 2.0 standard deviations were rejected)

Chapter 6

Maximum Likelihood Classification

6.1 Introduction

The Maximum Likelihood Classifier is a method commonly used in Remote Sensing applications. It uses information about the distribution of the samples to compute a distance measure that is used to determine the most likely class for a pixel. Its decision surfaces can take the form of parabolas, circles and ellipses, as shown in figure 6.1. In that figure, the unknown pixel will be classified taking in account not only the center of the samples for the classes but also the distribution of the pixels of these classes.

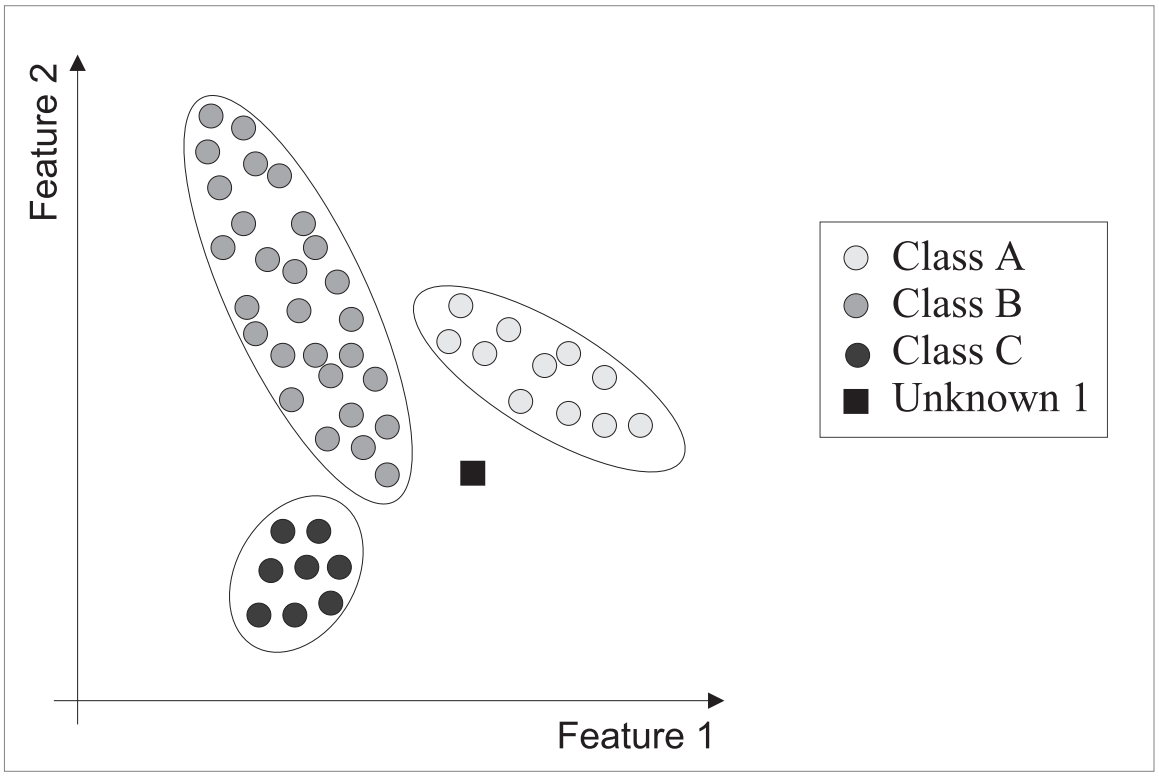

Figure 6.1: Minimum Distance Classification of a pixel in the 2-feature space

The Maximum Likelihood classifier assumes that the classes are unimodal and normally distributed. Its discriminant function is given by:

$$g_i(x) = \ln p(\omega_i) - \frac{1}{2}\ln|\Sigma_i| - \frac{1}{2}(x-\mu_i)^t\Sigma_i^{-1}(x-\mu_i) \tag{6.1}$$

where $p(\omega_i)$ is the a-priori probability for class i, μ_i and Σ_i are the mean and covariance matrix for the data of class i and $|\Sigma_i|$ is the determinate of the covariance matrix. Classification is

done by choosing the maximum $g(x)$ for all classes $i \in N$.

The a-priori probabilities $p(\omega)$ aren't always available or determinable, and can be considered equal for all classes. If all covariance matrices Σ_i are equal, equation 6.1 can be simplified to

$$g_i(x) = \frac{1}{2}(x - \mu_i)^t \Sigma^{-1}(x - \mu_i) \tag{6.2}$$

which is known as the *Mahalanobis Distance* classifier.

The Maximum Likelihood classifier will attempt to classify a pixel regardless of its likelihood. Since we are assuming normal distributions for the classes, the tails for the histograms for the classes will have very low values, and pixels could be assigned to those classes under certain conditions. Figure 6.2 (after Richards [2], pp. 185) shows the one-dimensional distribution for some classes and some points on it. In figure 6.2 the pixels represented by the letters s, u and v are close to the tails of the classes distributions, meaning that the probability of the pixels belonging to the classes is very small. The pixel represented by t is between two classes, and should be classified to one of them.

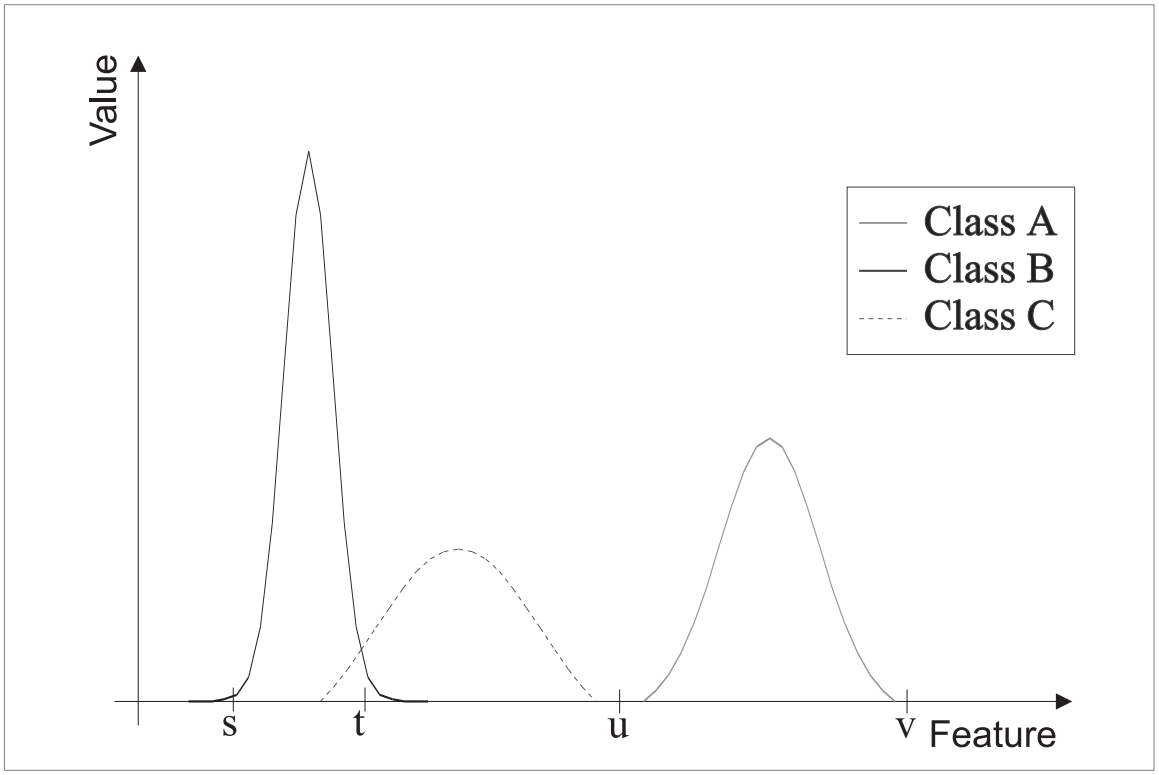

Figure 6.2: Classification of pixels near the tails of the distributions

When the probabilities are small, a threshold could be applied so only pixels with probabilities (or likelihoods) above it would be classified, and this threshold could depend on an accuracy factor. The threshold T_i for class i could be calculated as:

$$T_i = -X - \frac{1}{2}\ln|\Sigma_i| + \ln p(\omega_i) \tag{6.3}$$

where X is obtained from a chi-square table, and X is the value for which $C_N(\chi^2)$ equals the percentage of points we want to be classified (not rejected) and N is the number of features in the image (degrees of freedom in the χ^2 table). Using a threshold for classification a pixel x will be classified as class i if

$g_i(x) > g_j(x)$ for all $i \neq j$ and $g_i(x) > T_i$

A more complete explanation of the Maximum Likelihood classifier is given in [2] and [6].

6.2 Signatures for the Maximum Likelihood Classifier

Signatures for the Maximum Likelihood Classification can be generated from images with the `cmaxlik_signature` kroutine (described in section 6.2.1) which extracts signatures from the classes samples.

For a Maximum Likelihood Classifier signature we need some of the components shown in equation 6.1: the classes' mean vector μ and inverse of the covariance matrix Σ^{-1}. To help with the analysis of the signature's distributions and speed calculation of the likelihoods, the covariance matrix Σ and the negative of the logarithm of its determinate will also be stored in the signature. The data will be stored in separate planes in the depth direction, one for the mean, one for the covariance matrix, one for the inverse of the covariance matrix and one for the negative logarithm of the determinate of the covariance matrix.

6.2.1 The `cmaxlik_signature` kroutine

Object name: cmaxlik_signature
Icon name: MAXLIK Signature
Category: Image Classification
Subcategory: Maximum Likelihood

6.2.2 Parameters

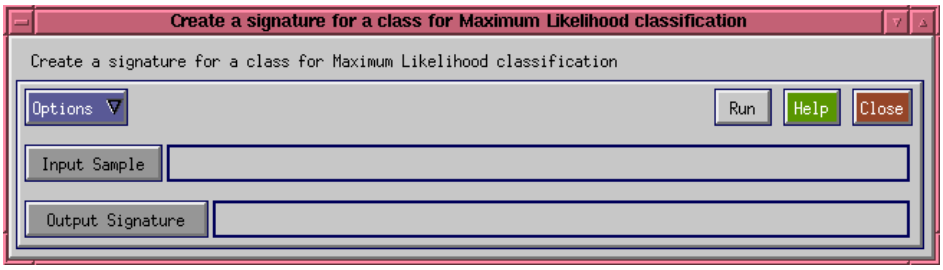

Figure 6.3: The `cmaxlik_signature` kroutine GUI

The parameters for the `cmaxlik_signature` kroutine are shown in its GUI pane (Figure 6.3):

- **Input Sample**: The input sample file name. The file can be in any format used by Khoros. The `cmaxlik_signature` kroutine will not work with objects which depth or time dimensions are different than one or which data type is complex, and the features should be represented along the elements dimension. In other words, for an image of size $W \times H$ with F features the data should have dimensions $(W \times H \times 1 \times 1 \times F)$. Mask information is used, only valid masked points will be considered for the signatures.

- **Output Signature**: The output signature file name. The output file will contain the signature for that sample. The samples object data dimension will be $(F \times F \times 4 \times 1 \times 1)$ for F features.

6.3 Image Classification with the Maximum Likelihood Classifier

After extracting the samples from the classes and its signatures, we can classify the whole image or part of it with the Maximum Likelihood Classifier. Classification is done with the `cmaxlik_classify` kroutine, described in section 6.3.1.

6.3.1 The `cmaxlik_classify` kroutine

Object name: cmaxlik_classify
Icon name: MAXLIK Classify
Category: Image Classification
Subcategory: Maximum Likelihood

6.3.2 Parameters

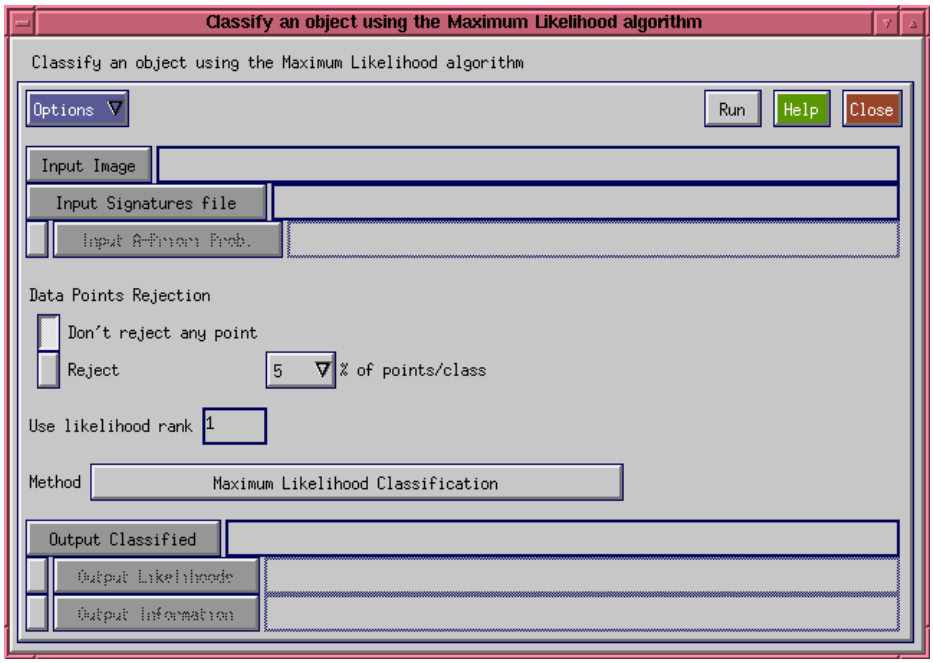

Figure 6.4: The `cmaxlik_classify` kroutine GUI

The parameters for the `cmaxlik_classify` kroutine are shown in its GUI pane (Figure 6.4):

- **Input Image**: The file name of the image to be classified. It does not needs to be the same file that generated the signatures, although it is common to use the same image for signatures and classification. The `cmaxlik_classify` operator can process objects with dimensions depth and time > 1. The input image must have the same element dimension as the used for the signatures. In other words, the input image should have dimensions $(W \times H \times D \times T \times E)$ where E should be the same as the input used for the `cmaxlik_signature` operator.

- **Input Signatures file**: The file name of the group of classes signatures. This file is usually obtained from the output of the `kappend` operator, please see section 2.2.1 for more information. Its dimensions should be $(F \times F \times 4 \times C \times 1)$ for F features and C classes.

- **Data Points Rejection**: If the **Don't reject any points** option is chosen, all pixels will be classified regardless of its likelihoods. If the **Reject** is chosen, the user can select a rejection percentage value for calculation of the threshold in equation 6.3. Possible percentage values are $0.1, 0.5, 1, 2.5, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 99$.

- **Use likelihood rank**: Usually we will be interested only in classifying the pixel to the class with the maximum likelihood, but for evaluation purposes sometimes it is useful to have the classification with the *second* maximum likelihood and so on. A value larger or equal to 1 and smaller or equal to the number of classes can be provided here to obtain

the n-rank maximum likelihood classification result. The default is 1 (for maximum likelihood).

- **Method**: If **Maximum Likelihood Classification** is selected, classification will be done using the equation 6.1. If **Minimum Mahalanobis Distance Classification** is selected, classification will be done using the equation 6.2.
- **Output Classified**: The result of the classification. If there were any masked points in the input, these points will be masked in the output. If any rejection was done, the rejected pixels will also be masked. Depends on the parameter **Use likelihood rank**. Its dimensions will be $(W \times H \times D \times T \times 1)$.
- **Output Likelihoods**: If chosen, the output likelihoods for all classes will be saved to this file name, without any ordering. This is useful for post-classification inspection of the distance values with the `cxinspector` operator. Its dimensions will be $(W \times H \times D \times T \times C)$ where C is the number of classes.
- **Output Information**: If chosen, pixel-to-pixel information about the classification process will be send out to this file. For big images, this file can get huge and it slows down the classification process, so avoid using it except for tests.

6.4 Utilities

6.4.1 Printing the Maximum Likelihood Signature

The contents of a Maximum Likelihood Signature (created by the `cmaxlik_signature` operator, described in section 6.2.1) can be printed for inspection or use in reports. Pure text, LaTeX [12] table, HTML table and CSV (comma-delimited) formats are supported.

6.4.1.1 The `cmaxlik_printsig` kroutine

Object name: cmaxlik_printsig
Icon name: MAXLIK Print Signature
Category: Image Classification
Subcategory: Maximum Likelihood

6.4.1.2 Parameters

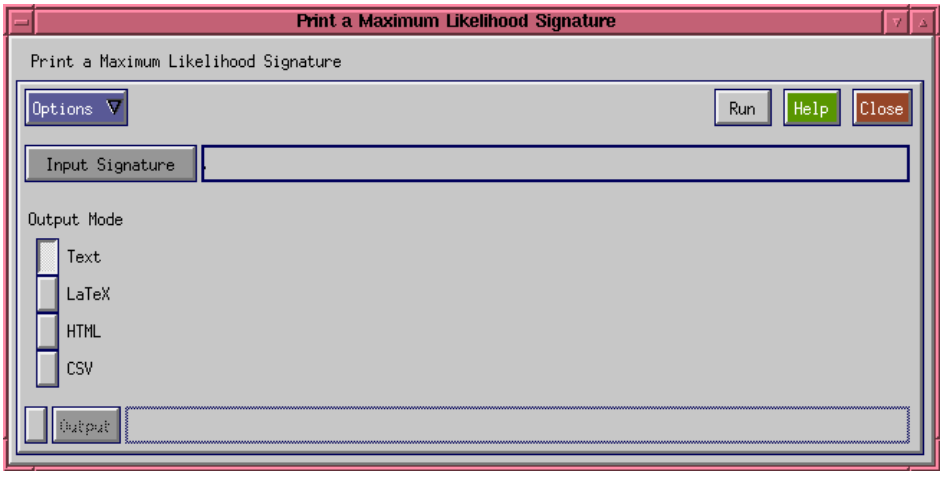

Figure 6.5: The `cmaxlik_printsig` kroutine GUI

The parameters for the `cmaxlik_printsig` kroutine are shown in its GUI pane (Figure 6.5):

- **Input**: The input file, generated by `cmaxlik_signature` or by joining several signatures with `kappend` - both single and multiple signatures are supported and will be printed in a single file. In other words, the input signatures should have dimensions $(F \times F \times 4 \times C \times 1)$ where F is the number of features and C is the number of classes.
- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.
- **Output**: The output file. If selected and set to a file, will save the contents of the signature in that file, if unselected will display it on the console.

6.5 Classification Example

6.5.1 Creating a Cantata Workspace

As we did in other examples, we will create a Cantata workspace that will contain both parts of the classification process: the Signature Extraction and the Classification itself. Remember that we can use the same set of signatures to classify more than one image if the images have the same set of classes, if it is the case it would be better to separate the signature extraction from the classification process.

The workspace to classify the Para Landsat TM-5 image (bands 1-5 and 7) with the Maximum Likelihood Classifier is shown in figure 6.6.

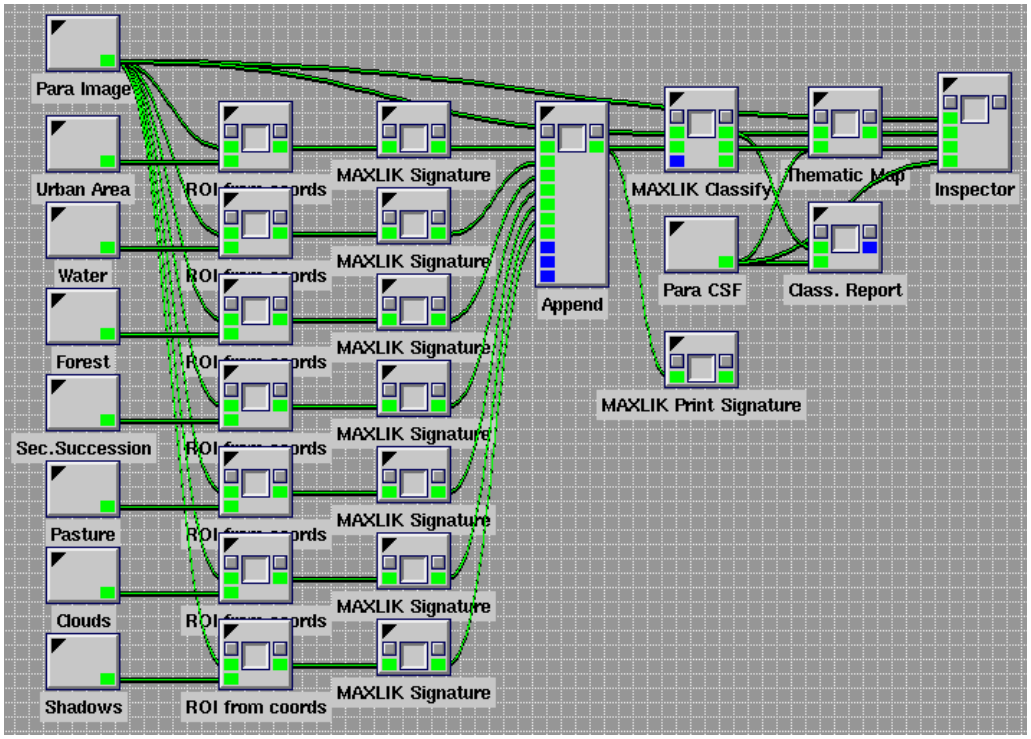

Figure 6.6: A Cantata Workspace for classification with the Maximum Likelihood Classifier

In the first column at the left, we have eight `User defined` glyphs with contains the Para image file name (for the first) and the coordinates for samples for the seven classes (Urban Area, Water, Forest, Secondary Succession, Pasture, Clouds and Shadows). These glyphs serves as input to the `ROI from coords` glyphs (for the `cROIfromcoords` kroutine, described in section 10.1.1), which will mark the points inside the coordinates as valid and then input this re-

sult to the `MAXLIK Signature` (for the `cmaxlik_signature` kroutine, described in section 6.2.1). All signatures are then joined together by the `Append` glyph (`kappend` operator), in the time dimension.

The file created by this procedure contains the signatures for all classes which are shown in tables 6.1 (Urban Area), 6.2 (Water), 6.3 (Forest), 6.4 (Secondary Succession), 6.5 (Pasture), 6.6 (Clouds) and 6.7 (Shadows).

Mean Values					
94.387	51.006	69.387	72.826	139.749	62.980
Covariance Matrix					
657.678	370.455	524.884	248.808	560.960	399.262
370.455	239.337	343.521	150.581	344.396	241.652
524.884	343.521	532.387	226.759	520.557	364.342
248.808	150.581	226.759	127.453	248.331	168.505
560.960	344.396	520.557	248.331	901.571	544.729
399.262	241.652	364.342	168.505	544.729	402.682
Inverse of Covariance Matrix					
0.014	-0.023	0.005	-0.008	0.001	-0.003
-0.023	0.097	-0.043	0.005	-0.001	0.002
0.005	-0.043	0.031	-0.011	0.000	-0.003
-0.008	0.005	-0.011	0.040	-0.003	0.002
0.001	-0.001	0.000	-0.003	0.006	-0.008
-0.003	0.002	-0.003	0.002	-0.008	0.017
-Log(Determinate(Covar.Matrix))					-26.827

Table 6.1: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Urban Area)

Mean Values					
52.560	19.491	16.329	10.485	3.529	0.308
Covariance Matrix					
1.757	0.096	0.130	0.234	0.114	0.044
0.096	0.468	0.208	0.460	0.238	0.050
0.130	0.208	1.738	0.854	0.527	0.103
0.234	0.460	0.854	2.888	1.500	0.315
0.114	0.238	0.527	1.500	2.600	0.246
0.044	0.050	0.103	0.315	0.246	0.388
Inverse of Covariance Matrix					
0.579	-0.083	-0.020	-0.027	0.005	-0.031
-0.083	2.566	-0.122	-0.369	0.005	0.006
-0.020	-0.122	0.682	-0.163	-0.031	-0.010
-0.027	-0.369	-0.163	0.620	-0.267	-0.241
0.005	0.005	-0.031	-0.267	0.556	-0.129
-0.031	0.006	-0.010	-0.241	-0.129	2.863
-Log(Determinate(Covar.Matrix))					-0.611

Table 6.2: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Water)

After the signatures are joined together we can use the resulting file and the original image file for classification. Classification is performed by the `MAXLIK Classify` glyph (for the `cmaxlik_classify` kroutine, described in section 6.3.1). For this example classification is done using rank 1 (maximum likelihood) with the Maximum Likelihood algorithm and without using

Mean Values					
52.698	21.526	18.521	62.753	45.612	8.395
Covariance Matrix					
2.681	0.792	0.949	2.086	2.047	0.801
0.792	1.460	0.906	2.978	1.883	0.640
0.949	0.906	1.936	1.964	2.126	0.819
2.086	2.978	1.964	33.834	8.148	1.519
2.047	1.883	2.126	8.148	23.603	4.444
0.801	0.640	0.819	1.519	4.444	2.320
Inverse of Covariance Matrix					
0.490	-0.130	-0.138	-0.007	-0.004	-0.073
-0.130	1.173	-0.370	-0.068	-0.009	-0.086
-0.138	-0.370	0.818	0.001	-0.010	-0.121
-0.007	-0.068	0.001	0.038	-0.010	0.015
-0.004	-0.009	-0.010	-0.010	0.071	-0.121
-0.073	-0.086	-0.121	0.015	-0.121	0.745
-Log(Determinate(Covar.Matrix))					-8.030

Table 6.3: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Forest)

Mean Values					
55.430	24.217	21.574	80.345	66.183	14.093
Covariance Matrix					
3.310	1.205	2.190	-4.472	2.348	2.392
1.205	1.678	1.540	0.527	2.442	1.515
2.190	1.540	3.835	-9.122	2.120	3.052
-4.472	0.527	-9.122	119.711	16.694	-11.010
2.348	2.442	2.120	16.694	23.231	4.917
2.392	1.515	3.052	-11.010	4.917	6.068
Inverse of Covariance Matrix					
0.535	-0.133	-0.186	0.000	-0.007	-0.079
-0.133	1.251	-0.452	-0.055	-0.012	-0.122
-0.186	-0.452	0.735	0.044	-0.013	-0.092
0.000	-0.055	0.044	0.019	-0.021	0.042
-0.007	-0.012	-0.013	-0.021	0.080	-0.090
-0.079	-0.122	-0.092	0.042	-0.090	0.422
-Log(Determinate(Covar.Matrix))					-10.136

Table 6.4: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Secondary Succession)

Mean Values					
62.269	28.897	32.676	53.561	87.936	30.846
Covariance Matrix					
8.512	4.488	9.007	-2.624	21.246	13.762
4.488	4.459	7.407	2.582	15.572	8.438
9.007	7.407	15.827	1.189	30.600	17.471
-2.624	2.582	1.189	34.999	6.178	-8.293
21.246	15.572	30.600	6.178	101.192	54.003
13.762	8.438	17.471	-8.293	54.003	38.414
Inverse of Covariance Matrix					
0.413	-0.158	-0.085	0.036	-0.013	-0.048
-0.158	1.260	-0.458	-0.101	0.008	-0.044
-0.085	-0.458	0.374	0.023	-0.032	0.011
0.036	-0.101	0.023	0.058	-0.034	0.060
-0.013	0.008	-0.032	-0.034	0.071	-0.090
-0.048	-0.044	0.011	0.060	-0.090	0.188
-Log(Determinate(Covar.Matrix))					-12.467

Table 6.5: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Pasture)

Mean Values					
237.200	134.285	171.092	141.315	210.607	102.468
Covariance Matrix					
783.038	563.074	753.670	369.713	467.360	187.061
563.074	498.815	659.234	324.052	402.338	159.305
753.670	659.234	884.888	435.601	550.828	222.602
369.713	324.052	435.601	232.328	304.995	135.910
467.360	402.338	550.828	304.995	584.747	286.378
187.061	159.305	222.602	135.910	286.378	181.402
Inverse of Covariance Matrix					
0.007	-0.001	-0.005	0.001	-0.000	0.000
-0.001	0.138	-0.099	-0.010	0.001	0.008
-0.005	-0.099	0.094	-0.028	-0.004	0.003
0.001	-0.010	-0.028	0.082	-0.003	-0.014
-0.000	0.001	-0.004	-0.003	0.015	-0.017
0.000	0.008	0.003	-0.014	-0.017	0.031
-Log(Determinate(Covar.Matrix))					-25.269

Table 6.6: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Clouds)

Mean Values					
51.211	19.211	16.149	33.359	19.009	2.978
Covariance Matrix					
6.204	2.701	4.202	5.847	9.271	5.256
2.701	2.602	3.313	6.968	8.160	3.883
4.202	3.313	6.332	9.471	12.806	6.578
5.847	6.968	9.471	45.007	37.953	11.998
9.271	8.160	12.806	37.953	52.829	19.677
5.256	3.883	6.578	11.998	19.677	10.785
Inverse of Covariance Matrix					
0.346	-0.222	-0.080	0.022	0.003	-0.071
-0.222	1.551	-0.441	-0.112	0.025	-0.102
-0.080	-0.441	0.653	-0.010	0.005	-0.198
0.022	-0.112	-0.010	0.072	-0.061	0.066
0.003	0.025	0.005	-0.061	0.115	-0.156
-0.071	-0.102	-0.198	0.066	-0.156	0.495
-Log(Determinate(Covar.Matrix))					-9.496

Table 6.7: Maximum Likelihood Signature for the Para Landsat TM-5 image (Class Shadows)

a-priori knowledge of the classes probabilities and without rejections. After classification a report and a thematic map are created, using a class specification file (section 13.3). The report is shown in table 6.8, where each pixel corresponds to $0.0009km^2(900m^2)$, and the resulting thematic map is shown in figure 6.8.

Class	Pixels	%	Area
Rejected or non-classified	0	0.00	0.00 km2
Urban Areas	11621	9.51	10.46 km2
Water	8472	6.93	7.62 km2
Forest	42993	35.17	38.69 km2
Secondary Succession	25962	21.24	23.37 km2
Pasture	20510	16.78	18.46 km2
Clouds	6741	5.51	6.07 km2
Cloud Shadow	5951	4.87	5.36 km2
TOTAL	122250	100.00	110.02 km2

Table 6.8: Classification report for the Para Landsat TM-5 image
(1 pixel = 0.0009 km2)

The original images, the result of the classifiers (as a thematic map), the output likelihoods (created by the classifier) and the class specification file are used as input for `cxinspector` (section 11.7.1) for visual inspection of the results.

Even without the ground truth image we can see that some pixels that should be classified as clouds were classified as urban areas, and some pixels that should be assigned to water were assigned to the class shadows. This is probably due to the large variances for classes Urban Areas and Shadows and to poor sampling of the classes.

Figure 6.7: The Pará Landsat TM-5 image (combination R=5, G=4, B=3)

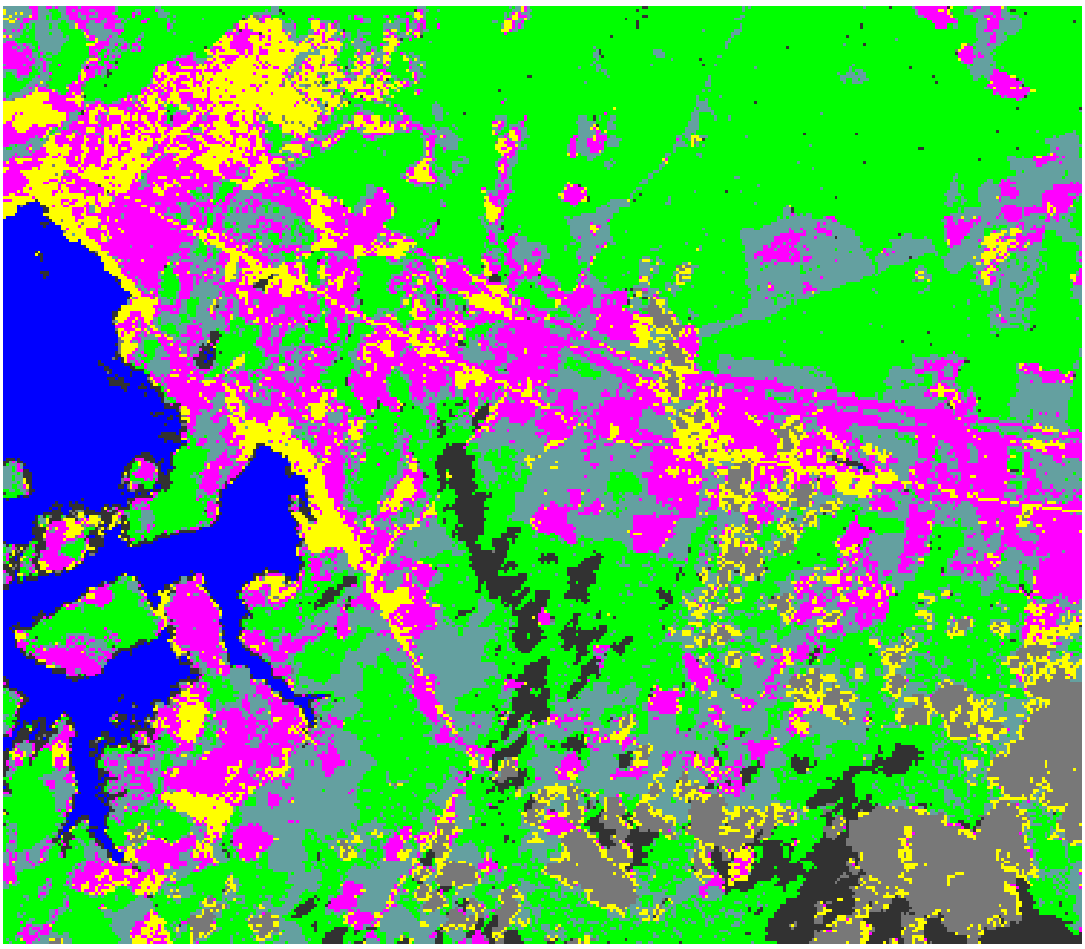

Figure 6.8: The Pará Landsat TM-5 image classified with the Maximum Likelihood Classifier

Chapter 7

Histogram Overlay Classification

7.1 Introduction

Important Note: If you're reading this is because you got an alpha release of this software and document. The official release will not have this message on the manual. Please check the Ejima Lab Khoros Page for more information (see section 1.5 for its URL and other useful addresses).
The operators that are working in this release are listed in appendix A. See also a list of operators that are not working in this alpha release in chapter C.

The Histogram Overlay Classification [8] is a simple classifier that classify areas instead of pixels by comparing the degree of overlap from these areas' histograms to the samples' histograms extracted from the classes. The overlap g between two classes i and j is calculated as

$$g_{i,j} = \left(\sum \min(h_i, h_j)\right) / N$$

where h_i is the histogram for class i, h_j is the histogram for class j and N is the total number of pixels in the area. For classification, the largest overlay between the sample area and the classes determines the class to be assigned to that area. The histograms for both known classes (signatures) and for the areas being classified must be normalized. This can be extended to multidimensional histograms. Figure 7.1 shows one example with two classes in one dimensional feature space. Visually, the histogram overlay measure is the overlay area between two histograms and shown as $g_A(x)$ and $g_B(x)$ in figure 7.1.

Signatures for the Histogram Overlay classifier are the normalized multidimensional histogram values and can be created with the `chover_signature` operator and joined together with the `kappend` operator. Classification for the Histogram Overlay classifier is done with the `chover_classify` operator.

7.2 Signatures for the Histogram Overlay Classification

7.3 Image Classification with the Histogram Overlay Classifier

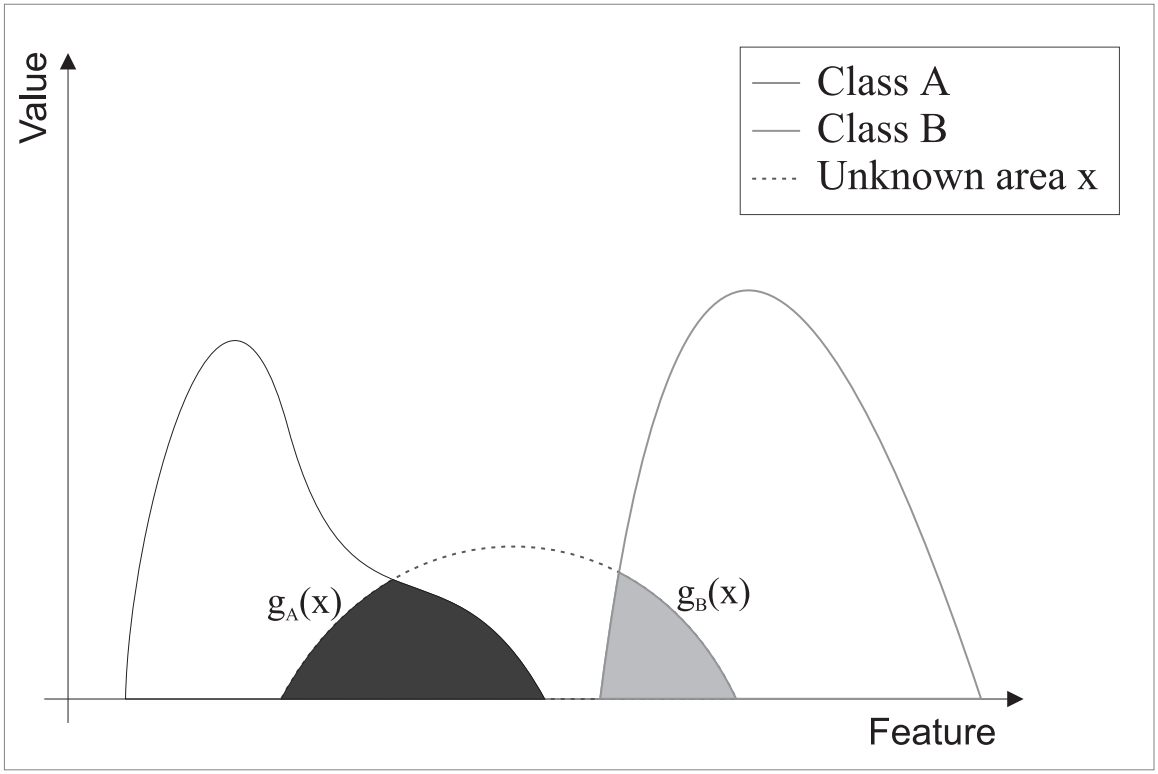

Figure 7.1: Histogram Overlay Classification of an area in the 1-feature space

Chapter 8

Back-Propagation Neural Network Classification

8.1 Introduction

In chapter 2 (figure 2.3) we saw that classification could be considered as a partition of the feature space. If we could estimate the parameters of the lines (or more commonly, hyperplanes) that separate the data into mutual exclusive regions, we could classify pixels in an image by checking the positions of these pixels' vectors against the boundary lines. Figure 8.1 shows a simple partition of a 2-class in 2-feature space using a decision boundary.

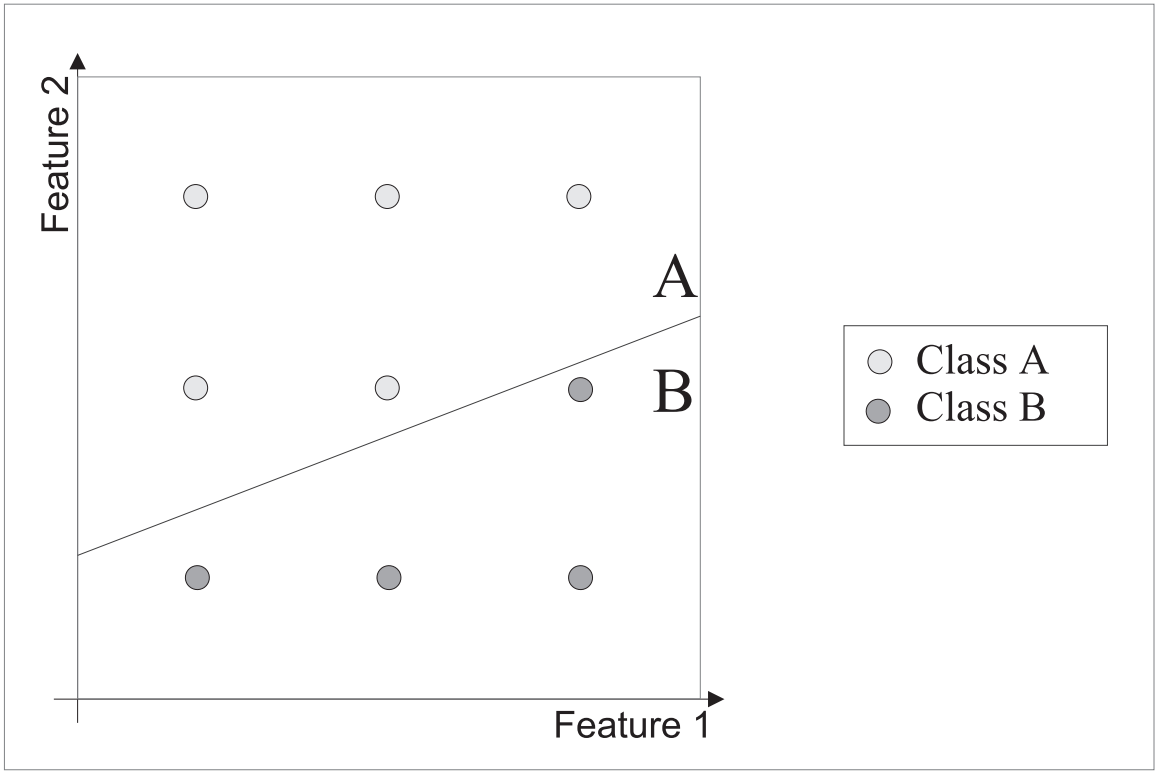

Figure 8.1: Partition of a 2-feature space with 2 classes

A Back-Propagation Neural Network classifier can use labeled input samples to estimate the parameters for a set of hyperplanes that will partition the feature space in most cases. The parameters for these hyperplanes (which will control the orientation, slope, etc. in $N-$dimensions) will be given by the weights of the network, which are the values that are altered between iterations of the training steps.

The estimation of the hyperplane parameters is done by repeatedly presenting to the network the input values, applying the weights (that initially are random), comparing the network output with the expected results, and readjusting weights that corresponds to the line slopes and intercepts - this step is called the *training* of the network and it is performed until the difference between the network output and the expected values is small enough or a maximum number of training steps is achieved. The basic training algorithm is:

1. **Initialize weights**: Before training the network all weights must be initialized to small random values. These weight values will be changed with the iterations of the training process, to (hopefully) *converge* to a set of weights that generate the correct responses for the training samples (i.e. results for the input vectors that are equal or close to the expected results). Masters [9] suggests that the network should be trained with different starting random weights to make sure the network will always converge independently of the starting weights.

2. **Present input vectors**: For each of the samples in the training set, the feature vector for that sample will be presented to the network and evaluated from the input to the output, generating an output vector for that sample and set of weights. This is done from the input layer to the output layer, one layer at a time, with the output for a layer being the input for the next layer.

3. **Compare the output vector with the expected output vector**: The output vector obtained with the presentation above will be compared with the desired response for each pattern and an error based on the difference will be calculated. This error will serve to correct the weights in the layers.

4. **Change weights**: The error obtained from comparing the obtained results with the expected results will be used to correct the weights for the preceding layer. In the multi-layer model this must be done from the output layer to the input layer. This is the called *back-propagation* of the error, hence the name of the neural network model.

5. **Stop the training process**: If the errors obtained in step 3 are smaller than a predetermined threshold, the algorithm will stop and the weights will be saved as the trained network for use by the classifier. Another way to stop the training algorithm is to give a maximum number of iterations for the training.

At the end of the training step, we will have a set of weights that can be used with the same network to classify unknown pixels. The classification algorithm steps are:

1. **Present input vectors**: For each feature vector in the input image present it as the input for the network.

2. **Apply the weights**: Using the input vector apply the weights obtained by training the network and get the output vector. This is done for each layer in the network, from the input to the output layer.

3. **Evaluate the output vector**: Extract the class from the feature vector. In this step some thresholding will be necessary, to determine if a neuron in the output vector is *activated* or not. In the model used in this toolbox, a correct classification result happens when one and only one of the neurons in the output layer is activated. There is the possibility that for a pixel none or more than one of the neurons in the output vector will be activated, in this case there is no way to determine the correct class for the input vector, and it should be rejected.

Since we trained the network to correctly reproduce the correct classes for the sample input vectors, we could expect that it will identify the correct classes for the input vectors. Unfortunately it is not always the case, if the network was trained but did not converge (if the training was interrupted after a number of iterations) some classification errors may occur, and the network can be overfitted to the training patterns and cannot generalize for the classification patterns. Some of these problems can be avoided by trying different designs for the network and using different sets of patterns to train and test the network before using it for classification.

The mathematical details of the training steps of the Back-Propagation Neural Network are beyond the scope of this document. An introduction on using the Back-Propagation Neural Network for image classification but without the implementation details is given in [2]. Other good, more detailed but still introductory readings can be found in [3] and [9]. Application of neural networks in image processing and classification can be found in [10]. Mathematical discussion of several neural network-related issues can be found in [11]. In this document we will focus on the practical details of implementing visual workspaces with the tools of the Classify toolbox for classification with neural networks.

8.2 Neural Network Architecture

A neural network scheme is shown in figure 8.2. In this figure we have a typical architecture for a neural network, with five neurons (or units, represented by circles) on the input layer corresponding to five features for an input pixel, two hidden layers[1] with respectively three and two neurons on it and three output neurons for classification of the pixels in three classes. The network neurons on one layer are connected to all neurons on the next layer. The connections work as summations of the weighted values which then are thresholded, creating the output value for that unit. In figure 8.2 the first input layer does not process anything at all, it just

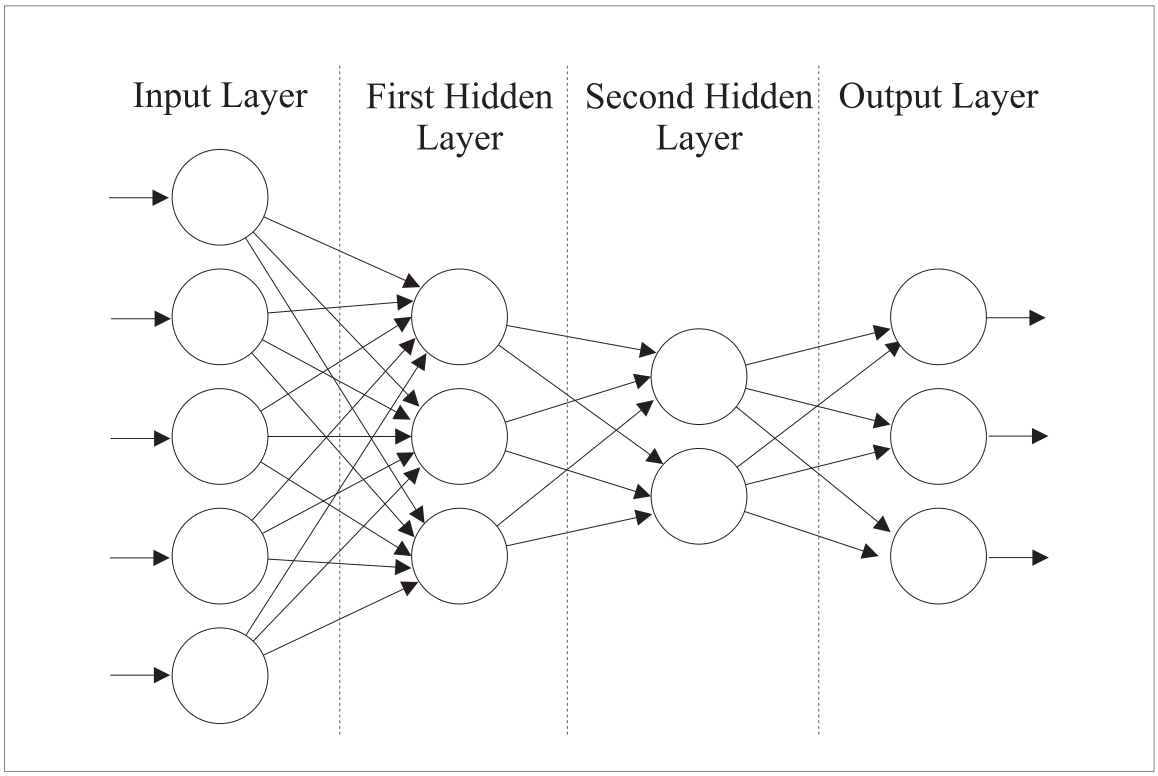

Figure 8.2: A multilayer neural network

[1]Details about the need for hidden layers for classification tasks and more explanations about the Perceptron, Neural Networks and the Back-Propagation training algorithm and more can be found in [3])

serve to pass the values of the feature vector to the second layer, and it is called a layer just for convenience. All the other layers (including the output layer) uses the input values from the neurons on the preceding layer and weights to calculate its outputs. Usually the number of layers in a network is expressed by the number of active layers, so the network in figure 8.2 has 3 layers.

In real classification tasks the boundaries are seldom simple as the ones shown in figure 8.1, usually combinations of the boundaries are necessary. The number of separating boundary lines (or hyperplanes) is controlled by the number of neurons in the first hidden layer, which in turn are equivalent to the number of features. If there is a second hidden layer, it will receive the inputs of the first hidden layer and will be able to produce almost arbitrary regions [3]. This is exemplified in figure 8.3. The operators for creating and training the Back-Propagation Neural Network classifier will use the number of features as the number of neurons on the input layer, will accept values from the user for both the number of hidden layers and the number of neurons on each layer and will determine the number of neurons on the output layer from the number of classes. In figure 8.3 (adapted from [3], pp. 87) we have three examples of network

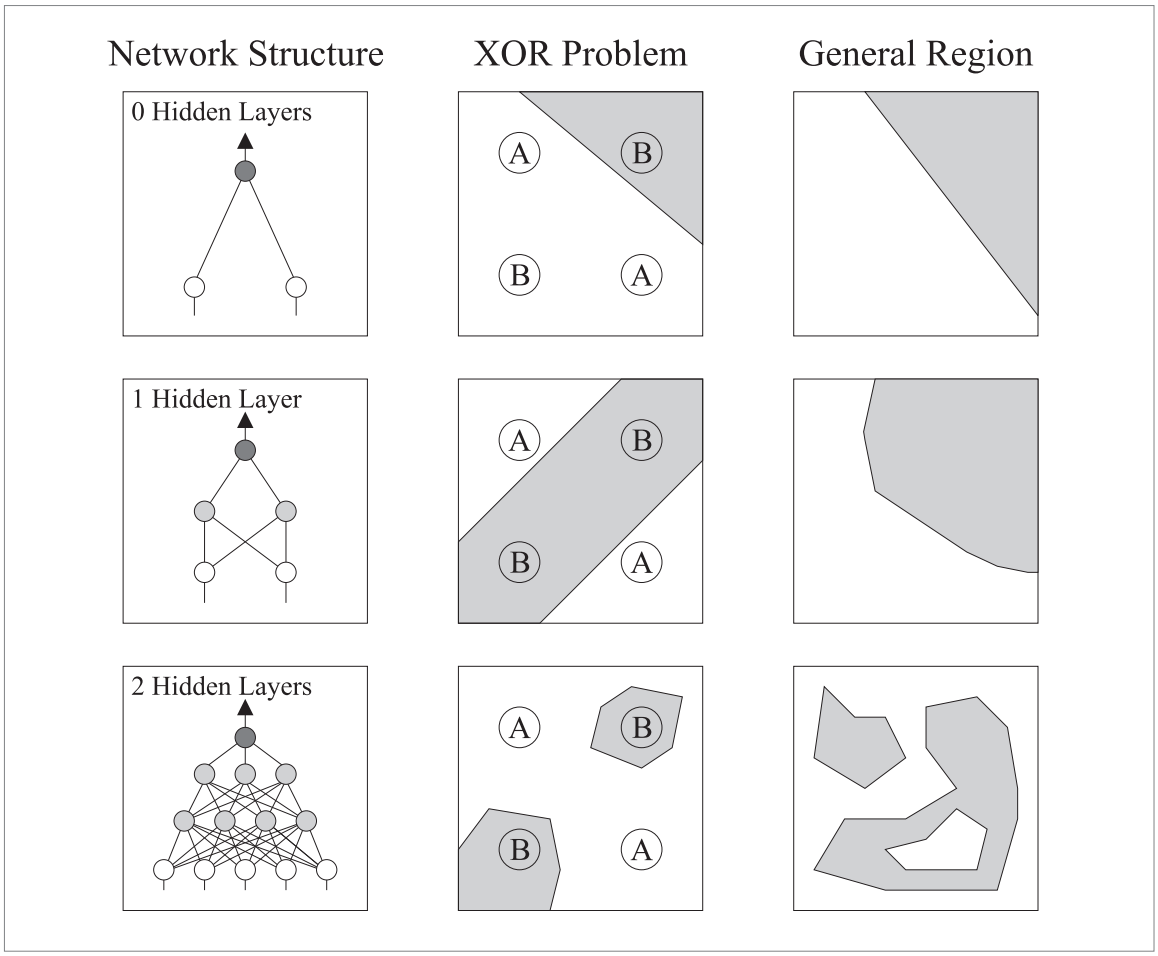

Figure 8.3: Decision Regions for multilayer neural networks

architectures, showing for each one the number of *processing layers* (i.e. excluding the input layer), how the network could solve the XOR problem (a traditional example of the need for hidden layers) and how the network could partition a feature space.

When using a Back-Propagation Neural Network for image classification several parameters must be chosen, and unfortunately there is no simple rule which determines the optimum (or at least good enough) values for these parameters, so for most classification tasks some of these parameters must be decided on a trial-and-error basis. Some of those parameters are:

- **Number of hidden layers**: Masters [9] shows that there is no theoretical reason to ever use more than two hidden layers. The programs here allow the use of more than two hidden layers, but keep in mind that the time required to train the network depends on the number of hidden layers (and units) we use. Details on the number of hidden layers can be found in [9], pp. 85.

- **Number of units (or neurons) in each hidden layer**: This is another factor that cannot be determined easily. Too few neurons may not be enough, but on the other hand too many will increase the risk that the network learns irrelevant details about the training set that will cause poor classification results later. Masters [9] suggests a rough guideline that states that the number of neurons in the hidden layers h be $h = \sqrt{n \times m}$ where m is the number of neurons on the output layer and n is the number of neurons in the input layer, for a neural network with one hidden layer. For a neural network with two hidden layers, the number of neurons in the first hidden layer will be $h_1 = m \times (\sqrt[3]{n/m})^2$ and for the second hidden layer will be $h_2 = m \times \sqrt[3]{n/m}$. This guideline must be applied carefully if there are few neurons in the input and output layers (as in many image classification cases), in these cases it is better to start with the values suggested by the guidelines slightly increased and test several configurations or use the number of samples instead of the number of input neurons as the value for n in the equations above.

- **Number of units (or neurons) in the input layer**: This parameter will be determined by the number of features for our feature vectors. The programs in this toolbox will automatically select this parameter from the input signatures for the trainer.

- **Number of units (or neurons) in the output layer**: This parameter will also be determined automatically from the number of classes we want to use for classification. Internally the trainer and classifier will use a pseudo-binary vector with size C where C is the number of classes. This pseudo-binary vector uses floating point values close to zero and one to ensure that the network will be able to converge. The vector elements are all "zero" except for the class index which will be "one". For example, the expected value for the class 3 in a 6-classes problem where "zero" is 0.1 and "one" is 0.9 will be $(0.1, 0.1, 0.9, 0.1, 0.1, 0.1)$. Both the training and the classifier programs will convert the single integer index for classes to this representation vector and back.

- **Maximum allowed error**: A neural network is trained by presenting it repeatedly with the input vectors and output expected results, and adjusting its weights until the difference between the expected results and the results calculated by the network is small enough. The network is considered trained when the maximum total error for all presentations is smaller than this parameter (or alternatively when the maximum individual value for the presentation of a sample to the network is smaller than a similar parameter). The problem with the selection of this parameter is that there is the possibility that the network will get stuck in a minimum error that will be above these thresholds (i.e. will not converge).

- **Maximum number of iterations**: If the network is not able to converge in the conditions presented in the item above, another way to stop the training is to decide a maximum number of iterations and stop training after this number is achieved. If this number is too small, there is the risk that the network will not be properly trained. If this number is too large and the network converges, it will stop training when its error is small enough, but if the network is unable to converge, it will probably take time to train the network and it will not be trained correctly.

8.2.1 Steps for training and using a Back-Propagation Neural Network for image classification

The steps for classifying an image with the Back-Propagation Neural Network are a little different from the steps for other classifiers in this Toolbox. The required steps are:

1. **Extraction of samples**: This step can be done in the same way it is done for the other classifiers: the user select regions of the image to use as samples, usually by extracting ROIs from the images.

2. **Signatures Extraction and labeling**: This step is similar to the one for the K-Nearest Neighbors classifier (chapter 4). Basically all points on the sample are joined together so they can be easily labeled with the `csigappend` operator. Description of the signatures and extraction methods are in section 8.3.

3. **Network training**: This is the most important step for classification with the Back-Propagation Neural Network: it is where the weights of the network will be calculated, and these weights control the decision boundaries (hyperplanes) that will classify the feature space. The training will be done with the algorithm described in section 8.1. Description of the operator and parameters for training the network are in section 8.4.

4. **Classification**: The input vectors in the image to be classified will be passed through the network and a class will be determined by the output vector, as described in the classification algorithm in section 8.1. Description of the operator and parameters for classifying images with the network are in section 8.5.

The training step of the Back-Propagation Neural Network can take time to be done, depending on the network parameters and nature of the data, and because the training algorithm is repeated several times. The classification step is considerably faster since it involves only the input of a pixel feature vector values on the input layer and the calculation of the input and output values in the hidden and output layers.

8.3 Signatures for the Back-Propagation Neural Network Classifier

The Back-Propagation Neural Network classifier does not use signatures in the same fashion as the other classifiers in this toolbox. Instead, a set of pixels is extracted as ROIs from the image samples, joined and labeled together with the `csigappend` operator, and used to train the neural network to estimate the weights that connect all neurons in the network. These weights are then used to classify the input pixel (feature vectors).

The first step in training the network is selecting the regions of interest that will be used as samples for the classes. These regions of interest must be preprocessed by the `cbpnn_signature` pane and joined together with the `csigappend` operator before training of the network. These steps are similar to those necessary for creating signatures for the K-Nearest Neighbor classifier (chapter 4). Please refer also to section 2.2.1 and figure 2.8 for more details.

8.3.1 The `cbpnn_signature` pane

Object name: cbpnn_signature
Icon name: BPNN Signature
Category: Image Classification
Subcategory: Neural Network

The `cbpnn_signature` is just a pane which calls the `ccompressROI` kroutine (section 10.3.1).

8.3.2 Parameters

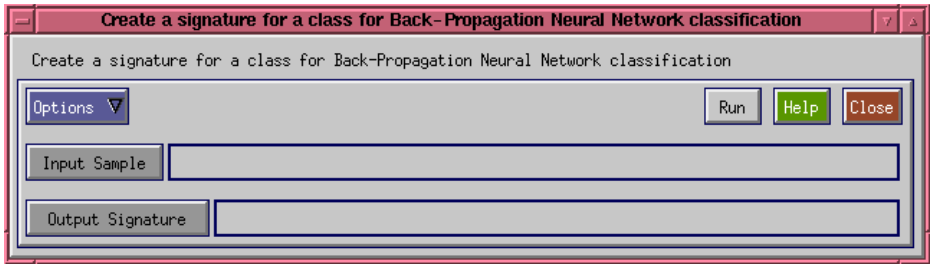

Figure 8.4: The `cbpnn_signature` kroutine GUI

The parameters for the `cbpnn_signature` pane are shown in its GUI pane (Figure 8.4):

- **Input Sample**: The input sample file name. The file can be in any format used by Khoros. The `cbpnn_signature` kroutine will not work with objects which depth or time dimensions are different than one or which data type is complex, and the features should be represented along the elements dimension. In other words, for an image of size $W \times H$ with F features the data should have dimensions $(W \times H \times 1 \times 1 \times F)$. Mask information is used, only valid masked points will be considered for the signatures.
- **Output Signature**: The output signature file name. The output file will contain the signature for that sample. The samples object data dimension will be $(N \times 1 \times 1 \times 1 \times F)$ for F features and N samples.

8.4 Training the Back-Propagation Neural Network

Before classification the Back-Propagation Neural Network must be trained to correctly recognize the classes for the input vectors in the signatures. The training algorithm is described in section 8.1. In this toolbox, training of the network is done by the `cbpnn_train` operator, which will create the network with the parameters passed by the user and train it with the signatures extracted by `cbpnn_signature` operator and joined and labeled with the `csigappend` operator.

8.4.1 The `cbpnn_train` kroutine

Object name: cbpnn_train
Icon name: BPNN Train
Category: Image Classification
Subcategory: Neural Network

8.4.2 Parameters

The parameters for the `cbpnn_train` kroutine are shown in its GUI pane (Figure 8.5):

- **Input Training Patterns**: The input patterns (signatures) for the classes. This input must be the output of a `csigappend` operator (section 10.5.1), since the samples must be labeled. This object must have dimensions $S \times 2 \times 1 \times 1 \times F$ where S is the number of samples in the object and F is the number of features.
- **Number of hidden layers**: The number of hidden layers to be used by the network. Valid values are 1 to 5, but for most cases one or two hidden layers will suffice.
- **Number of units/hidden layer (separated by spaces)**: The number of units (neurons) in each hidden layer. The values are integers separated by spaces.

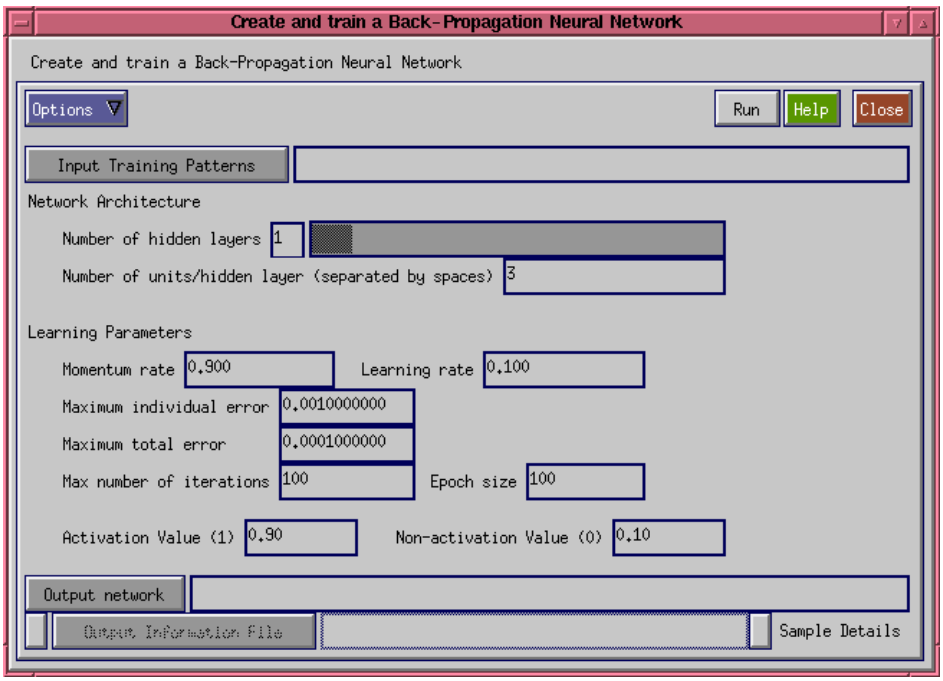

Figure 8.5: The `cbpnn_train` kroutine GUI

- **Momentum rate**: The momentum rate, usually denoted by α in the algorithms. This parameter will determine the change rate in the weights: if the changes are large (meaning that the network is far from convergence) the weight changes will be large, if the changes are small (meaning that the network is close to converge) the changes will be small.
- **Learning rate**: The learning rate, usually denoted by ϵ in the algorithms. This parameter controls the convergence of the network, and it is usually $\ll 1$ making the network take smaller steps towards the solution.
- **Maximum individual error**: One of the parameters that control the end of the training process. If the maximum individual error reaches this value, the network training will stop.
- **Maximum total error**: One of the parameters that control the end of the training process. If the maximum total error reaches this value, the network training will stop.
- **Max number of iterations**: One of the parameters that control the end of the training process. In each iteration the network will be processed **Epoch size** times. The network training will stop if the number of iterations reach this parameter.
- **Epoch size**: The number of times the network will be trained for each iteration (see above).
- **Activation value (1)**: Instead of using the samples' class indexes directly it will be more convenient for the network training to use a vector of values where all values will be close to zero (but should never be zero) and only one element corresponding to the class index will be close to one (but again never be one). This parameter and the next determine which values will be used for "one" and "zero". Values more distant from the extremes (e.g. 0.7 for "one" and 0.3 for "zero") can increase the convergence in some cases but lead to poor classification results.
- **Non-activation value (0)**: A floating point close to zero (see explanation above).

- **Output network**: A KDF file with the network information. This file dimensions will be determined by the number of layers and units on each layer. It should be used as input for the `cbpnn_classify` kroutine.
- **Output Information File**: If selected, for each iteration the network weights and errors will be printed. This can be useful for verification if the network really converged or if the training stopped because the maximum number of iterations was reached.
- **Sample Details**: If chosen, will print information about each sample evaluation in each iteration. If there is a large number of samples in the signatures, this will make the output file get very large.

8.5 Image Classification with the Back-Propagation Neural Network Classifier

With the network trained, we can use it for image classification. Classification is done by presenting the feature vectors (pixels) one by one to the network and by evaluating the results. The output for the network will be a vector of pseudo-binary values as described in section 8.2. Classification is done by the `cbpnn_classify` operator, described next.

8.5.1 The `cbpnn_classify` kroutine

Object name: cbpnn_classify
Icon name: BPNN Classify
Category: Image Classification
Subcategory: Neural Network

8.5.2 Parameters

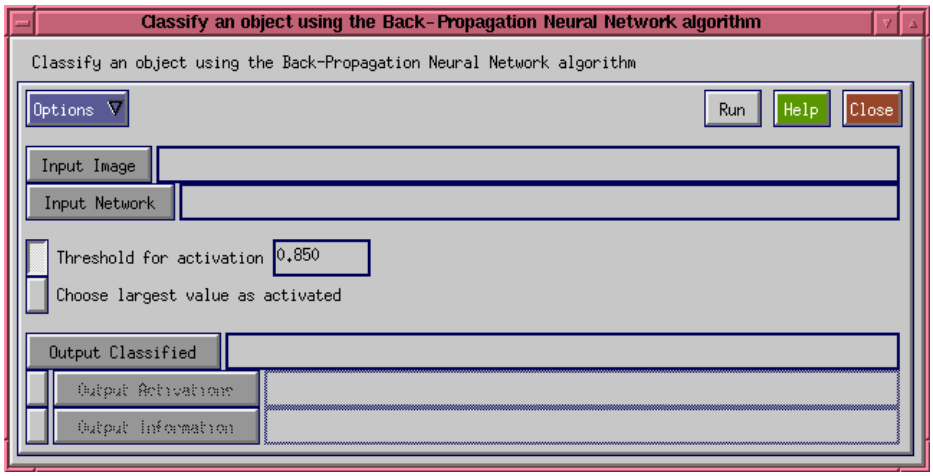

Figure 8.6: The `cbpnn_classify` kroutine GUI

The parameters for the `cbpnn_classify` kroutine are shown in its GUI pane (Figure 8.6):

- **Input Image**: Any image in a format supported by Khoros, with dimensions $W \times H \times D \times T \times E$, where E must have the same size as the number of neurons in the input layer of the network.
- **Input Network**: A network trained by `cbpnn_train`. All network parameters and weights will be read from this file.

- **Threshold for activation**: If chosen, the output class will be the only value in the output vector for the network that is above this threshold. If there are more than one values or none of them are above the threshold the pixel will be marked as rejected. This threshold should be set to a value a little bit smaller than the value selected for **Activation value (1)** for the `cbpnn_train` operator, otherwise many pixels will be rejected.
- **Choose largest value as activated**: If chosen, instead of getting a single value above the threshold it will consider as the correct class the largest activation value for the vector. There is no mathematical foundation for this technique but it will avoid rejection of pixels and if the network was well-trained it will classify correctly most pixels that would be rejected if a threshold was used.
- **Output Classified**: The results of the classification, with dimensions $W \times H \times D \times T \times 1$, where each element is an index for a class corresponding to the order the signatures were appended.
- **Output Activations**: The activations, pixel by pixel. This object dimensions will be $W \times H \times D \times T \times C$, where C is the number of classes used, corresponding to the number of neurons in the output layer of the network. This file can be used by `cxinspector` to evaluate the classification results.
- **Output Information**: If selected will print information about the network and classification results for each pixel, including activation values for each class.

8.6 Utilities

8.6.1 Verifying duplicated and conflicting samples for the network training

One problem that may lead to non-convergence of the network is when we have equal feature vectors assigned to different classes and attempt to try the network with these vectors. The training will be done but at some point conflicting values will arise, making the individual errors for those vectors increase instead of decrease.

The `cbpnn_sigcheck` kroutine can check a set of appended and labeled signatures (created with the `csigappend` operator) to see if any pair of vectors is similar but are assigned to different classes. Note that while it can discover such inconsistencies, these are not the only possible causes of failure to convergence in a network, but will hint to the need of resampling the classes.

8.6.1.1 The `cbpnn_sigcheck` kroutine

Object name: cbpnn_sigcheck
Icon name: BPNN Check Signatures
Category: Image Classification
Subcategory: Neural Network

8.6.1.2 Parameters

The parameters for the `cbpnn_sigcheck` pane are shown in its GUI pane (Figure 8.7):

- **Input Signatures**: This must be the output of a `csigappend` operator, i.e. this operator expects an object of dimensions $S \times 2 \times 1 \times 1 \times F$ where S is the number of samples for all classes and F is the number of features.
- **Tolerance**: The Euclidean distance will be calculated for the vectors being considered, if this distance is smaller or equal to the tolerance the vectors will be considered similar.
- **Output Information**: If chosen will print the information in the file specified in that field, otherwise will print it to the console.

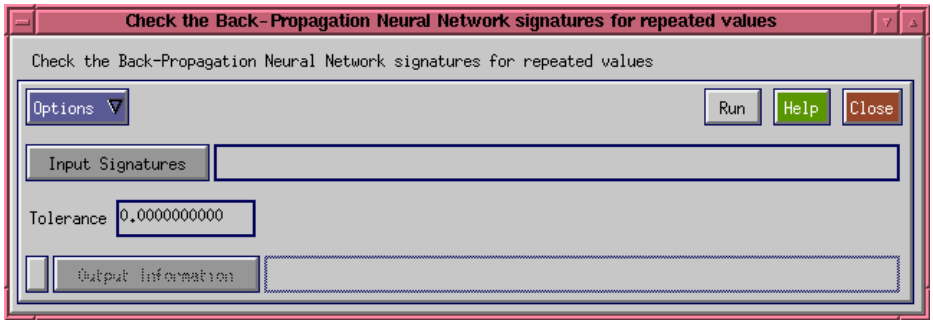

Figure 8.7: The `cbpnn_sigcheck` kroutine GUI

8.7 A Commented Example of Pattern Classification with the Back-Propagation Neural Network Classifier

To better exemplify the training and classification processes of the Back-Propagation Neural Network classifier, we will present a two-feature simple pattern and classify it. While this is not a good example of classification of complex patterns (like images, for example) it will serve as an example of some points of the training and classification processes.

In this example we will try to partition the data shown in figure 8.1 in two classes. The training data are two set of pairs of coordinates for points which belong to each class. A workspace to train and classify the test patterns is shown in figure 8.8.

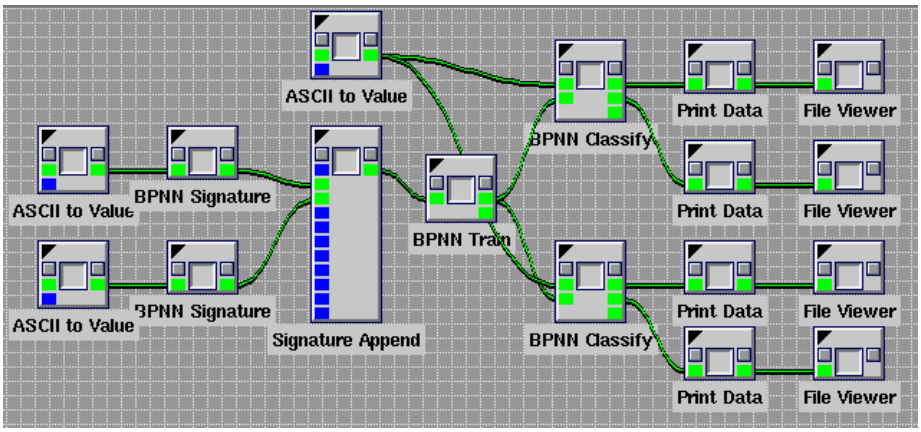

Figure 8.8: A Cantata Workspace for classification with the Back-Propagation Neural Network (example 1)

In the workspace in figure 8.8 we have two `ASCII to Value` glyphs in the left which will input the coordinate files (if you installed the sample data with your toolbox they will be in the directory `$CLASSIFY/sampledata`, files `NNDemo.circle.data` and `NNDemo.cross.data`). They will serve as input for the `BPNN Signature` glyphs, which will be joined and labeled by the `Signature Append` glyph, to serve as input for the `BPNN Train` glyph.

The network was trained with one hidden layer with one neuron, using 10000 maximum iterations. The input data and the partition lines obtained while training the network are shown in figure 8.9. The partition line was plotted with several values of N (the number of iterations) to show how the network gradually gets close to a solution (keep in mind that it is a very simple example since only one hidden neuron was enough to get a solution).

We can see that the first partition line (for $N = 0$) was very far from a solution, since the weights that control it were random values (all weights are random before the training steps). With a few steps the line starts to shift towards a solution for the division of the classes, and with $N = 1000$ a possible solution is achieved. The network continues to be trained, until the

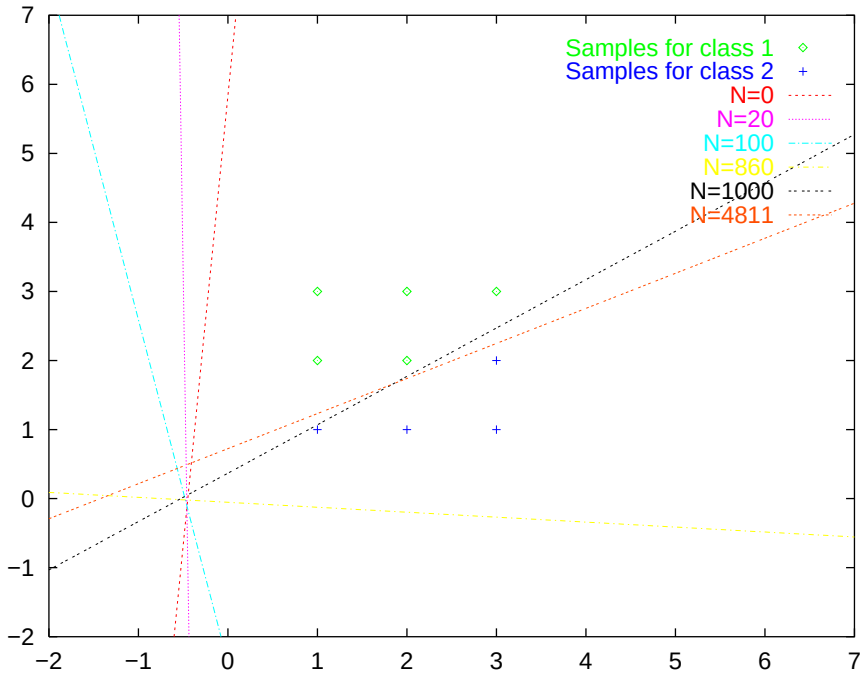

Figure 8.9: Training the BPNN for classification of the sample patterns

specified minimum error value is reached, with $N = 4811$.

To test the classifier, we used a similar data set but with the data more densely distributed and a little extrapolated. The classification results are shown in figure 8.10.

In that figure we see that some of the samples were rejected instead of classified, because they are too close to the partition line to be estimated in which side of it they belong. The activation of the output neurons for the point at (2.5,2), for example, are (0.491,0.509), meaning that it is slightly biased to belong to class 1. When we used classification with the threshold for activation (upper glyph in the workspace) the points were rejected but when we used the maximum activation to select the class, the pixels that were unclassified using the activation threshold were classified sometimes as class 1 and sometimes as class 2.

One could experiment with the network to see what happens when the training parameters are changed. For example, we tested the network using two hidden layers instead of one with respectively 10 and 4 units in each hidden layer and the network converged a little bit faster, but rejected several points when classifying the data, probably due to overfitting.

Even if some variations can train the network faster, there is no guarantee that a particular configuration will work for all classification problems. For any particular problem, the safest way is to try different parameters several times, to see if the network converges all the times.

8.8 Classification Example

As a classification example for the Back-Propagation Neural Network classifier, we will classify the Jelly Beans image in the same classes we used for classification with the Minimum Distance classifier (chapter 5). In this case, the number of points for the samples was reduced to avoid duplication of pixels for different classes.

8.8.1 Creating a Cantata Workspace

A Cantata workspace for classification of the Jelly Beans image is shown in figure 8.11. The workspace is similar to the one shown in figure 8.8 but since we are dealing with images here we can use the same operators we used in other classification examples to extract ROIs from the images and evaluate the classification results.

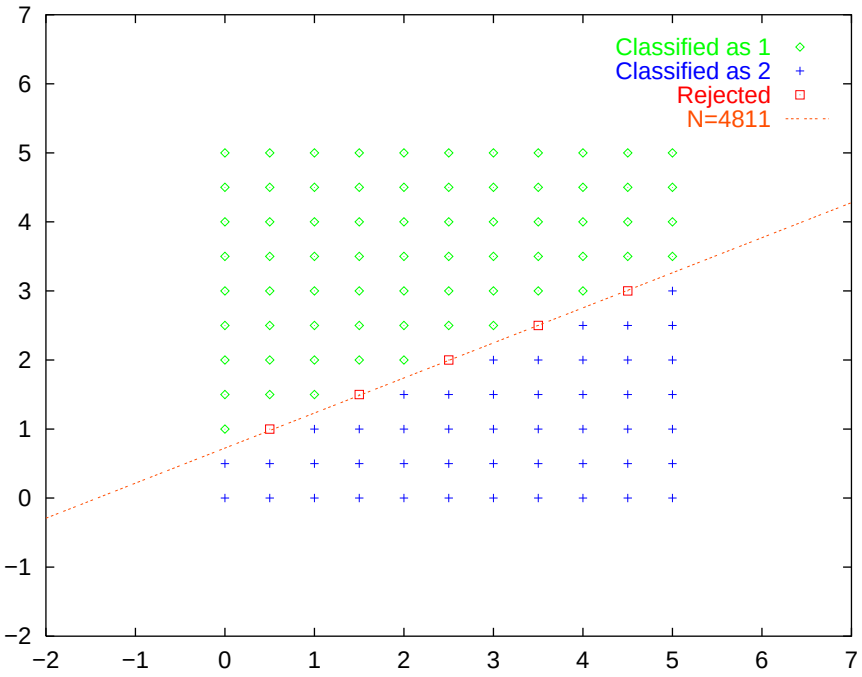

Figure 8.10: Classification of the the sample patterns with the BPNN

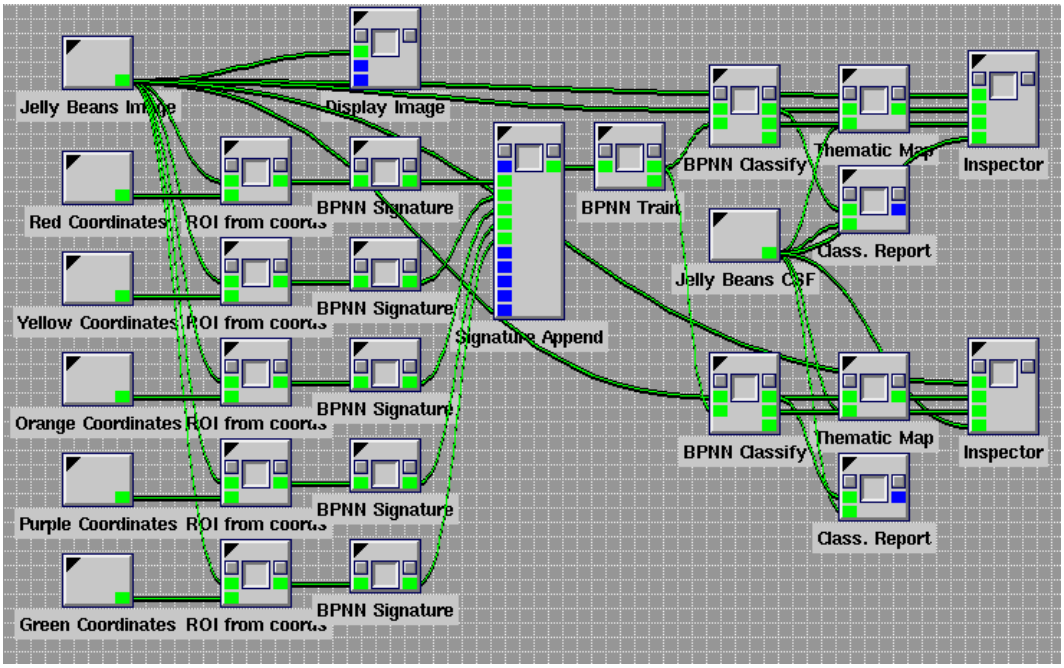

Figure 8.11: A Cantata Workspace for classification with the Back-Propagation Neural Network (example 2)

The network was created with two hidden layers with 100 and 20 neurons respectively. We set the maximum number of iterations to 1000, with an epoch size of 20 to check the evolution of the training process (usually the epoch size will be larger, to increase training speed). All the other parameters for the network were the suggested defaults.

8.8.2 Results for the Training and Classification

Again, we tried several configurations of the network to test which could be trained and classify the image. Besides the configuration mentioned above, we also tried networks with a single hidden layer with 20 and 100 neurons on it, but in several tests the network was unable to converge (in some cases it came closer and then the errors increased again).

The 100 and 20 neurons two-hidden layered network was trained in only 4 iterations (could be more or even less, depending on the initial random values for the weights). The trained network was used to classify the Jelly Beans image (figure 8.12), using the threshold 0.85 to determine which neurons in the output layer were active, giving the result shown in figure 8.13 and using the maximum activation value to determine the class, with the result shown in figure 8.14. The most noticeable error in these results was the classification of the blue jelly beans as green, and it is probably due to overfitting the network. One possible solution for this would be decrease the number of hidden neurons gradually and test the network.

Figure 8.12: The Jelly Beans Image

Figure 8.13: Classification results using a thresholded activation values to determine the classes (rejected pixels are shown in white)

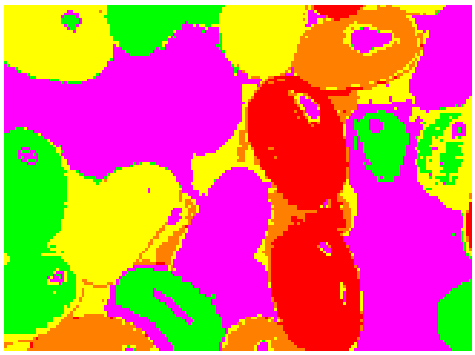

Figure 8.14: Classification results using the highest activation value to determine the classes

Chapter 9

Table Look-up Classification

9.1 Introduction

Important Note: If you're reading this is because you got an alpha release of this software and document. The official release will not have this message on the manual. Please check the Ejima Lab Khoros Page for more information (see section 1.5 for its URL and other useful addresses).
The operators that are working in this release are listed in appendix A. See also a list of operators that are not working in this alpha release in chapter C.

The Table Look-up Classifier is a simple classifier that can be used when the feature vector dimensions are small. If there are enough samples that can cover most (but not necessarily all) these vectors then the Table Look-up classifier will work by creating a partition of the feature space and classifying the pixels by using the labels that were already assigned to the discrete values in the feature space. Figure 9.1 shows the partition of the feature space with 2 features, each with 12 discrete values for it.

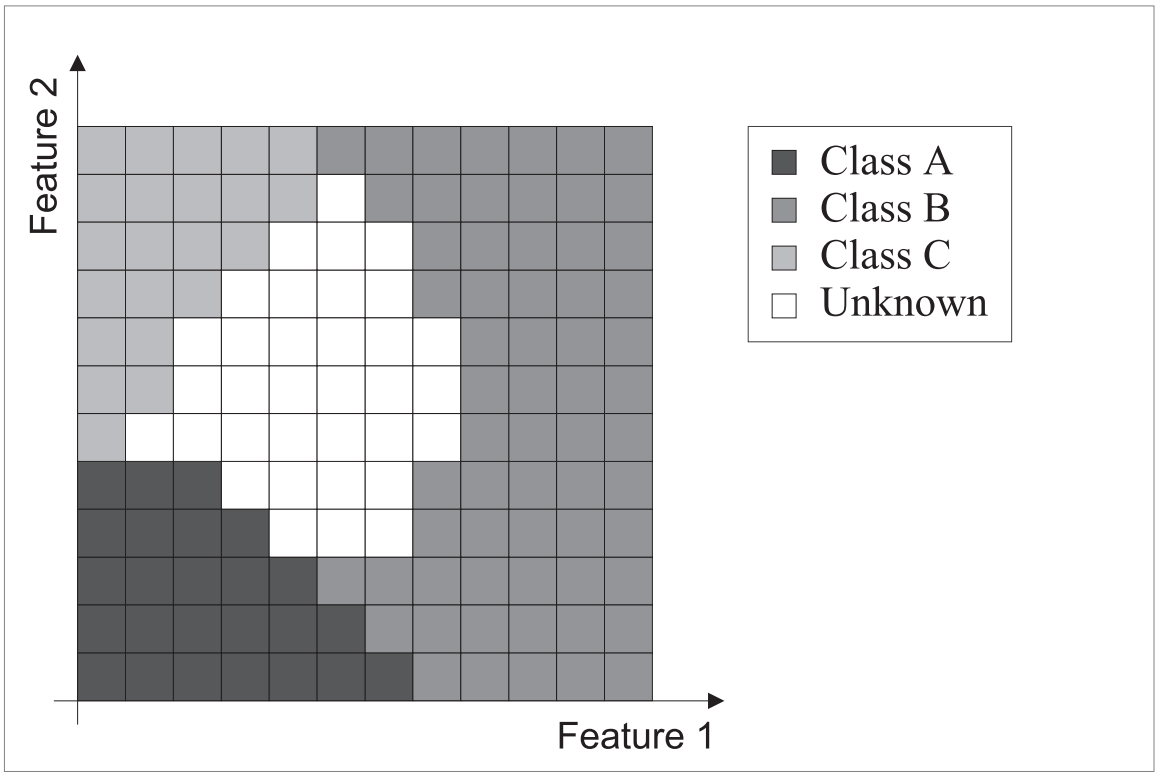

Figure 9.1: Table Look-up Feature Space Partition

To classify a pixel with the feature space partition shown in figure 9.1, the classifier just need to get the values from that pixel's vector and retrieve the corresponding class - a simple table look-up operation. In figure 9.1 only 110 samples were obtained from three classes so some of the discrete feature space slots were not assigned to a class. In this case, if any unknown feature vector happens to match a region without a label, the pixel will not be assigned to any class and will be marked as unclassified.

A problem that is not addressed correctly with the Table Look-up classifier is when there are discrete equal feature vectors assigned to more than one class. The signature labeling algorithm will consider that the last assignment will overwrite the previous.

The Table Look-up classifier can classify images quite fast even if the number of feature vectors is not so small - classification is just a matter of fetching a value from a multidimensional matrix. On the other hand, if the number of feature vectors is large, many samples will be required to represent the classes properly without leaving too many empty spaces on the partition. For example, if there are three features each one with 64 possible values there will be $64^3 = 262144$ possible vectors in the feature space, requiring at least the same number of sample pixels to fill completely the feature space (no unknown classes), and even if there were enough sample pixels there will be a chance that the same feature vector would point to two different classes.

9.2 Signatures for the Table Look-up Classifier

9.3 Image Classification with the Table Look-up Classifier

Chapter 10

Pre-Classification Techniques

10.1 Extracting ROI (regions-of-interest) from rectangular coordinates

In this toolbox the signatures for the classifiers are generated using parts of the images (or regions-of-interest, ROIs) which we know the corresponding classes. These regions are usually extracted from larger images. ROIs can be extracted interactively with `extractor` or can be determined by its bounding box or boxes.

The kroutine `cROIfromcoords` extract regions-of-interest with rectangular coordinates for its bounding box(es). This kroutine reads a Coord file (see section 13.1) with coordinates defining rectangular areas and mark the pixels inside these coordinates as ROI (regions-of-interest). ROIs can be specified by masking or set to a value, and ROIs can be selected as being the areas inside or outside the specified coordinates. This kroutine expects two-dimensional coordinates for the ROIs but apply the same ROI to all dimensions on the image - e.g. in an animation it will mark as ROIs the same area over all frames.

10.1.1 The `cROIfromcoords` kroutine

Object name: cROIfromcoords
Icon name: ROI from coords
Category: Data Manip
Subcategory: Size & Region Operators

10.1.2 Parameters

The parameters for the `cROIfromcoords` kroutine are shown in its GUI pane (Figure 10.1):

- **Input Image File**: The input image file name. The file can be in any format used by Khoros. The expected dimensions are $W \times H \times D \times T \times E$, and the mask will be processed and applied to $D \times T \times E$ image planes.
- **Input Coord File**: The input coordinate file name in the Coord file format (see 13.1).
- **ROIs are everything**: Select **Inside coords** to use all points inside the coordinates as ROI points or **Outside coords** to use all points outside the coordinates as ROI points.
- **Non-ROI values will be**: Select **Masked as 0 (non-valid)** to create a mask for the image, where only the ROI points will be marked as valid or **Assigned to Value** to assign values (specified by the **i** and **j** values, but if data type is not complex only **i** will be considered) to the non-ROI points. This second option is useful for setting all non-ROI points to black or white for printing.

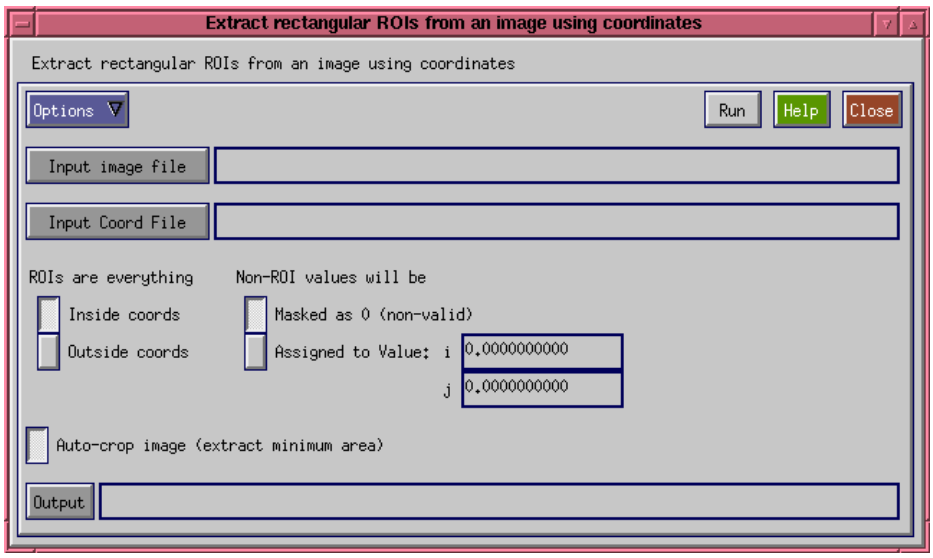

Figure 10.1: The `cROIfromcoords` kroutine GUI

- **Auto-crop image (extract minimum area)**: If set, will reduce the size of the image to the bounding box defined by the coordinates. Useful to reduce the size of the object, but should be used only when the ROIs are *not* going to be joined later by kappend or a similar routine because there is the possibility that the different objects created by this way will have different sizes and then joining them with kappend will create padding areas with values 0, which surely is not a good idea for some classification methods which will consider all those padding points as valid samples.
- **Output**: The output image file name. Its dimensions will be $W \times H \times D \times T \times E$

10.2 Appending ROI (regions-of-interest) coordinate files

If we have all the rectangular ROIs that will be used to classify an image, we can mask the image so only those regions will be "visible". If we assign labels to these coordinate files we can also generate a pseudo-ground-truth image for those regions. These files can be used to test the chosen samples by classifying only the regions used for the samples (see section 10.4.1).

The `cappendROIcoords` kroutine can append and label several single ROI coordinate files in one file.

10.2.1 The `cappendROIcoords` kroutine

Object name: cappendROIcoords
Icon name: Append ROI coords
Category: Data Manip
Subcategory: Size & Region Operators

10.2.2 Parameters

- **Input Pre-Appended**: If chosen, this will be the name for a file which already contains some appended ROI coordinates. This file is usually created with this operator and used when there are more than ten classes to be appended. If chosen, the labeling will start at the last label on this file plus 1. If chosen, this file must be in the Multiple Coord File format (section 13.2). If not chosen, the labeling will start at 1.

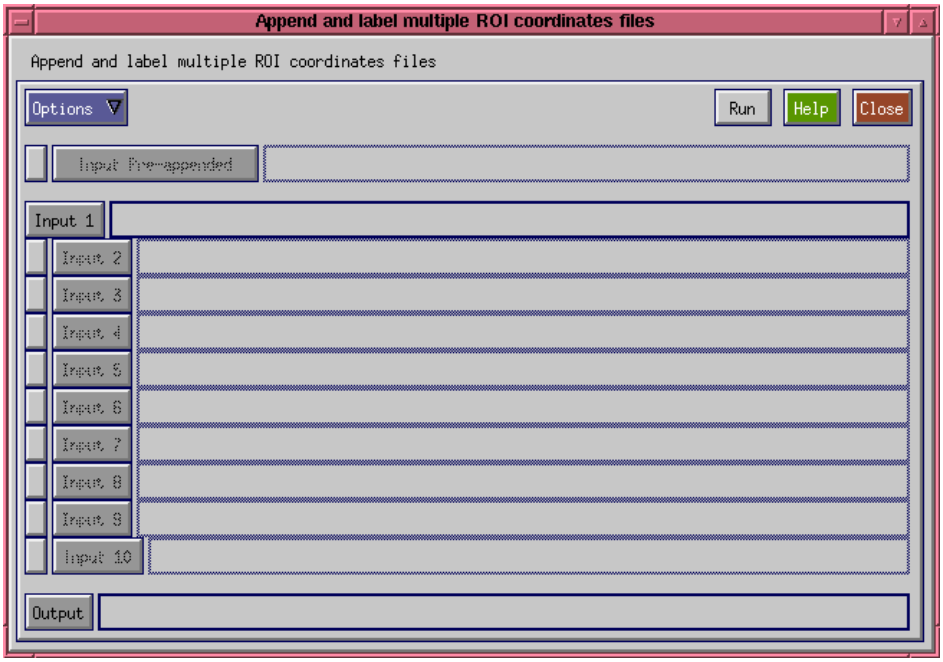

Figure 10.2: The `cappendROIcoords` kroutine GUI

- **Input 1**: The file name for the 1st ROI coords file. This and all other input signature objects must be in the Coord file format (section 13.1).
- **Input 2**: The file name for the 2nd ROI coords file (optional).
- **Input 3**: The file name for the 3rd ROI coords file (optional).
- **Input 4**: The file name for the 4th ROI coords file (optional).
- **Input 5**: The file name for the 5th ROI coords file (optional).
- **Input 6**: The file name for the 6th ROI coords file (optional).
- **Input 7**: The file name for the 7th ROI coords file (optional).
- **Input 8**: The file name for the 8th ROI coords file (optional).
- **Input 9**: The file name for the 9th ROI coords file (optional).
- **Input 10**: The file name for the 10th ROI coords file (optional).
- **Output**: The output object file name. It should be used as input to another copy of the `cappendROIcoords` (see **Input Pre-Appended** above) operator in case there are more than 10 classes. This file will be in the Multiple Coord File format (section 13.2).

10.3 Compressing the ROI (regions-of-interest)

When there are more than one ROI in an image, the mask segment will be used to mask all values that are outside the ROIs as non-valid - this often happens if we have non-contiguous ROIs for the same class. If the whole image is too large, the programs that will process this ROI will take too much time verifying which points are masked and which are not. If this step must be done several times, it would be convenient to separate first the masked from the non-masked points in the image.

The kroutine `ccompressROI` eliminates all points in the input object that are masked as non-valid, reproducing only the masked as valid points in the output object and eliminating

the mask. The side effect is that the original spatial information cannot be preserved on the output object, which will be a single-row object with the same number of elements and same values of the input object. This program should **not** be used if spatial information is important for further processing steps.

Basically the program scans all dimensions width, height, depth and time of an object gathering element-sized vectors which are masked as TRUE (or all elements vectors in case the input does not have a mask). These vectors are then dumped into the output object, which will not have a mask.

10.3.1 The `ccompressROI` kroutine

Object name: ccompressROI
Icon name: Compress Masked ROI
Category: Data Manip
Subcategory: Size & Region Operators

10.3.2 Parameters

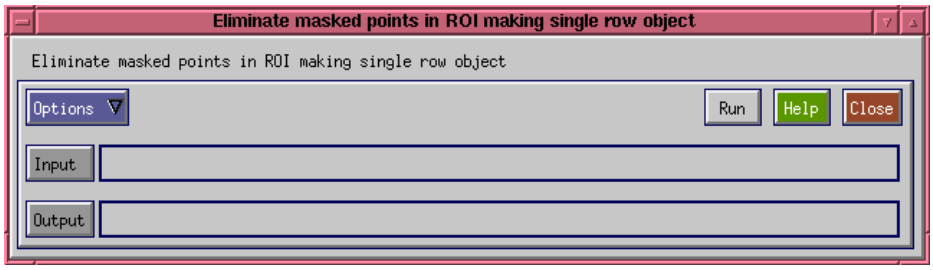

Figure 10.3: The `ccompressROI` kroutine GUI

The parameters for the `ccompressROI` kroutine are shown in its GUI pane (Figure 10.3):

- **Input**: The input object file name. The file can be in any format used by Khoros. The expected dimensions are $W \times H \times D \times T \times E$.
- **Output**: The output object file name. The output dimensions will be $S \times 1 \times 1 \times 1 \times E$ where $S = W \times H \times D \times T$.

10.4 Creating a masked image and pseudo ground truth image from multiple ROI coordinate files

One way to verify if the samples we extracted from the image are a good choice to represent the classes is to classify the samples with the signatures we calculated. If the regions we chose for extracting the samples are classified without errors we can assume that they represent the classes appropriately.

To classify the regions we need an image with only the pixels that belong to the regions-of-interest "visible". This image can be obtained by using all coordinates we used for sample extraction and masking as non-valid all pixels outside these coordinates.

To evaluate the classification result we need a "ground truth" image - an image where the pixels that will be used to extract signatures for a class A are already classified as A. This image is considered the result of a 100

Both the masked image to be classified and the pseudo ground truth file can be obtained with the appended and labeled signatures)from the `cappendROIcoords` operator), using the `cROIfrommcoords` operator.

10.4.1 The `cROIfrommcoords` kroutine

Object name: cROIfrommcoords
Icon name: ROI from multiple coords
Category: Data Manip
Subcategory: Size & Region Operators

10.4.2 Parameters

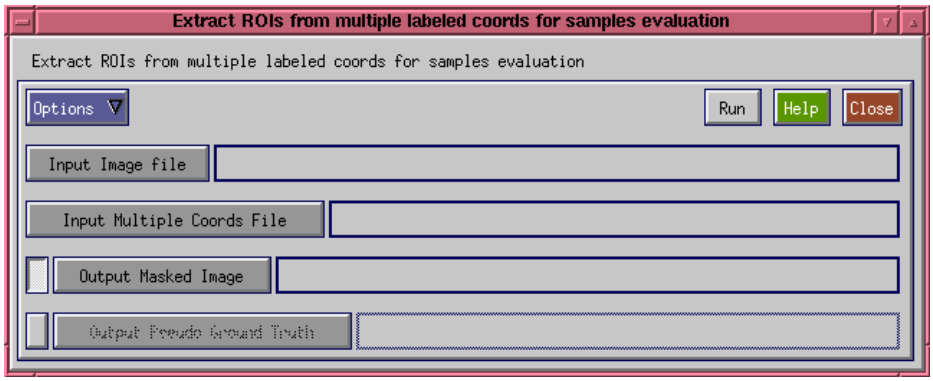

Figure 10.4: The `cROIfrommcoords` kroutine GUI

The parameters for the `cROIfrommcoords` kroutine are shown in its GUI pane (Figure 10.4):

- **Input Image File**: An image which will be used for extracting the ROIs and creating the pseudo ground truth image. Expected dimensions are $W \times H \times D \times T \times E$.
- **Input Multiple Coords File**: A text file with coordinates for ROIs for all classes in the classification task. This file must be in the Multiple Coord File format (section 13.2).
- **Output Masked Image**: The input image with a mask, where points inside the coordinates for the ROIs are masked as valid and all other as non-valid. Its dimensions will be $W \times H \times D \times T \times E$.
- **Output Pseudo Ground Truth**: The pseudo ground truth image, i.e. an image where points inside the coordinates for the ROIs will have a value corresponding to the labels of these regions, and all other pixel will be masked and with a value of 0. Its dimensions will be $W \times H \times D \times T \times 1$.

10.5 Appending and labeling non-uniform signatures

The signatures used in some classifiers are of constant size (when compared with signatures for other classes but for the same task) - e.g. the signatures for the Minimum Distance Classifier (chapter 5) all are of size $F \times 2 \times 1 \times 1 \times 1$ where F is the number of features, regardless of the class. Since these signatures are of constant size they can be appended together in one of their dimensions (usually the time dimension) for the classification process, with the `kappend` operator.

Other classifiers need signatures that usually aren't of constant size. For example, the K-Nearest Neighbors Classifier (chapter 4) uses samples of the images that can vary in number (but not in number of features) as signatures for a classification task. In this case we need to do two things: *label* the samples so we'll know from which class they came and *append* these labeled samples so they can be used by the classifier. Labels are created in the first element of the second line in the height direction (size of the input objects will have another height layer).

The operator `csigappend` will label and append up to 10 signatures into a single file. Optionally the operator can get a result of another copy of it as input, in order to label and append more than 10 signatures. Its user interface parameters are described in the next section.

10.5.1 The `csigappend` kroutine

Object name: csigappend
Icon name: Signature Append
Category: Image Classification
Subcategory: Pre-Classification

10.5.2 Parameters

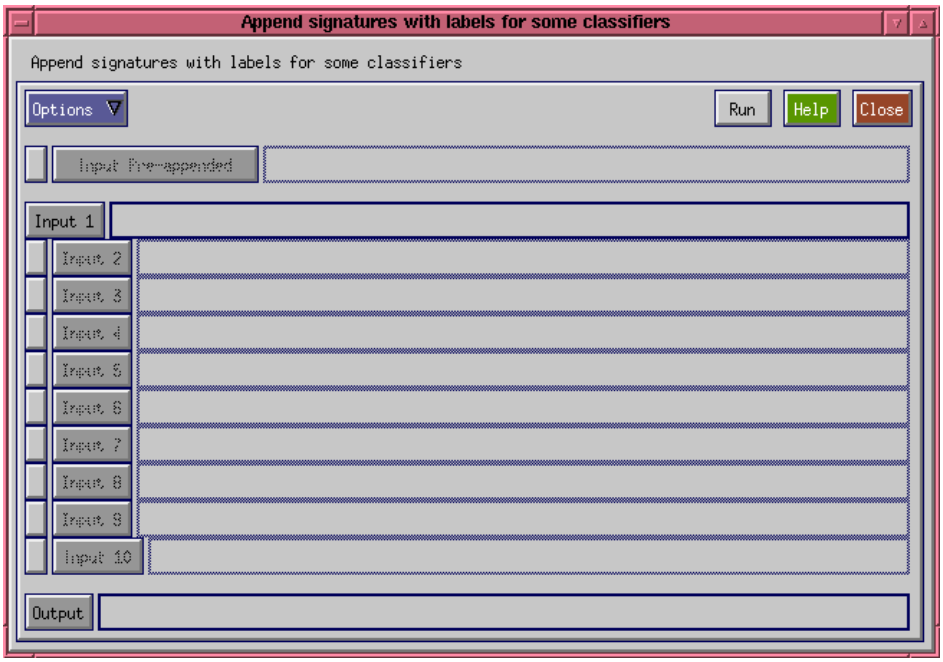

Figure 10.5: The `csigappend` kroutine GUI

The parameters for the `csigappend` kroutine are shown in its GUI pane (Figure 10.5):

- **Input Pre-Appended**: If chosen, this will be the file name for an object which already contains some appended signatures that was created with this operator. If chosen, the labeling will start at the last label on this object plus 1. If not chosen, the labeling will start at 1. This object must have dimensions $S \times 2 \times 1 \times 1 \times F$ where S is the number of samples already in the object and F is the number of features.
- **Input 1**: The file name for the 1st class signature. This and all other input signature objects must have dimensions $T \times 1 \times 1 \times 1 \times F$ where T is the number of samples on this object and F is the number of features.
- **Input 2**: The file name for the 2nd class signature (optional).
- **Input 3**: The file name for the 3rd class signature (optional).
- **Input 4**: The file name for the 4th class signature (optional).
- **Input 5**: The file name for the 5th class signature (optional).
- **Input 6**: The file name for the 6th class signature (optional).

- **Input 7**: The file name for the 7th class signature (optional).
- **Input 8**: The file name for the 8th class signature (optional).
- **Input 9**: The file name for the 9th class signature (optional).
- **Input 10**: The file name for the 10th class signature (optional).
- **Output**: The output object file name. It should be used as input to another copy of the `csigappend` (see **Input Pre-Appended** above) operator in case there are more than 10 classes. This object will have dimensions $(S + U) \times 2 \times 1 \times 1 \times F$ where S is the number of samples already in the object, U is the total number of samples in all input signature objects and F is the number of features.

Chapter 11

Post-Classification Techniques

11.1 Classification result spatial filtering

After classification, a mode filter can be applied to the results to smooth it by extracting from a region around a pixel the most represented (frequent) class in that region and reassigning the class for that pixel with this most frequent class. For example, if in the 3×3 neighborhood around a pixel X the classes distribution is $(1,1,1,2,2,1,2,3,3)$ where the central value is the value of X as obtained through a classifier, after filtering X will be reclassified as 1 since it is the most frequent class in the neighborhood. By considering the results of classification for the neighbor pixels when reevaluating one pixel, this filtering applies some spatial context information to the result.

Mode filtering can be done with the`celementmode` kroutine, described next.

11.1.1 The `celementmode` kroutine

Object name: celementmode
Icon name: Element Mode
Category: Image Classification
Subcategory: Post-Classification

11.1.2 Parameters

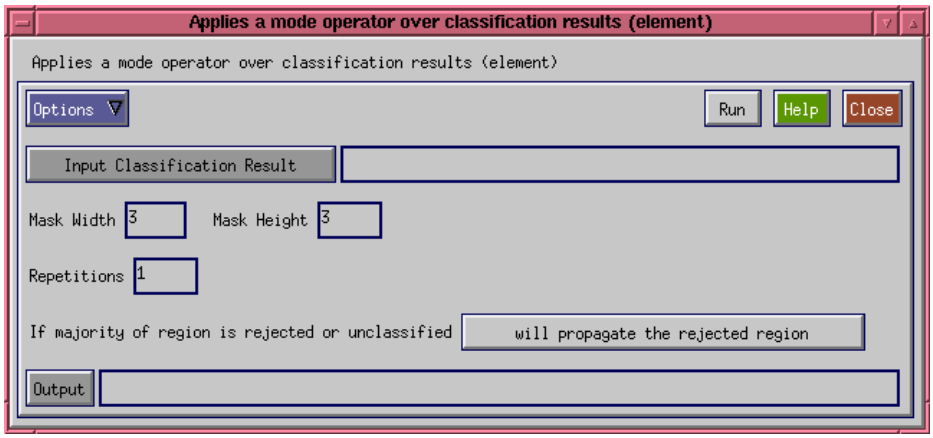

Figure 11.1: The `celementmode` kroutine GUI

The parameters for the `celementmode` kroutine are shown in its GUI pane (Figure 11.1):

- **Input Classification Result**: The classification result, usually the output from a classify operator or the thematic map operator. Its dimensions should be $W \times H \times D \times T \times 1$

(the algorithm will accept values for $E > 1$ and will process band to band).

- **Mask Width**: Width of the mask that will be used to select the neighbor pixels to the pixel of interest.
- **Mask Height**: Height of the mask that will be used to select the neighbor pixels to the pixel of interest.
- **Repetitions**: Number of times that the filtering algorithm will be repeated.
- **If majority of region is rejected or unclassified**: If set to **will propagate the rejected region**, the filtering will consider the masked points when evaluating the majority of points in a region, and will mask the output pixels if the majority of its neighbors is masked. This will have the effect of masking small classified regions near larger rejected regions. If set to **will use the majority of classified pixels**, will use the majority of the points that are classified, i.e. the masked points will not be considered unless the whole region being considered is masked. This will reduce the number of masked (rejected) points in a classification result. If there isn't a mask in the input, both options will have the same effect.
- **Output**: The filtered output image. It will have the same dimensions as the input image, and the map will be preserved (but not applied).

11.2 Probabilistic classification result spatial filtering

Another way to do post-classification filtering is consider not only the spatial distribution of the pixels on the classification result, but also the probabilities that were used to decide the pixel's class. (more on this later)

11.3 Thematic map generation

The classification result for the classifiers in this toolbox are usually an index from 1 to M where M is the number of classes used for a classification task. In order to make the results more presentable, colors can be assigned to each class to make it easier to distinguish from the others.

The `cthematicmap` operator can read a Class Specification File (section 13.3) with RGB values to create a map for the classification result which then can be displayed or printed in colors for easy visual evaluation.

11.3.1 The `cthematicmap` kroutine

Object name: cthematicmap
Icon name: Thematic Map
Category: Image Classification
Subcategory: Post-Classification

11.3.2 Parameters

The parameters for the `cthematicmap` kroutine are shown in its GUI pane (Figure 11.2):

- **Input**: The input image file name. This is usually the output of a classification program.
- **Class Specification File**: The class specification file name in the Class Specification file format (see 13.3).
- **Output**: The output image. The value segment will be the same as the input image, but a map segment will be created with the classes color indexes.

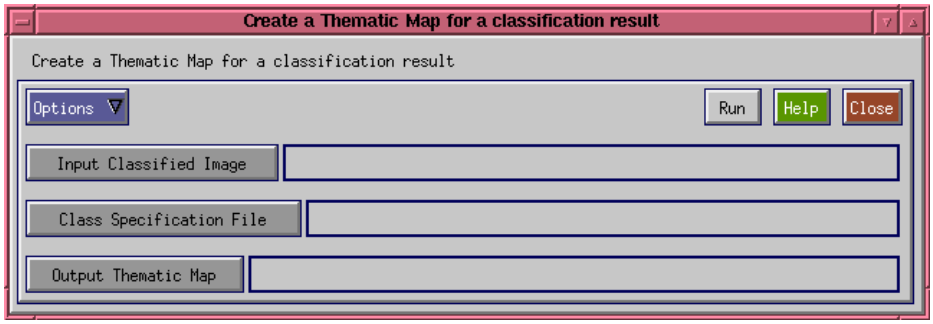

Figure 11.2: The `cthematicmap` kroutine GUI

11.3.3 Choosing colors for creation of the thematic map

The choice of colors for creating a thematic map can be complicated if there are too many classes in the image. To help on this task, this section will present a color table with the RGB values for 32 colors and gray levels chosen to be not too similar to any other colors in the table. These values could serve as starting points for specific thematic maps. Also, some operators in the Color toolbox (available on the same file repository you got the Classify toolbox from) can be used to interactively specify a color by its RGB values.

Table 11.1 shows the RGB values and names for 32 different colors. The name of the colors aren't based in any color naming system, and are listed on the table for convenience. The colors can be seen in figure 11.3, with the first and second rows of colors corresponding to the first column in table 11.1, and the third and fourth rows of colors corresponding to the second column in the table.

When chosing values for the entries in the Class Specification File (section 13.3) you should consider that colors will appear different when displayed and printed. Also, if you're displaying this file, there is no guarantee that your machine will display the colors correctly and some will appear similar (in my case, some also appear blueish).

R	G	B	Name	R	G	B	Name
255	255	255	White	255	0	0	Red
204	204	204	20 % Black	0	255	0	Green
153	153	153	40 % Black	0	0	255	Blue
102	102	102	60 % Black	0	255	255	Cyan
51	51	51	80 % Black	255	0	255	Magenta
0	0	0	Black	255	255	0	Yellow
255	204	51	Light Orange	204	255	0	Yellow-Green
242	135	6	Orange	17	187	158	Sea Green
255	204	204	Light Red	127	0	0	Dark Red
204	255	204	Light Green	0	127	0	Dark Green
204	204	255	Light Blue	0	0	127	Dark Blue
204	255	255	Light Cyan	0	127	127	Dark Cyan
255	204	255	Light Magenta	127	0	127	Dark Magenta
255	255	204	Light Yellow	127	127	0	Dark Yellow
172	114	0	Light Brown	255	102	153	Pink
106	70	0	Brown	181	103	233	Violet

Table 11.1: RGB values for easily discernible colors for thematic map creation

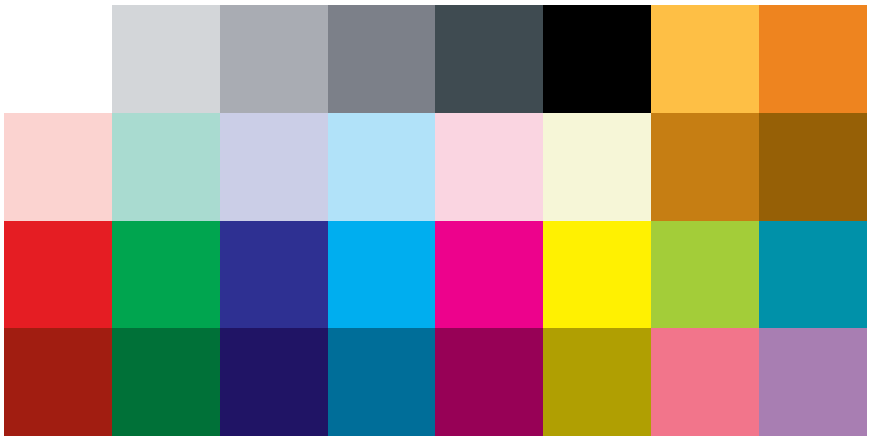

Figure 11.3: Easily discernible colors for thematic map creation

11.4 Report generation

While the thematic map can give a feeling for the classification results and can be easily compared with the input image, often the quantitative results for the classification are required. The operator `cclassreport` can create a table with the number of pixels for each class, and optionally give an area estimation for each class if the pixel area information is given. Reports can be generated in pure ASCII text format, CSV (comma-delimited) format, LaTeX formatted table [12] and HTML formatted table [13].

11.4.1 The `cclassreport` kroutine

Object name: cclassreport
Icon name: Class. Report
Category: Image Classification
Subcategory: Post-Classification

11.4.2 Parameters

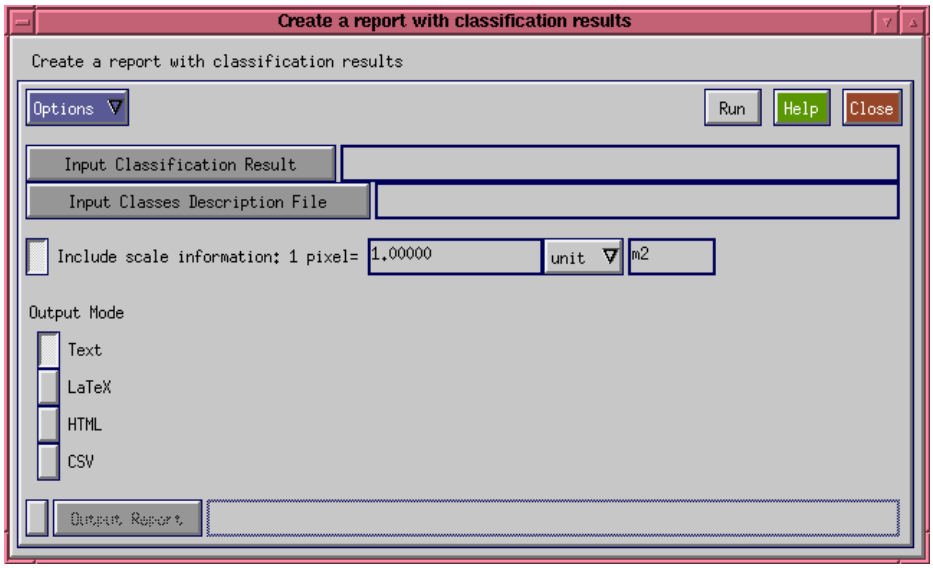

Figure 11.4: The `cclassreport` kroutine GUI

The parameters for the `cclassreport` kroutine are shown in its GUI pane (Figure 11.4):

- **Input Classification Result**: The output result for any classifier. Data can be of any size, but the element dimension size must be 1, and the data type should be integer.
- **Class Specification File**: The class specification file name in the Class Specification file format (see 13.3).
- **Include scale information:**: If selected, will print the value pixels and the corresponding scale. Will use a multiplier factor (scale) passed by the value in **1 pixel =** and the scale unit passed by **1 unit**.
- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.
- **Output Report**: The output file. If selected and set to a file, will save the contents of the report in that file, if unselected will display it on the console.

11.5 Classification results comparison

Often it is desirable to have the classification results described numerically instead of visually. To do this, a *Confusion or Error Matrix* is used, which is obtained by comparing the classification result with the *Ground Truth* (a term borrowed from Remote Sensing) which is the image representing the expected values for each pixel, that can be considered the reference for estimating the accuracy the classifier.

The `ccompare` operator (section 11.5.1) can create the confusion matrix from a classification result and its ground truth. The output is actually done in two tables in a single file, one table for the confusion matrix and other for the omission and commission errors, per-class and general accuracy.

11.5.1 The `ccompare` kroutine

Object name: ccompare
Icon name: Compare Results
Category: Image Classification
Subcategory: Post-Classification

11.5.2 Parameters

The parameters for the `ccompare` kroutine are shown in its GUI pane (Figure 11.5):

- **Input Classified Image**: The input classified object file name. The file can be in any format used by Khoros, and it is usually the direct result of a classifier, without a color map, i.e. before using the operator `cthematicmap` - the map segment is ignored anyway.
- **Input Truth Image**: The input ground truth or similar image file name. he file can be in any format used by Khoros, and its values must correspond to the classes obtained with the classifier (i.e. in the same range, and without a color map).
- **Class Specification File**: The Class Specification File (see section 13.3) that will be used to generate a report with the classes' labels instead of indexes only.
- **Output Mode**: If set to **Text**, will create a simple text file with the contents of the signature file. If set to **LaTeX** the contents will be formatted as a LaTeX table, ready for inclusion in LaTeX files. If set to **HTML** the contents will be generated as a HTML

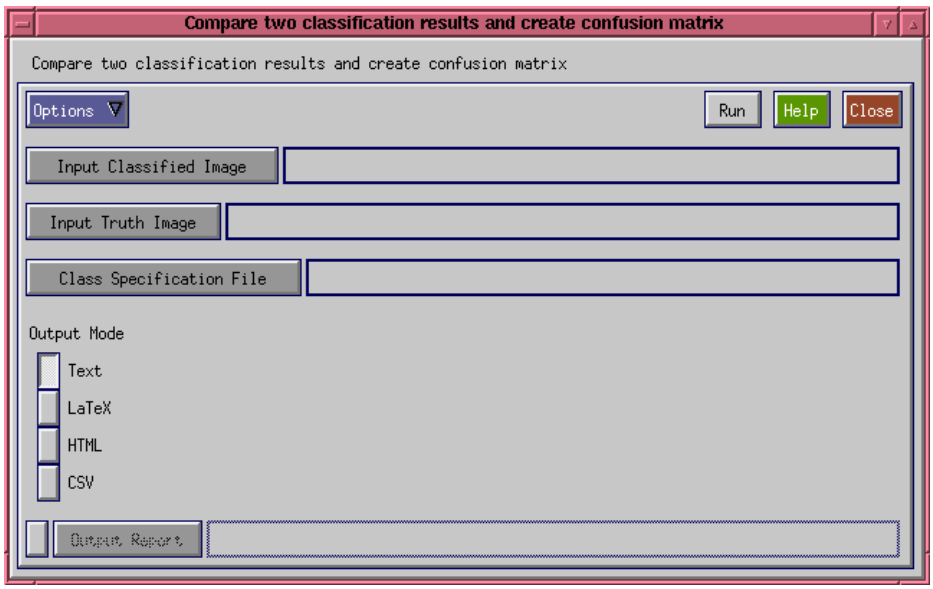

Figure 11.5: The `ccompare` kroutine GUI

table, ready for use with WWW browsers. If set to **CSV** the contents will be formatted as a CSV (comma-delimited) table, ready for use with some spreadsheet programs.

- **Output Report**: The output file. If selected and set to a file, will save the contents of the signature in that file, if unselected will display it on the console.

11.6 Interactive retouch of a classification result

Even for the simplest applications of image classification the results cannot be what the user expected. To enhance the results, the user can verify the signatures and samples, change some classification parameters or use other classifier that is better for that specific task.

When the results are close to what the user expected but some areas can be identified as incorrectly classified, an alternative for the somehow difficult fine tuning of the classifier is to retouch the classification result itself. This is also not simple, and must be done when the user is sure of which areas were incorrectly classified. The `cxretouch` xvroutine allows the user to retouch (paint), mask and filter regions in the output classified image for better results.

11.6.1 The `cxretouch` xvroutine

Object name: cxretouch
Icon name: Interactive Retouch
Category: Image Classification
Subcategory: Post-Classification

11.6.2 Parameters

The parameters for the `cxretouch` xvroutine are shown in its GUI pane (Figure 11.6):

- **Input Classified Image**: A classified image, with color map (i.e. a classification result with a thematic map applied to it). The `cxretouch` xvroutine can process images with dimensions $W \times H \times 1 \times 1 \times 1$.
- **Input Class Specification File**: The class specification file name in the Class Specification file format (see 13.3). Will be used to get the classes' names and associated colors for retouching.

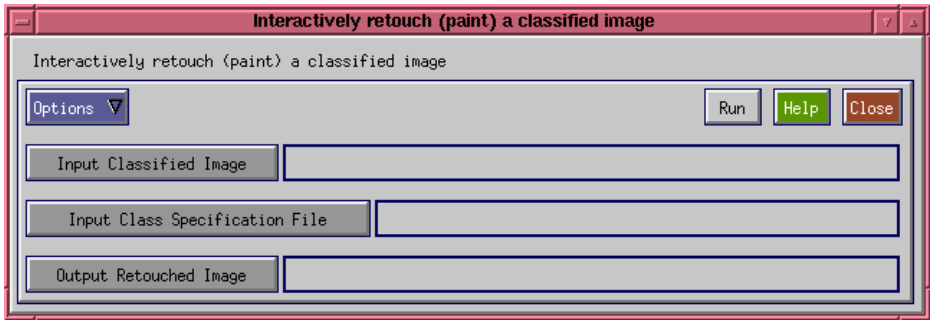

Figure 11.6: The `cxretouch` xvroutine GUI

- **Output Retouched Image**: The output retouched image, with dimensions $W \times H \times 1 \times 1 \times 1$.

11.6.3 Commands and Options

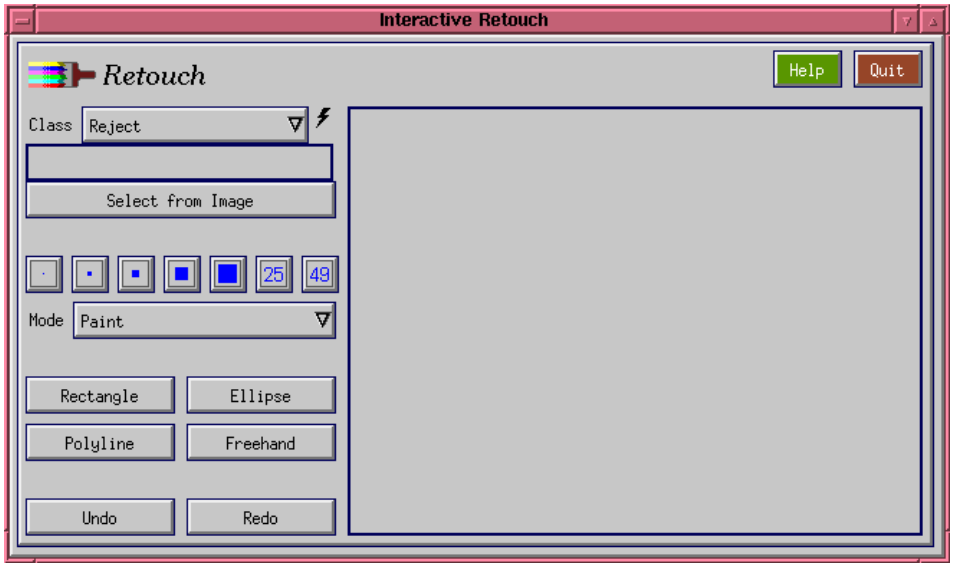

Figure 11.7: The `cxretouch` xvroutine form

All commands and options for the `cxretouch` operator are on its main form, shown (empty) in figure 11.7. On the right side there is a workspace that will be used to display the image being retouched, a 9×9 print pixel object, a image position object and if necessary a panicon object (will appear if the image is too big to fit in the workspace). On the left side there are several mode selector menus and buttons, from the top to the bottom:

- A class list selector that will display the classes found in the Class Specification file by names and the Reject class. This list serve to point the present class - i.e. the class and color that will be used for the paint operations.
- A small workspace that will display the color corresponding to the class in the list above.
- A "Select from Image" button that when pressed will wait until the user click in a region in the image, the class for the pixel where the user clicked will be used as the present class.
- Seven small buttons with when selected will paint, filter or mask small square areas in the image when the user clicks on it. Its sizes are 1×1, 3×3, 5×5, 9×9, 13×13, 25×25 and 49×49.

- A mode list selector that can be used together with the seven buttons above. Four "painting" modes are available:
  - Normal painting: all pixels in the area will be substituted by the class specified in the class list selector
  - Mode filter (3x3 kernel): all pixels in the area will be subjected to a 3x3 mode filter (erasing small speckles and smoothing regions)
  - Mode filter (7x7 kernel): all pixels in the area will be subjected to a 3x3 mode filter (erasing small speckles and smoothing regions)
  - Mask (reject): all pixels in the area will be masked (rejected)
- Four rectangular buttons that when pressed will call the ROI selection routine in the four available modes (rectangle, ellipse, polyline and freehand) for selection of a region where all pixels inside of it will be reclassified with the value of the present class. Note that these buttons actions does not depend on the mode list selector above.
- An undo and a redo buttons for primitive undo and redo operations (the last paint operation can be cancelled).

To paint, mask or filter a region in the image simply click the mouse button over it (if any of the square buttons is pressed) or select a region with the ROI extraction methods (if any of the rectangle, ellipse, polyline and freehand buttons is selected). For large and/or complex areas the operator will need some seconds to process the data.

11.7 Interactive inspection of a classification result

After classification the result image can be interactively inspected with the `cxinspector` operator. This operator will use the original image, classified image and class specification file to display information about the classification for each selected pixel in the images. If probabilities or distances information is available, it can also be used.

If the input and classified images has depth or time dimensions larger than one, only one frame of the image will be shown. The user can select which time or depth index will be used interactively.

11.7.1 The `cxinspector` xvroutine

Object name: cxinspector
Icon name: Inspector
Category: Image Classification
Subcategory: Post-Classification

11.7.2 Parameters

The parameters for the `cxinspector` xvroutine are shown in its GUI pane (Figure 11.8):

- **Input Original Image**: The original image (prior classification). For better visualization, the `cxinspector` operator can alternate the display of the image before and after classification. If there are three bands in the input image it will be displayed as a RGB image, otherwise a band at a time will be displayed. Expected dimensions are $W \times H \times D \times T \times E$.
- **Input Classified Image**: The output of any of the classifiers, or the output of a `cthematicmap` operator (section 11.3.1), in this case the display will be done using the colors specified by the thematic map. Expected dimensions are $W \times H \times D \times T \times 1$.

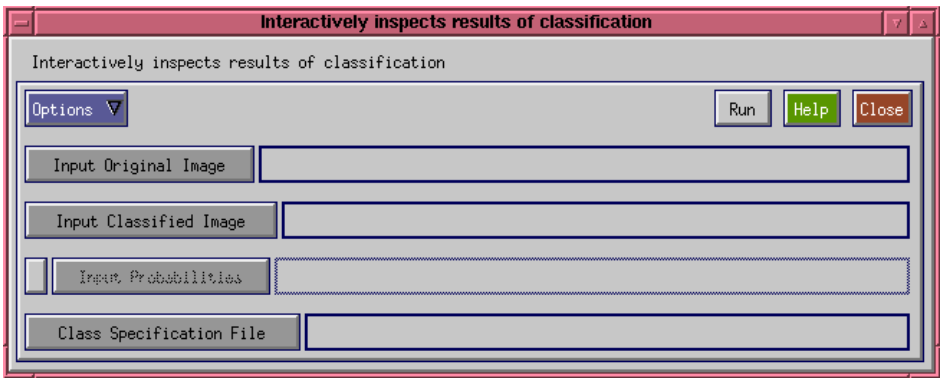

Figure 11.8: The `cxinspector` xvroutine GUI

- **Input Probabilities**: If selected, will accept a file name for a Output Probabilities file that can be generated with some of the classifiers on this toolbox, with the likelihoods, distances or similar information for each input vector and all possible classes. The dimensions for this file should be $W \times H \times D \times T \times C$ where C is the number of classes used for classification.

- **Class Specification File**: The class specification file name in the Class Specification file format (see 13.3). Will be used only to get the classes' names for report.

11.7.3 Commands and Options

There are few commands and options for the `cxinspector` operator. Its master form is shown in figure 11.9. There is a main workspace which will show the image, and two buttons: **Image Display**, which will bring the pane shown in figure 11.10 and **Output** which will bring the output window with the information for the pixels.

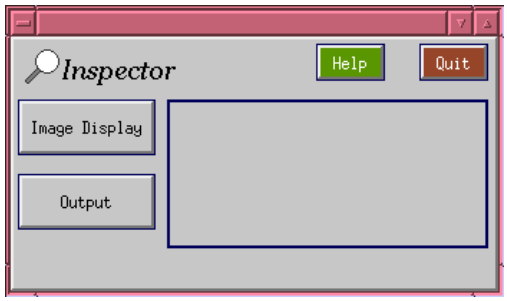

Figure 11.9: The `cxinspector` xvroutine master form

The Image Display Pane shown in figure 11.10 has the following elements:

- **Display Image**: If set to **Original (non-classified) image** will show the original image in the main workspace. If set to **Classification Result image** will show the classification result (for better results use the output of a `cthematicmap` operator).

- **Show Element (band)**: If the original image has 1 or 3 bands it will be displayed automatically as a gray level or color image respectively. If the number of bands is different, only one color band will be displayed at a time, the band can be selected with the value selected here.

- **Show Depth**: If there is more than one depth dimension on the input and classified images, the user can select which depth will be displayed here.

- **Show Time**: If there is more than one time dimension on the input and classified images, the user can select which depth will be displayed here.
- **Use private colormap**: By default the images will be displayed with a private colormap, uncheck this selection to use a common colormap.

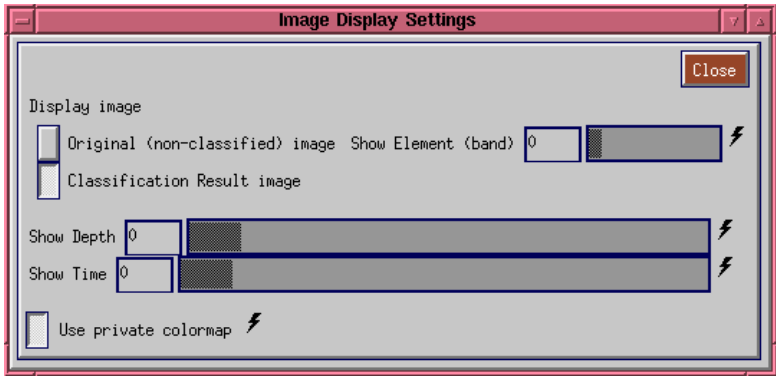

Figure 11.10: The `cxinspector` xvroutine image display pane

To obtain information about the classification for a pixel, just click over the pixel with the mouse. A small report will be created and shown in the output pane. The report will look like the following example:

```
Inspection report for image classified with the Minimum Distance classifier:
Point at (50,101,0,0):  Class 5 (Pasture )
Original Data Vector:   59.000  27.000  30.000  45.000  81.000  25.000
Class (ordered by probability)     Probability  Rejected
Pasture                                  4.163     No
Secondary Succession                    28.210     No
Forest                                  44.611     Yes
Cloud Shadow                            64.995     Yes
Water                                   89.879     Yes
Urban Areas                             90.310     No
Clouds                                 240.850     Yes
```

The part after "Class (ordered by probability)" will appear only if there is an input probabilities object.

Chapter 12

Other Utilities in the Classify Toolbox

12.1 Extracting the mean, standard deviation and covariance matrix from masked objects

Some classification methods in this toolbox (e.g. Minimum Distance Classifier in chapter 5 and Maximum Likelihood Classifier in chapter 6) uses some statistical properties of the data to create the signatures. The operator `mcovar_mask` described in section 12.1.1 extracts the mean, the standard deviation and the covariance matrix of a set of data vectors. While the operator itself is not used directly for signature generation, its library function is used.

The operator `mcovar_mask` uses all data in the dimensions $w \times h \times d \times t$ and creates a mean vector of e elements, a standard deviation vector of e elements and a covariance matrix of $e \times e$ elements.

The difference from this operator and the similar operator in the MATRIX toolbox is that this one consider the mask on the input object, not processing points which are masked, being suited for the sample selection routines that masks non-sample points.

12.1.1 The `mcovar_mask` kroutine

Object name: mcovar_mask
Icon name: Mean and Cov. Matrix
Category: Matrix
Subcategory: Linear Operators

12.1.2 Parameters

The parameters for the `mcovar_mask` kroutine are shown in its GUI pane (Figure 12.1):

- **Input**: The input object file name. The file can be in any format used by Khoros and can have any w,h,d,t,e dimensions.
- **Normalization method for Covariance Matrix**: The covariance matrix calculation divides it by the number of points N on the image, you can use either N by choosing **Normalize to N points** or $N - 1$ by choosing **Normalize to N-1 points**.
- **Output Mean Vector**: If chosen, will output the mean vector of the area (considering only points masked as valid) to that file name.
- **Output Std.Dev. Vector**: If chosen, will output the standard deviation vector of the area (considering only points masked as valid) to that file name.

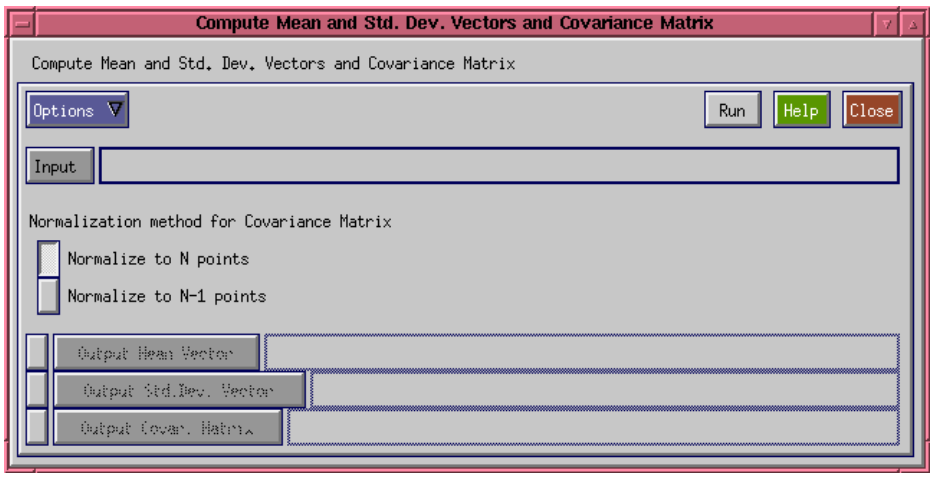

Figure 12.1: The `mcovar_mask` kroutine GUI

- **Output Covar. Matrix**: If chosen, will output the covariance matrix of the area (considering only points masked as valid) to that file name.

12.2 Unmapping RGB-based image objects

Some input RGB images (for example, images generated with paint programs, conversion programs and similar utilities) does not have a color map with them, since the values are specified directly in RGB. For some applications in this toolbox, the separation between data and color map are essential. Specifically, the `ccompare` operator (section 11.5.1) can compare two classification results and create a confusion matrix, but these objects are compared considering its value segments only. If an user has a ground truth image and wants to compare it with a classification result this ground truth image must be unmapped before so only the values (classes indexes) will be used for comparison.

The `kunmapdata` kroutine (section 12.2.1) can separate images from its maps in two ways: *automatic* and *non-automatic*. In the automatic mode, the user must specify an expected (maximum) number of colors that will be used to create the color map, and the program will scan all image to create the color map and unmap the data based on this color map. In the non-automatic mode the color map will be created from the Class Specification File and all pixels in the image will be unmapped from it. The non-automatic mode is faster but requires a correct Class Specification File to work.

If the number of colors in the input image exceeds the expected number of colors, all pixels with colors that cannot be unmapped will be masked in the output. If the user selected the non-automatic mode, pixels with colors which index are not present in the Class Specification File will be masked.

12.2.1 The `kunmapdata` kroutine

Object name: kunmapdata
Icon name: Unmap Data
Category: Data Manip
Subcategory: Map Operators

12.2.2 Parameters

The parameters for the `kunmapdata` kroutine are shown in its GUI pane (Figure 12.2):

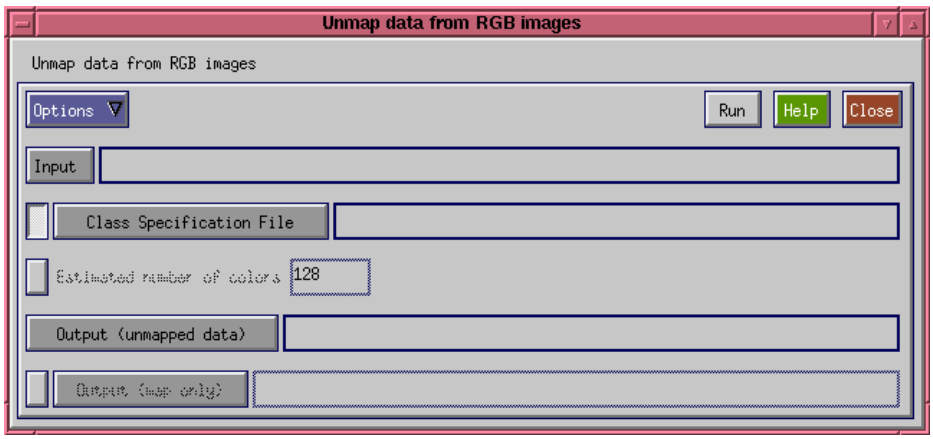

Figure 12.2: The `kunmapdata` kroutine GUI

- **Input**: The input object file name. The file can be in any format used by Khoros. The `kunmapdata` will accept only RGB image objects, i.e. with dimensions $W \times H \times 1 \times 1 \times 3$.
- **Class Specification File**: If selected, the program will use a Class Specification File (see section 13.3) to determine the number and colors that *should* be present in the input image. This option is mutually exclusive with both **Estimated number of colors** and **Output (map only)** options below, meaning that either the user enters a map with the Class Specification File or a map will be created automatically with `kunmapdata`. Maps can be created from Class Specification Files with the operator `cthematicmap` (section 11.3.1).
- **Estimated number of colors**: If chosen, will use its value to estimate the maximum number of colors in the input image to create automatically a colormap for it and output it on the file specified by **Output (map only)**.
- **Output (unmapped data)**: The output object file name, without the map information (a single plane of dimensions $w \times h \times 1 \times 1 \times 1$ where the values are entries for either the classes in the Class Specification File specified in **Class Specification File** or for the map specified in **Output (map only)**.
- **Output (map only)**: If chosen, will be used with the value specified in **Estimated number of colors** to create a map for the input image automatically.

12.3 Substituting or masking values in a classification result

For some classification applications it could be useful to manipulate the classification results to make changes in the classes indexes - for example, after a classification result, one could want to see what it would look like if two classes were considered the same. For simple visualization, that could be done with the `cthematicmap` operator (section 11.3.1) but direct substitution of the values in a classification result is also possible with the `ksubstitute` operator.

The `ksubstitute` routine scans the input data object and substitute some target values or mask them in the output object. Substitution pairs are passed in strings, separated by spaces, and are evaluated left to right.

Another application of the same routine is masking all values in an object except the ones specified - useful for post-evaluation of only pixels that belongs to a class in a classification result image.

12.3.1 The `ksubstitute` kroutine

Object name: ksubstitute

Icon name: Substitute Values
Category: Data Manip
Subcategory: Value Operators

12.3.2 Parameters

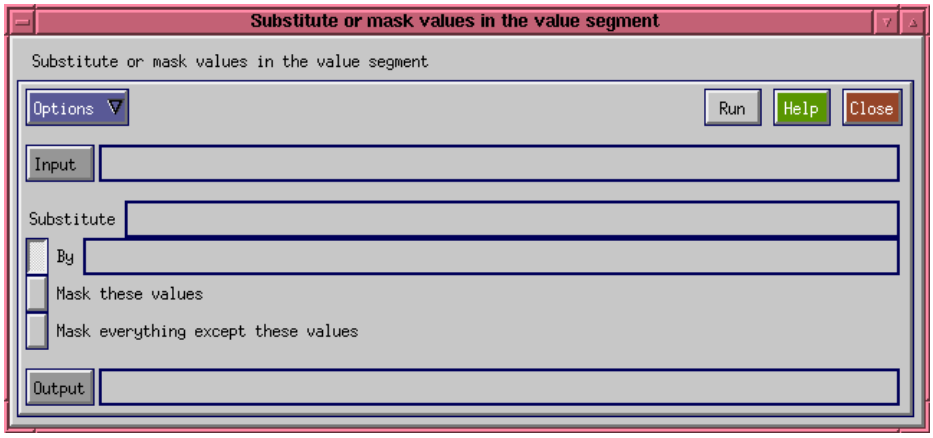

Figure 12.3: The `ksubstitute` kroutine GUI

The parameters for the `ksubstitute` kroutine are shown in its GUI pane (Figure 12.3):

- **Input**: The input object, which can have any dimensions.
- **Substitute**: A list of values to be substituted, separated by spaces.
- **By**: If chosen, it should contain a list of values that will substitute one by one the values on the list on **Substitute** (both lists must have the same number of elements or an error will occur).
- **Mask these values**: If chosen, instead of substituting the values in **Substitute** by the ones in **By** it will mask the values in **Substitute** as non-valid.
- **Output**: The output object that will have the same dimensions as the input.

Substitution is not done recursively: if a string `1 3` is passed in the **Substitute** field and a string `3 5` is passed in the **By** field, all original values `1` will be replaced by `3` but will not be replaced by `5` in the same call to the routine.

12.4 Copying mask of an object into another object

12.4.1 The `kinsertmask` kroutine

Object name: kinsertmask
Icon name: Insert Mask
Category: Data Manip
Subcategory: Segment Operators

12.4.2 Parameters

The parameters for the `linsertmask` kroutine are shown in its GUI pane (Figure 12.4):

- blah

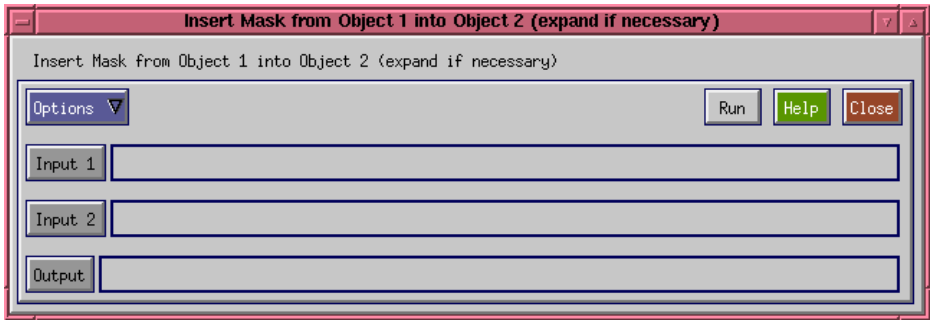

Figure 12.4: The `kinsertmask` kroutine GUI

12.5 Estimating the time required to run an operator

Sometimes it is useful to have an estimate of how much time is required to run an operator, specially for some operators in this toolbox. An estimation of the time in seconds and microseconds can be obtained using the `cstopwatch` operator. Basically it is used at least in pairs, one marking the time before an operator is executed and another marking the time after execution and using the time marked by the first to calculate the difference in seconds. Some control connections dependency is necessary to estimate the times.

The time difference is measures using the `gettimeofday()` C library function, which I don't know if is portable to other Unix systems. The time measurement precision depends also on the hardware.

12.5.1 The `cstopwatch` kroutine

Object name: cstopwatch
Icon name: Stopwatch
Category: Program Utilities
Subcategory: Other

12.5.2 Parameters

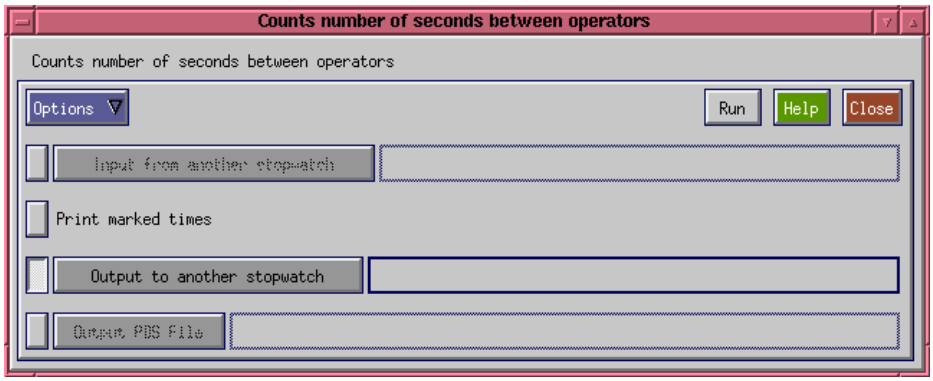

Figure 12.5: The `cstopwatch` kroutine GUI

The parameters for the `cstopwatch` kroutine are shown in its GUI pane (Figure 12.5):

- **Input from another stopwatch**: If set, will use this file as an input to calculate the difference of time in seconds between two calls to `cstopwatch`.
- **Print marked times**: If set, will print all the times marked so far (including the one generated with this operator)

- **Output to another stopwatch**: If set, will generate a file with the times measured so far. If not set or if **Print marked times** is selected the times measured so far will be printed out in the console.
- **Output PDS File**: If set, will output the time to a Polymorphic data object with dimensions $1 \times 1 \times 1 \times 1 \times 1$ with the last time measure that can be used in other applications.

Figure 12.6 shows how the `cstopwatch` operator can be used in a Cantata workspace. In this case we want to measure the execution time for the `BPNN Classify` operator. We put two `cstopwatch` operators, one "before" and another "after" the call to `BPNN Classify` and set the control connections so the execution order will be `cstopwatch` → `BPNN Classify` → `cstopwatch`. To have a better measure of the time it is better to ensure that right after the call to the operator being measure (in the example, `BPNN Classify`) **only** the `cstopwatch` operator will be executed, and all other operators will be executed after `cstopwatch`. Note also that there is a data connection between the two `cstopwatch` operators. This can be done also with more than two `cstopwatch` operators.

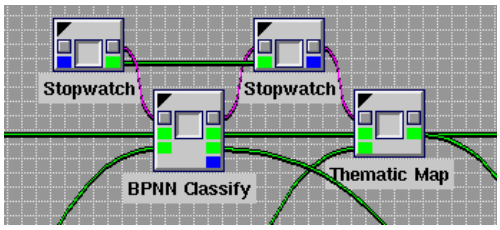

Figure 12.6: An example of usage of the `cstopwatch` kroutine in a Cantata workspace

Chapter 13

File Formats for the Classify Toolbox

13.1 Coord File Format

The Coord file format is a text format used to specify rectangular coordinates for ROI extraction (see 10.1.1) in files. The file format is very simple:

- Each line corresponds to a ROI
- Lines starting with `#` or `/` are considered as comments
- Lines with less than 7 characters on it are ignored
- Each line should have 4 coordinates: start column, start row, width, height.
- Coordinates for the ROIs can overlap

An example can be:

```
#coordinates for ROIs containing RED jelly beans
93 32 14 11
101 88 17 22
```

Of course, the file name and extension can be freely chosen.

13.2 Multiple Coord File Format

The Multiple Coord file format is a text format used to specify rectangular coordinates for multiple class ROI extraction. Files in this format are usually generated by the `cappendROIcoords` operator (see 10.2.1) to serve as input for the `cROIfrommcoords` operator (see 10.4.1). The file format is also very simple:

- Each line corresponds to a ROI
- Lines starting with `#` or `/` are considered as comments
- Lines with less than 7 characters on it are ignored
- Each line should have 5 coordinates: class index, start column, start row, width, height.
- Coordinates for the ROIs can overlap

An example can be:

```
#coordinates for ROIs for classes 1,2 and 3
1 23 42 17 9
1 19 3 42 72
2 109 107 12 40
2 118 120 9 7
2 125 122 2 10
3 50 47 12 8
```

Of course, the file name and extension can be freely chosen.

13.3 Class Specification File Format

The Class Specification File Format is a text format used to specify classes labels and display colors for creation of thematic maps with `cthematicmap` (section 11.3.1) or reports with `cclassreport` (section 11.4.1). The file format is:

- Each line corresponds to a class
- Lines starting with `#` or `/` are considered as comments
- The first token in a line is the class index - 1 is the index for the first class, 2 for the second, etc. The class 0 cannot be used, it is usually assigned to unclassified pixels in the images.
- The class indexes does not need to be in order, the order will be determined by the first parameter
- The next three tokens in a line are the red, green and blue values for the color for the thematic map. Color values must be on range 0-255.
- After these four initial tokens, all remaining text will be considered as the class label.

An example can be:

```
#file with class specifications for a simple classification task
2 0 0 255 This is the second class, represented by the color blue
1 255 0 0 This is the first class, represented by the color red
#note that the classes index is determined explicitly by the
#first parameter in each line
```

Of course, the file name and extension can be freely chosen.

13.4 A-Priori Probabilities File Format

The A-Priori Probabilities File Format is a text format used to specify a-priori probabilities for each class (see section 2.1.3). The a-priori probabilities will be used with some classifiers to bias the classification results. The file format is:

- Each line corresponds to a class
- Lines starting with `#` or `/` are considered as comments
- The first token in a line is the class index - 1 is the index for the first class, 2 for the second, etc. The class 0 cannot be used, it is usually assigned to unclassified pixels in the images.

- The second and last token in a line is a positive double value such as $0 < value \leq 1$, where *value* is the a-priori probability for that class. Values that are outside the range will generate an error.
- The class indexes does not need to be in order, the order will be determined by the first parameter.
- All classes in the classification task must have an entry in this file, otherwise an error will be generated.

An example can be:

```
#file with a-priori information for some classes
#class 2, a-priori probability is 0.75
2 0.75
#class 1, a-priori probability is 0.25
1 0.25
```

Of course, the file name and extension can be freely chosen. This file can be generated with the `cxapriori` program (not implemented in this version).

Appendix A

Operators on the Classify Toolbox

This tables shows all the operators on the Classify Toolbox, their description and index on this document. Only operators which are complete (including on-line documentation) are with the Index field filled.

A.1 Parallelepiped Classification

Binary Name	Icon Name	Description	Index
cparallel_signature	PARAL. Signature	Create a signature for a class for Parallelepiped classification	Section 3.2.1
cparallel_classify	PARAL. Classify	Classify an object using the Parallelepiped algorithm	Section 3.3.1
cparallel_printsig	PARAL. Print Signature	Print a Parallelepiped Signature	Section 3.4.1.1

Table A.1: Operators for the Parallelepiped Classifier

A.2 K-Nearest Neighbors Classification

Binary Name	Icon Name	Description	Index
cknn_signature	KNN Signature	Create a signature for a class for K-Nearest Neighbors Classification	Section 4.2.2
cknn_classify	KNN Classify	Classify an object using the K-Nearest Neighbors algorithm	Section 4.3.1
cknn_printsig	KNN Print Signature	Print a K-Nearest Neighbors Signature	Section 4.4.1.1

Table A.2: Operators for the K-Nearest Neighbors Classifier

A.3 Minimum Distance Classification

Binary Name	Icon Name	Description	Index
cmindist_signature	MINDIST Signature	Create a signature for a class for Minimum Distance classification	Section 5.2.1
cmindist_classify	MINDIST Classify	Classify an object using the Minimum Distance algorithm	Section 5.3.1
cmindist_printsig	MINDIST Print Signature	Print a Minimum Distance Signature	Section 5.4.1.1

Table A.3: Operators for the Minimum Distance Classifier

A.4 Maximum Likelihood Classification

Binary Name	Icon Name	Description	Index
cmaxlik_signature	MAXLIK Signature	Create a signature for a class for Maximum Likelihood classification	Section 6.2.1
cmaxlik_classify	MAXLIK Classify	Classify an object using the Maximum Likelihood algorithm	Section 6.3.1
cmaxlik_printsig	MAXLIK Print Signature	Print a Maximum Likelihood signature	Section 6.4.1.1

Table A.4: Operators for the Maximum Likelihood Classifier

A.5 Back-Propagation Neural Network Classification

Binary Name	Icon Name	Description	Index
cbpnn_signature	BPNN Signature	Create a signature for a class for Back-Propagation Neural Network classification	Section 8.3.1
cbpnet_train	BPNN Train	Create and train a Back-Propagation Neural Network	Section 8.4.1
cbpnet_class	BPNN Classify	Classify an object using a Back-Propagation Neural Network	Section 8.5.1
cbpnn_sigcheck	BPNN Check Signature	Check the Back-Propagation Neural Network signatures for repeated values	Section 8.6.1.1

Table A.5: Operators for the Back-Propagation Neural Network Classifier

A.6 Histogram Overlay Classification

Binary Name	Icon Name	Description	Index
chover_signature	HOVER Signature	Create a signature for a class for Histogram Overlay classification	
chover_classify	HOVER Classify	Classify an object using the Histogram Overlay algorithm	
chover_printsig	HOVER Print Signature	Print a Histogram Overlay signature	

Table A.6: Operators for the Histogram Overlay Classifier

A.7 Table Look-up Classification

Binary Name	Icon Name	Description	Index
ctlu_signature	TLU Signature	Create a signature for a class for Table Look-Up Classification	
ctlu_classify	TLU Classify	Classify an object using the Table Look-Up algorithm	
ctlu_printsig	TLU Print Signature	Print a Table Look-Up Signature	

Table A.7: Operators for the Table Look-up Classifier

A.8 Pre-Classification Utilities

Binary Name	Icon Name	Description	Index
cROIfromcoords	ROI from coords	Extract rectangular ROIs from an image using coordinates	Section 10.1.1
cappendROIcoords	Append ROI coords	Append and label multiple ROI coordinates files	Section 10.2.1
ccompressROI	Compress Masked ROI	Eliminate masked points in ROI making single row object	Section 10.3.1
cROIfrommcoords	ROI from multiple coords	Extract ROIs from multiple labeled coords for samples evaluation	Section 10.4.1
csigappend	Signature Append	Append signatures with labels to use with some classifiers	Section 10.5.1

Table A.8: Pre-Classification Utilities

A.9 Post-Classification Utilities

Binary Name	Icon Name	Description	Index
celementmode	Element Mode	Applies a mode operator over classification results (element)	Section 11.1.1
cprobmode	Prob. Element Mode	Applies a probabilistic mode operator over classification results (element)	
cthematicmap	Thematic Map	Create a Thematic Map for a classification result	Section 11.3.1
cclassreport	Class. Report	Create a report with classification results	Section 11.4.1
ccompare	Compare Results	Compare two classification results and create confusion matrix	Section 11.5.1
cxinspector	Inspector	Interactively inspects results of classification	Section 11.7.1
cxretouch	Interactive Retouch	Interactively retouch (paint) a classified image	

Table A.9: Post-Classification Utilities

A.10 Other Utilities

Binary Name	Icon Name	Description	Index
mcovar_mask	Mean and Cov. Matrix	Compute Mean and Std. Dev. Vectors and Covariance Matrix	Section 12.1.1
kunmapdata	Unmap Data	Unmap data from RGB images	Section 12.2.1
ksubstitute	Substitute Values	Substitute or mask values in the value segment	Section 12.3.1
kinsertmask	Insert Mask	Insert Mask from Object 1 into Object 2 (expand if necessary)	Section 12.4.1
cstopwatch	Stopwatch	Count number of seconds between operators	Section 12.5.1

Table A.10: Other Utilities

Appendix B

Installing the Classify Toolbox

B.1 Obtaining the Classify Toolbox

The latest version of the Classify Toolbox can be downloaded from the Ejima Lab Khoros Page (see section 1.5 for its URL). As soon as a new version is available, it will be also submitted to the Khoral FTP Server, and will be announced in the Khoros Mailing List. For the addresses of the Khoral FTP Server and instructions on how to join the Khoros Mailing List please refer to the Khoral Research Home Page (URL also listed in section 1.5).

Before downloading and installing the Classify Toolbox make sure you have:

- Khoros 2.1 installed. The present version of this toolbox was designed from scratch with Khoros 2.1 with all patches applied, and will **not** work with previous versions. If you don't have Khoros 2.1 installed, I'd suggest you to get it, refer to the Khoral Research Home Page for instructions on how to obtain Khoros 2.1.
- The toolboxes **Design**, **Datamanip** and **Matrix** installed. The Classify toolbox depends on these toolboxes.
- At least 18 MB of disk space to download and install the Toolbox.

B.2 Installing the Classify Toolbox

1. Download the toolbox. Go to your Khoros directory and uncompress the files there. This will create a directory `Classify`.
2. Edit your `.Toolboxes` file and add an entry like
   `CLASSIFY:/home/santos/Khoros/Classify;machdefs=false,globaldir=true`
   Change the `/home/santos/Khoros/Classify` part to the directory you used to install the toolbox. Make sure that the `machdefs` and `globaldir` are set to `true` or `false` accordingly to the other entries on the `.Toolboxes` file. In case of doubt, refer to the `Toolboxes` file in the directory you have Khoros installed.
3. Go to the toolbox directory and do the commands:
   ```
   cd objects
   make Makefile
   make Makefiles
   make clean
   make install
   ```

If there weren't any compilation errors, the toolbox should be ready to use. If you get lots of compilation errors about functions that cannot be found, go to the source directory of the

operator which generated the error and check its `Imakefile`, if there is a line
`#define HasFortranDependence NO`
change it to
`#define HasFortranDependence YES`
and recompile the program.

Some users reported that they must change all Imakefiles in order to compile the toolbox. As far as I know this is not necessary, but I didn't had the chance to compile it under other architectures and I suspect that putting the `#define HasFortranDependence YES` in **all** operators will create larger executable files in some architectures. I hope to solve this in a future version.

B.3 Technical Information

The Classify Toolbox was developed in a portable computer with a Pentium 90 Mhz microprocessor, running FreeBSD version 2.2-BETA and using Khoros 2.1 with all patches available until February 1998 applied.

I was able to compile and run the programs on another FreeBSD machine. If you have any problem with the programs on this toolbox, please check the list of known bugs in appendix C and if it is not there, contact the author (see section 1.5).

Appendix C

Future Work

C.1 List of things to do and bugs to fix

List of operators that are not working in this alpha release (will be when the non-beta release is out):

- An operator to print the trained network for the Back-Propagation Neural Network classifier
- All operators for the Histogram Overlay classifier
- All operators for the Table Look-up classifier
- The `cxapriori` operator
- The `cprobmode` operator

Future work on this toolbox include (more or less in the order I'm planning to do):

- Fix the bugs (see below).
- Complete the toolbox with the operators listed above.
- Add routines and methods to evaluate samples and signatures.
- Do visual versions for some classifiers where the user can verify the classification result as it is being generated.
- Add some unsupervised classifiers - maybe.
- Try to enhance some of the algorithms for speed.
- Enhance mask support - for speed I make the assumption that all pixels in an element vector are masked with the same value as the first. Classification errors will result if it is not the case.

List of known bugs:

- For a reason I wasn't able to trace, some classifiers finish the classification task but the glyphs in Cantata keep running. Usually re-running the glyph will work. If a classifier is taking too much time, and you chose to create an information file for the classifier, see the bottom of the file to check which was the last pixel that was classified, if the file shows that the whole image is classified but the glyph is still running, re-run it from Cantata. I had this problem in two FreeBSD machines I used to develop and test the toolbox, and I sometimes had the same problem with the `putimage` operator, so maybe it is a local problem.

- In the `cROIfromcoords` operator, some functions give unexpected results. For example, given a single rectangular region and choosing that everything outside the region is to be masked and asking for auto-cropping will give a rectangular masked region as output. It is not really a bug since it does what the user ask for, but can lead to unexpected results if the user is not sure about its usage.

You can contact the author by e-mail (listed in section 1.5) with bug reports, questions and suggestions for improvement of this toolbox. There is a questionnaire distributed with the toolbox (file `$CLASSIFY/repos/ToolboxInfo`) with what I plan to do next in regard to this toolbox, if you can please answer it and send it to me by e-mail.

C.2 Changes from the alpha release to the alpha-2 release

- Several classification and related routines were rewritten to be faster. The time for the classifiers implemented in the Alpha version and this release are shown in table C.1. The times were measured in a Pentium 90 MHZ notebook running FreeBSD (single user). Times in other machines (probably with other programs running simultaneously) may vary. For the next release the Back-Propagation Neural Network trainer will also be rewritten.

- The operator `croifromcoords` was renamed to `cROIfromcoords` (section 10.1.1).

- The signature extraction for the KNN classifier (`cknn_signature`, section 4.2.2) have a new option to avoid subrepresentation of classes.

Classifier	Alpha release	Alpha-2 release
Parallepeliped classification	281	7
K-Nearest Neighbors Signature (reducing 20000 points to 2%)	1007	790
K-Nearest Neighbors Signature (reducing 750 points to 2%)	6	3.5
K-Nearest Neighbors classification (K=5)	603	50
K-Nearest Neighbors classification (R=10)	(don't have the times)	197
Minimum Distance classification (no rejections)	44	5
Minimum Distance classification (rejecting 2 std.dev)	61	6
Maximum Likelihood classification (no rejections)	450	52
Maximum Likelihood classification (5% rejections)	553	55
Maximum Likelihood classification (25% rejections)	553	52
Back-Propagation Neural Network classification	91	32

Table C.1: Time in seconds for running the operators

Bibliography

[1] Robert M. Haralick and Linda G. Shapiro, *Glossary of Computer Vision Terms*, Pattern Recognition, Vol. 24, No. 1, pp. 69-93, 1991.

[2] John A. Richards, *Remote Sensing Digital Image Analysis - An Introduction.* (Second, Revised and Enlarged Edition), Springer-Verlag, 1993. ISBN 0-387-54840-8.

[3] R. Beale and T. Jackson, *Neural Computing: An Introduction*, Adam Hilger (IOP Publishing), 1990. ISBN 0-85274-262-2.

[4] Yoh-Han Pao, *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley Publishing Company. ISBN 0-201-12584-6.

[5] K. S. Fu, *Applications of Pattern Recognition*, CRC Press, 1986. ISBN 0-8493-5729-2.

[6] Richard O. Duda and Peter E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, Inc., 1973. ISBN 0-471-22361-1.

[7] John R. Jensen, *Introductory Digital Image Processing - A Remote Sensing Perspective*, Prentice-Hall, 1986, ISBN 0-13-500828-X.

[8] Y. Inomata and S. Ogata, *Supervised and unsupervised classification by histogram overlay techniques*, Int. Journal of Remote Sensing, Vol. 14, No. 14, pp. 2605-2616, 1993.

[9] Timothy Masters, *Practical Neural Network Recipes in C++*, Academic Press Inc., 1993. ISBN 0-12-479040-2.

[10] Arun D. Kulkarni, *Artificial Neural Networks for Image Understanding*, Van Nostrand Reinhold publishers, 1994. ISBN 0-442-00921-6.

[11] B. Müller, J. Reinhardt and M. T. Strickland, *Neural Networks - An Introduction*, Second Edition, series *Physics of Neural Networks*, Springer-Verlag, 1995. ISBN 3-540-60207-0.

[12] Leslie Lamport, LaTeX *User's Guide & Reference Manual*, Addison-Wesley Publishing Company, 1986. ISBN 0-201-15790-X.

[13] E. D. Raggett, *HTML Tables*, RFC 1942, 05/15/1996.

[14] Khoros Manuals