Machine Learning
(Course 395)

Computer Based Coursework Manual

Lecturers:
Maja Pantic
Stavros Petridis

CBC Helpers:

Eftychia Fotiadou
Markos Georgopoulos
Jiankang Deng
Jean Kossaifi
Kritapat Sonsri-In
Linh Tran
Robert Walecki
Mengjiao Wang
Yujiang Wang
Christopher Bowles
Jie Pu
Yiming Lin

Imperial College London

LONDON 2017

Table of Contents

1. Introduction

The purpose of this Computer-Based Coursework (CBC) is to provide the students with hands-on experience in implementing and testing basic machine learning techniques. The techniques that will be examined are Decision Trees (DT) and Artificial Neural Networks (ANN). Each of these techniques will be used in order to identify six basic emotions from the human facial expressions (anger, disgust, fear, happiness, sadness and surprise) either based on a labelled set of facial Action Units (AUs) or directly from the image pixels. AUs correspond to contractions of human facial muscles, which underlie each and every facial expression, including facial expressions of the six basic emotions. More theory and details on Facial Action Units and their relation to emotions will be given in section 3.
The implementation of the aforementioned techniques requires understanding of these techniques. For this reason, following the lectures of the course is strongly advised.

2. Organization

a) Working Method

Implementation of the algorithms will be done using MATLAB (or Python). The students will work in groups of 4 students. They are expected to work together on implementation of each machine learning technique and the related emotion recognizer. The groups will be formed shortly after the first lecture (for lecture schedule see http://ibug.doc.ic.ac.uk/courses/machine-learning-course-395/) and a CBC helper will be assigned to each group. The implementation will be either done from scratch (DT) or by using special toolboxes (ANN). After an assignment is completed, the code generated by each group will be evaluated by the CBC helpers. This will be done in the lab and by using a separate test set that will not be available to the students. The implemented algorithms will have to score a predefined minimum classification rate on this unknown test set. In addition, each group must hand in, via the CATE system, a report of approximately **4-5 pages** (excluding result matrices and graphs), and explaining details of the implementation process of each algorithm along with comments on the acquired results. Your code (which should run on any lab computer) should also be included in the CATE submission. All files (including the report) have to be combined in one archive file.

You should also inform us about your team members by email by noon January 25th with the following information:

- Student login
- Correspondence email
- CID
- First and last Name
- Degree, course/study taken, and the current year in that course.

Fill in the excel form (you can find it on the course website under the section "Group Forming" or on http://ibug.doc.ic.ac.uk/media/uploads/documents/courses/ml-cbc-groupform.xls) with the above information and email it to us (machinelearningtas@gmail.com). If you cannot form a team with 4 members, then email us the above information and we will assign you to a team. Please note that we only accept

requests for groups of 4. Members in a request for a different group size will be assigned randomly to separate groups.
Deliverables for every assignment will be described at the end of every section describing the assignment in question. Each group is responsible for the way in which the assignments are implemented and the reports are prepared and presented. These reports provide feedback on the performance of the group as a whole.

b) Role of the CBC helpers

The role of the CBC helpers is to monitor the implementation of the assignments by the students. The CBC helpers, however, will not make any substantive contribution to the implementation process. Final grading will be exclusively done by the lecturer of the course, who will, nevertheless, ask for the recommendations of the CBC helpers concerning the group progress.

c) Communication

Communication between the students and the CBC helpers is very important, and will be done in labs during the CBC sessions or via email using the following address:

machinelearningtas@gmail.com

Please **ALWAYS** mention your group number and your assigned helper in the subject line of your email; this makes it easier for us to divide the work. In addition, students should visit the website of the course, at http://ibug.doc.ic.ac.uk/courses/machine-learning-course-395/, in order to download the required data files and various MATLAB functions needed in order to complete the assignments of this CBC. Also many useful links and information will be posted onto this website.

d) Time Management

In total, there are 3 assignments to be completed. As mentioned before, after the completion of each assignment a report of approximately 4-5 pages of text must be handed in. The deadlines for handing in each assignment are as follows:

- Assignment 1: No hand in required.
- Assignment 2: Monday February 13th – noon.
- Assignment 3: Monday March 6th – noon.

e) Grading

In this CBC, we expect each group member to actively participate in the implementation of the algorithms. Each individual assignment will be graded based on the submitted report and code. The final CBC grade will be computed as follows:

$$assignment\_grade = 0.75 * report\_content + 0.15 * code\_performance + 0.1 * report\_quality$$

$$CBC\_grade = 0.4 * assignment\_grade\ [Ass\ 2] + 0.6 * assignment\_grade\ [Ass\ 3]$$

code_performance refers to the generalisation of the trained algorithms on new unseen examples, *report_content* refers to what is provided in the report (e.g., results, analysis and discussion of the results and how the questions in each assignment have been answered) and *report_quality* refers to quality of presentation.

NOTE: **CBC accounts for 33% of the final grade for the Machine Learning Course**. In other words, final grade = 0.67*exam_grade + 0.33*CBC_grade. For example, if the exam_grade = 32/100 and the CBC_grade = 80/100, then final_grade = 48/100.

f) Assignment submission guidelines

In order to avoid negative consequences related to CBC assignment submission, *strictly* follow the points listed below.

- You should *work in groups*. Take note that only *one report per group* will be accepted.
- Send a *timely* email to the THs with the *full* list of group members, and the following information for each and every group member (use the excel form from the website – http://ibug.doc.ic.ac.uk/media/uploads/documents/ml-cbc-groupform.xls):
    - Student login
    - Correspondence email
    - CID
    - Full first Name
    - Family Name
    - Degree, course/study taken, and the current year in that course.
- The *text in your report* should be approximately 4-5 pages.
- Make sure you mention your group number in each of your reports, as well as at each communication with the CBC helpers.
- *Strictly follow* the assignment *submission deadlines and times* specified on CATE.
- Each and every group member *individually has to confirm* on CATE that they are part of that particular group, for *each and every assignment submission* (under the pre-determined group leader) before each assignment submission deadline.

g) Outline of the manual

The remaining of this CBC manual is organized as follows. Section 3 introduces the Facial Action Coding System (FACS). It explains the meaning of each AU as well as the relation between the AUs and the six basic emotions. Section 4 introduces the basic system-evaluation concepts including K-fold cross-validation, confusion matrices, recall and precision rates. Section 5 contains the first (optional) assignment by providing an introduction on MATLAB fundamentals via a number of exercises. Sections 6, 7, and 8 explain the assignments 2-4 and the machine learning techniques that have to be implemented.

3. The Facial Action Coding System and the basic emotions

One of the great challenges of our times in computer science research is the automatic recognition of human facial expressions. Machines capable of performing this task have many applications in areas as diverse as behavioural sciences, security, medicine, gaming and human-machine interaction (HMI). The importance of facial expressions in inter-human communication has been shown by numerous cognitive scientists. For instance, we use our facial expressions to synchronize a conversation, to show how we feel and to signal agreement, denial, understanding or confusion, to name just a few. Because humans communicate in a far more natural way with each other then they do with machines, it is a logical step to design machines that can emulate inter-human interaction in order to come to the same natural interaction between man and machine. To do so, machines should be able to detect and understand our facial expressions, as they are an essential part of inter-human communication.

a) FACS

Traditionally, facial expression recognition systems attempt to recognize a discrete set of facial expressions. This set usually includes six 'basic' emotions: anger, disgust, fear, happiness, sadness and surprise. However, the number of possible facial expressions that humans can use numbers about 10,000, many of which cannot be put in one of the six basic emotion categories (think for example of expressions of boredom, 'I don't know', or a brow-flash greeting). In addition, there is more than one ways to display the same feeling or emotion. Therefore, describing a facial expression in such loose terms as 'happy', 'sad' or 'surprised' is certainly not very exact, greatly depending on who is describing the currently displayed facial expression while leaving a large variation of displayed expressions possible within the emotion classes. The activation of the facial muscles on the other hand can be described very precisely, as each muscle or group of muscles can be said to be either relaxed or contracted at any given time. As every human has the same configuration of facial muscles, describing a facial expression in terms of facial muscle activations would result in the same description of a facial expression, regardless of the person displaying the expression and regardless of who was asked to describe the facial expression. The Facial Action Coding System (FACS[1,2]), proposed by psychologists Ekman and Friesen, describes all the possible facial muscle (de)activations that cause a visible change in the appearance of the face. Every muscle activation that causes visible appearance changes is called an Action Unit (AU). The FACS consists of 44 AUs (see Fig. 1 for examples).

[1] P. Ekman and W.V. Friesen, The Facial Action Coding System: A Technique for the Measurement of Facial Movement, San Francisco: Consulting Psychologist, 1978

[2] P. Ekman, W.V. Friesen and J.C. Hager, "The Facial Action Coding System: A Technique for the Measurement of Facial Movement", San Francisco: Consulting Psychologist, 2002

Figure 1: Examples of AUs

b) Action Units and emotions

The same psychologists who proposed the FACS also claimed that there exist six 'basic' emotions (anger, disgust, fear, happiness, sadness and surprise) that are universally displayed and recognized in the same way. As we already mentioned, many research groups have proposed systems that are able to recognize these six basic emotions. Almost all proposed emotion detectors recognize emotions directly from raw data. In this CBC we will use a different approach to emotion detection. Instead of directly classifying a set of features extracted from the images into emotion categories, we will use AUs as an intermediate layer of abstraction. The rules that map AUs present in a facial expression into one of the six basic emotions are given in Table 1. In this CBC we will not use these rules directly but instead we will try to learn emotional classification of AUs using different machine learning techniques. Also, in this CBC we consider the step of AU detection to be solved. Students are provided with a dataset that consists of a list of AUs and the corresponding emotion label.

Figure 2. Typical smile includes activation of AU6, AU12 and AU25.

Emotion	AUs	Emotion	AUs
Happy	{12}	Fear	{1,2,4}
	{6,12}		{1,2,4,5,20,
Sadness	{1,4}		25||26||27}
	{1,4,11||15}		{1,2,4,5,25||26||27}
	{1,4,15,17}		{1,2,4,5}
	{6,15}		{1,2,5,25||26||27}
	{11,17}		{5,20,25||26||27}
	{1}		{5,20}
Surprise	{1,2,5,26||27}		{20}
	{1,2,5}	Anger	{4,5,7,10,22,23,25||26}
	{1,2,26||27}		{4,5,7,10,23,25||26}
	{5,26||27}		{4,5,7,17,23||24}
Disgust	{9||10,17}		{4,5,7,23||24}
	{9||10,16,25||26}		{4,5||7}
	{9||10}		{17,24}

Table 1. Rules for mapping Action Units to emotions, according to FACS. A||B means 'either A or B'

c) DATA

The data for this CBC will be provided to the students in the form of mat files. Each file contains the following two variables:

- A matrix *x*, which is an *N×45* matrix, where *N* is the total number of examples and 45 is the total number of AUs that can be activated or not. In case an AU is activated, the value of the corresponding column will be 1. Otherwise, it will be 0. For instance, the following row

  [1 1 0 0 1 0 …0]

  would mean that AU1, AU2 and AU5 are activated.
- A vector *y* of dimensions *N×1*, containing the emotion labels of the corresponding examples. These labels are numbered from 1 to 6, and correspond to the emotions anger, disgust, fear, happiness, sadness and surprise respectively.

In addition, the students will be provided with functions that map emotion labels (numbers 1 to 6) to actual emotions (anger, disgust, fear, happiness, sadness, surprise) and back. These files are called *emolab2str.m* and *str2emolab.m* respectively.
During this CBC, the students will work with two types of data: *clean* and *noisy* data, each given as a separate mat file. *Clean* data was obtained by human experts. The AU and emotion information in this type of data is considered correct for every example. On the other hand, the AUs in the *noisy* data were obtained by an automated system for AU recognition[3]. Since the system is not 100% accurate, the output of the system can contain wrongly detected AUs and some AUs can be missing.

4. System Evaluation

In this section, the basic system evaluation concepts that will be used throughout this CBC are given. These include:

- K-fold Cross Validation
- The Confusion Matrix
- Recall and Precision Rates
- The F_α-measure

a) Basic terms

Class is a collection of similar objects, which in this CBC is a set of examples with the same emotion label. The set of labels is denoted by $\Omega = \{l: 1 \leq l \leq 6\}$, where each integer stands for an emotion as described in the previous section.

Features or attributes are characteristics of objects. In this CBC it is AUs. If a feature (AU) *f* is activated (present) for an object (example) *n*, then the value of the element a_{fn} of the matrix generated as described in 3c) is 1. Otherwise, it is 0. You will be given N examples $z_n \in \Re^{45}, 1 \leq n \leq N$, as each of the examples has 45 AUs (attributes) that are

3 M.F. Valstar and M. Pantic, "Fully automatic facial action unit detection and temporal analysis", Proc. IEEE Int'l Conf. Computer Vision and Pattern Recognition, vol 3, p. 149, 2006

either activated (with value 1) or not (with value 0). The class label of example z_n is denoted by $l(z_n) \in \Omega$.

Classifier is any function: $D: \Re^m \to \Omega$, where m is the number of attributes. In this CBC, you will create algorithms for finding classifiers $D: \Re^{45} \to \{l: 1 \leq l \leq 6\}$, where l is an emotion label. You will consider a set of six discriminant functions $G = \{g_l(x): 1 \leq l \leq 6\}$, where x is an example and $g_l: \Re^{45} \to \Re$, each giving a score for lth class. Usually, an example x is given a label in the class of the highest score, the labelling choice called the maximum membership rule. That is, $D(x) = \omega_* \in \Omega \leftrightarrow g_*(x) = \max\limits_{1 \leq l \leq 6}\{g_l(x)\}$. When there is a tie, i.e. an example is given two or more labels, a possible solution could be to randomly allocate one of the tied labels. When no label has been allocated, then a possible solution could be to allocate randomly one of all six labels.

b) Training and testing

After classifier D has been trained with training examples, we will test its performance on a new set of data, test examples. Its performance may be measured in terms of error rate, i.e. a quotient of number of test examples classified incorrectly and the total number of examples.

$$Error(D) = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} \{1 - \Im(l(z_n), s_n)\},$$

where N_{test} is the number of examples z_n tested, $1 \leq n \leq N_{test}$, s_n is the label given by classifier D to z_n and $\Im(l(z_n), s_n) = 1$ iff $l(z_n) = s_n$ and $\Im(l(z_n), s_n) = 0$, otherwise.

It is a good practice to have three sets of data: the training data, the validation data and the test data. The first set is used to train classifiers, the second is used to optimise the parameters of classifiers (e.g. the number of hidden neurons when neural networks are used), and the third set is used to calculate the error rate for the final set of parameters.

The procedure for training a classifier is as follows:

1) The training data are used to train multiple classifiers using a different set of parameters each time (e.g. number of hidden neurons for neural networks).
2) The trained classifiers are tested on the validation set and the classifier which results in the best performance is selected. This is called parameter optimization because we select the set of parameters that led to the best classifier and in case we need to train a new classifier on the training set we will use this optimal set already found.
3) The test error is calculated on the test data for evaluating the performance of the classifiers.

It is a good practice to stop the training process when the difference between the training error and the validation error (obtained on classifying validation data) starts to increase, which is illustrated by the diagram below (Fig. 3). If the values of the validation error increase while the values of the training error steadily decreases then a situation of overfitting may have occurred. That is, the classifiers allocate the label perfectly on the training data, but poorly for the validation (new) data. It may be due to fitting the characterestics of the training data, which are not present in a general pool of the data (or at least not in the validation data).

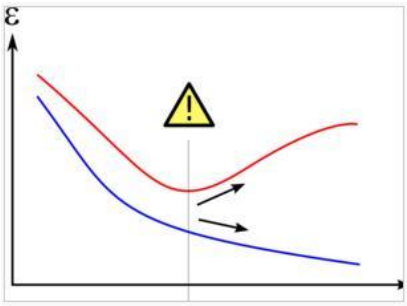

Fig.3: The values of training error are shown in blue, the values of the validation error in red for each iteration (as joined points by a smooth curve). On the horizontal axis, we have number of iterations and on the vertical axis, the values of training and validation errors.

c) Cross-validation

Since the amount of data for training and testing is limited, we can reserve part of the data for testing. To guarantee that the part retained for testing is representative, one may employ K-fold cross-validation. One splits the data into K folds (parts) and hold out one for testing while using the other K-1 folds for training. The process is repeated K times, each time a different fold is retained for testing. The total error estimate is the arithmetic mean of Error(D) obtained for each of K times of testing.

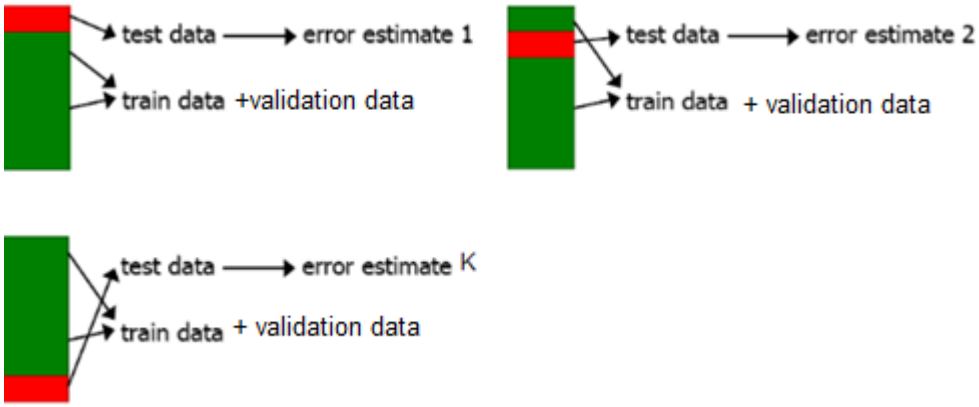

Fig.4: *K-fold* cross-validation process.

In this CBC, you will perform 10-fold cross-validation, in which you will split the dataset into 10 folds and subsequently use each one for testing. Note that the 9 folds should be further divided into training and validation sets.

d) Confusion matrix

A confusion matrix is a visualization tool typically used to present the results attained by a learner. Each column of the matrix represents the instances in a predicted class, while each row represents the instances in an actual class. One benefit of a confusion matrix is that it is easy to see if the system is confusing two classes (i.e. commonly mislabelling one as

another). In the example confusion matrix below (Table 2), of the 8 actual cats, the system predicted that three were dogs, and of the six dogs, it predicted that one was a rabbit and two were cats. We can see from the matrix that the system in question has trouble distinguishing between cats and dogs, but can make the distinction between rabbits and other types of animals pretty well.

		Predicted Class		
		Cat	Dog	Rabbit
Actual Class	Cat	5	3	0
	Dog	2	3	1
	Rabbit	0	2	11

Table 2: A simple confusion matrix

True positives (TPs) are the examples that were classified correctly as members of a given class. In Table 2, if we consider the class Cat as the positive one we have 5 TPs. True negatives (TNs) are the examples that were classified correctly as members of the negative classes (dog and rabbit). In Table 2 we have 3+2+1+11=17 TNs. False positives (FPs) are the examples that were classified incorrectly as members of the positive class. In Table 2 we have 2+0=2 FPs. They are found in the column of Predicted Class Cat. False negatives (FPs) are the examples that were classified incorrectly as members of the negative classes. In Table 2 we have 3+0=3 FNs. They are found in the row of Actual Class Cat.

		Predicted Class	
		Cat	Other
Actual class	Cat	5 (TP)	3 (FN)
	Other	2 (FP)	17 (TN)

Table 3: The number of TPs, TNs, FPs, FNs for class Cat

e) Recall and Precision Rates

To be able to compare the two classifiers, the recall and precision rates are used. Recall and Precision Rates measure the quality of an information retrieval process, e.g. a classification process. Recall Rate describes the completeness of the retrieval. It is defined as the portion of the positive examples, i.e. TPs retrieved by the process versus the total number of existing positive examples (including the ones not retrieved by the process), i.e. TPs and FNs. Precision Rate describes the actual accuracy of the retrieval, and is defined as the portion of the positive examples (TPs) that exist in the total number of examples retrieved (TPs and FPs). Based on the recall and precision rates, we can justify if a classifier is better than another, i.e. if its recall and precision rates are significantly better.

$$Recall\ rate = \frac{TP}{TP + FN} \times 100\% \quad Precision\ rate = \frac{TP}{TP + FP} \times 100\%$$

For the example of class Cat discussed above we obtained:

$$Recall\ rate = \frac{5}{5+3} \times 100\% \approx 63\%$$
$$Precision\ rate = \frac{5}{5+2} \times 100\% \approx 71\%$$

f) F_α measure

While recall and precision rates can be individually used to determine the quality of a classifier, it is often more convenient to have a single measure to do the same assessment. The F_α measure combines the recall and precision rates in a single equation:

$$F_\alpha = (1+\alpha)\frac{precision * recall}{\alpha * precision + recall},$$

where α defines how recall and precision rates will be weighted. In case recall and precision rates are evenly weighted then the F_1 measure is defined as follows:

$$F_1 = 2\frac{precision * recall}{precision + recall}.$$

For the example of class Cat discussed above we obtained:

$$F_1 = 2 \times \frac{63\% \times 71\%}{63\% + 71\%} \approx 67\%$$

5. Assignment 1: MATLAB Exercises

Assignment 1 is an optional assignment. It aims to provide a brief introduction to some basic concepts of MATLAB without assessing students' acquisition, application and integration of this basic knowledge. The students, are strongly encouraged to go through all the material, experiment with various functions, and use the MATLAB help files extensively (accessible via the main MATLAB window).
Type “doc” on the MATLAB command line to open the help browser. MATLAB blogs also provide useful information about how to program in MATLAB (http://blogs.mathworks.com).

a) Vectors and Arrays

A vector in MATLAB can be easily created by entering each element between brackets and assigning it to a variable, e.g. :

```
a = [1 2 3 4 5 6 9 8 7]
```

Let's say you want to create a vector with elements between 0 and 20 evenly spaced in increments of 2:

```
t = 0:2:20
```

MATLAB will return:

```
t =
     0   2   4   6   8   10   12   14   16   18   20
```

Manipulating vectors is almost as easy as creating them. First, suppose you would like to add 2 to each of the elements in vector 'a'. The equation for that looks like:

```
b = a + 2

b =
     3   4   5   6   7   8   11   10   9
```

Now suppose you would like to add two vectors together. If the two vectors are the same length, it is easy. Simply add the two as shown below:

```
c = a + b

c =
     4   6   8   10   12   14   20   18   16
```

In case the vectors have different lengths, then an error massage will be generated.
Entering matrices into MATLAB is the same as entering a vector, except each row of elements is separated by a semicolon (;) or a return:

```
B = [1 2 3 4;5 6 7 8;9 10 11 12]

B =
     1    2    3     4
     5    6    7     8
     9   10   11    12

B = [ 1  2  3  4
      5  6  7  8
      9 10 11 12]
```

```
B =
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

Matrices in MATLAB can be manipulated in many ways. For one, you can find the transpose of a matrix using the apostrophe key:

```
C = B'

C =
     1     5     9
     2     6    10
     3     7    11
     4     8    12
```

Now, you can multiply the two matrices B and C together. Remember that order matters when multiplying matrices.

```
D = B * C

D =
    30    70   110
    70   174   278
   110   278   446

D  = C * B

D =
   107   122   137   152
   122   140   158   176
   137   158   179   200
   152   176   200   224
```

Another option for matrix manipulation is that you can multiply the corresponding elements of two matrices using the .* operator (the matrices must be the same size to do this).

```
E = [1 2;3 4]
F = [2 3;4 5]
G = E .* F
E =
     1     2
     3     4
F =
     2     3
     4     5
G =
     2     6
    12    20
```

MATLAB also allows multidimensional arrays, that is, arrays with more than two subscripts. For example,

```
R = randn(3,4,5);
```

creates a 3-by-4-by-5 array with a total of 3x4x5 = 60 normally distributed random elements.

b) Cell arrays and structures

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example

```
C = {A sum(A) prod(prod(A))}
```

produces a 1-by-3 cell array. There are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces, for example *C{1}* retrieves the first cell of the array. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change `A`, nothing happens to `C`.

Three-dimensional arrays can be used to store a sequence of matrices of the *same* size. Cell arrays can be used to store sequences of matrices of *different* sizes. For example,

```
M = cell(8,1);
for n = 1:8
   M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```
M =
    [           1]
    [ 2x2  double]
    [ 3x3  double]
    [ 4x4  double]
    [ 5x5  double]
    [ 6x6  double]
    [ 7x7  double]
    [ 8x8  double]
```

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields.

```
S =
     name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

Like everything else in MATLAB, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

Or, an entire element can be added with a single statement.

```
S(3) = struct('name','Jerry Garcia',...
              'score',70,'grade','C')
```

Now the structure is large enough that only a summary is printed.

```
S =
1x3 struct array with fields:
    name
    score
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are all based on the notation of a *comma separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

This is a comma separated list. Without any other punctuation, it is not very useful. It assigns the three scores, one at a time, to the default variable `ans` and dutifully prints out the result of each assignment. But when you enclose the expression in square brackets,

```
[S.score]
```

it is the same as

```
[S(1).score, S(2).score, S(3).score]
```

which produces a numeric row vector containing all of the scores.

```
ans =
    83    91    70
```

Similarly, typing

```
S.name
```

just assigns the names, one at time, to `ans`. But enclosing the expression in curly braces,

```
{S.name}
```

creates a 1-by-3 cell array containing the three names.

```
ans =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

And

```
char(S.name)
```

calls the `char` function with three arguments to create a character array from the `name` fields,

```
ans =
Ed Plum
Toni Miller
Jerry Garcia
```

c) Functions

To make life easier, MATLAB includes many standard functions. Each function is a block of code that accomplishes a specific task. MATLAB contains all of the standard functions such as sin, cos, log, exp, sqrt, as well as many others. Commonly used constants such as pi, and i or j for the square root of -1, are also incorporated into MATLAB.

```
sin(pi/4)

ans =

    0.7071
```

To determine the usage of any function, type `help [function name]` at the MATLAB command window.
MATLAB allows you to write your own functions with the *function* command. The basic syntax of a function is:

```
function [output1,output2] = filename(input1,input2,input3)
```

A function can input or output as many variables as are needed. Below is a simple example of what a function, `add.m`, might look like:

```
function [var3] = add(var1,var2)
%add is a function that adds two numbers
var3 = var1+var2;
```

If you save these three lines in a file called "add.m" in the MATLAB directory, then you can use it by typing at the command line:

```
y = add(3,8)
```

Obviously, most functions will be more complex than the one demonstrated here. This example just shows what the basic form looks like.

d) Loops

If you want to repeat some action in a predetermined way, you can use the *for* or *while loop*. All of the loop structures in MATLAB are started with a keyword such as "for", or "while" and they all end with the word "end".
The *for loop* is written around some set of statements, and you must tell MATLAB where to start and where to end. Basically, you give a vector in the "for" statement, and MATLAB will loop through for each value in the vector: For example, a simple loop will go around four times each time changing a loop variable, *j*:

```
for j=1:4,
    j
end
```

If you don't like the *for loop*, you can also use a *while loop*. The *while loop* repeats a sequence of commands as long as some condition is met. For example, the code that follows will print the value of the *j* variable until this is equal to 4:

```
j=0
while j<5
      j
      j=j+1;
end
```

You can find more information about *for loops* on http://blogs.mathworks.com/loren/2006/07/19/how-for-works/

e) Reading from files / Writing to files

Before we can read anything from a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and it goes off to find it on the disk. If it can't find the file, it

returns with an error; even if the file does exist, we might not be allowed to read from it. So, we need to check the value returned by *fopen* to make sure that all went well. A typical call looks like this:

```
fid = fopen(filename, 'r');
if (fid == -1)
  error('cannot open file for reading');
end
```

There are two input arguments to *fopen*: the first is a string with the name of the file to open, and the second is a short string which indicates the operations we wish to undertake. The string 'r' means "we are going to read data which already exists in the file." We assign the result of *fopen* to the variable *fid*. This will be an integer, called the "file descriptor," which we can use later on to tell MATLAB where to look for input.

There are several ways to read data from a file we have just opened. In order to read binary data from the file, we can use the *fread* command as follows:

```
A = fread(fid, count)
```

where *fid* is given by *fopen* and *count* is the number of elements that we want to read. At the end of the *fread*, MATLAB sets the file pointer to the next byte to be read. A subsequent *fread* will begin at the location of the file pointer. For reading multiple elements from the file a loop can be used in combination with *fread*.

If we want to read a whole line from the file we can use the *fgets* command. For multiple lines we can combine this command with a loop, e.g. :

```
while (done_yet == 0)

    line = fgets(fid);
    if (line == -1)
      done_yet = 1;
    end
```

Before we can write anything into a file, we need to open it via the *fopen* function. We tell MATLAB the name of the file, and give the second argument 'w', which stands for 'we are about to write data into this file'.

```
fid = fopen(filename, 'w');
if (fid == -1)
  error('cannot open file for writing');
end
```

When we open a file for reading, it's an error if the file doesn't exist. But when we open a file for writing, it's not an error: the file will be created if it doesn't exist. If the file does exist, all its contents will be destroyed, and replaced with the material we place into it via subsequent calls to *fprintf*. Be sure that you really do want to destroy an existing file before you call *fopen*!

There are several ways to write data to a file we have just opened. In order to write binary data from the file, we can use the *fwrite* command, whose syntax is exactly the same as *fread*. In the same way, for writing multiple elements to a file, *fwrite* can be combined with a loop.

If we want to write data in a formatted way, we can use the *fprintf function, e.g. :*

```
fprintf(fid, '%d %d %d \n', a, b, c);
```

which will write the values of *a,b,c* into the file with handle *fid*, leaving a space between them. The string *%d* specifies the precision in which the values will be written (single), while the string \n denotes the end of the line.
At the very end of the program, after all the data has been read or written, it is good practice to close a file:

```
fclose(fid);
```

f) Avoiding "Divide by zero" warnings

In order to avoid "Divide by zero" warnings you can use the *eps* function. *Eps(X)* is the positive distance from *abs(X)* to the next larger in magnitude floating point number of the same precision as X. For example if you wish to divide A by B, but B can sometimes be zero which will return *Inf* and it may cause errors in your program, then use *eps* as shown:

C = A / B; % If B is 0 then C is *Inf*
C = A / (B + eps); % Even if B is 0 then C will just take a very large value and not *Inf*.

g) Profiler / Debugging

The *profiler* helps you optimize M-files by tracking their execution time. For each function in the M-file, profile records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. To open the *profiler* graphical user interface select Desktop->Profiler. So if the execution of your code is slow you can use the *profiler* to identify those lines of code that are slow to execute and improve them.

Another useful function that can be used for debugging is the *dbstop* function. It stops the execution of the program when a specific event happens. For example the commands

dbstop if error
dbstop if warning

stop execution when any M-file you subsequently run produces a run-time error/warning, putting MATLAB in debug mode, paused at the line that generated the error. See the MATLAB help for more details. Alternatively, you can use the graphical user interface to define the events that have to take place in order to stop the program. Just select Debug menu -> Stop if Errors/Warnings.

6) Assignment 2: Decision Trees Algorithm

The goal of this assignment is to implement a decision tree algorithm. The results of your experiments should be discussed in the report. You should also deliver the code you have written.

a)Implementation

Part I: Loading data

Make sure the clean data (*x*, *y*) is loaded in the workspace where *x* is an $N \times 45$ array, *N* is the total number of examples and 45 is the number of action units (or features/attributes) and *y* is an $N \times 1$ vector, containing the labels of the corresponding examples. These labels are numbered from 1 to 6, the same as the total number of emotions. In order to construct a decision tree for a specific emotion, the labels in *y* should be remapped according to that particular emotion. For example, if you train for happiness, with label 4, then the labels with that value should be set to 1 (positive examples) and all the others to 0 (negative examples).

Part II: Creating Decision Tree

You need to write a function that takes as arguments a matrix of examples, where each row is one example and each column is one attribute, a row vector of attributes, and the target vector which contains the binary targets. The target vector will split the training data (examples) into positive examples for a given target and negative examples (all the other labels). The table below provides a pseudo code for the function.

function DECISION-TREE-LEARNING(*examples,attributes,binary_targets*) **returns** a decision tree for a given target label

```
if all examples have the same value of binary_targets
then return a leaf node with this value
else if attributes is empty
    then return a leaf node with value = MAJORITY-VALUE(binary_targets)
else
    best_attribute ← CHOOSE-BEST-DECISION-ATTRIBUTE(examples,attributes, binary_targets)
    tree ← a new decision tree with root as best_attribute
    for each possible value υ_i of best_attribute do  (note that there are 2 values: 0 and 1)
        add a branch to tree corresponding to best_attibute = υ_i
        {examples_i , binary_targets _i}← {elements of examples with best_attribute = υ_i and the
                                          corresponding binary_targets_i }
        if examples_i is empty
        then return a leaf node with value = MAJORITY-VALUE(binary_targets)
        else subtree ← DECISION-TREE-LEARNING(examples_i ,attributes-{best_attribute},
                                                binary_targets_i)
return tree
```

Table 1. Pseudo code for the decision tree algorithm

The function MAJORITY-VALUE(*binary_targets*) returns the mode of the *binary_targets*. The function CHOOSE-BEST-DECISION-ATTRIBUTE chooses the attribute that results in

the highest information gain. Suppose that the set of training data has p positive and n negative examples. Each attribute has two values 0 and 1. Suppose p_0 is the number of positive examples for the subset of the training data for which the attribute has the value 0, and n_0 is the number of the negative examples in this subset. Suppose p_1 is the number of positive examples for the subset of the training data for which the attribute has the value 1, and n_1 is the number of the negative examples in this subset. Then,

$Gain(attribute) = I(p,n) - Remainder(attribute)$, where

$I(p,n) = -\frac{p}{p+n}\boldsymbol{log_2}\left(\frac{p}{p+n}\right) - \frac{n}{p+n}\boldsymbol{log_2}(\frac{n}{p+n})$ and

$Remainder(attribute) = \frac{p_0+n_0}{p+n} I(p_0,n_0) + \frac{p_1+n_1}{p+n} I(p_1,n_1)$.

The resulting tree must be a MATLAB structure (*struct*) with the following fields:

- *tree.op* : a label for the corresponding node (e.g. the attribute that the node is testing). It must be empty for the leaf node.
- *tree.kids* : a cell array which will contain the subtrees that initiate from the corresponding node. Since the resulting tree will be binary, the size of this cell array must be 1x2, where the entries will contain the left and right subtrees respectively. This must be empty for the leaf node since a leaf has no kids, i.e. tree.kids = [].
- *tree.class* : a label for the leaf node. This field can have the following possible values:
    - *0 - 1*: the value of the examples (*negative-positive*, respectively) if it is the same for all examples, or with value as it is defined by the MAJORITY-VALUE function (in the case *attributes* is empty).
    - It must be empty for an internal node, since the tree returns a label only in the leaf node.

This tree structure is essential for the visualization of the resulting tree by using the ***DrawDecisionTree.m*** function, which is provided. Alternatively, if you use Python you should write your own visualisation function.

Part III: Evaluation

Now that you know the basic concepts of decision tree learning, you can use the clean dataset provided to train 6 trees, one for each emotion, and visualize them using the DrawDecisionTree function. Then, evaluate your decision trees using 10-fold cross validation on both the clean and noisy datasets. 6 trees should be created in each fold, and each example needs to be classified as one of the 6 emotions. You should expect that slightly different trees will be created per each fold, since the training data that you use each time will be slightly different. Use your resulting decision trees to classify your data in your test set. Write a function:

- predictions = testTrees(T, x2),

which takes your trained trees (all six) T and the features *x2* and produces a vector of label *predictions*. Both *x2* and *predictions* should be in the same format as x, y provided to you. Think how you will combine the six trees to get a single output for a given input sample. Try at least 2 different ways of combining the six trees.

NOTE: In case you use Python you should provide clear instructions how to test your code.

Report average cross validation classification results (for both clean and noisy data):

- Confusion matrix.

  (***Hint:*** you should get a single 6x6 matrix)
  (***Hint***: you will be asked to produce confusion matrices in almost all the assignments so you may wish to write a general purpose function for computing a confusion matrix)

- Average recall and precision rates per class.

  (***Hint:*** you can derive them directly from the previously computed confusion matrix)

- The F_1-measures derived from the recall and precision rates of the previous step.
- Average classification rate (NOTE: classification rate = 1 – classification error)

Comment on the results of both datasets, e.g. which emotions are recognised with high/low accuracy, which emotions are confused.

Part IV: Pruning function

Run the *pruning_example* function, which is provided, using the clean and noisy datasets.

b) Questions

In your report you will have to answer the following questions.

Noisy-Clean Datasets Question

Is there any difference in the performance when using the clean and noisy datasets? If yes/no explain why. Discuss the differences in the overall performance and per emotion.

Ambiguity Question

Each example needs to get only a single emotion assigned to it, between 1 and 6. Explain how you made sure this is always the case in your decision tree algorithm. Describe the different approaches you followed (at least two) to solve this problem and the advantages/disadvantages of each approach. Compare the performance of your approaches on

both clean and noisy datasets and explain if your findings are consistent with what you described above.

Pruning Question

Briefly explain how the *pruning_example* function works. One figure with two different curves should be generated for each dataset (clean and noisy). Include the two figures in your report and explain what these curves are and why they have this shape. What is the difference between them? What is the optimal tree size in each case?

c) Deliverables

For the completion of this part of the CBC, the following have to be submitted electronically via CATE:

1. All the code you have written.
2. The 6 trees you have trained on the entire clean dataset (in .mat format).
3. A report of approximately 4-5 pages (excluding figures and tables) containing the following:
    - brief summary of implementation details (e.g., how you performed cross-validation, how you selected the best attribute in each node, how you compute the average results, anything that you think it is important in your system implementation);
    - diagrams of the six trees trained on the entire dataset;
    - commented results of the evaluation including the average confusion matrix, the average classification rate and the average precision, recall rates and F_1-measure for each of the six classes; for both clean and noisy datasets.
    - Answers to noisy-clean, ambiguity and pruning questions.

d) Grading scheme

Final Grade = 0.75* Report content + 0.15* Code performance + 0.1* Report quality

Code Performance = CR on unseen data + 15

Code (total : 100)

- Results on new test data : 100

  Make sure that your testTrees function runs. If not you will be asked to resubmit the code and lose 30% of the code mark.

Report content (total : 100)

- Implementation details : 15
- Tree figures: 5
- Confusion matrix : 5
- Recall/precision/Fmeasure/Classification rate : 5
- Analysis of the cross validation experiments: 10
- Answer to the clean-noisy question: 15
- Answer to the ambiguity question : 25
- Answer to the pruning question : 20

Report quality (total : 100)

- Quality of presentation.