ISELL User Guide

Olcay Taner YILDIZ

October 12, 2009

Contents

Chapter 1

How to Compile & Execute

One needs only gcc to compile the ISELL program. Go to the source directory, and just write,

```
user@pc:~$make
```

which will compile the ISELL C source code and creates isell executable program. The compilation is done with -ansi -pedantic options, therefore any programming struct or any nonstandard (ANSI standard) function call will result in an error in the compilation.

When you have the executable program, just write

```
user@pc:~$./isell
```

to run the program. You can give an optional batch script file to the program. For example,

```
user@pc:~$./isell script.txt
```

will run ISELL program, compiles and interprets the commands in the script.txt.

Chapter 2

External Files

ISELL needs four files to operate. These are dictionary.txt, order.txt, hyperparameters.xml and parameters.xml.

2.1 dictionary.txt

This file contains functions written in the enhanced postscript (eps) language. The functions are

circle Draws a circle to an eps file. It takes three parameters /y0, /x0 and /r representing the y and x axis coordinates of the center of the circle, and the radius of the circle respectively.

cshow Prints a string object as centred. It takes only the string as an argument.

clshow Prints a string object as left aligned. It takes only the string as an argument.

crshow Prints a string object as right aligned. It takes only the string as an argument.

fillrect Draws and fills a rectangle. It takes five parameters /upperY, /upperX, /lowerY, /lowerX and /gray representing the y and x axis coordinates of the upper-left corner of the rectangle, y and x axis coordinates of the lower-right corner of the rectangle and gray-scale depth of the color to fill with respectively.

drawrect Draws a rectangle. It takes four parameters /upperY, /upperX, /lowerY and /lowerX representing the y and x axis coordinates of the upper-

left corner of the rectangle, y and x axis coordinates of the lower-right corner of the rectangle respectively.

xaxis Draws the x axis. It takes five parameters /sameY, /secondX, /firstX, /tickwidth, and /tickcount representing y axis position of the x axis, start and end of the x axis, width of each tick and number of ticks to plot on the x axis respectively.

yaxis Draws the y axis. It takes five parameters /sameX, /secondY, /firstY, /tickwidth, and /tickcount representing x axis position of the y axis, start and end of the y axis, width of each tick and number of ticks to plot on the y axis respectively.

verticalfill Fills a rectangle by drawing vertical lines in it. It takes five parameters /upperY, /upperX, /lowerY, /lowerX, and /tickwidth representing the y and x axis coordinates of the upper-left corner of the rectangle, y and x axis coordinates of the lower-right corner of the rectangle and the distance between two vertical lines respectively.

horizontalfill Fills a rectangle by drawing horizontal lines in it. It takes five parameters /upperY, /upperX, /lowerY, /lowerX, and /tickwidth representing the y and x axis coordinates of the upper-left corner of the rectangle, y and x axis coordinates of the lower-right corner of the rectangle and the distance between two horizontal lines respectively.

squarefill Fills a rectangle by drawing small squares in it. It takes five parameters /upperY, /upperX, /lowerY, /lowerX, and /tickwidth representing the y and x axis coordinates of the upper-left corner of the rectangle, y and x axis coordinates of the lower-right corner of the rectangle and the side length of each small square respectively.

dashedline Draws a dashed line. It takes five parameters /secondY, /secondX, /firstY, /firstX, and /dashtype representing the y and x axis coordinates of the starting and ending points of the line and type of the dash respectively.

line Draws a usual line. It takes four parameters /secondY, /secondX, /firstY, /firstX, and /dashtype representing the y and x axis coordinates of the starting and ending points of the line respectively.

2.2 command.txt

This file contains the commands (and its properties) available in the ISELL program. The position of each command is important. In the process.c file, there is a function **process_order(int res)** which process each command in the interpreter (or batch script file). The parameter *res* corresponds to the index of the command and is the position of the command in the command.txt file. For example the command *exit* is the first command in the command.txt file and if *res* is 0, this command is processed.

There are 10 lines for each command in the command.txt file.

1. index. COMMAND
2. Name of the command.
3. Group of the command. When the user enters the command help, the commands are grouped according to this field (and the subgroup) of the command. The program automatically groups the commands. Therefore if a nonexistent group is entered to this area, there will be no problem, it will be displayed correctly.
4. Subgroup of the command. The program automatically groups the commands. Therefore if a nonexistent subgroup is entered to this area, it will be displayed correctly.
5. Explanations of the return values of the command. Most commands in the ISELL environment return one (output variable of type real or integer such as out1, output variable of type string such as sout1) or more values (array output variable such as aout1, aout2, etc.) into the output variable(s). These output variables can be used as usual variables in the script.
6. Name of the output variables returned by this command.
7. Grammar of the command. This field of the command is used in compiling the script files. If the parameters of the command are not in the types as specified here, ISELL will return error 1016 in the compilation. Possible parameter types are <stringval>, <integerval>, and <realval> for one string, integer, and real constant value (such as 3.4, 2) respectively; <stringvaln>, <integervaln>, and <realvaln> for one or more string, integer, and real constant values respectively; <stringvar>, <integervar>, <realvar>, <filevar>, <matrixvar> for string, integer, real, file, and matrix variables (such as i, j, st, myfile) respectively; <onoff> for on or off constant value.

8. Explanations of the parameters of the command.

9. Explanation of the command itself (what it does, how it does, citation to the algorithm, etc.).

10. File name(s) of one or more example scripts. These example scripts can be found in the **examples** directory.

2.2.1 To add a new command?

In order to add a new command to the ISELL program follow the following steps:

1. Select a command name. Be sure there isn't another command with the same name.

2. Add 10 new lines to the end of the command.txt. Fill the lines with appropriate definitions, grammar and explanations. Remember ISELL will not open if the number of lines in the command.txt is not divisible by 10.

3. Add a new case to the switch in the **process_order** function. Be sure the number in the case is one less than the index of the new command.

4. If the command takes parameters (most of the time it does), call the **checkparams** function. **checkparams** will check if the user has given enough parameters to the command. First argument of the **checkparams** function is always *paramcount* variable, which shows the number of parameters user has entered. Second argument of the **checkparams** function is the number of parameters the command expects. Let say this number is p. Then there are p more arguments of the **checkparams** function which correspond to the error messages the user will see when he/she did't enter enough parameters. If the user did't enter first parameter, he/she will see the first message (Third argument of the checkparams function). If the user entered first parameter but did't enter the second parameter, he/she will see the second message (Fourth parameter of the checkparams function). In the ISELL program the error messages has their own codes. They are constant strings defined in messages.h file.

5. You can also check the type and bounds of the constant integer and real value parameters. Use **atoi_check** and **atof_check** functions for integer and real value parameters respectively. The third and fourth

arguments of those functions are the minimum and maximum value allowed for the constant integer or real parameter.

6. Do not forget to set the output variable(s) if the command has it (them). Two main functions exist for this course namely **set_default_variable** and **set_default_string_variable**.

2.3 hyperparameters.xml

Most of the algorithms in the machine learning have hyperparameters to tune such as k (nearest neighbor count) for nearest neighbor algorithm, C for support vector machine algorithms, learning rate for gradient descent training as in neural networks or logistic regression. In ISELL, these hyperparameters can be tuned automatically using cross-validation. Before dividing into train or test sets, ISELL separates out some part of the original dataset (validation set) for hyperparameter tuning. *cvratio* is the command to set the percentage of the validation set. For each possible value of the hyperparameter, the training is done and the best hyperparameter value (the one that performs best on the validation set) is used for testing (**tune_hyperparameters_of_the_supervised_algorithm**). In order to tune hyperparameter of an algorithm, the script must contain line

```
hyperparametertune on
```

For each hyperparameter there is a <**parameter**> tag, which contains (i) name of the hyperparameter (ii) all possible values the hyperparameter can take as a sublist. Each possible value of the hyperparameter is surrounded by <value>, </value> tags. As an example:

```
<parameter name="k">
<value>1</value>
<value>2</value>
<value>3</value>
<value>4</value>
<value>5</value>
<value>6</value>
<value>7</value>
<value>8</value>
<value>9</value>
<value>10</value>
</parameter>
```

2.3.1 To add a new hyperparameter?

In order to add a new hyperparameter to the ISELL program follow the following steps:

1. Select a name for the hyperparameter.
2. Add a new <parameter> tag to hyperparameters.xml.
3. Add possible values of the new hyperparameter to hyperparameters.xml.
4. Modify **tune_hyperparameters_of_the_supervised_algorithm** function so that the specific algorithm can be run with hyperparameters tuned.
5. If the original algorithm does not need other validation data to use, one must add

```
case ...:if (get_parameter_o("hyperparametertune") == ON)
           return YES;
         else
           return NO;
```

lines to **is_cvdata_needed** function.

2.4 parameters.xml

There are three classes of parameters in ISELL program. These are:

- The parameters of the learning algorithm such as k (nearest neighbor count) for nearest neighbor algorithm, C for support vector machine algorithms, learning rate for gradient descent training as in neural networks or logistic regression.
- Parameters independent from the learning algorithm such as training and test set percentages (60 percent of the data will be for training, other for testing), number of folds (10 fold, 30 fold etc.), type of validation (k-fold crossvalidation, $5\times$ 2 crossvalidation, $10\times$ 10 crossvalidation, etc.), statistical test used in comparing classifiers or regressors (Wilcoxon sign-rank test, Sign test, Paired t test, etc.).
- Parameters of the ISELL program (most of them on-off type parameters) such as will the source code be displayed while it is running, will the results of the algorithms or tests etc. be displayed on the screen or forwarded to an output file.

The parameters can be in one of six types namely, INTEGER, FLOAT, STRING, ONOFF, LIST, and ARRAY.

INTEGER The parameter can take integer values. Example parameters are: *nearcount* (nearest neighbor count) for the nearest neighbor algorithm, $foldcount$ number of folds in crossvalidation, *precision* floating-point precision in displaying real numbers on the screen or in a file, *maxhidden* maximum number of hidden nodes in a hidden layer of multilayer percentron algorithm, *tailed* type of statistical test two tailed (2) or one tailed (1), *perceptronrun* number of epochs to train a gradient descent algorithm, *optimizecount* number of optimization steps in RIPPER rule learning algorithm.

```
<parameter name="perceptronrun" type="INTEGER" value="10"/>
<parameter name="foldcount" type="INTEGER" value="2"/>
```

FLOAT The parameter can take floating point values. Example parameters are: *learning_rate* initial learning rate used in training gradient descent algorithms, $confidencelevel$ confidence level of a statistical test to reject the null hypothesis, *noiselevel* noise percentage used while producing noisy data, *svmC*, *svmnu* C and nu parameters of the C-SVM (support vector machines) and nu-SVM learning algorithms, *testpercentage* percentage of examples to put aside before dividing the data into training and validation sets, *cvratio* percentage of the validation set with respect to the original dataset.

```
<parameter name="learning_rate" type="FLOAT" value="0.10"/>
<parameter name="variance_explained" type="FLOAT" value="0.99"/>
```

STRING The parameter can take string values. Example parameters are: $imagefile$ when current figure will be saved this file name will be used, $modelfilename$ when the model of the current learning algorithm will be saved this file name will be used, $posteriorfilename$ when the posterior probabilities of the test instances will be saved this file name will be used.

```
<parameter name="testcodefilename" type="STRING" value=""/>
<parameter name="posteriorfilename" type="STRING" value=""/>
```

ONOFF The parameter can take only on or off values. On and off values are stored as 1 and 0 in ISELL. Example parameters are: *timeon* if it is on, the execution time of each command is reported, *displaycodeon* if

it is on, the program prints each line it is executing, *parallel* if it is on, parallel programming (executing) is enabled, *accuracy* if it is on, accuracy of the classification algorithms is reported, otherwise error is reported, *savetestfiles* if it is on, the source code of the testing phase of the learning algorithm is saved in a given rogramming language (which is also a LIST type parameter), *hold* same as hold on in the Matlab drawing environment, *showticks* if it is on, ticks are displayed on the axes.

```
<parameter name="timeon" type="ONOFF" value="OFF"/>
<parameter name="displaycodeon" type="ONOFF" value="ON"/>
```

LIST The parameter takes a value from a list of strings. Each string in the list is stored between <value> and </value> tags. Example parameters are: *currentos* operating system where ISELL program runs (it can take three different values as Windows, Solaris, Linux), *testtype* name of the statistical test used is comparisons, *correctiontype* name of the confidence level correction method, *modelselectionmethod* name of the model selection technique, *prunetype* name of the pruning algorithm used in decision tree pruning, *kerneltype* type of kernel used in Support Vector Machines, *language* name of the programming language.

```
<parameter name="correctiontype" type="LIST" value="bonferroni">
   <value>bonferroni</value>
   <value>holm</value>
   <value>nocorrection</value>
</parameter>
<parameter name="kerneltype" type="LIST" value="linear">
   <value>linear</value>
   <value>polynom</value>
   <value>rbf</value>
   <value>sigmoid</value>
</parameter>
```

ARRAY The parameter is an array of values. Array elements can be INTEGER or FLOAT. <size> tag represents the size of the array, <value> tag represents default value of the elements of the array. Example parameters are: *hiddennodes* number of hidden nodes in each hidden layer of the multilayer perceptron, *statecounts* number of states in the Hidden Markov Model for each class.

```
<parameter name="hiddennodes" type="ARRAY" subtype="INTEGER" size="3" va
<parameter name="svmcweights" type="ARRAY" subtype="FLOAT" size="50" val
```

2.4.1 To add a new parameter?

In order to add a new parameter to the ISELL program follow the following steps:

1. Select a name for the parameter. Be sure there isn't another parameter with the same name.
2. Add one or more lines to the parameters.xml file. If the parameter is of LIST type, add also all possible values. If the parameter is of ARRAY type, add the subtype and size of the array.
3. Specify the default value of the parameter in the <value> tag.

If the parameter is a changable parameter from the script, one needs add a new command (Section 2.2.1) in order to be able to change its value.

Chapter 3

Dataset Definition

In order to define a dataset or datasets to ISELL program, one needs first put all dataset directories in a single directory (data directory). Each dataset directory must contain at least one dataset file. The command to define the data directory is **setdatadir**. For example, the partial script

```
setdatadir /home/olcay/datasets
```

defines the data directory as /home/olcay/datasets. After defining the data directory, one loads a dataset definition file. The command for this purpose is **loaddatasets**. For example, the partial script

```
loaddatasets mydataset.xml
```

loads the datasets given in the file mydataset.xml. Here mydataset.xml must be a dataset definition file with a predefined format. An example mydataset.xml is given below:

```
<datasets>
<dataset name="iris" task="classification" size="150"
         directory="iris" filetype="text"
         separator="," filename="iris.data">
   <classes>
      <class value="iris-setosa" size="50"/>
      <class value="iris-versicolor" size="50"/>
      <class value="iris-virginica" size="50"/>
   </classes>
   <attributes>
      <attribute type="Real" available="True" min="4.3" max="7.9"/>
      <attribute type="Real" available="True" min="2.0" max="4.4"/>
      <attribute type="Real" available="True" min="1.0" max="6.9"/>
```

```
      <attribute type="Real" available="True" min="0.1" max="2.5"/>
      <attribute type="Output" available="True">
         <value>iris-setosa</value>
         <value>iris-versicolor</value>
         <value>iris-virginica</value>
      </attribute>
   </attributes>
</dataset>
</datasets>
```

Although the tags and properties are self-explanatory we give the explanations below:

name Name of the dataset

task Learning task for the dataset. Can be classification or regression.

size Number of instances in the dataset.

directory Name of the dataset directory that contains the dataset file.

filetype Only text files are supported for now.

separator The attributes in the dataset file are separated with a character. If the separator character is space just write " ".

filename Name of the dataset file.

classes The surrounding tag that contains the name of the classes and the number of instances in each class.

attributes The surrounding tag that contains the attributes.

type Type of the attribute. Three types of attributes are supported in ISELL. real integer and string (discrete) attributes. Output attribute has type output.

min Minimum value of the attribute.

max Maximum value of the attribute.

available Some attributes may be unusable in the learning.

Chapter 4

ISELL Script Language

4.1 Introduction

We begin by considering a simple script.

```
'A first program in ISELL
writescreen Hello_World!
```

Output:

```
Hello World!
```

The line

```
'A first program in ISELL
```

begins with índicates that this line is a *comment*. Comments in ISELL can not be extended by more than one line.

The line

```
writescreen Hello_World!
```

instructs the interpreter to print a string of characters. In ISELL language every statement must end with the new line character. When the preceding writescreen command is executed, it prints the message Hello World! on the screen. The characters normally print exactly as they appear. Notice that the character '_' was not printed on the screen. Since the parameters of every command are separated via space characters, in order to print space character one needs another character namely '_'.

4.1.1 Data Types and Variables

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it.

Variable Names

Variables are the basic data object in ISELL. They can be declared anywhere in the program and can only be used afterwards. The declaration of a variable is

data_type variable_name

and the declaration of more than one variable is

data_type variable_name1 variable_name2 ... variable_namen

Sample variable definitions are given below:

```
integer x1 x2
real y
string st
file infile outfile
matrix m1 m2 mymatrix
```

There are some restrictions on the names of variables.

1. Names are made up of letters and digits. So ab9c is a valid variable name, whereas ab[not.
2. The underscore can not be used inside a variable.
3. Upper case and lower case letters are not distinct, so x and X are the same.
4. Keywords are not reserved: You can use them as variable names.

Data types

There are five basic data types in ISELL (given below).

Table 4.1: Data types in ISELL

integer	an integer
real	double precision floating point
string	an array of characters
file	a file variable used in order to read and write text files
matrix	matrix variable

Declarations

All variables must be declared before use, although certain declarations can be made explicitly by context. A declaration specifies a type, and contains a list of one or more variables of that type, as in

```
integer lower upper step
string st
```

Variables can be distributed among declarations in any fashion; the lists above could equally well be written as

```
integer lower
integer upper
integer step
string st
```

This latter takes more space, but is convenient for adding a comment to each declaration or for subsequent modifications. A variable can not be initialized in its declaration.

4.1.2 Operators

There are four types of operators in ISELL.

1. Arithmetic operators
2. Equality and relational operators
3. Assignment operators
4. Increment and decrement operators

Arithmetic Operators

ISELL arithmetic operators are given below. First four of them are regular arithmetic operators. Addition, subtraction, multiplication and division. The fifth of them is the modulus operator, which is the mod operation in mathematics. It gives the remainder when the first number is divided by the second number.

Table 4.2: ISELL arithmetic operators

Operation	Algebraic expression	ISELL expression
Addition	f+7	f+7
Subtraction	p−c	p−c
Multiplication	bm	b*m
Division	$\frac{x}{y}$	x/y
Modulus	r mod s	r@s
Power	x^y	xˆy
Logarithm	$\log_x y$	x$y

All operators can be applied to real and integer data types, whereas there are only specific commands for file (Table 4.3), matrix (Table 4.4) and string (Table 4.5) data types to operate on.

Table 4.3: ISELL file commands

Command	Operation
openfile	Opens a file for reading or writing
closefile	Closes a file
writefile	Writes a string or a value to a file
readfile	Reads a value to a variable from a file
readarrayfile	Reads a group of values to a string array

Equality and Relational Operators

Although we make operations on variables and constants, we may also compare them. For this purpose, we use the equality and relational operators. These operators can be applied to integer, real and string data types.

Assignment Operators

The operations such as

Table 4.4: ISELL matrix commands

Command	Operation
readposterior	Reads the posterior file into the matrix variable
readcposterior	Reads the posteriors of the correct class into the matrix variable
mresize	Takes a list of columns and produces a new matrix which contains only those columns
mcovariance	Takes the covariance of the first matrix and returns it as the second matrix
mcorrelation	Takes the correlation of the first matrix and returns it as the second matrix
meigenvectors	Finds the eigenvectors of the first matrix and returns it as the rows of the second matrix in the order of increasing eigenvalues
mtranspose	Takes the transpose of the matrix
msum	Returns the sum of the elements of the matrix
mtrace	Returns the trace of a square matrix
mprint	Prints the given matrix to the given file
loadmatrix	Loads a matrix from a file
maverage	Finds the column averages of the matrix
mstdev	Finds the column standard deviations of the matrix

Table 4.5: ISELL string commands

Command	Operation
tokenize	Divides the input string into a list of tokens separated via a given list of characters
countchar	Returns number of occurences of a given character in a given string variable
stringlength	Returns the length of a string variable
charatindex	Returns the character at the given position in the given string variable

```
x = x + y
z = z - j
```

where we add one variable to another variable or subtract one variable from another variable occurs so often in the programs that there are extra oper-

Table 4.6: ISELL equality and relational operators

Operator	ISELL operator	Meaning of ISELL condition
$=$	== or =	x is equal to y
$\neq$	!= or <>	x is not equal to y
$>$	>	x is greater than y
$<$	<	x is less than y
$\geq$	>=	x is greater than or equal to y
$\leq$	<=	x is less than or equal to y

ators (assignment operators) dedicated to them. With these operators all of the arithmetic operators are extended to be assignment operators. So the examples above can be simplified as

```
x += y
z -= j
```

Note that in ISELL assignments operators are commands, therefore they can only be used in prefix notation (Table 4.7).

Table 4.7: ISELL assignment commands

Assignment operator	Sample	Explanation
$+=$	+= c 7	c = c + 7
$-=$	−= d 4	d = d − 4
$*=$	*= e 5	e = e * 5
$/=$	/= f 3	f = f / 3
@=	@= g 9	g = g mod 9
^=	^= x 3	$x = x^3$
$=	$= y 4	y = $\log_4 y$

Increment and Decrement Operators

In the looping, we usually add 1 to the loop variable (increment) or subtract 1 from the loop variable (decrement). Since loops are one of the main structures of a program, in order to simplify the syntax of these sentences, ISELL has extra increment and decrement commands. For example

```
++ c
-- b
```

mean as

```
c = c + 1;
b = b - 1;
```

Table 4.8: ISELL increment and decrement operators

Operator	Sample	Explanation
++	++ a	Increment a by 1
−−	−− b	Decrement b by 1

4.2 If-switch

4.2.1 If-else Structure

Let say we want to make a decision in the program, for example we want to determine if a person's grade is greater than or equal to 60, and if it is so, to print a message such as "Congrulations! You passed.". We will do this in the program using an **if** statement as:

```
if >= grade 60
  writescreen Congrulations!_You_passed.
```

The syntax of the **if-else** statement is:

if relational_operator relational_operand1 relational_operand2
    $statementblock_1$
else
    $statementblock_2$
endif

Else part of this statement is optional.

When the compiler runs this program first *expression* is evaluated; if it is true, $statementblock_1$ is executed. Otherwise if the expression is false, and there is an **else** part, $statementblock_2$ is executed. The code for each statementblock can be a null statement,

```
if == x 4
endif
```

or a single statement,

```
if == y 4
  ++ y
endif
```

or a group of statements.

```
if == x 4
  ++ y
  += z %y%
endif
```

The expression in the if statement is called a *condition*. Each condition must be a single relational operation. A relational expression is formed by using the relational operators and the equality operators. (Discussed in the previous section). For example,

```
if > x 8
```

is a conditional expression that checks if the variable x is larger than 8 or not. ISELL currently does not support complex relational expressions.

4.2.2 Switch

Suppose you want to run C4.5 algorithm on 4 datasets (*iris*, *vote*, *mushroom*, and *bupa*). Your code will be as:

```
if = algindex 1
  c45 iris cv 1
else
  if = algindex 2
    c45 vote cv 1
  else
    if = algindex 3
      c45 mushroom cv 1
    else
      c45 bupa cv 1
    endif
  endif
endif
```

This multi-way decision can be more easily done with the reserved word **switch**. The **switch** statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. The syntax of **switch** is:

switch variable
    case $constant\_expression_1$
        $statements_1$
    case $constant\_expression_2$
        $statements_2$
    ...
endswitch

If value of the variable is equal to $constant\_expression_1$, then all $statements_1$ will be executed and the program continues after the **endswitch**. If expression is equal to $constant\_expression_2$, then all $statements_2$ will be executed and the program continues after the **endswitch**. If expression is not equal to none of the constant expressions, none of the statements will be executed and again the program continues after the **endswitch**.

Now our example program can be done with switch as follows:

```
switch algindex
  case 1
    = dataset iris
  case 2
    = dataset vote
  case 3
    = dataset mushroom
  case 4
    = dataset bupa
endswitch
c45 %dataset% cv 1
```

Some comments for switch:

- Each **case** is labeled by one constant or constant expression. In the example program, 1, 2, 3, and 4 are integer-valued constants.
- If a **case** matches the expression value, execution starts at that case. For example, if *algindex* is 2, the execution starts at

  ```
  = dataset vote
  ```

- All **case** expressions must be different.
- Cases can occur in any order. So in the example program you can exchange the positions of cases 1, 2, 3, and 4.

4.3 Loops

4.3.1 Repetition Structures

Most programs involve repetition or *looping*. A *loop* is a group of instructions the computer executes repeatedly while some *loop-continuation condition* remains true. There are two types of repetition:

1. Counter-controlled repetition
2. Sentinel-controlled repetition

Counter-controlled repetition is sometimes called *definite repetition* because we know in advance exactly how many times the loop will be executed. On the other hand, sentinel-controlled repetition is sometimes called *indefinite repetition* because it is not known in advance how many times the loop will be executed.

In counter-controlled repetition, a *control variable* is used to count the number of repetitions. The control variable is incremented or decremented (usually by 1) each time the group of instructions is performed. When the value of the control variable indicates that the correct number of repetitions has been performed, the loop terminates and the computer continues executing with the statement after the repetition structure.

Counter-controlled repetition requires:

1. The *name* of the control variable (or loop counter)
2. The *initial value* of the control variable
3. The *increment* (or *decrement*) by which the control variable is modified each time through the loop.
4. The condition that tests for the *final value* of the control variable (i.e., whether looping should continue).

Sentinel values are used to control repetition when:

1. The precise number of repetitions is not known in advance, and
2. The loop includes statements that obtain data each time the loop is performed.

The sentinel value indicates “end of data”. The sentinel is entered after all regular data items have been supplied to the program. Sentinels must be distinct from regular data items. For example, if the user will enter positive numbers, the sentinel can be any nonpositive value.

4.3.2 While statement

While statement has the following syntax:

```
while relational_operator relational_operand1 relational_operand2
    statementblock
endwhile
```

In this **while** structure, first the relationals expression is evaluated (See Section 4.2.1). If it is true, statementblock is executed and expression is re-evaluated. This cycle continues until expression becomes false, at which point execution resumes after statement. Let say we want to sum all numbers between 1 and 10 and print the result. The code will be

```
integer counter sum
= sum 0
= counter 1
while <= counter 10
  += sum %counter%
  ++ counter
endwhile
writescreen %sum%
```

Output:

```
55
```

This sample program prints the sum of the numbers from 1 to *counter*, where *counter* goes from 1 to 10. The declarations

```
integer counter sum
= counter 1
```

names the control variable, declares it to be integer and sets it to an *initial value* of 1.

The statement

```
++ counter
```

increments the loop counter by 1 each time the loop is performed. The loop-continuation condition

```
<= counter 10
```

in the **while** structure tests if the value of the control variable is less than or equal to 10 (the last value for which the condition is true).

4.3.3 For statement

For statement has the following syntax:

for *loop_control_variable* $expression_1$ $expression_2$
    statementblock
endfor *loop_control_variable*

$expression_1$ initializes the loop's control variable, $expression_2$ is the loop-continuation condition. The **for** repetition structure handles all the details of counter-controlled repetition. We can write our example program now with for:

```
integer counter sum
= sum 0
for counter 1 10
  += sum %counter%
endfor counter
writescreen %sum%
```

The program operates as follows. When the **for** structure begins to execute, the control variable *counter* is initialized to 1. Then, the loop-continuation condition *counter*<=10 is checked. Because the initial value of *counter* is 1, the condition is satisfied so *sum* is incremented by *counter* and assignment operator add the value of *counter* to *sum*. The control variable *counter* is then incremented by 1, and the loop begins again with the loop-continuation test. This process continues until the fail of the loop-continuation test (when *counter* is 11) and repetition terminates.

Some comments for **for** loop:

1. The initialization and loop-continuation condition can contain arithmetic expressions.

2. The increment may be negative (in which case you must add a fifth parameter -1 to the for statement)

3. If the loop-continuation condition is initially false, the body portion of the loop is not performed. Instead, execution proceeds with the statement following the for structure.

4.3.4 Nested loops

The loops that we have seen so far can also be nested. With two loops nested, we can have two or more **for** loops or **while** loops. If two loops are nested,

to finish one step of the outer loop, one must finish all the steps of the inner loop. Therefore, if the outer loop mas $m$steps and the inner loop has $n$steps, the statements in the inner loop will be executed mn times, whereas the statements out of the inner loop but in the outer loop are executed $m$times.

In the nested loop example, we want to run KNN algorithm (with $k = 3$) 100 times (with different train and test sets) on 4 datasets (*iris*, *vote*, *mushroom*, and *bupa*).

```
integer i j
string dataset
nearcount 3
for i 1 4
  switch i
    case 1
      = dataset iris
    case 2
      = dataset vote
    case 3
      = dataset mushroom
    case 4
      = dataset bupa
  endswitch
  for j 1 100
    knn %dataset% cv %j%
  endfor j
endfor i
```

The outer loop counts each number between 1 and 4 and sets the dataset name accordingly. The inner loop runs knn algorithm 100 times for the datasets selected in the outer loop.

4.4 Arrays

4.4.1 Arrays

An array is a group of memory locations. These locations are related by the fact that they have all the same name and the same type. To refer to a particular location or element within the array, we specify the name of the array and the position number of the particular element within the array.

Let see an example array c (Table 4.9). This array contains twelve elements. Any one of these elements may be referred to by giving the name

of the array followed by the position number of the particular element in square brackets([]). Thus, the first element of array c is referred to as $c[1]$, the second element of array c is referred to as $c[2]$, in general the i'th element of array c is referred to as $c[i]$.

Table 4.9: An example array c

c[1]	-2
c[2]	23
c[3]	-100
c[4]	1
c[5]	0
c[6]	3
c[7]	-245
c[8]	6
c[9]	34
c[10]	12
c[11]	7
c[12]	34

The position number is usually called subscript. The subscript may be an integer or an integer expression. For example, if $a = 5$ and $b = 6$, then the statement

```
+= c[%a+b%] 2
```

results in array element $c[11]$ being incremented by 2.

Also to divide the value of the seventh element of array c by 2 and assign the result to variable x, we would write

```
= x %c[6]/2%
```

4.4.2 Declaring Arrays

Arrays occupy space in memory. We specify the number of allocations required by each array, so that the computer reserves memory for each array. The syntax for array declaration is

elementtype_type array_name[array_size]

For example, to tell the computer reserve 12 elements for integer array c, the declaration

```
integer c[12]
```

is used. Array variables can also be declared with a single declaration

elementtype_type array_name1[array_size1] array_name2[array_size2] ...

such as

```
integer b[100] x[27]
```

It is important to remember that arrays are automatically initialized to zero by ISELL.

Let see an example, which compute the sum of the elements of the array.

```
integer a[100] i sum
for i 1 100
  = a[i] %i%
endfor i
= sum 0
for i 1 100
  += sum %a[i]%
endfor i
writescreen %sum%
```

Output:

```
5050
```

In this example program, array *a* is declared with 100 elements and initialized as {1, 2, 3, ..., 100}. In order to take the sum of the elements of array *a*, one must add all elements of array *a* (*a*[1], *a*[2], ..., *a*[100]) to *sum*. We can do this by making a loop and by setting the loop counter to count from 1 to 100.

4.4.3 Multidimensional Arrays

Arrays in ISELL can have multiple dimensions. For example, we use two dimensional array to represent tables of values consisting of information in rows and columns. To identify a particular element of such a table, we must specify two subscripts. The first identifies the row in which the element is contained, and the second identifies the column in which the element is contained. Let see an example of a two dimensional array *c*.

Every element in the array *c* is identified by an element name of the form *c*[*i*,*j*] *c* is the name of the array, and *i* and *j* are the subscripts that uniquely identify each element in *c*.

Table 4.10: An example two dimensional array c

	Column 1	Column 2	Column 3	Column 4
Row 1	c[1,1]	c[1,2]	c[1,3]	c[1,4]
Row 2	c[2,1]	c[2,2]	c[2,3]	c[2,4]
Row 3	c[3,1]	c[3,2]	c[3,3]	c[3,4]

All elements of a multidimensional array is initialized to zero automatically like a onedimensional array.

We now give three examples, where the first example is adding two matrices, the second example is multiplying these two matrices and the third example is transposing a matrix.
Addition:

```
integer i j
integer a[10,10] b[10,10] c[10,10]
for i 1 10
  for j 1 10
    = c[i,j] %a[i,j]+b[i,j]%
  endfor j
endfor i
```

To add two matrices, we add the elements of those two matrices one by one. Therefore, the i,j'th element of the resulting c array is equal to the sum of the i,j'th elements of the arrays a and b.
Multiplication:

```
integer i j k total
integer a[10,10] b[10,10] c[10,10]
for i 1 10
  for j 1 10
    = total 0
    for k 1 10
      += total %a[i,k]*b[k,j]%
    endfor k
    = c[i,j] %total%
  endfor j
endfor i
```

The i,j'th element of the resulting array c is obtained by summing up the products of the row i of the array a and the column j of the array b.
Transpose:

```
integer i j
for i 1 10
  for j 1 10
    = b[i,j] %a[j,i]%
  endfor j
endfor i
```

The *i*,*j*'th element of the resulting array *b* is equal to the *j*,*i*'th element of the array *a*.

Chapter 5

Operating System Commands

Some of the operating systems commands can be used in ISELL. These commands are given in Table 5.

Table 5.1: Operating system commands available in ISELL

Command	Explanation
cls or reset	Clear the screen
copy or cp	Copies one file to a given file
cd	Change current directory to the given directory
del or rm	Delete(s) one or more files
dir or ls	Lists contents of the current directory
move or mv	Moves (renames) one file to another file
rename	Changes the name of the given file to another
mkdir	Creates a new directory
rmdir	Removes the given directory

Chapter 6

Distributions & Statistical Tests

6.1 Probability Distributions

Probability distributions can be categorized into two as discrete and continuous probability distributions. Each probability distribution is founded on a probability density function, which is the probability that the variable X coming from that probability distribution has the value of x. For discrete distributions, this value can be calculated as single point probability but for continuous distributions it has to be expressed in terms of an integral between two points.

6.1.1 Normal Distribution

The probability density function for normal distribution is,

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{6.1}$$

If $\mu = 0$ and $\sigma = 1$, it is called *standard normal distribution*. To convert normal distribution to standard normal distribution, we subtract the mean and divide to the square root of the variance. The parameters, mean and variance are,

$$\begin{aligned} E[X] &= \mu \\ Var[X] &= \sigma^2 \end{aligned} \tag{6.2}$$

Many statistical tests are based on the assumption that the data comes from a normal distribution. Central limit theorem states that the sum of independent and identically distributed random variables approaches to normal

distribution as the sample size becomes large (usually larger than 30). The sum of normally distributed random variables is also normally distributed.

There is also a connection between random numbers generated using uniform distribution and normal distribution. By summing 12 randomly generated numbers and subtracting 6, we get a standard normally distributed random variable.

Sample Script:

```
>>normal 2.0
0.977
>>normal 1.0
0.841
>>normalinv 0.95
1.645
>>normalinv 0.99
2.326
```

6.1.2 Chi Square Distribution

The probability density function for chi square distribution is,

$$f(x) = \frac{e^{\frac{-x}{2}} x^{\frac{v}{2}-1}}{2^{\frac{v}{2}}\Gamma(\frac{v}{2})} \tag{6.3}$$

where v is the degree of freedom of the chi square distribution and Γ is the Gamma function. The parameters, mean and variance are,

$$\begin{aligned} E[X] &= v \\ Var[X] &= \sqrt{2v} \end{aligned} \tag{6.4}$$

The sum of squares of N normally distributed random variables is chi-square distributed with N degrees of freedom.

$$\chi_n = \mathcal{N}_1 + \mathcal{N}_2 + \ldots + \mathcal{N}_n \tag{6.5}$$

The sum of N chi-square distributed random variables is also chi-square distributed. The sum of the degrees of freedoms of the N random variables will be the degree of freedom of the resulting random variable.

$$\chi_{v_1+v_2+\ldots+v_n} = \chi_{v_1} + \chi_{v_2} + \ldots + \chi_{v_n} \tag{6.6}$$

Sample Script:

```
>>chi 2.0 3
0.572
>>chi 2.0 5
0.849
>>chiinv 0.05 2
5.991
>>chiinv 0.01 2
9.210
```

6.1.3 F Distribution

Probability density function for F distribution is,

$$f(x) = \frac{\Gamma(\frac{v_1+v_2}{2})(\frac{v_1}{v_2})^{\frac{v_1}{2}} x^{\frac{v_1}{2}-1}}{\Gamma(\frac{v_1}{2})\Gamma(\frac{v_2}{2})(1+\frac{v_1 x}{v_2})^{\frac{v_1+v_2}{2}}} \tag{6.7}$$

where v_1 and v_2 are degrees of freedom of the F distribution and Γ is the Gamma function. The parameters, mean and variance are,

$$E[X] = \frac{v_2}{v_2-2}$$
$$Var[X] = \sqrt{\frac{2v_2^2(v_1+v_2-2)}{v_1(v_2-2)^2(v_2-4)}} \tag{6.8}$$

F distribution is the ratio of two chi-square distributions with degrees of freedom v_1 and v_2 respectively, where each chi-square is first divided by its degree of freedom.

$$F_{v_1,v_2} = \frac{\frac{\chi_{v_1}}{v_1}}{\frac{\chi_{v_2}}{v_2}} \tag{6.9}$$

If $v_2 = 1$, then F distribution value is equal to the square of the t distribution's value with v_1 degrees of freedom.

$$F_{v_1,1} = t_{v_1}^2 \tag{6.10}$$

Sample Script:

```
>>f 2.0 5 10
0.164
>>f 3.0 5 10
0.066
```

```
>>finv 0.05 5 10
3.326
>>finv 0.01 5 10
5.636
```

6.1.4 t Distribution

The probability density function for t distribution is,

$$f(x) = \frac{(1 + \frac{x^2}{v})^{-\frac{v+1}{2}}}{B(0.5, 0.5v)\sqrt{v}} \tag{6.11}$$

where v is the degree of freedom of the t distribution and B is the Beta function. The parameters, mean and variance are,

$$\begin{aligned} E[X] &= 0 \\ Var[X] &= \sqrt{\frac{v}{v-2}} \end{aligned} \tag{6.12}$$

t distribution approaches to normal distribution as v gets larger.
Sample Script:

```
>>t 1.0 10
0.170
>>t 2.0 10
0.037
>>tinv 0.05 10
1.812
>>tinv 0.01 10
2.764
```

6.2 Statistical Tests

Statistical tests in the literature can be divided into two groups. Parametric tests and nonparametric tests.

In parametric methods, there is usually an assumption on the underlying distribution of the data. Most parametric methods are based on normality assumption because the theory behind the test can only be verified with the normal distribution. One problem with parametric methods is that we can not be sure about the underlying probability distribution, which is usually normal distribution. One way to overcome this problem is to perform hypothesis tests on classifiers' error rates to detect nonnormality.

In contrast to parametric methods, nonparametric methods do not assume a particular population probability distribution, which makes them valid for any population with any probability distribution. When the data is discrete, we can only use nonparametric methods.

Second type of grouping of statistical tests divides tests into three parts. Pairwise tests, range tests and multiple tests. These types of tests differ from each other on the number of classifiers they compare. Pairwise comparison methods compare two classifiers at a time, multiple comparison methods can compare all classifiers at once and range tests can compare any number of classifiers between two and K.

For each of the tests we say that, we have K classifiers on which we make N-fold crossvalidation, where each classifier is tested on N validation sets. So we have $K \times N$ results and let Y_{ij} represents the number of errors model i does on validation set j. Each classifier has a mean of $m_i, i = 1, 2, \ldots, k$. All of the classifiers have a general mean of m. For parametric tests we assume that, all of the K samples are independent and normally distributed. There is no such an assumption for nonparametric tests.

When we compare classifiers by using statistical tests, we will use the data given below. This data contains four classifiers and there are 10 trials for each classifier.

Table 6.1: Example error percentages of four classifiers

error1.txt	error2.txt	error3.txt	error4.txt
9	15	18	21
11	13	16	19
10	17	15	17
13	14	17	23
8	15	13	18
12	16	15	20
10	15	14	16
11	17	17	21
7	14	15	21
10	13	16	19

6.2.1 One-Way Anova Test

Hypothesis

The null hypothesis of this test is

$$H_0 : m_1 = m_2 = \ldots = m_K$$

against the alternative

H_1: At least one of the samples has a larger or smaller mean than at least one of the other samples

Method

The method starts by calculating MST as

$$MST = N\frac{\sum_{i=1}^{K}(m_i - m)^2}{K-1} \tag{6.13}$$

This is the total standard error between the means of the classifiers. We calculate MSE as

$$MSE = \frac{\sum_{i=1}^{K}\sum_{j=1}^{N}(Y_{ij} - m_i)^2}{KN-K} \tag{6.14}$$

which is total mean square error. If $\frac{MST}{MSE}$ ratio is smaller than the F distribution value with $K-1$ and $KN-K$ degrees of freedom, then H_0 is accepted. Otherwise it is rejected.

Assumptions

- All samples are independent and composed of normal random variables with unknown mean μ_i and unknown variance σ^2.

Sample Script:

```
anova error1.txt error2.txt error3.txt error4.txt
```

6.2.2 K-Fold Paired t Test

The name of this test comes from the fact that first it uses k-fold cross-validation, second it pairs test errors of the classifiers and third it uses t distribution.

Hypothesis

The null hypothesis of this test is

$$H_0 : m_1 = m_2$$

against the alternative

H_1: One of the classifiers has larger or smaller error than the other.

Method

p_i is the difference between the error rates of the two classifiers as $p_i = Y_{1i} - Y_{2i}$.

The estimator for mean

$$m = \frac{\sum_{i=1}^{k} p_i}{k} \tag{6.15}$$

and variance

$$S^2 = \frac{\sum_{i=1}^{k} (p_i - m)^2}{k - 1} \tag{6.16}$$

We calculate the test statistic

$$L = \frac{\sqrt{k} m}{S} \tag{6.17}$$

The null hypothesis is accepted with α level of confidence, if this value is between $-t_{\alpha/2,k-1}$ and $t_{\alpha/2,k-1}$, otherwise it is rejected.

Assumptions

- The difference of the classifier's errors on the test set is approximately normally distributed with zero mean and unknown variance σ^2.

Sample Script

```
cvttest error1.txt error2.txt
```

6.2.3 5×2 cv t Test

This test is proposed by [1]. In this test five replications of 2–fold cross–validation are performed.

Hypothesis

The null hypothesis of this test is

$$H_0 : m_1 = m_2$$

against the alternative

H_1: One of the classifiers has larger or smaller test error than the other.

Method

$p_i^{(j)}$ is the difference between the error rates of the two classifiers as $p_i^1 = Y_{1(2i-1)} - Y_{2(2i-1)}$ and $p_i^2 = Y_{1(2i)} - Y_{2(2i)}$. s_i^2 is the estimated variance of the i'th replication as $s_i^2 = (p_i^{(1)} - \bar{p_i})^2 + (p_i^{(2)} - \bar{p_i})^2$.

Total variance is then

$$VAR = \sum_{j=1}^{5} s_i^2 \tag{6.18}$$

Now the null hypothesis H_0 is accepted with α level of confidence if the value $\frac{p_i^1}{\sqrt{\frac{VAR}{5}}}$ is between $-t_{\alpha/2,5}$ and $t_{\alpha/2,5}$. Otherwise it is rejected.

Assumptions

- The difference of the classifier's errors on the test sets, is treated as approximately normally distributed with zero mean and unknown variance σ^2.
- $p_i^{(1)}$ and $p_i^{(2)}$ are independent normals.
- Each of the estimated variances s_i^2 are assumed to be independent.
- $p_i^{(1)}$ and the sum of the variances of the differences are independent of each other.

Sample Script

```
tpaired5x2 error1.txt error2.txt
```

6.2.4 5×2 cv F Test

This test first appeared in [2]. It is a variant of the previous version 5×2 cv t test. This version of the test has lower Type I error and higher power.

Hypothesis

The null hypothesis of this test is

$$H_0 : m_1 = m_2$$

against the alternative

H_1: One of the classifiers has larger or smaller test error than the other.

Method

We have $n = 10$ in this test. 10 comes from 5x2 cross validation, five replications of 2–fold cross–validation. $p_i^{(j)}$ is the difference between the error rates of the two classifiers as $p_i^1 = Y_{1(2i-1)} - Y_{2(2i-1)}$ and $p_i^2 = Y_{1(2i)} - Y_{2(2i)}$. s_i^2 is the estimated variance for each of the five replication as $s_i^2 = (p_i^{(1)} - \bar{p_i})^2 + (p_i^{(2)} - \bar{p_i})^2$.

To test this, first we calculate sum of squares of $p_i^{(j)}$'s:

$$DIFFS = \sum_{i=1}^{5}\sum_{j=1}^{2}(p_i^{(j)})^2 \tag{6.19}$$

Second we calculate sum of variances of five replications

$$VAR = 2\sum_{j=1}^{5} s_i^2 \tag{6.20}$$

Now the null hypothesis H_0 is accepted with α level of confidence, if $\frac{DIFFS}{VAR}$ is smaller than the probability value of F distribution with degrees of freedoms 10, 5 and α level of confidence.

Assumptions

- The difference of the classifier's errors on the test sets, is treated as approximately normally distributed with zero mean and unknown variance σ^2.
- $p_i^{(1)}$ and $p_i^{(2)}$ are independent normals.
- Each of the estimated variances s_i^2 are assumed to be independent.
- The sum of squares of the differences of the error rates and sum of the variances of the differences are independent of each other.

Sample Script

```
ftest error1.txt error2.txt
```

6.2.5 Newman-Keuls Test

The design and probability distribution used is the same as Tukey's procedure. But the form of the test is changed. This test is first proposed by Newman(1939). Later in 1952, Keuls proposed the same test in another journal, Euphytica.

Hypothesis

The null hypothesis of this test is that all means in a subset of L means are equal

$$H_0 : m_1 = m_2 = \ldots = m_L$$

against the alternative

H_1: At least one of the samples has a larger or smaller mean than at least one of the other samples in a subset L samples of K samples of data.

If H_0 is rejected, we make comparisons on all the means and find which means are significantly different from each other.

Method

The basic idea in multiple range tests is comparing the best and the worst of the subset of classifiers selected. If they are not significantly different from each other, then we can say that all of the classifiers between this two methods are also not significantly different.

To be able to do that, we first put the classifiers means in increasing order. We then test for equality of K means, after $K-1$ means, $K-2$ means until 2 means. But at each time when we compare L means, we only make $K-L+1$ tests. First we check the equality of means m_1 and m_L, then m_2 and m_{L+1}, ..., m_{K-L+1} and m_K.

If L means are equal to each other, then we do not need to make any further comparisons between these means. So we do not have to control any of the subsets of the means between m_i and m_j, when m_i and m_j are not significantly different from each other.

The comparisons of m_i and m_j is done by subtracting the two means and comparing the difference with $q^{0.95}_{K(N-1),M}\sqrt{\frac{MSE}{N}}$, where M is the number of means in the subset compared, $q^{\alpha}_{f_1,f_2}$ is the studentized range distribution with α confidence and f_1, f_2 degrees of freedom. If the difference is smaller than the calculated number we accept the null hypothesis, otherwise we reject it.

Assumptions

- Y_{1i} and Y_{2i} are normally distributed random variables with means μ_i and a common variance σ^2.

Sample Script

```
newmankeuls error1.txt error2.txt error3.txt error4.txt
```

6.2.6 Sign Test

The sign test is one of the nonparametric oldest tests. It was first used in 18th century to compare the number of males born with the number of females born in 82 years in England. The aim of the sign test is to check for equality of the number of positive examples with the number of negative examples. If one can separate the data into two groups of examples, sign test could be applied.

Hypothesis

The hypothesis of this test is that the number of positive examples are equal to the number of negative examples.

$$H_0 : P(+) = P(-)$$

against the alternative

$$H_1 \text{:}\ P(+) \neq P(-).$$

Method

The data consists of observations of a bivariate random sample (X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n), where there are N examples. Within each pair (X_i, Y_i) a comparison is done and each pair is classified as '+' if $X_i > Y_i$, '–' if $X_i < Y_i$ and 0 if $X_i = Y_i$. Number of positive examples are represented as P(+), number of negative examples are represented as $P(-)$ and total number of examples are represented as $P(+) + P(-) = N$. The test statistic uses snadard normal distribution with $\alpha/2$ level of confidence. Test statistic t is given as:

$$t = \frac{1}{2}(n + z_{\alpha/2}\sqrt{n}) \tag{6.21}$$

If $P(+)$ is between t and $N-t$, then the null hypothesis is accepted, otherwise the null hypothesis is rejected.

Assumptions

- The bivariate random variables (X_i, Y_i) are mutually independent.
- Each pair (X_i, Y_i) may be represented as positive, negative or 0.

Sample Script

```
signtest error1.txt error2.txt
```

6.2.7 Wilcoxon Rank Test

This test first appeared in Wilcoxon (1945) for equal sample sizes and generalized in Mann and Whitney (1947) for unequal sample sizes.

Hypothesis

The hypothesis of this test is

$$H_0 : m_1 = m_2$$

against the alternative

H_1: One of the classifiers has larger or smaller test error than the other.

Method

To make this test, first the ranks of the results of the classifiers R_{ij} (the rank of the j'th trial of the i'th classifier) must be determined. To give ranks, we order all of the Y_{ij}'s, and give the best one the first rank, the second best one the second rank etc. If ties occur, we give the average rank to all Y_{ij}s. After finding ranks, we find the sum of the ranks of the first classifier, T_1.

$$T_1 = \sum_{i=1}^{N} R_{1i} \tag{6.22}$$

If there are ties in the data, we get new T_L by using the following formula:

$$T_1 = \frac{T_1 - N\frac{2N+1}{2}}{\sqrt{\frac{N^2}{2N(2N-1)}\sum_{i,j} R_{ij} - \frac{N^2(2N+1)^2}{4(2N-1)}}} \tag{6.23}$$

By using T_1 we get the following statistic:

$$S = \frac{T_1 + 0.5 - \frac{N(2N+1)}{2}}{\sqrt{\frac{N^2(2N+1)}{12}}} \tag{6.24}$$

If S is between $-z_{\alpha/2}$ and $z_{\alpha/2}$, then the null hypothesis is accepted with α level of confidence, otherwise it is rejected.

Assumptions

- Both samples are random samples from their respective populations.
- There is independence within each sample and between the two samples.

Sample Script

```
wilcoxon error1.txt error2.txt
```

6.2.8 Kruskal-Wallis Test

The Wilcoxon test for two samples was extended to include more than two samples by Kruskal and Wallis (1952). This test can be used with unequal samples sizes.

Hypothesis

The hypothesis of this test is

$$H_0 : m_1 = m_2 = \ldots = m_K$$

against the alternative

H_1: At least one of the samples has a larger or smaller mean than at least one of the other samples

Method

The ranks R_{ij} are assigned to the examples as in Wilcoxon test. The test statistic is calculated as:

$$T = \frac{12}{KN(KN+1)} \sum_{i=1}^{K} \frac{R_i^2}{N} - 3(KN+1) \tag{6.25}$$

,where R_i is the sum of ranks of the i'th classifier and $\sum_{i=1}^{K} R_i = \frac{KN(KN+1)}{2}$.

If there are ties in the dataset, test statistic is defined as:

$$T = \frac{1}{S^2}(\sum_{i=1}^{k} \frac{R_i^2}{n_i} - \frac{N(N+1)^2}{4}) \tag{6.26}$$

where S^2 is,

$$S^2 = \frac{1}{N-1}(\sum_{i,j} R(X_{ij})^2 - N\frac{(N+1)^2}{4}) \tag{6.27}$$

We test the null hypothesis by comparing this value with chi-square distribution with K-1 degrees of freedom. If the value is smaller than the chi-square distribution value, we accept the null hypothesis, otherwise we reject it.

Assumptions

- All samples are random samples from their respective populations.
- There is independence within each sample and among the various samples.

Sample Script

```
kruskalwallis error1.txt error2.txt error3.txt error4.txt
```

6.2.9 Van Der Waerden Test

Van Der Waerden (1952/1953) proposed another comparison of K samples as in Kruskal-Wallis test. This test can also be used with unequal sample sizes.

Hypothesis

The hypothesis of this test is

$$H_0 : m_1 = m_2 = \ldots = m_K$$

against the alternative

H_1: At least one of the samples has a larger or smaller mean than at least one of the other samples

Method

Like Kruskal-Wallis test, we calculate ranks R_{ij} of the examples. We convert each rank R_{ij} into the $\frac{R_{ij}}{KN+1}$ quantile of a standard normal random variable.

$$A_{ij} = the \frac{R_{ij}}{KN+1}{}' th\ quantile \tag{6.28}$$

We calculate the mean of these A_{ij} for each classifier as:

$$\hat{A}_i = \frac{1}{N} \sum_{j=1}^{N} A_{ij} \tag{6.29}$$

The variance of these scores is

$$S^2 = \frac{1}{KN - 1} \sum_{KN} A_{ij}^2 \tag{6.30}$$

The test statistic is defined as

$$T = \frac{1}{S^2} \sum_{i=1}^{K} N \hat{A_i}^2 \tag{6.31}$$

We test the null hypothesis by comparing this value with chi-square distribution with K-1 degrees of freedom. If the value is smaller than the chi-square distribution value, we accept the null hypothesis, otherwise we reject it.

Assumptions

- All samples are random samples from their respective populations.
- There is independence within each sample and among the various samples.

Sample Script

```
vanderwaerdan error1.txt error2.txt error3.txt error4.txt
```

Chapter 7

Supervised Learning

7.1 Experimental Setup

To run a supervised learning algorithm, first you need to set the datasets directory (**setdatadir** command) then load the datasets (**loaddatasets** command). To see how these commands work, see Chapter 3.

In ISELL, you can run a supervised algorithm in three ways:

Cross-validation In k fold cross-validation, the dataset is divided into k equal parts. To generate a train set and validation set pair, we keep one of the k parts out as the validation set and combine the remaining $k-1$ parts to produce the training set. If we do this for each of the k different parts, we get k training and validation set pairs. An example script for k fold cross-validation is:

```
runcount 1
foldcount 10
c45 iris cv 1
```

If you want 5×2 cross-validation [1], you will use **runcount** command to make 5 times 2-fold cross-validation. For example:

```
runcount 5
foldcount 2
nearcount 3
knn vote cv 1
```

will run K-nearest neighbor algorithm with $K = 3$ on *vote* dataset using 5×2-fold cross-validation.

Bootstrap Another cross-validation technique is *bootstrap* [3]. The basic idea is to randomly draw datasets with replacement from the training set where each sample contains the same number of instances as the training set. This is done K times producing K bootstrap datasets. An example script for k bootstrap is:

```
runcount 100
nearestmean bupa bootstrap 1
```

will run 100 times nearest-mean algorithm on *bupa* dataset using bootstrap sampling technique.

Single train and test set If you have your training and test set pairs ready, you can also use ISELL to train the supervised algorithm on your training set and test on your test set. To do this, just write:

```
logistic mushroom mytrain.txt mytest.txt
```

will train logistic regression on mytrain.txt train set and will test on mytest.txt test set, where both files are formatted according to the *mushroom* dataset.

You can also separate some part of the dataset as a test set before doing any bootstrap or cross-validation. ISELL will then test the algorithm not on the validation set (for cross-validation) but on the test set. In this case, there is only one test set, but since there are 10 different training sets, there will be 10 different classifiers (regressors) and each will be tested on this single test set to produce 10 error rate. Use **testpercentage** command for this purpose. For example:

```
testpercentage 0.333333
runcount 1
foldcount 10
c45 iris cv 1
```

will separate 1/3 part of the dataset as the test set, then will do 10-fold crossvalidation on the remaining 2/3 part of the dataset.

To save the outputs (error rate, confusion matrix, etc.) of the supervised algorithm, you can use the following commands:

output When you run a supervised algorithm on a dataset, the expected error(s) are written on the screen. In order to forward these errors to an output file, you can use the output command. For example:

```
output error.txt
runcount 1
foldcount 10
c45 iris cv 1
```

will run C4.5 algorithm on *iris* dataset and will print the errors on 10 folds to the output file error.txt. Opening a new output file will close the previous output file automatically.

printposterior

posteriorfile If you are running a classification algorithm, you may not only want the expected error but also the posterior probabilities for each class of each test instance. In order to print the posterior probabilities use **printposterior** (set to **on**) and set the **posteriorfile** to the posterior output file. For example:

```
output error.txt
printposterior on
posteriorfile posterior.txt
runcount 1
foldcount 10
c45 iris cv 1
```

printconfusion

confusionfile If you are running a classification algorithm, you may not only want the expected error but also the confusion matrix. In order to print the confusion matrix use **printconfusion** (set to **on**) and set the **confusionfile** to the confusion output file. For example:

```
output error.txt
printconfusion on
confusionfile confusion.txt
runcount 1
foldcount 10
c45 iris cv 1
```

savemodel

modelfile You can also save the model (if you are running k-fold crossvalidation k models) of the supervised learning algorithm. Use **savemodel** (set to **on**) and set the **modelfile** to the model output file. For example:

```
output error.txt
savemodel on
modelfile model.txt
runcount 1
foldcount 10
c45 iris cv 1
```

Then you can run **testmodel** command to test the saved model on a separate test set. For example:

```
testmodel modelfile newtest.txt
```

will test the model saved before (10 decision trees) on newtest.txt test set.

7.2 Classification Algorithms

7.2.1 SELECTMAX

SELECTMAX decides based on the prior class probability without looking at the input. No parameters.

7.2.2 NEARESTMEAN

NEARESTMEAN keeps the mean vector of the features for each class and chooses the class with the nearest mean using Euclidean distance [4]. No parameters.

7.2.3 GAUSSIAN

GAUSSIAN is the parametric classifier where each class is represented by a Gaussian and a common covariance matrix is shared by all classes [4]. No parameters.

7.2.4 NAIVEBAYES

NAIVEBAYES is the classic classifier where each feature is assumed to be Gaussian distributed [4] and each feature is independent from other features. No parameters.

7.2.5 QDACLASS

QDACLASS is parametric classifier where each class is represented by a Gaussian and covariance matrices are different for each class [4]. No parameters.

7.2.6 C4.5RULES

C4.5RULES starts with the C4.5 tree and converts it to a rule set by writing each path from the root to a leaf as a rule. The initial rule set therefore has as many rules as there are leaves in the tree [5]. These generated rules may contain superfluous conditions and may cause overfitting. C4.5Rules prunes rules by removing those conditions whose removal do not increase the error rate on the training set. After pruning the conditions in the rules, the best subset of the rules is searched according to MDL [6]. The search is exhaustive if the number of rules is small (less than or equal to ten), otherwise it is done by best first search starting from ten different subsets of rules. No parameters.

7.2.7 KNN

KNN is the nearest-neighbor classification algorithm [7]. Number of nearest neighbors are set using **nearcount** command.

7.2.8 C45

C45 is the archetypal decision tree method [5]. A decision tree is made up of internal decision nodes and terminal leaves. The input vector is composed of d attributes, $\boldsymbol{x} = [x_1, \ldots, x_d]^T$, and the aim in classification is to assign $\boldsymbol{x}$ to one of K mutually exclusive and exhaustive classes, $\mathcal{C}_1, \ldots, \mathcal{C}_K$. Each internal node m implements a decision function, $f_m(\boldsymbol{x})$, where each branch of the node corresponds to one outcome of the decision. Each leaf of the tree carries a class label. In *univariate* decision trees (for example in C4.5), the decision at internal node m uses only one attribute, i.e., one dimension of $\boldsymbol{x}$, x_j.

Type of pruning is set using **prunetype** command. If type of pruning is pre-pruning, we need a data size threshold, because, in pre-pruning, the tree is not fully constructed until zero training error but is kept simple by early termination. At any node, if the dataset reaching that node is small, even if it is not pure, it is not further split and a leaf node is created instead of growing a subtree. This threshold is set using **pruningthreshold** command.

7.2.9 LDT

LDT is Linear discriminant tree algorithm. The details of the algorithm are given in [8]. The parameters are the same as C4.5.

7.2.10 LDACLASS

LDACLASS is the Multiple Discriminant Analysis classifier [7]. If the covariance matrix is singular, Principal Component Analysis (PCA) is used to decrease dimensionality. The percentage of variance explained in PCA is set using **variance_explained** command.

7.2.11 LOGISTIC

LOGISTIC is the linear logistic discriminator trained to minimize cross-entropy by gradient-descent. Discrete features are converted to numeric features by 1-of-n encoding. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command.

7.2.12 IREP, IREP2, RIPPER

IREP, IREP2 and RIPPER are rule learning algorithms. RIPPER learns rules from scratch starting from an empty rule set. It has two phases: In the first phase, it builds an initial set of rules, one at a time, and in the second phase, it optimizes the rule set m times, typically twice [9].

In the first phase, RIPPER learns rules one by one. First, propositional conditions are added one at a time to a rule, at each step choosing the condition that maximizes the information gain. Choosing the best is done exhaustively by searching all possible split points ($x_i < \theta$ or $x_i \geq \theta$ for numeric features and $x_i = v$ for discrete features). When we insert a new condition, the coverage of the rule (number of instances that a rule covers) decreases, since some of the examples that are covered by the rule do not obey the new condition. We stop adding conditions further, when the current rule does not cover any negative examples.

The rule is then pruned to alleviate overfitting. Conditions are removed one by one, where each time the condition that mostly increases a rule value metric (based on the accuracy on the prune set) is selected for removal. We stop removing conditions when the rule value metric can not be increased further.

RIPPER uses MDL and in RIPPER, rules are added to a rule set to minimize the total description length. The total description length of a rule set is the number of bits to represent the rules plus the number of bits needed to identify the exceptions to the rules in the training set. We stop adding rules when the total description length of the rule set is larger than 64 bits from the minimum description length obtained so far or if the error rate of the new rule is larger than 0.5.

In the second phase, rules in the rule set are optimized. Two alternatives are grown for each rule. The first candidate, the replacement rule, is grown starting with an empty rule, whereas the second candidate, the revision rule, is grown starting with the current rule. These two rules and the original rule are compared and the one with the smallest description length is selected and put in place of the original rule.

RIPPER learns rules to separate a positive class from a negative class. A ($K > 2$)-class problem is converted into a sequence of $K - 1$ two-class problems. This is done first by sorting classes in the order of increasing prior probabilities. In order to solve the ith two-class problem, $i = 1, \ldots, K - 1$, C_i is taken as the positive class and the union of classes C_{i+1}, C_{i+2}, ..., C_K is taken as the negative class. For each two-class problem, there are the two phases of learning a rule set and optimizing it, as we discussed above. We therefore generate a sequence of rules and the most probable class is the final default rule without conditions and it matches any instance which is not covered by any previous rule.

Number of optimization steps are set using **optimizecount** command.

7.2.13 RBF

RBF is the Radial Basis Function network classification algorithm [4]. Number of Gaussians are set using **hiddennodes** command. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command.

7.2.14 MLPGENERIC

MLPGENERIC is the well-known multilayer perceptron classification algorithm, where one can train the neural network with any number of hidden layers (from zero that is linear perceptron). Number of hidden nodes in each layer is set using **hiddennodes** command. Initial learning rate is set using

learningrate command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command.

7.2.15 DNC

DNC [10] builds MLP networks for classification with a single hidden layer. It starts with small number of hidden units and incrementally adds hidden nodes to the network until a satisfactory solution is found. Hidden nodes are added one at a time and to the same hidden layer. The weights of the newly added hidden node are initialized randomly to a small number. The whole network (both the old hidden nodes and the new one) is re-trained after each addition. The decision of adding a new hidden node is made by considering the flatness of the average error curve. Number of hidden nodes in each layer is set using **hiddennodes** command. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command. Error threshold, error drop ratio and the window size parameters of the DNC algorithm are set with **errorthreshold**, **errordropratio** and **windowsize** commands.

7.2.16 MULTILDT

MULTILDT is the multivariate tree where each decision node is linear [11] and uses all inputs as opposed to a univariate decision tree, such as C4.5. When the inputs are correlated, looking at one feature may be too restrictive. A *linear multivariate tree*, at each internal node, uses a linear combination of all attributes.

$$f_m(\boldsymbol{x}) : \boldsymbol{w}_m^T \boldsymbol{x} + w_{m0} = \sum_{j=1}^{d} w_{mj} x_j + w_{m0} > 0 \tag{7.1}$$

To be able to apply the weighted sum, all the attributes should be numeric and discrete values need be represented numerically (usually by 1-of-L encoding) before. The weighted sum returns a number and the node is binary. In this linear case, each decision node divides the input space into two with a hyperplane of arbitrary orientation and position where successive decision

nodes on a path from the root to a leaf further divide these into two and the leaf nodes define polyhedra in the input space. The parameters are the same as C4.5. For PCA when required, variance explained is set using **variance_explained** command.

7.2.17 OMNILDT

OMNILDT is the omnivariate decision tree algorithm [12]. In omnivariate decision tree, at each node, we train and compare all three possible nodes; univariate, linear multivariate, and nonlinear multivariate, and using a statistical test, we choose the best and continue tree induction recursively. Each node implements a binary split to induce simple and interpretable trees. To group $K > 2$ classes into two, we use Guo and Gelfand's [13] exchange heuristic which uses class information.

7.2.18 SVM, NUSVM

SVM (or NUSVM) is the Support vector machine classifier with different type of kernels. The source code is exported from the LIBSVM 2.82 library [14]. C (or nu), Sigma, gamma and bias values are set using **svmC** (or **svmnu**), **sigma**, **svmgamma**, and **svmcoef0** commands respectively. Kernel type is set using **kerneltype** command and if the kernel is poynom kernel, the degree of the polynomial kernel is set using **svmdegree** command.

7.3 Regression Algorithms

7.3.1 ONEFEATURE

ONEFEATURE finds the best axis-aligned split $x_i < c$. Using that split, the training instances are divided into two parts. For each part, the estimated output value is calculated by taking the average of the output values. This is a regression stump with only one node and two leaves. No parameters.

7.3.2 SELECTAVERAGE

SELECTAVERAGE finds the average of the output values of the training data. It uses this value as the estimated output value for all test cases. This the regression version of SELECTMAX. No parameters.

7.3.3 LINEARREG

LINEARREG is the classic linear regression algorithm. No parameters.

7.3.4 QUANTIZEREG

QUANTIZEREG divides each input feature into ten equal-sized bins to generate hyperrectangles in the instance space and calculates the average output in each bin. To combat the curse of dimensionality, Principal Component Analysis (PCA) is used to decrease dimensionality to two and regressogram is applied in this two dimensional space. Number of bins are set using *partitioncount* command.

7.3.5 KNNREG

KNNREG is the nearest-neighbor regression algorithm [7]. Number of nearest neighbors are set using **nearcount** command.

7.3.6 GPROCESSREG

GPROCESSREG does Gaussian process regression as described in [15]. Sigma, gamma and bias values are set using **sigma**, **svmgamma**, and **svmcoef0** commands respectively. Kernel type is set using **kerneltype** command and if the kernel is poynom kernel, the degree of the polynomial kernel is set using **svmdegree** command.

7.3.7 RBFREG

RBF is the Radial Basis Function network regression algorithm [4]. Number of Gaussians are set using **hiddennodes** command. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command.

7.3.8 MLPGENERICREG

MLPGENERIC is the well-known multilayer perceptron regression algorithm, where one can train the neural network with any number of hidden layers (from zero that is linear perceptron). Number of hidden nodes in each

layer is set using **hiddennodes** command. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command.

7.3.9 DNCREG

DNC [10] builds MLP networks for regression with a single hidden layer. It starts with small number of hidden units and incrementally adds hidden nodes to the network until a satisfactory solution is found. Hidden nodes are added one at a time and to the same hidden layer. The weights of the newly added hidden node are initialized randomly to a small number. The whole network (both the old hidden nodes and the new one) is re-trained after each addition. The decision of adding a new hidden node is made by considering the flatness of the average error curve. Number of hidden nodes in each layer is set using **hiddennodes** command. Initial learning rate is set using **learningrate** command. Number of epochs is set using **perceptronrun** command. After each epoch, the learning rate is decreased by multiplying it with a constant factor. This factor is set using **etadecrease** command. If momentum is used, set alpha parameter of the momentum using **alpha** command. Error threshold, error drop ratio and the window size parameters of the DNC algorithm are set with **errorthreshold**, **errordropratio** and **windowsize** commands.

7.3.10 SVMREG, NUSVMREG

SVMREG (or NUSVMREG) is the Support vector machine regressor with different type of kernels. The source code is exported from the LIBSVM 2.82 library [14]. C (or nu), p, Sigma, gamma and bias values are set using **svmC** (or **svmnu**), **svmp**, **sigma**, **svmgamma**, and **svmcoef0** commands respectively. Kernel type is set using **kerneltype** command and if the kernel is poynom kernel, the degree of the polynomial kernel is set using **svmdegree** command.

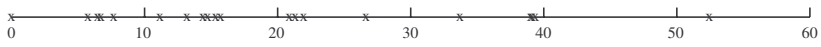
Chapter 8

Graphics Environment

8.1 Plot Commands

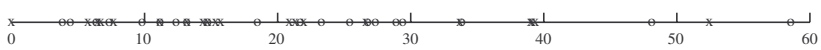
plotx Plots one dimensional data.

```
plotx data1.txt
legendposition none
```



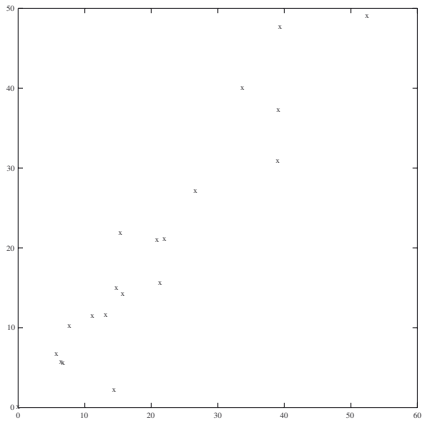
Another example:

```
plotx data1.txt
hold on
plotx data2.txt
legendposition none
```



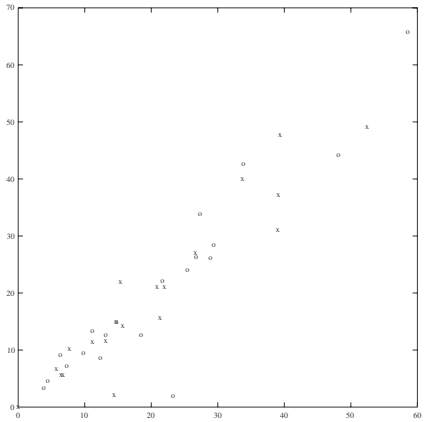
plotxy Plots two dimensional data.

```
plotxy data3.txt
legendposition none
```



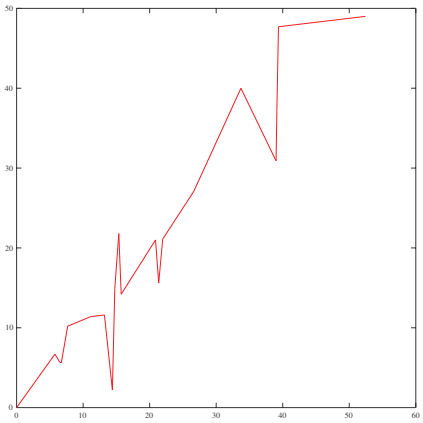
Another example:

```
plotxy data3.txt
hold on
plotxy data4.txt
legendposition none
```



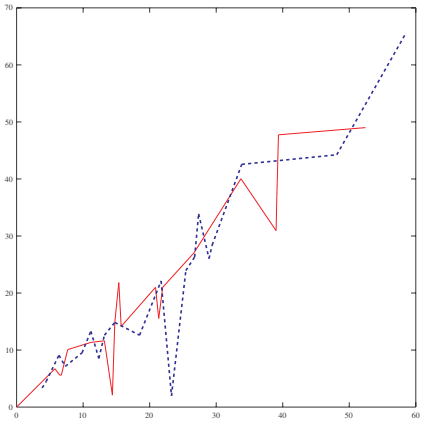
plotxyline Plots two dimensional data and draws lines between neighbor points.

```
plotxyline data5.txt red 1
legendposition none
```



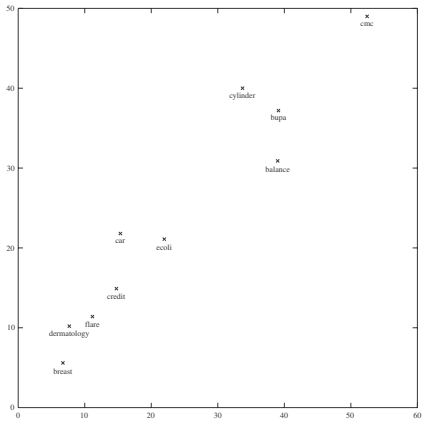
Another example:

```
plotxyline data5.txt red 1
hold on
plotxyline data6.txt blue 2
legendposition none
```



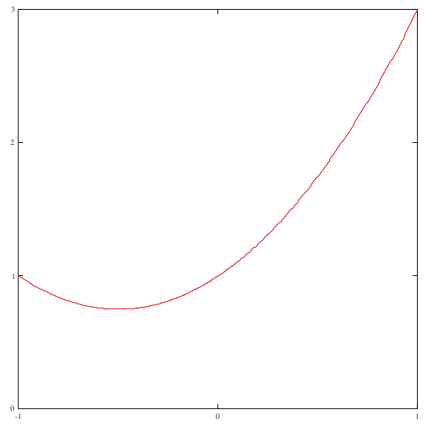
plotxynames Plots two dimensional data where each point is labeled with a given name

```
plotxynames data8.txt
```



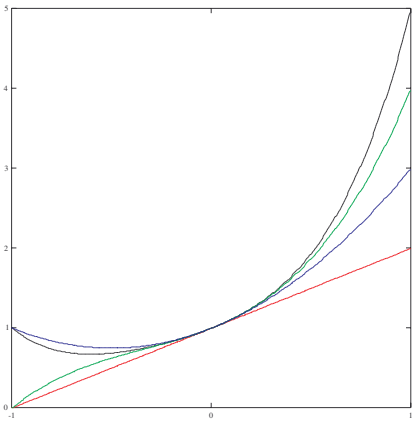
polyplot Plots a given polynomial between two values

```
polyplot -1 1 x2+x+1 red
legendposition none
```



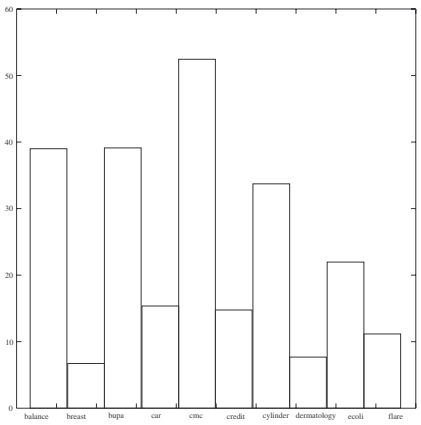
Another example:

```
polyplot -1 1 x+1 red
hold on
polyplot -1 1 x2+x+1 blue
polyplot -1 1 x3+x2+x+1 green
polyplot -1 1 x4+x3+x2+x+1 black
legendposition none
```



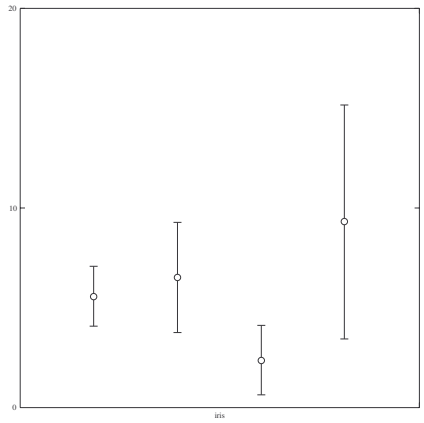
plotay Plots dataset and error (or accuracy) information

```
plotay data7.txt
legendposition none
```



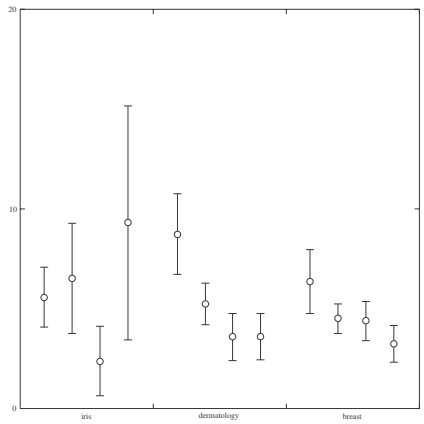
plotmv Plots mean and standard deviation of the files

```
plotmv 1.c45 1.knn 1.lda 1.lp
xaxisnames iris
```



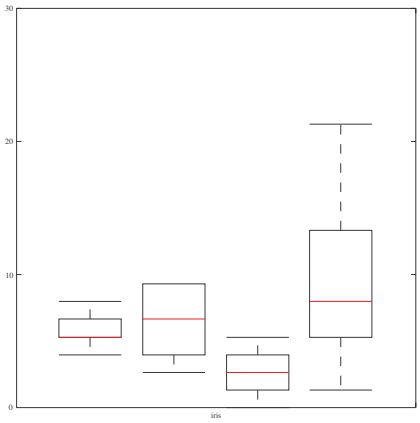
Another example:

```
plotmv 1.c45 1.knn 1.lda 1.lp
hold on
plotmv 2.c45 2.knn 2.lda 2.lp
plotmv 3.c45 3.knn 3.lda 3.lp
xaxisnames iris dermatology breast
```



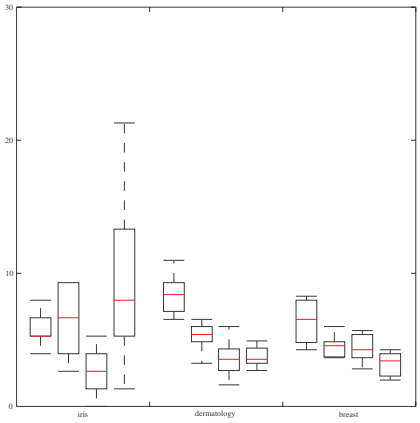
boxplot Draws the boxplot of the files

```
boxplot 1.c45 1.knn 1.lda 1.lp
xaxisnames iris
```



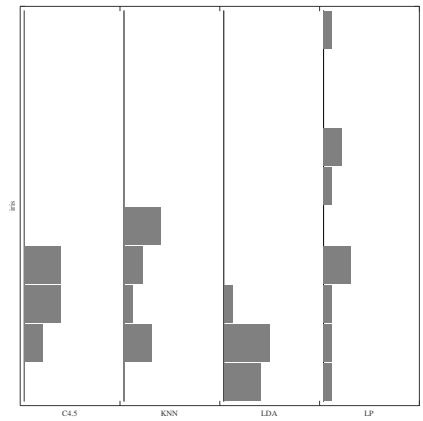
Another example:

```
boxplot 1.c45 1.knn 1.lda 1.lp
hold on
boxplot 2.c45 2.knn 2.lda 2.lp
boxplot 3.c45 3.knn 3.lda 3.lp
xaxisnames iris dermatology breast
```



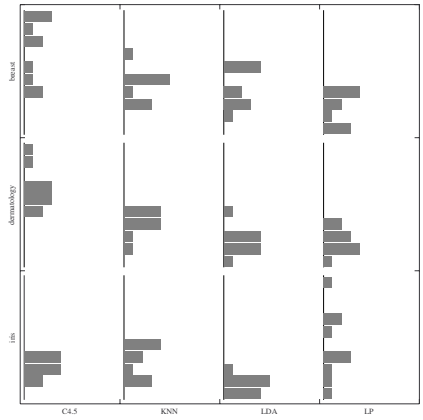
boxplot Draws the histogram plot of the files

```
histplot 1.c45 1.knn 1.lda 1.lp
xaxisnames C4.5 KNN LDA LP
yaxisnames iris
```



Another example:

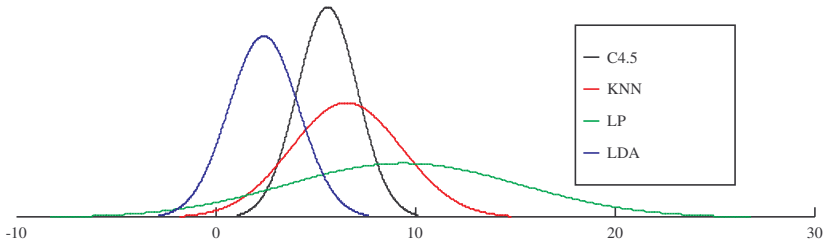
```
histplot 1.c45 1.knn 1.lda 1.lp
hold on
histplot 2.c45 2.knn 2.lda 2.lp
histplot 3.c45 3.knn 3.lda 3.lp
xaxisnames C4.5 KNN LDA LP
yaxisnames iris dermatology breast
```



plotgauss Plots gaussian fits for the data in the file

```
plotgauss 1.c45
hold on
plotgauss 1.knn
plotgauss 1.lda
```

```
plotgauss 1.lp
legendnames C4.5 KNN LDA LP
```



Bibliography

[1] T. G. Dietterich, "Approximate statistical tests for comparing supervised classification learning classifiers," *Neural Computation*, vol. 10, pp. 1895–1923, 1998.

[2] E. Alpaydın, "Combined 5×2 cv f test for comparing supervised classification learning classifiers," *Neural Computation*, vol. 11, pp. 1975–1982, 1999.

[3] B. Efron, "Computers and the theory of statistics," *SIAM Review*, vol. 21, pp. 460–480, 1979.

[4] E. Alpaydın, *Introduction to Machine Learning*. The MIT Press, 2004.

[5] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Meteo, CA: Morgan Kaufmann, 1993.

[6] J. R. Quinlan and R. L. Rivest, "Inferring decision trees using the minimum description length principle," *Information and Computation*, vol. 80, pp. 227–248, 1989.

[7] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[8] O. T. Yıldız and E. Alpaydın, "Linear discriminant trees," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 19, no. 3, 2005.

[9] W. W. Cohen, "Fast effective rule induction," in *The Twelfth International Conference on Machine Learning*, 1995, pp. 115–123.

[10] T. Ash, "Dynamic node creation in backpropogation networks," *Connection Science*, vol. 1, no. 4, pp. 365–375, 1989.

[11] O. T. Yıldız and E. Alpaydın, "Linear discriminant trees," in *17th International Conference on Machine Learning*. Morgan Kaufmann, 2000, pp. 1175–1182.

[12] ——, "Omnivariate decision trees," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1539–1546, 2001.

[13] H. Guo and S. B. Gelfand, "Classification trees with neural network feature extraction," *IEEE Transactions on Neural Networks*, vol. 3, pp. 923–933, 1992.

[14] C. C. Chang and C. J. Lin, *LIBSVM: a library for support vector machines*, 2001. [Online]. Available: http://www.csie.ntu.edu.tw/∼cjlin/libsvm

[15] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.