Internal Use Only（非公開）

TR－SLT－0065

Clustering of Backchannels in Japanese Spontaneous Speech

Wenzel Svojanovsky, Rainer Gruhn

March 30, 2004

概要

Human language, especially spontaneous speech, carries more information than just spoken words. This research analyzes prosodic features of the backchannel "うん" based on F0, duration, and energy of the signal.

Training and test data are subsets extracted from a 150 hour corpus of spontaneous conversational speech from one Japanese female collected in the ESP project. The data is partially labeled with 8 types of intentional labels by human experts.

The "うん" segments are automatically clustered and classified into one of several speech act classes.

（株）国際電気通信基礎技術研究所
音声言語コミュニケーション研究所
〒619－0288「けいはんな学研都市」光台二丁目2番地2 TEL：0774－95－1301

Advanced Telecommunication Research Institute International
Spoken Language Translation Research Laboratories
2-2-2 Hikaridai "Keihanna Science City" 619-0288,Japan
Telephone:+81-774-95-1301
Fax :+81-774-95-1308

Contents

Chapter 1

Introduction

1.1 Motivation

Spontaneous speech is more difficult to understand than read speech. This applies both to humans and machines. Indeed, the performance of automatic speech recognition drops substantially, when analyzing spontaneous speech. Today speech recognizers are powerful enough to recognize the words in a dialog. But still the meaning can not be "understood".
Indeed, a transcription of a spontaneous dialog give hardly enough information of the complete meaning. Humans use *backchannels*, such as "mhm" and "hm" to control the dialog flow. What they say, is not important at this time, it depends on how they say it.

Those backchannels can have several meanings. In case of "un" in Japanese spontaneous speech, can have either have the meaning of "disagree" and "agree". It also can just give a signal of listing.

Only from the transcription, a meaning is hardly to understand. This is why speech recognizers should be able not just to understand the word "un" it should get an idea of the meaning.

A Japanese female was wearing a portable MiniDisk recorder and a headset and recorded her speech all day. This approach deals with an excerpt of this data, containing 2649 token of the utterance "un". 482 token are labeled to

one of the groups "listen", "understand", "interest", "agree", "agree overall", "call back", "understand but disagree", and "emotion".

1.2 Theory

1.2.1 What Is Prosody

Prosody is (according to) both the phenomena, that involve the acoustic parameters of pitch, duration and intensity, and the phenomena, that involve the phonological organization at levels above the segment. A appropriate definition is

> Prosody is a systematic organization of various linguistic units into an utterance or a coherent group of utterances in the process of speech production. Its realization involves both segmental features of speech, and serves to convey not only linguistic information, but also paralinguistic and non-linguistic information.

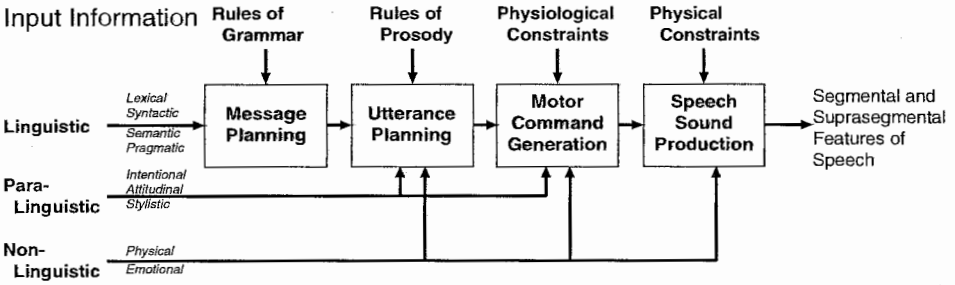

Figure 1.1: Processes by which various types of information are manifested in the segmental and suprasegmental features of speech

The process of speech sound production generates individual characteristics of speech. Figure 1.1 displays this multi–stage process and explains the difficulty of finding clear and unique correspondence between physically observable characteristics of speech and the underlying prosodic organization of an utterance.

1.2.2 Prosodic Features

In this approach, the following prosodic features of the categories duration, pitch, energy and glottal characteristics are excerpted from the data and stared in prosodic feature vectors.

dur: the duration of the token

pmean: the mean power during the backchannel

pmin: the minimum power in the backchannel

pmax: the maximum power in the backchannel

ppos: the position of the maximum power relative to the duration of the backchannel

fmean: the mean pitch during the backchannel

fmin: the minimum pitch in the backchannel

fmax: the maximum pitch in the backchannel

fpos: the position of the maximum pitch relative to the duration of the backchannel

fvcd: the occurrence of voiced pitch relative to the duration of the backchannel

fgrad: the the gradient from the position of the minimum and the position of the maximum of the pitch, relative to the duration

adduction quotient (H1-H2): the difference between the first and the second harmonic indicates a vowel. The value changes when the open quotient rises. Researchers use *(H1-H2)* as an indication of open or adduction quotient.

spectral tilt (H1-A3): the amplitude of the third formant relative to the first harmonic *(H1-A3)* is an evidence for spectral tilt and displays the abruptness of the cut off of the airflow.

Chapter 2

Experiments

2.1 Data

The data for the experiments consists of 2649 token of "un". 484 of these tokens are labeled to one out of eight the classes.
The prosodic feature vectors are extracted from the signal by the *pros.py* application.

For the experimental configuration, the vector set needs to be divided into three subsets:

training set: This set consists of the unlabeled tokens. It forms the set which is used for clustering.

development set: This set consists of about 50% of the labeled set and is used to assign labels to the clusters and build a classifier.

test set: The remaining labeled vectors for testing the performance of the classifier.

Table 2.1 shows the frequency of the tokens per class and how they are segmented over the two labeled sets.

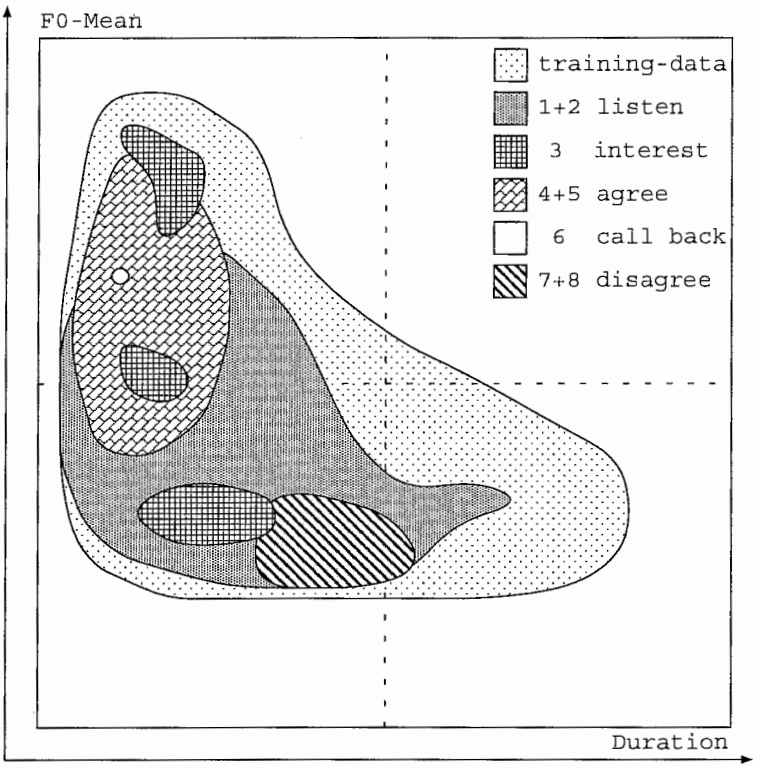

Figure 2.1: Visualization of training and development set. Class 1+2 covers a large part of the space, while class 3 is scattered. Class 6 consists of only one vector. Class 7+8 distinguish from other classes but 1+2.

Observations of the data came to the conclusion that the classes 1 and 2, 4 and 5, and 7 and 8 can be merged. They sound similar, and they are hardly separable. Also the meaning of the speech act is related. Figure 2.1 illustrates the distribution of the classes in the feature space, reduced to the dimension *F0mean* and *Dur*.

number	speech act	quantity	dev	test
1	listen	377	189	188
2	understand	46	23	23
3	interest	19	10	9
4	affirm	14	7	7
5	affirm overall	5	2	3
6	callback	1	1	0
7	disagree	15	7	8
8	emotion	5	3	2

Table 2.1: Label number and according speech act and the frequency in the development set (dev) and the test set (test)

2.2 Configuration

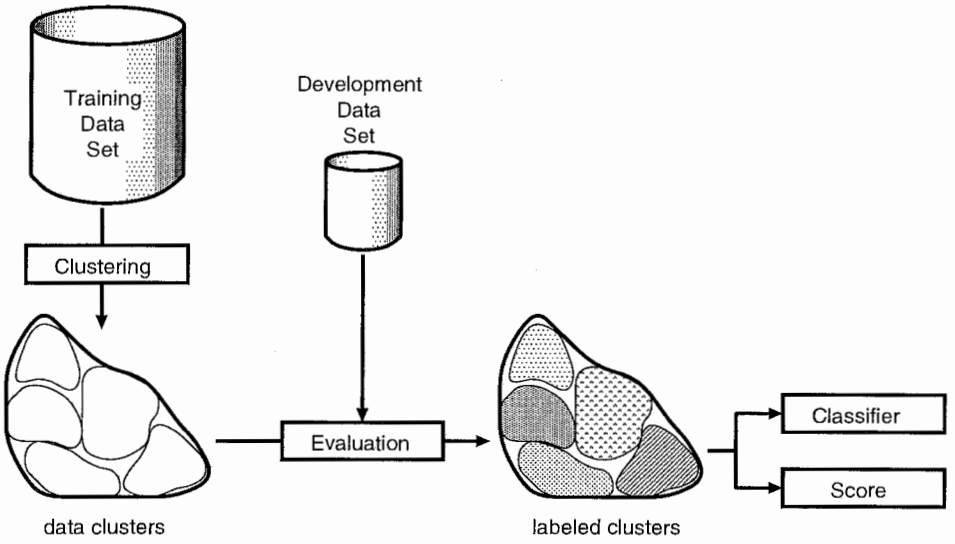

Figure 2.2: The configuration of the clustering algorithm

The figure 2.2 shows the configuration of the setup. The python module *Cluster.py* clusters the *training set* into a given amount of clusters. Several different experimental setups are tested in this approach.

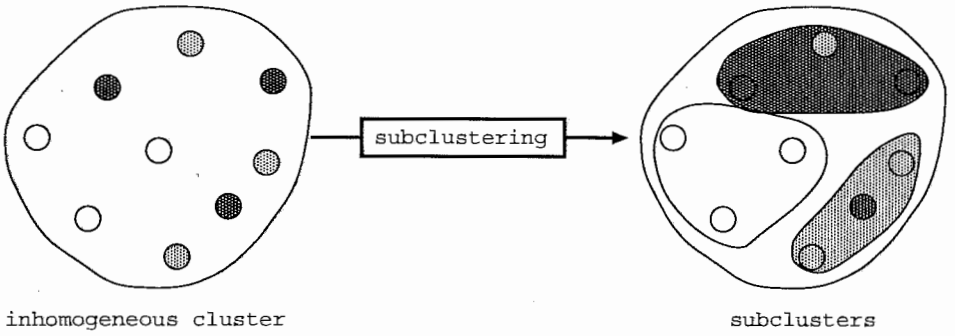

Figure 2.3: Inhomogeneous clusters are subclustered

After the clustering labels are assigned to the clusters. Each cluster is represented by each centroid. The vectors of the *development set* are assigned

to the clusters using the *nearest neighbor* method.
According to the labeled vectors in the clusters, a label is assigned to the clusters. Since the labels are spread inhomogeneously over the clusters, some clusters are subclustered. The process of subclustering is displayed in Figure 2.3. After the subclustering the clusters can remain inhomogeneous.
After the labels are assigned to the clusters resp. subclusters, the classifier can be generated and is test on the *test set*.

2.3 Experimental Setup

In total 72 different experimental setups are tested during this approach. These setups different in the clustering method, the clusters distance, the number of clusters and the vector distance measure.

2.3.1 Clustering Methods

The following cluster methods are tested during the experiments.

Bottom Up: This method starts with as many clusters as vectors in the set. Each vector belongs to its own cluster. The two nearest clusters are merged. This step continues until the number of clusters is reduced to a specific value. The distance of the clusters is declared separately.

Top Down: In this method all vectors are initially gathered in one big cluster. Then this cluster is split into two new clusters. Any vectors in this cluster are assigned to one of two new clusters which are represented by two vectors from this cluster. These two vectors can be the two furthest neighbors in the old cluster. Since the calculation of the furthest neighbors cost time, an improved algorithm termed as *fast split* is established. In latter one, the furthest vector to the average center is calculated. This vector is the first representative of the new cluster, the second one is the furthest neighbor to the first.

k–Means: The k–Means algorithm classifies all data to a given number of clusters. The representatives of the clusters are recalculated and the clusters are emptied. Then the data is classified again. With each step, the shape of the clusters become clearer. The algorithm stops after a specific number of steps or a special criteria. The quality of the cluster strongly depends on the initial clusters. In this approach the initial clusters where represented by arbitrarily chosen vectors from the training data set. Three different initial vectors are tested. The algorithm runs 5 times.

Split & Merge: This method is a combination of the *bottom up* and *top down* approach. The widest cluster is split into two new clusters until a certain number of clusters is produced. Then the nearest clusters are merged again. The number to clusters to produce changes with every single step.

Bottom up + k–Means: In this method the clusters will be produced with the *bottom up* method explained above. But during the clustering process after a frequently number of merging steps, the *k–Means* algorithm based on the already existing clusters runs. Then the *bottom up* continues.

Split & Merge + k–Means: This a extended version of *split & merge* algorithm. Every time the algorithm swaps from merge mode to split mode and from split mode to merge mode the *k–Means* algorithm runs to reshape the produced clusters.

2.3.2 Cluster Distance

Four types of distance measure are used used to identify a pair of clusters for merging, so it plays a major role during a *bottom up* related clustering.

Center Distance: The average center vector of the cluster is calculated. This center is an artificial vector and does not refer to an audio signal. This kind of cluster distance uses the distance between the center vectors.

Representative Distance: Different to the center is the representative data vector of the cluster. After the calculation of the cluster center, the nearest vector in the cluster to the center becomes the representative vector. The use of the representative is less accurate, especially in clusters with a low number of vectors, but its advantage is the high acceleration of the clustering by the use of a recalculated distance matrix and the referring to an audio signal.

Average Vector Distance: In this method the average distance value is the mean distance of *all* vector pairs between both clusters.

Furthest Neighbor: This method measures the distance between the furthest vectors from both clusters, which is the maximum distance of *all* vector pairs between both clusters.

2.3.3 Vector Distance

The *Euclidean distance* between two vectors is used in this approach. To emphasize some prosodic features, a weighting factor during the distance calculation is added to scale the significance of a single feature.
In total eight different weightings are tested. Therefore a *Split & Merge + k–Means* algorithm tested all combinations of features weightings using the factor 0 and 1. Only weighting are regarded, which operated in a 3 or higher dimensional feature space. These 8100 combinations are tested to approach an optimal weighting.

2.3.4 Number Of Clusters

During this approach the data are clustered into 20 or 40 clusters. 20 and 40 are rather arbitrarily chosen numbers. 20 seems to be a reasonable number, because the visualization of the tokens indicated, that one cluster per class will be insufficient, especially for the widely spread classes 1+2 ("listen") and 3 ("interest").
If the number of clusters is increased further, e.g. to 40, the clusters become too specialized. A very high number would probably cause a very good performance on the *development set*, but a poor performance on the *test set*.

2.4 Subclustering

If there are more than 20 vector assigned to one cluster and the labels are inhomogeneous, this cluster gets subclustered.
The subclusters are generated by a *k–Means* algorithm with random initialization.

2.5 Classification And Classifier Generation

The classifier is generated from the clusters and the *development set*. The vectors in the development set are assigned to the single clusters and a meaning is extracted from the labeled vectors in the clusters.
If the labels are too scattered in one cluster, the cluster is subclustered.

If a feature vector is classified, the *Classifier* looks up the nearest cluster from a list of the cluster centroids. If this centroid is bound to a label, the classifier will return the label of that centroid, else this cluster is subclustered and the nearest subcluster centroid is looked up.
This is a 2-level classifier, because the information of the subclustering is not used on the first level. Adding the subcluster centroids to the other centroids would influence the shape of the neighboring clusters.
The schematic structure of the classifier is explained in Figure 2.4.

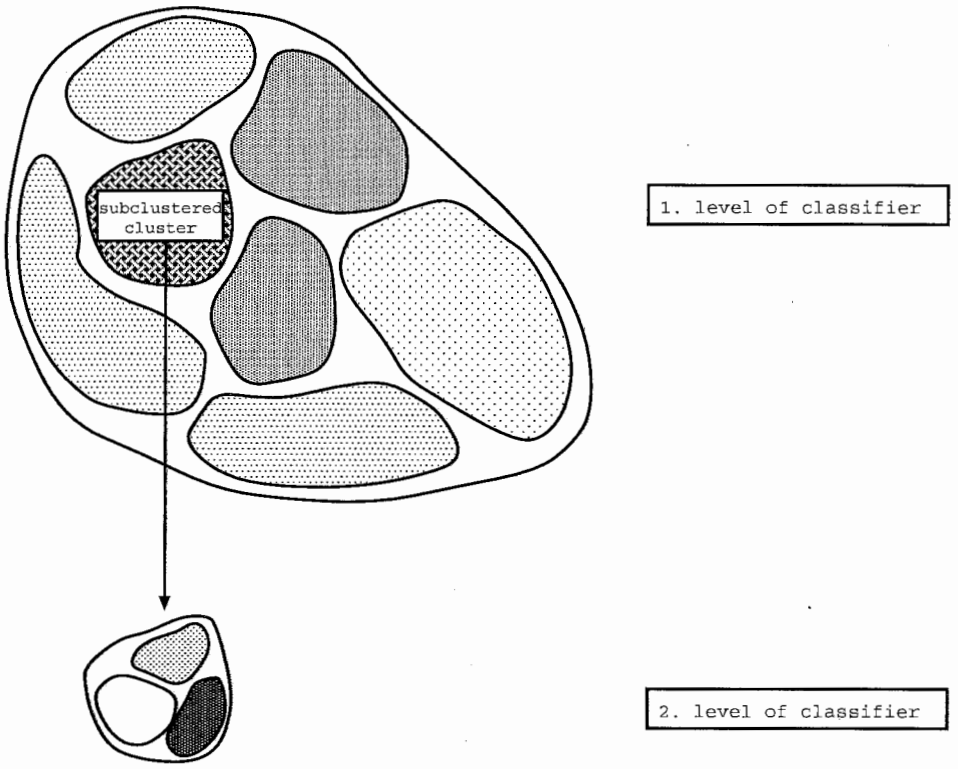

Figure 2.4: Schematic illustration of the 2-level classifier. A feature vector is assigned to a cluster on the first level. If this cluster is subclustered, the vector is assigned to one of the subclusters.

Chapter 3

Conclusions

A *bottom up* clustering method and the *furthest neighbor* achieved the best results at the evaluation. This algorithm performed an averaged class recognition rate of 64.5%. This classifier classifies 80.5% of the vectors in the test data correctly.
Some prosodic features seem to be more significant than others. So *duration* and *pitch* features are more important for a reasonable classifier than *energy* features and *glottal characteristics*.
The confusion matrix of the best performed classifier is given in 3.1. A clear separation of the classes "disagree" and "agree" seem to be possible, while "backchannels" like "listen" and "understand" hardly stand out from the other classes. A clear classification of "interest" also seems to be difficult.

The portability of this approach to another backchannel "honma" is tested but need to be investigated further.

	classified as							
	1+2		3		4+5		7+8	
class	#	%	#	%	#	%	#	%
1+2	181	85.8	5	2.4	9	4.3	16	7.6
3	3	33.3	4	44.4	2	22.2	0	0.0
4+5	2	20.0	1	1.0	7	70.0	0	0.0
7+8	4	40.0	0	0.0	0	0.0	6	60.0

Table 3.1: Confusion matrix on test set using a weighted 2-level classification. Absolute number (#) and percentage of vectors in each class is given. An averaged class recognition of 65.1% and a token recognition rate of 82.5% (198/240) is performed.

Chapter 4

Software Documentation

4.1 Overview

This chapter will familiarize the software which is used during this approach.

Figure 4.1 shows the flowchart of the whole experiment. The audio files are stored in a list file. The *pros.py* application will extract the prosodic feature vectors and save them as "ndv" file. These *normalized data vectors* can be loaded by the *Cluster.py* python module. A meaning is assigned to the resulting clusters and a classifier is generated with the *SubCluster.py* module.

4.2 pros.py Application

This application calculates prosodic features from a short signal and allows a convenient and useful visualization of this tokens.

This GUI is written for "Python 2.3" and the "The Snack Sound Toolkit 2.2.3". The installation of both programs will not be discussed in this manual. This software may also run on other Python and Snack versions, but it has not been tested.

4.2.1 Prosodic Features Extraction

When the GUI is started, the empty field is shown. `File -> Load` opens a small dialog to load data. Several file types can be chosen here. The option "`Filename -- Label File`" will load a file, which contains the filenames of the audio signals and the labels, which this file gets. All other calculations will be done automatically then.
It is important to type the complete filename including the file extension. Latter one is usually added automatically, but not in this case.
The structure of the different file formats are discussed in section 4.3.2. Each line of the file contains the filename and the label, separated with a ",", like in the example. Example to calculate prosodic features:

```
filename01.wav, label1
filename02.wav, label2
filename03.wav, label3
```

The program will extract the prosodic features from "filename01.wav", create a sound object and add the label "label1" to this sound object. Then the extraction of the features from "filename02.wav" with the label "label2", and so on.

After these calculations the save dialog will open to get sure, that the calculated data does not get lost.
After saving the data, the main field in the GUI shows several white circle. Each circle represents a sound token.

4.2.2 The GUI

On the picture 4.2 shows the GUI of the pros.py application. The main canvas illustrates the circles which represent the single prosodic objects. When the mouse points on one of these circles, the object will appear with a yellow circle around it. The token is active then. The meters on the right display the different prosodic features. Pressing the "L" button, the yellow circle will jump to the directly former object. Pressing the "L" button again, the activation will return to the first object. This functionality makes the comparing of two objects with each other very easy!

In the file menu reveals four option: *Save, Load, Import,* and *Quit.*

Save: All tokens, independent if visible, hidden, selected,... are saved to a file. The file format can be chosen from the option menu. The extension will be added to the filename in the entry.

Load: New vectors can be added by loading them from a file. This dialog offers five file types. The "lst"–file format loads clusters created with the *Cluster.py* module. The option "Filename – Label File" will calculate the prosodic features from a signal. This process is explained above.
The other file formats are discussed in section 4.3.2.

Import: From older approaches, the resulting vectors can be imported. This function is not needed anymore. These vectors did not use the gradient of the pitch.

Quit: Quit the application.

The *pros.py* application allows several functionalities to handle the tokens. One token is always *active*. It is marked with a yellow circle around is. The prosodic features of this token are displayed in the meters on the right.
A token can be in several states. In can be *visible, selected*, and *hidden*. If a token is *hidden* it cannot be selected. A hidden token can *never* be selected. A hidden token is also not visible.
A *selected* token can be visible and invisible. In both states it can be moved in the canvas, be added to a group or a cluster, deleted, ... A click with the left mouse button (LMB) will select this token and unselect all other selected tokens. If the `Crtl`–key pressed during the click, the other tokens will not be unselected. The token will chance the state of selection. So a single token can also be unselected with `Ctrl`+LMB. A token is *visible*, if the constraints from the meters are fulfilled. It it possible to change the meters "viewport", which defines constraints to the canvas. A token is not visible then, if is is outside of the viewport of the meters. But it still can be selected, so moving, deleting, grouping, ...effect this token, if it is selected. A hidden token is also not visible, but since it cannot be selected, no action effects the object.

The *Select* menu offers two options to hide and unhide token. "Hide Selected" will set the selected token to *hidden* state and unselect them. "Unhide All" will set all hidden tokens to *not hidden*. Since then, they can be selected

and effected again.

The *meters* on the right side of the GUI show a thin white line on the top and bottom. Keeping the LMB pressed and drag along the meter, will change the meter's viewport. Only tokens, whose value of that feature is in this viewport are visible. The meters' viewport can be reset with `Select -> Reset meters`.

The coordinates of the canvas can be selected freely. Therefore next to each coordinate axis an option menu can determ the prosodic feature, which shall be displays in the canvas. In total there are three axis. Two for the main canvas and a third for another small canvas, which displays a third dimension.
Additional to the prosodic features three extra features can be displayed, the *X, Y* and *Z* feature. The value will be 0 at initialization. A moving of the tokens in this axis will be saved in the datavector of the "adi"–file format.

All events which change the state of the canvas causes a *redraw*. This redraw can also be called by pressing the "R"–key. The redraw causes all tokens to return to its original place in the canvas. The position in the X, Y, and Z feature will not change.
These events that cause the redraw is the loading of new vectors, the manipulation of the viewport and a change in the axis.

A token can belong to one or several groups. If a token belong to a group, the upper part of the circle will appear in the specific group color. A token can belong to more than one group, in the case the color if the first group will be displayed.
It can also belong to a cluster, which is changes the color of the lower half of the token. The group names and group colors can be changed in the setup file, which will be discussed soon.

A click on the right mouse button (RMB) will pop up a small menu, which allows a playing of the audio, adding to a group, hide, and delete the token. *Play* plays the "wav"–file which is assigned to that token. `Ctrl`+LMB will also plays the audio without popping up this menu.
Add to Group shows all groups and allow to add this the selected tokens to the group. The group name "no group" will delete the token from all groups it belongs. The "Group menu" is not implemented yet. It should become a menu where the group colors and group names can be changed. In the moment this has to be done by the setup file.

Delete from Group is only active, if the token belongs to a group. It displays all groups the token belongs and allow to delete the token from a specific group.
Hide hides the token. It will become invisible, unselected and is not effected by any event anymore. But is is not deleted. It can be recovered by `Select -> Unhide All` and is not hidden then anymore. When data is saved to a file, also the hidden objects will be saved.
Draw Spectrum is not implemented in this version. It should draw the spectrum of this token.
Delete deletes the object. It will not be saved or can be recovered again.

Additional to the already explained features, the *Select menu* offers further useful functions. All events caused by the select menu only effect the selected token. So eventually the function *Select All* should be called first from that menu. *Select Visible* will select all visible objects. All other previous selected objects will get unselected. The use of *Invert Selection* selects all unselected tokens and unselect all selected token.

Since the label of the tokens may be illustrated with a color, the select menu offers the function *Group by label*, which adds *all selected* tokens to the group which has the same name as the label.

Select Group will invert the selection of all tokens which belong to that specific group (this does not mean the first group). So a selection of all tokens but the token from a specific group can be selected by `Select All` + `Select Group`.

4.2.3 Setup Of The pros.py Application

When the pros.py application is started, it tries to execute the file "pros.setup". This file makes a redefinition of some global variables possible. If the file does not exist, the default values will started, which are optimized for the "un" data.
For example in this file the floor and ceiling values of the meters, the group names, and the group colors.

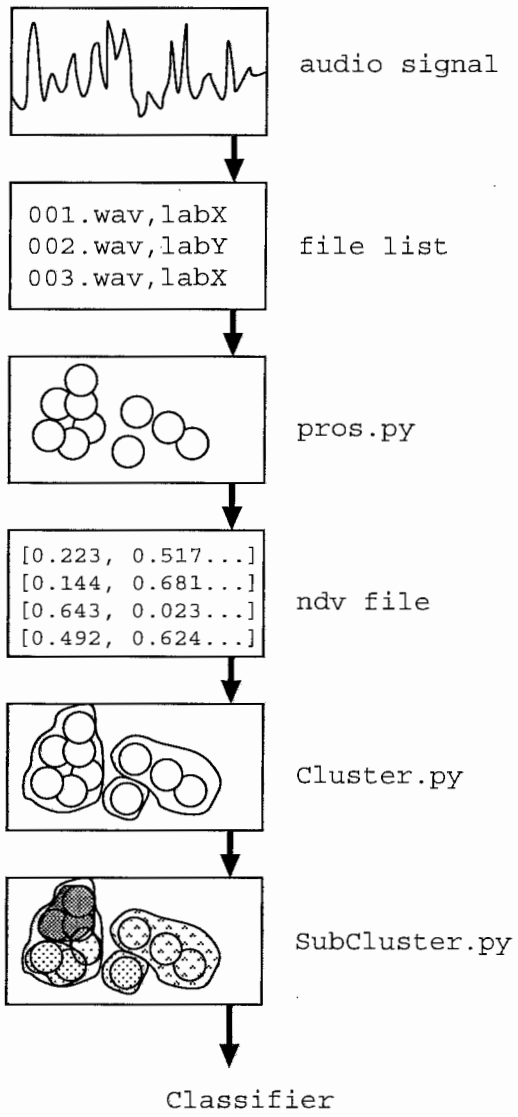

Figure 4.1: Flowchart of the experiment

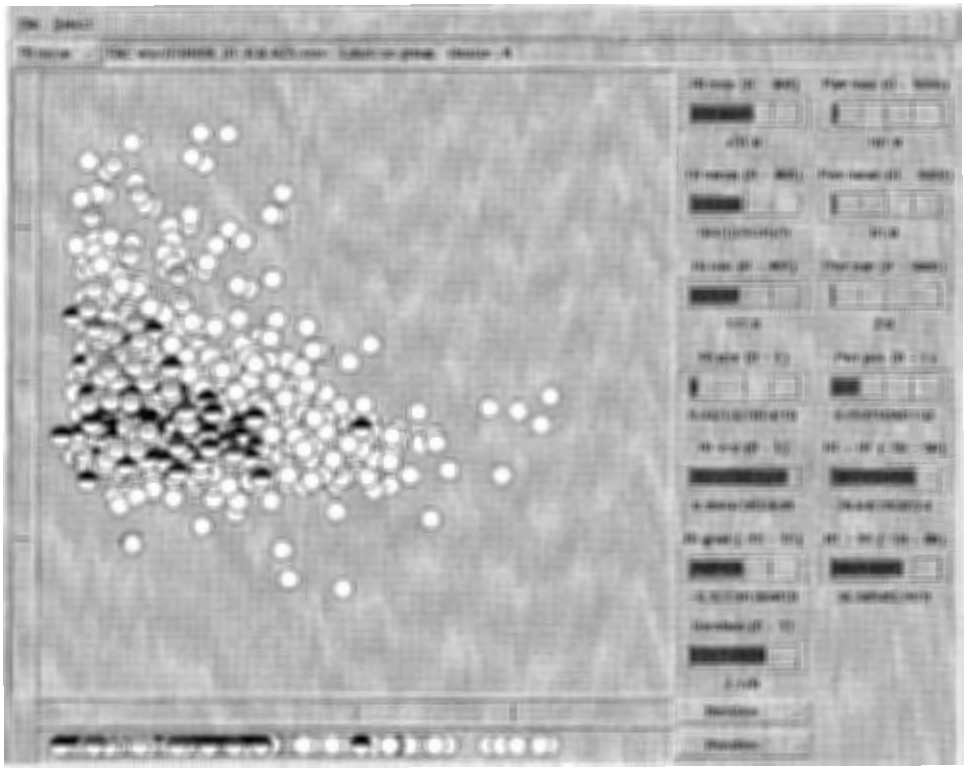

Figure 4.2: Visualization GUI pros.py application

4.3 File Format

4.3.1 Vector Files

The classifier in this approach accepts only the "ndv" format, which stands for *normalized data vector*. To create this file, the "pros.py" application is able to export the data in this file format. The application handles five kinds of files.

adi (All Data Information): This file saves all information of the sound objects, which can be given added with pros.py. All objects are represented by a list of data. The list is similar to the one which is used in the *dvc* file. In this case, additional information is appended to that list. The special coordinates X, Y, and Z, which can be chosen in the "pros.py" application is saved, also a list of the groups the objects belongs and the number of the cluster. The first line in this file is a header, which explains the order of the features.

dvc (Data VeCtor): In this file all relevant prosodic information is stored. The first line is a header, which illustrates the order of the features in the feature vectors. Each line of the file contains a list, which can easily be loaded on python. This list contains all the information of the feature vector. The single coefficients are not normalized. It is the exact prosodic feature value, which is extracted from the audio file.

ndv (Normalized Data Vector): This is the file which is handles by the classifier and the clustering program. The design of the file is the same like in the *dvc* file format. But in this case, all coefficients are normalized. This Normalization is done by the "pros.py" application. The floor and ceiling value for the single feature group can be declared in the "pros.setup" file.

lst (List): This file does not use a header. It contains a list of ndv files, which are created by the cluster application. This way a complete list of clusters can be loaded. Each file represents its own cluster. The number of the cluster is the order the filenames.

Filename - Label File: To calculate prosodic features from audio files, a list of these files and their labels can be loaded in "pros.py" application and the feature vectors will be calculated automatically. The label and the filename are separated with a comma. A file can look like this:
file01.wav,backchannel
file02.wav,happy
file03.wav,backchannel
file04.wav,understand

When a file is loaded, the header is ignored. Changing the order of the features in the header will not effect the vectors.

The following code shows an example, how a list can be loaded in python:

```
## open the file
file = open ('vectors.ndv', 'r')
## read file content
content = file.readlines()
## close file
file.close()
## content is a list of strings

## compile
c = compile('vec = '+content[4], 'loading', 'exec')
## execute compiled line
exec c
## vec is now a list of the fourth vector
## in 'vector.ndv' The first line (content[0])
## is the header of the nvd-file
```

4.3.2 Classifier File

The classifier in this approach works on *voronoi regions*. Each cluster is represented by one vector and the a token is classified to the cluster with the nearest feature vector.
The classifier uses some additional information when a vector is assigned to a cluster. This information is the distance modifier dictionary and the class merge list. Both are saved in the first six lines of the classifier header. If

both are not used, they should be at least in the file as empty list or empty dictionary.
The third line contains the distance modifying dictionary and the fifth line contains the merge list. The first, second, forth, and sixth line can be ignored. They give information to a human reader of the file, no information for the classifier.
The following lines are the representative vectors of the clusters. The label of these feature vectors may *not* the original labels. In this case, they represent the meaning of this cluster.
If a cluster is subclsutered, the label is not a class, it will look like "subclusters+5+". The number (here 5) is the number of the cluster.
After all representatives of the main clusters the subclusters are listed. Each subclustered cluster begins the a headline, like the "label" is had before. In this case it would be "subclusters+5+". The representative vectors with the meaning of the clusters as label follow.

Short cutout of classifier file. This classifier consists of 11 main clusters. Cluster 5 is subclustered into 3 subclusters, cluster 10 into 5

```
classifier file
dictionary for weightings =
{'H1H2': 1.0, 'pwrMax': 0.0, 'pwrMin': 0.0,...}
representatives of clusters
merge list =
[['listen', 'understand', 'backchannel'],...]
representatives of clusters
['wav/FAN008_09-281.019.wav', 'backchannel',...]
['wav/FAN008_17-417.043.wav', 'backchannel',...]
['wav/FAN007_12-830.037.wav', 'backchannel',...]
['wav/FAN004_01-46.47.wav', 'agree',...]
['wav/FAN007_11-456.373.wav', 'backchannel',...]
['wav/FAN002_10-25.945.wav', 'subclusters+5+',...]
['wav/FAN009_01-569.564.wav', 'backchannel',...]
['wav/FAN008_12-664.372.wav', 'backchannel',...]
['wav/FAN003_07-930.096.wav', 'agree',...]
['wav/FAN002_10-44.364.wav', 'backchannel',...]
['wav/FAN002_10-28.898.wav', 'subclusters+10+',...]
subclusters+5+
['wav/FAN008_16-190.473.wav', 'backchannel',...]
```

```
['wav/FAN007_12-491.674.wav', 'backchannel',...]
['wav/FAN004_07-35.783.wav', 'agree',...]
subclusters+10+
['wav/FAN011_01-181.341.wav', 'backchannel',...]
['wav/FAN004_07-559.622.wav', 'disagree',...]
['wav/FAN010_10-322.286.wav', 'backchannel',...]
['wav/FAN010_01-583.731.wav', 'disagree',...]
['wav/FAN003_10-54.609.wav', 'agree',...]
```

4.4 DataVector.py Module

This documentation explains the functionality of the DataVector module for python. The feature vectors in this approach are represented by objects of the class DataVector.
In principle all kind of data vectors can be clustered with "Cluster.py" as long the DataVector module is reimplemented correctly.
First of all, this module is described like is is now. Then is explained, how it may be changed.

The DataVector module MUST contain the following three functions:

getDistance(vec1, vec2): to calculate the distance between two vectors.

loadFromString(string): to load a DataVector object from a string

calcCentroid([vectors):] to calculate the centroid of vector given in a list

Also the class called "DataVector" with the method "getString". The objects of the class have to have at least the attribute "cluster" and "matrixIndex". Both attributes are integers and first one should be initialized with 0 and latter one with -1.

First of all, the DataVector module declares a class called "DataVector". The objects in this approach have an attribute for each prosodic feature. Additionally the filename of the features. The constructor ''__init__'' reads the demands the filename of the vector and its label.

_sub:

_add: These are special methods to define the call of the operator methods. Since in this approach these are static 13 dimensional vectors, the adding and subtracting is the standard operation.

getNorm(): This method returns the Euclidean norm of the DataVector object. The significance of the feature space can be modified with the "modiFactor" dictionary.

getString(): To save the vectors to a file, this method returns a string, from which the vector can be loaded again.

getList(): This method saves the features of the vector to a list. This method is used in *getString* to save the vectors to a file.

loadFromList(): To get the feature data from a list, which can be loaded from a file, this method recreates the feature coefficients of the DataVector object from the list.

The following functions are implemented in the module but do not belong to the the class *DataVector* itself.

calcCentroid([DataVector):] This functions calculates the centroid of the DataVector objects, which are in the stored in the list given by the parameter. This function is called when a cluster is updated and the new representative of the cluster is calculated.

loadFromString(string): To load a DataVector object from a file, this function constructs an instance of DataVector and restores the features, which are saved in the string. In this implementation, *loadFromString* creates a new list and calls *loadFromList* to fix the new instance.

getDistance(DataVector1, DataVector2): During the clustering process the distance between two DataVector objects is measured by the call if this function.
In this implementation it calculates the Euclidean distance between the vectors. This distance can be modified by the factors stored in the modiFactor dictionary. This makes a scaling of the single features possible.

4.5 Cluster.py Module

This is a documentation for the Cluster.py Module for Python. It shall help to cluster and save DataVector objects, defined in DataVector.py. The Clusters created with module can be saved and loaded in "pros.py" for visualization.

Importing this module is easy. Just type:

```
import Cluster
```

or in combination with other modules:

```
import sys, os, Cluster, string
```

Cluster.py defines a class called "Cluster" which offers several methods to handle clusters, such as merging, splitting, or deleting. The class "Cluster" can and should be used as a cluster manager. The *static* functions are sufficient to handle Clustering algorithms and to load and save the clusters.

After the module is loaded, the script gain access to the class "cluster" by typing `Cluster.Cluster`. The first "`Cluster`" identifies the module, the second to identify the class. The code line `Cluster.Cluster.create("hello cluster")` will create a new cluster and add the information to the cluster manager.

Annotation:
Since it is a lot of effort always to type `Cluster.Cluster`, in front of a method, the trick of defining a pointer to the class can save time, bytes, and nerves.

```
CC = Cluster.Cluster
CC.create("hello cluster")
CC.create("a new cluster")
```

It shortens the code and makes it more clearly. This kind of shortcut will be used in this documentation.

4.5.1 Clusters

The following attributes are defined in each cluster.

name

number identifier for the cluster

A cluster has carries several information, which are necessary to handle it efficiently. Each cluster has an individual number, given by the cluster manager. The cluster can be identified by the number. The cluster also has an individual name, which can be given by the user. It is possible to identify the cluster by the name. Merging and splitting will change the name of the cluster, so it is not a reliable identifier. Merging or splitting causes a deletion of one cluster which also deletes the name and the number.

index number in cluster list

The index is the most important identifier in the manager. It is the number in the cluster list and allows direct manipulation of the attributes or calling non-static methods. Both is *not recommended*. This is the job of the cluster manager and should only be used, if the manager itself is insufficient for your problem. `Cluster.Cluster.clusters[3].number = 12` could cause trouble, because the number of a cluster is changes here without changing the number dictionary or the consistence of the cluster identifiers. The index can be get from a dictionary.

```
idx1 = Cluster.Cluster.getIndexNumber(12)
idx2 = Cluster.Cluster.getIndexName("hello cluster")
```

For shorter code, use:

```
CC = Cluster.Cluster
idx1 = CC.getIndexNumber(12)
idx2 = CC.getIndexName("hello cluster")
```

This annotation will be used through the documentation and will not be explained further in this and the following chapter.

The static methods "getIndexNumber" and "getIndexName" look up the index in the static dictionary "numberIndexDict" resp. "nameIndexDict". "idx1" carries now the index of the cluster with the number 12 and "idx2" of the cluster with the name "hello cluster"! Both can be the same. If the number or the name does not exist in the dictionary -1 will be returned.

vectors list of data vectors in the cluster

vectorCount number of vectors in the cluster

changed counting for changes in the cluster

Each cluster contains a specific number of vectors. The number is counted and saved in "vectorCount". "vectors" is a list which saves the pointers to the *DataVector* objects. If vectors are deleted or added to a cluster this could change the width, or other values in the cluster. This is why the changes are counted in "changed".

center center of cluster

representative representative of cluster

furthest furthest vector to the center in the cluster

meanDist mean distance of all vectors to the center

To cluster data, the information "how the cluster looks" is important. "center", "representative", and "furthest" are DataVector objects which are calculated after the call of the the "update" method.
"center" is the mean center of all vectors in the cluster.
"representative" is a pointer to the vector with the shortest distance to the "center" vector. The advantage of the "representative" is, that it is a vector loaded like all other vectors. So it can be used for precalculation and points to a "wav"–file to make a listening to the cluster possible. The clear disadvantage is, that it could be relatively far from the center, especially in clusters with a small number of vectors.

"furthest" is the furthest vector to the center. "furthest" is used in fast splitting the cluster.
"meanDist" is the average distance to the center of the cluster and represents in some way the width of the cluster.

4.5.2 Cluster Manager

The cluster manager is the center of this clustering module. New clusters can be created and are stored in static lists and are automatically included in algorithms. As long as possible, only the manager with the following methods should be used. In the following examples the shortcut "`CC`" will be used instead of "`Cluster.Cluster`". This shortcut can also be implemented in the python code using "`CC = Cluster.Cluster`"
Following static attributes uses the cluster manager:

vectorData a list to store all vectors for clustering

clusters a list where all "Cluster" objects are saved

nameIndexDict a dictionary to get the index of a cluster in "clusters"

numberIndexDict a dictionary to get the index of a cluster in "clusters"

distMatrix a matrix where all distances between the vectors a saved

newClusterNumber a counter for the cluster numbers

The following *static* methods handle the vector and cluster management.

create(string)

```
clusterNumber = CC.create("hello cluster")
```

This method creates a new cluster with the name given in the parameter string. The new cluster is automatically added to the cluster list of the cluster manager. "create" returns the individual number of the cluster.

getIndexName(string)
getIndexNumber(integer)

```
index1 = CC.getIndexName("hello cluster")
index2 = CC.getIndexNumber(12)
```

Returns the actual index of a cluster identified by the name or number of the cluster. The index is important for the most methods of the cluster manager. The index may change during clustering, because of merging or deleting or splitting clusters. So it is important to get the actual index.

loadVectors(filename, index = -1)

```
number1 = CC.create("cluster for vectorfile1")
## create new cluster
number2 = CC.create("cluster for vectorfile2")
## create new cluster
index1 = CC.getIndexNumber(number1)
## get index of new cluster
index2 = CC.getIndexNumber(number2)
## get index of new cluster
CC.loadVectors("vectorfile1.ndv",index1)
## load vectors and add to cluster
CC.loadVectors("vectorfile2.ndv",index2)
## load vectors and add to cluster
CC.loadVectors("vectordata_remain.ndv") ## load vectors
```

"loadVectors" will load a "ndv" file, which can be created with the *pros.py* application. The vectors in the file will be all appended to "vectorData" and if the index (second parameter) is set, the new vectors will also be added to the cluster with the index in manager cluster list.
The code above creates two new clusters with individual names. The number of these new clusters are saved in "number1" and "number2". Then the index of these clusters are saved in "index1" and "index2". The manager starts loading "vectorfile1.ndv" and append the vectors in this file to "vectordata" and also add to the cluster with the name "cluster for vectorfile1". The same happens with the vectors in "vectorfile2.ndv", but they are added to the cluster named "cluster for vectorfile2".
The vectors in "vectordata_remain.ndv" are only loaded and appended to "vectorData".

loadVectorsToClusters(filename)

```
CC.loadVectorsToClusters("vectors.ndv")
```

"loadVectorsToClusters" will load all vectors from the "ndv"–file and append to "vectorData". It will also create a new cluster for each vector. This is the way a *Bottom up* algorithm can be initialized.

loadRepresentatives(filename)

```
CC.loadRepresentatives("cluster.crp")
```

"loadRepresentatives" loads a "crp"–file which contains both the center vector and the representative vector of a cluster. For each pair of vectors a cluster is created and the "representative" and "center" vector objects are set. The vector in the "crp" file are *not* appended to "vectorData".

loadClusters(filename)

```
CC.loadClusters("clusters.lst")
```

"loadClusters" reloads the clusters from the "lst" file, created with "saveClusters". All vectors will be appended to "vectorData" and new clusters are created, containing the vectors before saving.

calcDistMatrix

```
CC.calcDistMatrix()
```

The distance matrix is of the type *DistMatrix*, a class which is also defined in the *Cluster.py* module. This method will calculate the distance matrix for all vectors loaded to "vectorData". The call of this method is only needed once. Since it takes time (depending on the number of vectors in "vectorData"), it is not calculated automatically. Some methods need to look up the distance in this matrix, because the looking up is quiet more faster than

calculating the distance again.

```
getDistanceFurthest(idx1, idx2)
getDistanceRep(idx1, idx2)
getDistanceCenter(idx1, idx2)
getDistanceAverage(idx1, idx2)
```

```
CC.calcDistMatrix()
idx1 = CC.getIndexNumber(42)
idx2 = CC.getIndexNumber(23)
name1 = CC.clusters[idx1].name
name2 = CC.clusters[idx2].name
print "distance between "+name1+" and "+name2" :"
print "distance of centers          :", CC.getDistanceCenter(idx1, idx2)
print "distance of representatives  :", CC.getDistanceRep(idx1, idx2)
print "distance of furthest pair    :", CC.getDistanceFurthest(idx1, idx2)
print "average distance of vectors  :", CC.getDistanceAverage(idx1, idx2)
```

Once both indices are given, "getDistance???" will calculate the distance between both clusters. There are several ways to calculate the distance between clusters. One way is to calculate the center of the clusters and use this distance of these vectors.
"getDistanceCenter" does it this way. The disadvantage is, that this method is very slow. No precalculation is possible and the center must be recalculated after every change in the cluster. This takes time and a bottom-up method using this cluster distance is probably the slowest.
Similar to the distance measure of the center, the representative of the clusters can be used. "getDistanceRep" calculates the distance between both representatives and returns the value. Due to precalculation with "calcDistMatrix" the value can be read quickly. If the distance matrix is not calculated, the value will be calculated like in "getDistanceCenter" and has no speed advantages.
"getDistanceFurthest" calculates the distance of the furthest vector pair between both clusters. Therefore several distances have to checked. This is why the distance matrix has to be calculated with "calcDistMatrix".
"getDistanceAverage" calculates the average distance of each vector pair from both clusters. This method also needs the distance matrix which is calculated with "calcDistMatrix".

addVector(self, *vectors)

```
vec1 = DataVector.DataVector("nofile","dummy vector 1")
vec2 = DataVector.DataVector("nofile","dummy vector 2")
vec3 = DataVector.DataVector("nofile","dummy vector 3")
idx = CC.getIndexNumber(42)
CC.clusters[idx].addVector(vec1)
CC.clusters[idx].addVector(vec2, vec3)
```

"addVector" is a non–static method, which is frequently used. It adds one or more object of DataVectors to a cluster. The cluster is often accessed directly through the cluster list in the cluster manager. This example show, how three vectors can be added to a cluster. The cluster information like the mean distance or the center vector is not updated, due to the reason of time. This can be done by the non-static method "update". The line "`CC.clusters[idx].addVector(vec1, vec2, vec3)`" would have added all three vectors to the cluster and so displace the last two lines of the example.

update(self, limit=0)

```
vec1 = DataVector.DataVector("nofile","dummy vector 1")
vec2 = DataVector.DataVector("nofile","dummy vector 2")
vec3 = DataVector.DataVector("nofile","dummy vector 3")
idx = CC.getIndexNumber(42)
CC.clusters[idx].addVector(vec1, vec2, vec3)
CC.clusters[idx].update(2)
```

The "update" method is a method to (re-) calculate the main information of a cluster. These information are:

center: a DataVector object which points directly into the center of the cluster

representative: the nearest DataVector object to the center in the cluster

meanDist: the average distance from all vectors in the cluster to the center

furthest: the furthest vector from from the center in the cluster

Everytime a cluster experience a change like the adding of a vector or merging of two clusters, ...the cluster internal variable "changed" is increased. The update will only proceed, if the parameter "limit" is higher than "changed". So "`update(0)`" resp. "`update()`" will start calculating, if there was any change at the cluster.

classifyVectorNearestRep(vec)
classifyVectorNearestCenter(vec)

```
vec1 = DataVector.DataVector("nofile","dummy vector 1")
vec2 = DataVector.DataVector("nofile","dummy vector 2")
idx1 = CC.classifyVectorNearestRep(vec1)
idx2 = CC.classifyVectorNearestCenter(vec2)
number1 = CC.clusters[idx1].number
name2 = CC.clusters[idx2].name
print "first vector was put in cluster number : %i" % number1
print "second vector was put in the cluster '" + name2 +"'."
```

The "classifyVectorNearest???" method searches the cluster, which has the nearest center or representative to the given DataVector object. The object will be added to this cluster. The method returns the index of the cluster.

mergeClusters(idx1, idx2)
mergeClustersNumber(nb1, nb2)
mergeClustersName(nm1, nm2)

```
idx1 = CC.getIndexName("cluster 1")
idx2 = CC.getIndexName("cluster 2")
CC.mergeClusters(idx1, idx2)
CC.mergeClustersNumber(42, 23)
CC.mergeClustersName("hello", "cluster")
```

"mergeClusters" merges two clusters to one and deletes the source clusters. This may cause changes to the indices of the clusters, also the names or numbers of the deleted clusters ore deleted from the dictionary and the clusters

cannot be accessed this way anymore.
The clusters can also be identified by the name or the number of the clusters.

deleteCluster(index)

```
idx = CC.getIndexName("cluster to delete")
CC.deleteCluster(idx)
```

The cluster with the given index will be deleted. A deletion will cause changes to the indices of the clusters. Saves indices can be worthless. Therefore, to access a specific cluster, use the name or the number of the cluster and get the actual index of this cluster with "getIndexName" resp. "getIndexNumber".

splitCluster(index)
splitClusterFurthest(index)

```
CC.splitCluster(23)
CC.splitClusterFurthest(42)
```

There are several ways to split a cluster. Two are implemented in this manager. Both ways choose two vectors from the cluster and add each other vector to the nearest new cluster.
The one way implemented in "splitCluster" is a fast way. During the update, the furthest vector to the center is calculated. A second check of all vectors in the cluster searches the vector which is the furthest vector to the furthest vector from the center. The search of the two vectors is in $O(n)$, while n is the number of vectors in the cluster. The other way, which is implemented in "splitClusterFurthest" searches the furthest neighbors in the clusters. This method runs in $O(n^2)$.
This pair may the the same like in the faster algorithm in some cases, however experiments proofed, that "splitClusterFurthest" perform better clusters than the faster implementation.

findNearestClusters(type = "represent")

```
result = CC.findNearestClusters("represent")
print "nearest pair of representatives : %i and %i" % (result[0], result[1])
print "the distance is : %3.2f" % result[2]
```

"findNearestClusters" searches the pair with the minimum distance. The kind of distance can be chosen by the parameter. "represent", "center", "average", and "furthest" are allowed. The method return a list with three elements. The firth two are the indices of the cluster pair, the third one is the distance between the clusters

findWidestCluster()

```
widthIndex = CC.findWidestCluster()
width = CC.clusters[widthIndex].meanDist
print "widest cluster has index %i" % widthIndex
print "the mean distance is %3.2f : " % width
```

To get the index of the cluster with the widest mean distance of the vectors in the cluster to the center, use "findWidestCluster". The index of the cluster will be returned.

doKMeans(represent = "center")

```
CC.doKMeans("represent")
```

All clusters are updated first, so the clusters carry the information of the center and representative vectors. Than all clusters are emptied; the center and representative information is still available. All vectors loaded will be classified and added to the clusters again. This shall cause a better distribution of the vectors. The classification to the clusters depends on the parameter, where "represent" and "center" is valid.

doBottomUp(bound = 50, distBound = 100.0)

```
CC.loadVectors(loadFile)
CC.calcDistMatrix()
CC.doBottomUp(20)
```

This static method is a special implementation for clustering. It uses the cluster representative and the possibility to inactivate vectors in the DistMatrix. This implementation is the fastest method for the Bottom Up clustering. But the recognition performance of the clusters does not seem to be a good solution for the given problem, but maybe for another. No clusters should be initialized when starting this clustering method. The vectors to cluster have to be loaded first. The parameter "bound" is the number of clusters and the parameter "distBound" determines a cluster distance, when all clusters have a distance more than "distBound" from each other, the algorithm stops.

`countLabels(self, labelList)`: The method "countLabel" returns a dictionary which contains the labels as keys and the number of labels in this clusters. This method is used in "evaluate Clusters" and is probably not needed separately.

`resetClusters()` To reset the cluster manager, this method is needed. All vectors and clusters are deleted. So clustering can start again.

```
evaluateClusters
(labelList, furtherInf = "",
filename = "result.txt", mergeList = []):
```

```
CC.loadVectors(loadFile)
CC.loadRepresentatives(repFile)
for vec in CC.vectorData:
        CC.classifyVectorNearestRep(vec)
labelList = ["lab1", "lab2", "lab3", "lab4"]
mergeList = [["lab1", "lab2", "lab12"],["lab3", "lab4", "lab34"]]
CC.evaluateClusters(labelList, "", "result.txt", mergeList)
```

The method evaluates the clusters. It is used to analyze the separability of the data. The result is saved in the file "result.txt". A header to this result file can be added with the string of given with the parameter "furtherInf". If a mergeList is given in the parameters, the evaluation will

start twice. The first time the without merging the classes, the second time with the merged classes. The mergeList is a list of small list of labels. The first two elements in this "minilist" are the classes which are merged. The third element is the new name of the class. The classes are merged from left to right in the list. This way three or more classes can be merged using a mergelist such as `[["class1", "class2", "tempMerge"], ["tempMerge", "class3", "merge of class1, class2, and class3"]]`.

`saveCluster(self, filename = "noname.ndv")` A cluster can be saved using this method. The first line is a short header then then the vectors which are in that cluster are written. Usually single clusters are not saved. This method is called from the static method "saveClusters".

`saveClusters(filename = "clusters", clusterFilename = "cluster")`: This static method saves all important cluster information. The first parameter is the filename for the information of all clusters. The file ending "lst" and "crp" will be added. The "crp"–file saves the pair of cluster representatives. The first vector is the representative. The second vector is the cluster centroid. To evaluate the clusters or to build the classifier, only the representatives have to be loaded. The "lst"–file contains the filenames of the clusters.

4.5.3 DistMatrix

The class DistMatrix is implemented in "Cluster.py" It allows a precalculation of the distances between all or a specific number of DataVector objects. With its help the distance measure between two vectors can be looked up quickly. Some algorithms during clustering need a DistanceMatrix, otherwise they would run incredible slowly.

The implementation of this matrix is a 1-dimensional array. The distance between two vectors is saved only once. So this array shapes a upper triangle matrix and consists of n*(n-1)/2 elements.

The DistMatrix uses dictionaries for additional data. So "indexDict" saves the y index of the vector and "vectorDict" saves the pointer to the vector.

The index of the vectors in the matrix are also saved in the DataVector object in the attribute "matrixIndex".

To create a DistMatrix, the static method "create" should be used, NOT the "__init__" standard constructor. A list of DataVector objects has to be the parameter for this method. It returns the pointer to the matrix. No vectors can be added to the DistMatrix. So if it is needed to add new vectors to this matrix, create a new matrix (takes time) or a new method has to be implemented.

To get a value out of the DistMatrix object, the method "get" and the indices of the vectors has to be used. Similar with the change of a value, in this case use "set". If the indices are not known, they can be looked up in "indexDict".

Example:

```
...
## some code which creates three DataVector objects
## named vec1, vec2, vec3.
matrix = DistMatrix.create([vec1, vec2, vec3])    ## creates the matrix
idx1 = matrix.indexDict[vec1]              ## get indeces
idx2 = matrix.indexDict[vec2]
dist1And2 = matrix.get(idx1, idx2)          ## get distance between vec1 and vec2
matrix.set(idx1, idx2, 100.0)              ## the distance in the matrix between
                        ## vec1 and vec2 is changed to 100.0
...
```

To find the maximum or the minimum distance in the matrix, the command "getMaximum" and "getMinimum" can be used. They are probably faster than a implementation based on "get" and "indexDict". This method returns a list. This list contains the two indices of the vector and the minimum value. Example

```
...
result = matrix.getMinimum()
minIndex1 = result[0]                ## get the indeces of the vectors
minIndex2 = result[1]
minVector1 = matrix.vectorDict[minIndex1]    ## get vectors with minimum distanc
minVector2 = matrix.vectorDict[minIndex2]
print "the mimimum value is %f" % result[2]    ## print minimum value
...
```

Another advantage the "getMinimum" and "getMaximum" methods bring is the possibility of deactivating and activating the vectors in the matrix. Inactive vectors are not included in the minimum resp. maximum search. Therefore the DistMatrix objects have an array named "active" which saves 1 for active and 0 for inactive. When the matrix is created, all the vectors are initialized as active (1). The representative clustering method uses this flag to enhance the speed of the algorithm and to make an easy implementation possible.

None o really interest as long not manipulating the code. All the values are saved in an array named "entry". The dimension is saved in the attribute "dim". As additional information also an array "yIndex" exists. Those values are for faster access to the values in "entry". The order in entry is like in an upper right triangle matrix read from left to right, top to bottom. A four-vector matrix or five-vector matrix would ordered like these

```
     1    2    3    4
1    -    0    1    2
2    -    -    3    4
3    -    -    -    5
4    -    -    -    -

     1    2    3    4    5
1    -    0    1    2    3
```

2	-	-	4	5	6
3	-	-	-	7	8
4	-	-	-	-	9
5	-	-	-	-	-

4.6 Subcluster.py

To create a classifier, the module "SubCluster.py" is needed. It is a cluster manager like the "Cluster.py" module.

The main difference between both is, that this module handle the meaning of clusters and is able to create subclusters, in case a cluster is too large. But the clusters managed with "SubCluster.py" cannot be changed completely, such as merging and splitting.

This manager offers the following static methods to handle the clusters and the classifier.

init(clusterfilename, developfilename) This method loads the clusters from the *clusterfilename* which is the "lst"-file created with the "Cluster.py" module. Also a set of *DataVector* objects is loaded from the "ndv"-file, which is given by the *developfilename*.
Latter set is to assign a meaning resp. a class to the clusters. In this step, the "init" method assign the vectors in the development set to the clusters. Depending on the number of vectors on one class and the total number of vectors in this class, a ratio is calculated. The maximum ratio defines the class resp. meaning of this cluster.

mergeClasses(mergeList) Since several classes could be merged to one, the *mergeList* gives the needed information. If this method is called, all labels in the *development set* are updated and the meaning of the clusters is recalculated.
The *mergeList* is from the same format like in the "Cluster.py" module. It is a list of small lists consists of two or three string. The first both strings are the classes to merge and the optional third string defines the new name of the classes. [["class1", "class2", "class1+2"], ["class5", "class7", "class5+7"]]

createClassifier(filename) This method saves the current state of the subcluster manager as a classifier. Later modifications are not saved until this method is not called again. To the filename no extension is

added. The structure of the classifier is discussed in the file format section 4.3.2.
From this file, a classifier can be reloaded. The information which vectors are in the clusters is not saved. This information is not needed for classifying.

loadClassifier(filename) The file created with *createClassifier* can be reloaded. The clusters and subclusters will be created and a meaning is assigned to them, also the cluster representatives.
Now any *DataVector* can be classified to a cluster and a meaning can be estimated.

classify(vec) This method classifies a *DataVector* object to a cluster. If this cluster is subclustered, the meaning is calculated from the subclusters. The class resp. meaning of the cluster or subcluster this vector is classified will be returned.

checkMeaning(vec) Since some classes may be merged, the "old" label of a vector may not be correct anymore. This method reads the label from a vector and returns the class name this vector belongs to.

putInConfusionMatrix(vec) The "SubCluster.py" module is able to save a confusion matrix. So if a classifier is loaded, a *DataVector* object can be put into this confusion matrix. The original class of this vector and the class to which is is classified will be considered in this matrix.

printConfusionMatrix() This method prints the confusion matrix on "stdout", calculates the averaged class recognition rate and the vector misclassification rate.

This cluster manager does not offer a lot of instance methods. Only the following two (and the constructor "__init__") are implemented in this manager. The constructor will not be discussed here, because the clusters should be initialized by the use of the static "init" method, explained above.
The clusters in this manager can be accessed with "SubCluster.SubCluster.subclusters[index]" The index is the same index when the "Cluster.py" manager is used.

calcMeaning() If any modifications such as the adding of more labeled

vectors happens to the cluster or subcluster, this method recalculates the meaning to this cluster.

createSubclusters(n = 0) This is a very important method. It clusters a cluster into several subclusters. The number of subclusters can be chosen by the parameter n. If no parameter is given or it is set to 0, the number of subclusters will be the logarithm dualis of the number of development vectors in this cluster.
The subclusters are created with the *k–Means* algorithm with random initialization.

4.7 Code Examples

4.7.1 Top Down

The following code display how few code is needed to implement a *Top Down* algorithm. The clusters will be split with the fast split method.
In this code, the DataVector objects saved in "data_train.ndv" are loaded and put into one large start cluster. The widest cluster (the cluster with the highest mean distance from the centroid) will be split into two new clusters, until 20 clusters exist.

```
## load the module "Cluster.py"
import Cluster

## create a shortcut
CC = Cluster.Cluster

## To create a cluster for the beginning of the algorithm
## save the number of this start cluster
clsNumber = CC.create("startcluster")

## get the index of the start cluster
clsIndex  = CC.numberIndexDict[clsNumber]

## load the DataVectors from "data_train.ndv" and
## add all these vectors to the start cluster
CC.loadVectors("data_train.ndv", clsIndex)

## The index of the start cluster will be "0",
## so this alternative shorter code would cause the same effect
## CC.create("startcluster")
## CC.loadVectors("data_train.ndv", 0)

## while the number of cluster is smaller than 20
while (len(CC.clusters) < 20):

    ## get the Index of the widest clusters
    idx = CC.findWidestCluster()
```

```
        ## split this cluster into two new Clusters
        CC.splitCluster(idx)

        ## There are two ways to split a cluster
        ## ''splitClusterFurthest'' is a slower but
        ## more accurate way to split the cluster
        ## CC.splitClusterFurthest(idx)

## this line saves the clusters.
## It creates the files "clusters.crp" and "clusters.lst"
## the the singe cluster-files "cluster1.ndv", "cluster2.ndv",...
CC.saveClusters("clusters", "cluster")
```

4.7.2 Bottom Up

The following code display the implementation of a *Bottom Up* algorithm. The two nearest clusters will be merged to one.
The DataVector objects in "data_train.ndv" are loaded and each vector becomes a cluster. Then the nearest clusters are merged. This repeats until 20 clusters are left.

```
## load the module "Cluster.py"
import Cluster

## create a shortcut
CC = Cluster.Cluster

## all DataVectorObjects are loaded and
## for each object a new cluster will be created
CC.loadVectorsToClusters("data_train.ndv")

## to accellerate the calculation of the distance
## between two DataVector objects, a distance matrix
## precalculates the results
## this can take some minutes, depends on the amount of
## data and the way of distance measure
CC.calcDistMatrix()
```

```
## while the amount of clusters is greater than 20
while (len(CC.clusters) > 20):

     ## find both nearest clusters
     ## the way of distance measure between the clusters
     ## is the furthest neighbor. ''findNearestClusters''
     ## returns a list containing both indeces of these
     ## nearest neigbors and the distance
     idxList = CC.findNearestClusters("furthest")

     ## merge the nearest clusters. The indeces
     ## are located on 0 and 1 in the list
     CC.mergeClusters(idxList[0], idxList[1])


## this line saves the clusters.
## It creates the files "clusters.crp" and "clusters.lst"
## the the singe cluster-files "cluster1.ndv", "cluster2.ndv",...
CC.saveClusters("clusters", "cluster")
```

4.7.3 K–Means

This example shows a way how to implement a k–Means algorithm with the module "Cluster.py". The perfomrance of k–Means depends on the initialization.
In this example, the algorithm is initialized by the first 20 vectors. The vectors are loaded and the clusters are initialized. The the k–Means run three times.
This example itself will not perform well. Normally the k–Means have to run several times and fulfill some conditions to stop. Also the initialization may be completely inadequate.

```
## load the module "Cluster.py"
import Cluster

## create a shortcut
```

```
CC = Cluster.Cluster

## this load the DataVectors objects to CC.vectorData
## no clusters are created yet
CC.loadVectors("data_train.ndv")

## take the first 20 DataVector obejcts
## create a new cluster without name and put
## the object into the new cluster
for i in range(20):
     clsNumber = CC.create("")
     clsIndex  = CC.numberIndexDict[clsNumber]
     ## clsIndex will be i
     CC.clusters[clsIndex].addVector(CC.vectorData[i])

## doKMeans calculates the centroid of the clusters (update)
## free all vectors in the cluster and assign
## all vectors in CC.vectorData to the clusters
CC.doKMeans("center")
CC.doKMeans("center")
CC.doKMeans("center")

## this line saves the clusters.
## It creates the files "clusters.crp" and "clusters.lst"
## the the singe cluster-files "cluster1.ndv", "cluster2.ndv",...
CC.saveClusters("clusters", "cluster")
```

4.7.4 Create A Classifier

After the clusters are saved to "lst"- and "crp"-files a classifier, can be created. If no clusters need to be subclustered, the creation of a classifier is very easy. The "SubCluster.py" module will handle the subclustering and the creation of the classifier.
The following code loads the clusters which are stores in "clusters.lst" and load the *DataVector* objects from the "data_develop.ndv" file. "init" creates the new clusters and try to assign a class to them. Then the class "listen" and "understand" are merged to "backchannel". Cluster 5 will be subclustered into 10 smaller cluster. The meaning of the new clusters has to be recalcu-

lated. Then the classifier is created and saved to the file "classtest.clf"

```
## import the DataVector and SubCluster module
import SubCluster, DataVector

## create a shortcut
SC = SubCluster.SubClusters

## load the clusters and init the meaning of the
## clusters with the labeled vector in "datadevelop.ndv"
SC.init("clusters.lst", "data_develop.ndv")

## a short example for a mergeList. In this example
## the classes "listen" and "understand" are merged
## and have the new name "backchannel"
mergeList = [["listen", "understand", "backchannel"]]

## load the mergeList to the SubClusters
SC.mergeClasses(mergeList)

## sublcuster cluster 5 into 10 subclusters
SC.subclusters[5].createSubClusters(10)

## calculate the new meaning of the
## cluster5 and its subclusters
SC.subclusters[5].calcMeaning()

## create a classifier and save it to the file
SC.createClassifier("classtest.clf")
```

4.7.5 Load A Classier And Classify A Vector

This code demonstrates, how a classifier is loaded and vectors are classified. The classification will be printed on screen and a confusion matrix is calculated.
First the vectors are loaded from the "data_test.ndv" file. After this the classifier is loaded and the single vectors are classified. The "classify" method does not influence the confusion matrix. So the vector is classified and the

result is print on screen. But if the data is labeled data, a confusion matrix can be calculated.

```
## import modules
import SubCluster, DataVector

## create shortcut
SC = SubCluster.SubClusters

## read the vector data, which shall be classified
## open the vector file "data_test.ndv"
file = open("data_test.ndv", "r")

## read the data from the file
data = file.readlines()

## close file
file.close()

## ignore header line
data[0] = ""

## create a list of the DataVecot objects
vectorList = []

## read all strings in the file data
for s in data:

    ## ignore empty lines in data
    if ((s == "") or (s == "\n")): continue

    ## load object
    obj = DataVector.loadFromString(s)

    ## save object in list!
    vectorList.append(obj)

## load the former created classifier
```

```
## from the "classtest.clf" file
SC.loadClassifier("classtest.clf")

## take each vector in the test file
for vec in vectorList:

     ## classify and add result to confusion matrix
     SC.putInConfusionMatrix(vec)

     ## print result ob classification
     print vec.label + " classified as " SC.classify(vec)

## print confusionmatrix
SC.printConfusionMatrix()
```