Chapel Hill Information Processing Language 2 (CHIPL2): Evaluating Information Systems

http://Chipl.org

Robert M. Losee
http://ils.unc.edu/~losee
(U. of North Carolina-Chapel Hill)

October 19, 2015

Contents

Chapter 1

Introduction

Welcome to CHIPL2. This software allows one to measure amounts of information, as well as to model the information flow in systems, including people and computers, as well as other informative processes. There is an assumption that the reader has handy a copy of *Information from Processes* (`http://InformationFromProcesses.org`) which describes many of the ideas underlying the programs discussed here [4].

CHIPL2 is software built on top of the open source SageMath package. It can be accessed several different ways, as described below. Following chapters describe various measures of information and and informational measures of other phenomena. Part II provides specific application areas, with examples.

Note that there are *example files* included with the CHIPL2 release. These sometimes use commands that are not (yet) documented in the manual below. The names of these files usually indicate the function performed by the example.

1.1 Chapel Hill Information Processing Language (CHIPL2)

CHIPL2 is an information processing language built on top of an open source package, SageMath (`http://SageMath.org`). To execute CHIPL2 programs, one needs access to SageMath, either on the web or by loading the software on to your own personal computer or server.

1.1.1 Running CHIPL2 on the Web

CHIPL2 is most easily executed on the web. First, download the CHIPL2 download files, a ".zip" file from `http://CHIPL.org` into a directory or folder on your system and then unzip the file.

Go to the Sagemath site in the cloud, `https://cloud.sagemath.org` and create an account, using your preferred email address and a password of your choosing.

Remember the password, possibly writing it down, in violation of all the rules. Once logged in, create a “Project” with a name of your choosing. Once you have selected the project and are inside it, click on the “New” below the “Projects” on the upper left and then click on the “Drop Files to Upload” and you will then be able to navigate to the file names on your computer where you downloaded the CHIPL2 files from http://CHIPL.org and upload them to your sagemath cloud account.

This places the core of CHIPL2 into sagemath. To test that things work, note that when the starter Sage worksheet is selected by clicking on the names (click on ”files” if the files are not displayed) or the information icons on the far left, the “Run” option appears in the upperleft. When this happens, click on “Run” for one of these worksheets. Extensions for worksheets are always “.sagews” while the InfoBase that contains CHIPL2 itself is in InfoBase.sage. Try modifying the data in one of the worksheets and then execute it again.

If you are unable to download the files from the CHIPL website, I can email those at an ”.edu” domain a copy of the basic files as an email attachment.

1.1.2 Running SageMath Directly on your Personal Computer

Some operating systems, such as Linux and Mac OSX, can execute SageMath directly. There are instructions on the SageMath website on how to proceed through the install process. **NOTE to Windows Users**: *if you wish to download SageMath, you must skip to the next section on using virtual machines as there is no directly executable version of SageMath for Windows.* To download sagemath, go to `http://SageMath.org` and click on “Download.” Select the operating system you use and download SageMath from the closest website having the software. The software will be in a compressed or disk image form, as is most downloaded software. Using whatever software is appropriate for the software compression format being used, uncompress the software. If you are confused about decompressing the software, you might search for the file extension (e.g., “.lzma” or “.gz”) on the internet to find out how to decompress the SageMath software and possibly how to locate software that will perform the decompression. Next install SageMath, as per the instructions included with your version of the software.

You may then download the InfoBase.sage file containing CHIPL2 from the `http://CHIPL.org` website.

1.1.3 Running SageMath on a Virtual Machine on your Computer

Several personal computer operating systems support virtual machines, such as Oracle VM VirtualBox, executing on the personal computer. Some of the virtual machines are free, while some cost money. If you choose this method of executing SageMath, first install a virtual machine of your choice. Next, go to `http://SageMath.org` and click on the Download option. After selecting the appropriate machine and geographic location from which to download the software, select a version of SageMath that is offered to you and download it. On Windows, it will usually have an ".ova" extension (on the right end of the filename.) Once the .ova file is installed in the virtual machine, you may execute SageMath. Depending on the system you are using, you may directly read the file from your personal computer or you may "cut and paste" the text from your personal computer into the virtual machine when it is executing SageMath.

1.2 Data Manipulation: Entering Data

Data in CHIPL2 is largely entered in the form so taht there is a list of objects, with each list having a label or name, which is entered as a string, that is, in double quotes. A list is a set of strings (or rarely integers or floating point numbers) with each string being in quotes. The beginning and end of the list is marked by matching square braces, so that a list might look like

```
["Alice", "Bob", "Clarice"]
```

The primary means of entering data into CHIPL2 is with the `SetData()` command. Having three friends named Al, Barbara, and Chelsea might be entered as

```
SetData(["Al", "Barbara", "Chelsea"], "friends")
```

One will be able to process this data using statements such as

```
Entropy("friends")
```

which produces the following output: $\frac{\log(3)}{\log(2)}$. One can crudely examine the data in the system by entering

```
PrintAllData()
```

In many cases, one should understand the data that is set as an attribute, with the name of the attribute and the list of values for the attribute. If one enters data in case format, that is, one enters all the attribute values for a given case, such as a book and its subject headings, or a person and their information in a database, referred to by database specialists as a tuple, one may wish to transpose the information from a number of cases to produce the attribute-organized material. For example, the commands

```
SetData(["feature1","feature2","feature4"],"book1")
SetData(["feature1","feature3","feature4"],"book2")
SetData(["feature2","feature3","feature5"],"book3")
CasesToAttributes(["book1","book2"],["feature2","feature3"],[],
     useAllFeatures=False,binaryFeatureValues=True)
```

will create and execute `SetData` commands for each of the features or attribute values in each of the three books. The lists that are set by the `SetData()` command wil have either the value of the name of the feature or will have the value "Null."

The `CasesToAttributes()` command produces attribute lists for each of the features, as specified in the `CaseToAttribues()` command. The first operand of "book1" and "book2" specifies the cases that are to be used. The second operand specifies that features that are to be used in the production of attributes. The value "useAllFeatures=False" later in the command means that the only the features in the second operand are to be used, while when "useAllFeatures" is "True" all features in the cases in the first operand are used. The third operand specifies some attribute names that are used to overide the attribute values specified in the second operand. In this case, the second operand specifies the features that are inclluded, while the third operand is the name of the operand. A second tag, "binaryFeatureValues=True" specifies that the features that are included are used or the value "Null" is used when the feature is absent.

One can exercise control by proving a list of the features that are to be used in the second operand with the names for the attributes being provided by the third operand, e.g.,

```
SetData(["feature1","feature2","feature4"],"book1")
SetData(["feature1","feature3","feature4"],"book2")
SetData(["feature2","feature3","feature5"],"book3")
CasesToAttributes(["book1","book2"],["feature2","feature3"],["NAME2","NAME3"],
      useAllFeatures=False,binaryFeatureValues=True)
```

which will select only features 2 and 3 when creating attribute sets. Thus, if one only used this second `CasesToAttributes()`, one would not be able to calculate the entropy for features 1 or 5 but could calculate the entropy for features 2 (1.00 bits) or 3 (1.50 bits.) Note that to see the output of this, as well as many other commands, one may used the

```
PrintAllData()
```

command.

1.2.1 DataSets and Joins

A dataset as an entity can be established by using the

```
SetDataSet(dataSetName, listOfCaseNames, listOfAttributeNames)
```

command. Thus one might see

```
    SetDataSet("myDataSet",["case1","case2","case3"],["attribute1","attribute
```

Join Cases may be joined together, so that, for example, a dataset with names and phone numbers may be joined with another dataset with names and addresses. The following options must be specified for the `CaseToAttributes()` commands: useAllFeatures=False and binaryFeatureValues=False when these attributes are to be used for database applications and the `JoinCases()` command. The command itself

```
JoinCases(dataSetName1,dataSetName2,newDataSetName,
    joinAttributeName1,joinAttributeName2)
```

Assume there are two pre-existing datasets, phoneData with two attributes, personalName and personalPhone, and the dataset addressData, with attributes personalNameAddress and personalAddress. These may be joined together to produced a new dataset with the name addressBook as

```
    SetData(["Bob","123-4567"],"Bobphone")
    SetData(["Al","987-9876"],"Alphone")
    SetData(["Bob","8 Hilltop Circle"],"BobAddress")
    SetData(["Al","106 Lakewood Lane"],"AlAddress")
    CasesToAttributes(["Bobphone","Alphone"],[],
           ["persName","persPhone"],
           useAllFeatures=False,binaryFeatureValues=False)
    CasesToAttributes(["BobAddress","AlAddress"],[],
           ["persAddrName","persAddress"],
           useAllFeatures=False,binaryFeatureValues=False)
    SetDataSet("EveryonePhone",["Bobphone","Alphone"],
           ["persName","persPhone"] )
    SetDataSet("EveryoneAddress",["BobAddress","AlAddress"],
           ["persAddrName","persAddress"])
    JoinCases("EveryonePhone","EveryoneAddress","CombinedDB",
           "persName","persAddrName")
```

To see the output of the `JoinCases()` command, the `PrintAllData()` command may be used to see what is produced.

1.3 Processing Multiple Data Items

In many cases, it is desirable to apply a single function to pairs of items in a range of data items. This can be done in CHIPL2 with the

```
ProcessListOfCases("f", "object1", "object2")
```

command. This can be used to apply function ”f” to the two objects specified. In circumstances where either or both objects is a list, the data objects associated with each name in the list will be used.

Using the command

```
ProcessListOfCases("Similarity", ["data1", "data2", "data3"],
   typeOfProcessing="matchMToN")
```

applies the `Similarity()` function to each pair of variables in the list of data items. There are a number of options for the typeOfProcessing variable: `"matchMToN"`, which applies the function to each item in the data list with each other item, `"matchNTo1"`, which applies the function to each item in the first datalist with the next (third) operand, which is expected to be a single data object's name, `"match1To1"`, which applies the function to both the second and the third operand (this options is usually more complex than is necessary, since one can usually just apply the function, e.g., `"Similarity()"`, to the two data items themselves rather than using the `ProcessListOfCases()` function. Another `typeOfProcessing` option is `"nextInList"`, which takes the second operand, the first list of data objects, and applies the function between the first data object and the second, the second and the third, and generally the nth to the $(n+1)$th item, and, with a list of m data objects, object m is compared in a wrap-around operation to the first object.

Functions used in the `ProcessListOfCases()` often accept parameters. For example, the `Similarity()` function accepts the type of function that might be desired: simple match, dice, jacquard, or information theoretic. One can pass values to the function for parameters with an ” = ” value by assigning values to the parameter1 or parameter2 values. For example, when using the `Similarity()` function, one might set `parameter1 = "similarityType = 'simpleMatch'"`. This has the effect of passing the `"similarityType = 'simpleMatch'"` string to the `Similarity()` function, if that is the function passed in the `ProcessListOfCases()` function. Note that the `simpleMatch` option needs to be placed in single quotes, so that when a `"similarityType = 'simpleMatch'"` operand is passed to the `Similarity()` function, the proper `similarityType` is selected. However, using `parameter2 = "compareSimilarlyStructuredVectors = False"` in the `ProcessListOfCases()` will result in the `compareSimilarlyStructuredVectors` parameter passed to be set to `False`. Note that `False` and `True` are never included in quotes, unless they are part of a larger expression, in which case the larger expression may be in quotes.

As an example, the following code produces a “dictionary” containing the names for the objects (or pairs of objects) as well as the returned values:

```
SetData(["feature1","feature2","feature4"],
    "book1")

SetData(["feature1","feature3","feature4"],
    "book2")

SetData(["feature2","feature3","feature5"],
     "book3")

SetData(["feature2","feature3","feature5",
     "feature6"],"book4")

ProcessListOfCases("Similarity",

               ["book1","book2","book3","book4"],

               typeOfProcessing = "nextInList",

               parameter1 = "similarityType='simpleMatch'",

               parameter2 = "compareSimilarlyStructuredVectors=False")
```

This will compare `book1` with `book2`, `book2` to `book3`, `book3` to `book4`, and `book4` to `book1`.

Assigning the value of this function to a variable and then printing out the variable as follows will produce a sorted list of values, along with the associated labels:

```
x = ProcessListOfCases(..........)

PrintByValue(x,descending=True)
```

1.4 Miscellaneous Functions

1.4.1 Random Operations: Making-Up Data

When modeling systems, it may be desirable to produce random numbers so that the information in a system can be studied under various conditions.

Random Binary Numbers Binary numbers may be randomly generated with the RandomBinary function, which produces a zero or a one so that an infinite number of calls will have the proportion of ones as p and the proportion of zeroes as $1 - p$. The syntax for producing a random binary number with probability of a one is

$$RandomBinary(p)$$

Executing this as

$$RandomBinary(0.5)$$

will produce a random binary values each time it is executed such as $1, 0, 1, 0, 0$

Random Integers Random integers can be generated so that a range of numbers may be generated from $first$ to $last$ inclusive with

$$RandomInteger(first, last)$$

Generating random numbers from 0 to 4, inclusive, would be written as

$$RandomInteger(0, 4)$$

which might produce results such as $3, 4, 0, 2, 0, 0$.

1.4.2 Hierarchies of Informative Processes

Hierarchies of processes may be implemented using a stack of processes. When I talk with someone, my "mind" and its cognitive components are feeding my natural language processing component in my brain and it in turn controls my vocal chords. A listener hears the vibration produced by my vocal chords with their ear, which sends signals to the natural language processing component, which then sends the message to the listener's mind.

As an example of stacked processes producing a hierarchy, consider

```
lRArray=[]
for i in range(10):
    lst = [["randmod",1,"randmod",1], \
       ["randmod",1,"randmod",1,2], \
       ["randmod",1,"randmod",1]]
    x = StackOfProcesses(lst,2014,lRArray)
    printStackOfProcesses(lst,lRArray)
```

which produces

```
---------------                            ----------------
|  randmod    | >>   MIM= 0.00 bits   >> |  randmod     |
|             | >>  Equiv= 2.12 bits  >> |              |
---------------                            ----------------
```

```
        V                                           ^
        V                                           ^
---------------                           ----------------
|  randmod    | >>   MIM= 0.47 bits    >> |  randmod     |
|             | >>  Equiv= 1.29 bits   >> |              |
---------------                           ----------------
        V                                           ^
        V                                           ^
---------------                           ----------------
|  randmod    | >>   MIM= 0.47 bits    >> |  randmod     |
|             | >>  Equiv= 1.29 bits   >> |              |
---------------                           ----------------
```

The list (lst) created is of a set of sublists, with each sublist containing the process name (randmod, which modifies the input with low probability) and the value passed to it for the input process, that feeds a process below it, while the second process and its input represents the process that accepts information moving up toward the output (on the right in most cases.) An optional fifth argument in each sublist represents the number of possible ”feet” that may come out of the input and output processes at this level. If no fifth argument is provided, the default number of feet is 1. The printing process only displays the left most tower of processes and the rightmost. When multiple feet are produced for a given level of processes, the internal processes are not shown due to the graphic complexity but are processed and included in the numeric values given.

The game of “telephone” may be modeled using stacks of processes, one connected to another. A simple approach to this would be to model 20 feet, each with a slight probability of noise being produced in the transmitted signal, while the process above this would be an accurate transmitter of information. Thus, the only errors would occur due to people talking to someone else where the communication was misunderstood, and, at the same time, there are no errors being introduced at higher levels.

1.4.3 Miscellaneous Commands

String to List The string to list function is used for accepting strings

`StringToList(string)`

This command will take the string “This is a sample string” and the command

```
    StringToList("This is a sample string")
```

produces the following output:

```
[‘‘This", ‘‘is", ‘‘a", ‘‘sample", ‘‘string"].
```

Generating Sequences of Labels The command

```
LabelToList(labelprefix,firstvalueofsuffix,lastvalueofsuffix)
```

is used to produce sets of labels or strings. Thus the command

```
    LabelToList("Booktitle",1,5)
```

will produce the following output: [“Booktitle1”, “Booktitle2”, “Booktitle3”, “Booktitle4”, “Booktitle5”], which can be used in situations where a list of case names or attribute names is needed.

If one wishes to generate a list of data names that have already been produced, one might use

```
SetData(["1","2"],"Abcde")
SetData(["3","4","5","Abc")
SetData(["6","7"],"Xyz")
LabelToList("Abc",wildcard=True)
```

which produces a list of all the names starting with the string ”Abc”: [”Abcde”, ”Abc”]

Debugging Programs may be debugged using two helpful commands. using

```
    PrintAllData()
```

will produce a display of all attributes that have been established with a SetData() command. The

```
    ClearAllData()
```

command will remove all existing cases and attributes.

Chapter 2

Self-Information

2.1 Self-Information

Shannon suggested that the amount of information in learning about the occurrence of a single event is inversely related to the probability of the event. More formally, the amount of information $I()$ associated with event x is

$$I(x) = -\log \Pr(x) = \log \left(\frac{1}{\Pr(x)} \right).$$

The amount of self-information[1] for the probability of $p = 0.10$ is 3.3 bits. When the probability $p = 0.5$ the self-information is 1.0 bit, what one derives with the probability that a coin lands as "heads" with a coin toss with a fair, two sided coin. When the probability $p = 0.9999$ the information becomes 0.00014 bits.

The request for self-information is expressed in Chapel Hill Information Processing Language 2 as

```
SelfInformation(0.5)
```

which produces $\frac{0.693147180559945}{\log(2)}$ or 1.0.

As the probability increases, moving to the right in the graph below, the amount of information decreases, moving downward.

[1] The `SelfInformation()` command and the `Information()` command operate identically.

Chapter 3

Entropy

Entropy is the average information received over the uncertainty in the range of occuring values for the variable in question. With the entropy $H()$ for random variable X computed as

$$H(X) = \sum_i \Pr(x_i) \log(\frac{1}{\Pr(x_i)})$$

the entropy of random variable X is the information associated with each of the x_i values, weighted by the probability of that value. The entropy for a binary variable with a given probability peaks at the probability of 0.5.

We can examine the practical application of entropy with two datasets:

```
SetData([0, 0, 0, 1, 1, 1], 'data1')
SetData([0, 0, 1, 1, 1, 1], 'data2')
```

The entropy of the first set of values or attributes (set 'data1') may be computed as

```
Entropy('data1')
```

or 1 bit. The entropy for the second set, 'data2', is

```
Entropy('data2')
```

producing 0.92 bits. Dataset 'data1' has three zeros and three ones, and thus has the entropy of 1.0 bit, the same amount of information as would exist with the toss of a coin. Dataset 'data2' has more ones than zeroes and thus has less average information, less uncertainty, per symbol.

Expected Values The average of a set of values may be computed using the traditional methods taught to most school children of adding the numbers together and then dividing by the number of numbers, thus, for the list of values or random variable X with n values,

$$Avg(X) = \frac{\sum x_i}{n}. \tag{3.1}$$

For the set of example numbers $(1, 2, 3, 3)$, this produces $(1+2+3+3)/4 = \frac{9}{4}$. An alternative approach, used in computing expected values, is to weight each of the values by its probability of occurrences and then adding them. Thus, one might compute

$$Avg(X) = \sum_i \Pr(x_i)x_i \tag{3.2}$$

so that the following is computed: $1/4 * 1 + 1/4 * 2 + 1/2 * 3 = \frac{9}{4}$

This method of computing the average was used in computing the entropy above as the average information.

3.0.1 Joint Probability

The *joint probability* of two random variables or datasets is computed from treating the items in both datasets as pairs of items. With the same two datasets,

```
SetData([0, 0, 0, 1, 1, 1], 'data1')
SetData([0, 0, 1, 1, 1, 1], 'data2')
```

we may compute the probability of the pair of values 0 and 0 occurring in both the first and second datasets above as

```
JointProbability(0,0,'data1','data2')
```

which produces $\frac{1}{3}$.

This may be viewed as the average percent of time that a particular set of values occurs, across the two datasets.

3.0.2 Assumptions of Statistical Independence

Joint probabilities are often estimated as though the probability of each item is statistically independent of the other value. The probability of an entity in dataset $'data1'$ above is a 0 is $\frac{1}{2}$ and the probability that dataset $'data2'$ has a 1 is $\frac{2}{3}$. The joint probability of these two might be estimated by multiplying these two together, thus $\frac{1}{2} \times \frac{2}{3} = \frac{1}{3}$ Note that this differs from the correct computation of the joint probability of $\frac{1}{6}$, which does not assume statistical independence.

The estimate assuming independence is often close to the correct value and, with estimated values sometimes being too large and sometimes being too small, it is a reasonable estimation technique to use, although using the correct computation and value is always better if the accurate result is needed than using the approximating estimate. The so-called *curse of dimensionality* suggests that it is often very difficult to estimate complex joint probabilities with large numbers of variables, such as the probability that one is 23 years and 1 day old, male, is wearing striped socks, has a forest green shirt, and has a birthmark on the left elbow. It might be difficult to find enough people like this to accurately estimate this joint probability, while it might be easier to find the probability of each variable separately and then multiply them together.

3.0.3 Joint Entropy

The joint entropy of the two datasets above represents the joint information of the values in the two datasets, computed over both datasets. Computed as

```
JointEntropyXAndY('data1','data2')
```

for datasets $'data1'$ and $'data2'$, the joint entropy[1] is $\frac{\log(6)}{6\ \log(2)} + \frac{\log(3)}{3\ \log(2)} + \frac{1}{2} = 1.46$ bits. Note that the entropy for the first set, taken by itself, is

```
Entropy('data1')
```

or 1.0 bits, while the entropy for the second set, taken by itself, is

```
Entropy('data2')
```

with 0.92 bits. The joint entropy of 'data1' and 'data2' is different due to the dependencies incorported into joint probabilities and joint entropy.

[1]The JointEntropyXAndY() command and the JointEntropy() commands are identical.

Chapter 4

Conditional Entropy and Equivocation

The conditional entropy[1] $H(X|Y)$ is the entropy of one random variable given another random variable. This view of conditional entropy is useful when examining any cause and effect situation, whether a communication channel, where we are concerned with the amount of entropy at the output, given the entropy at the input, or a simple linkage, although these can all be modeled as a channel. The vertical line in $H(X|Y)$, $|$, can be read as the English language term "given." The entropy, the average information received about the values in the first dataset, given the second dataset, is computed as

```
    ConditionalEntropyXGivenY('data1','data2')
```

$= 0.541$.

Equivocation is another term for conditional entropy, emphasizing the amount of information needed (or randomness that exists) about one random variable when one knows a second random variable. It may be viewed as a measure of the level of *ambiguity* in the first random variable. As Roman notes, considering a channel that moves from X to Y, "$H(X|Y)$ is the amount of information about X that doesn't make it through the channel to Y [5, p. 73]. Conditional entropy or equivocation can be treated as a measure of information loss between *any cause and effect situation*, such as a channel.

There is a certain amount of randomness that already exists about dataset $'data1'$, computed as the entropy of this dataset,

```
    Entropy('data1')
```

$= 1$. The conditional entropy is computed as the joint entropy of the two datasets minus the entropy of the second set. Thus,

[1]The `ConditionalEntropyXAndY()` command and the `ConditionalEntropy()` command operate identically.

```
JointEntropyXAndY('data1','data2') - Entropy('data2')
```

or 1.46 bits minus 0.918 bits, which equals

```
ConditionalEntropyXGivenY('data1','data2')
```

or 0.541 bits.

Similar Datasets Consider two datasets that are very similar, a modification of the earlier datasets, such as

```
SetData([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], 'data3')
SetData([0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1], 'data4')
```

In this situation, the conditional entropy

```
ConditionalEntropyXGivenY('data3','data4')
```

= 0.345 bits. There is less randomness in the output with these datasets, compared to the earlier data, because the input is a much stronger predictor of the output.

Independent Datasets Consider two datasets where the value of one provides little information about the other, such as

```
SetData([0, 1, 0, 1], 'inddata1')
SetData([1, 1, 0, 0], 'inddata2')
```

In this situation, the conditional entropy

```
ConditionalEntropyXGivenY('inddata1','inddata2')
```

= 1.00 bits, the largest amount of amibiguity that can be obtained with binary data. This is computed as the joint entropy of the two minus the entropy of the second. Thus,

```
JointEntropyXAndY('inddata1','inddata2') - Entropy('inddata2')
```

or 2.00 bits minus 1.00 bits, which equals

```
ConditionalEntropyXGivenY('inddata1','inddata2')
```

or 1.00 bits.

There is more randomness in the output, because the input is a non-predictor of the output.

Dependent Datasets Consider two datasets where the value of one provides all possible information about the other, such as

```
SetData([0, 0, 1, 1], 'depdata1')
SetData([1, 1, 0, 0], 'depdata2')
```

In this situation, the conditional entropy

```
ConditionalEntropyXGivenY('depdata1','depdata2')
```

= 0.000 bits, the smallest amout of ambiguity or equivocation that can occur with binary data. This is computed as the joint entropy of the two minus the entropy of the second. Thus,

```
JointEntropyXAndY('depdata1','depdata2') - Entropy('depdata2')
```

or 1.00 bits minus 1.00 bits, which equals

```
ConditionalEntropyXGivenY('depdata1','depdata2')
```

or 0.000 bits.

There is less randomness in the output than with the independent data above because the input is a full predictor of the output.

Chapter 5

Relationships between Informative Events

5.1 Mutual Information

The mutual information[1] between two random variables is the amount of information that one provides about the other. Sometimes the information that two random variables provide about each other is referred to as *average mutual information*, but we follow the more common tradition of referring to this as *mutual information*.

Independent Datasets If we examine the two independent datasets above, 'inddata1' and 'inddata2', we find that

```
MutualInformationXAndY('inddata1','inddata2')
```

0.000 bits. As one would expect, there is no information in one variable about the other when the datasets are independent.

Dependent Datasets However, if we examine the two dependent datasets above, we find that

```
MutualInformationXAndY('depdata1','depdata2')
```

1.00 bits, the largest amout information that a binary variable can have about another binary variable.

The mutual information between two variables may be treated as the entropy of one variable minus the conditional entropy of this variable given the other variable. Thus, we find that

[1]The `MutualInformationXAndY()` command and `MutualInformation()` command operate identically.

```
MutualInformationXAndY('inddata1','inddata2')
```

may be evaluated as

```
Entropy('inddata1') - ConditionalEntropyXGivenY('inddata1','inddata2')
```

or 0.000 = 1.00 - 1.00

5.1.1 Related Measures

A normalized form of mutual information [6]

```
NormalizedMutualInformationXAndY("indata1', 'indata2', numeric=True)
```

The Entropy Correlation Coefficient [1] also serves as a measure of mutual information:

```
EntropyCorrelationCoefficient('indata1', 'indata2', numeric=True)
```

5.2 Information Distance and Similarity

Information distance, the dissimilarity of two informational entities, may be computed as

```
SetData(["A","A","A","B","B"],"data1")
SetData(["A","B","B","B","B"],"data2")
Distance("data1","data2")
```

which equals 0.823934816631.

Similarly, the informational similarity of these two entities is

```
Similarity("data1","data2")
```

which equals 0.176065183369.

The similarity and distance between two information entities is assumed here to be between entities of equal length or size. Given some caveats, the similarity is “the number of bits of information that is shared between the two strings per bit of informaiton of the string with the most information” [3, p. 3254]. This similarity serves as a scaled value, from 0 to 1, representing the number of bits of information in one entity per bit in the other.

Part I

Applications

Chapter 6

Data in Databases

Data may be stored in a variety of forms. One popular model for storing data is the notion of a table, with columns for *attributes.* Each row represents a *tuple*, a specific set of relationships, while the table as a whole is often referred to as a *relation.*

An example table is provided here:

Name	Preference	Address	Department
Al	Y	123 Shady Lane	Accounting
Bob	Y	111 Hilltop Circle	Accounting
Caitlyn	N	111 Hilltop Circle	Accounting
Debra	N	53 Westbury Drive	Payroll

This contains data with four tuples, each one describing a relationship between four attributes: a personal name, whether one prefers ice cream to candy, a home address, and the department in which they work.

This data may be coded for use in sagemath:

```
SetData(['Al', 'Bob', 'Caitlyn', 'Debra'], 'FirstName')
SetData(['Y', 'Y', 'N', 'N'], 'Preference')
SetData(['123 Shady Lane','111 Hilltop Circle',
'111 Hilltop Circle', '230 Westbury Drive'], 'Address')
SetData(['Accounting', 'Accounting', 'Accounting', 'Payroll'],
'Department')
```

Each attribute in the table is entered as a set of data values.

6.0.1 The Entropy in an Attribute

Each column in the table contains a set of data values. The average information in receiving an attribute, the entropy of the attribute, may be computed for the FirstName as

```
    Entropy('FirstName')
```

which equals $\frac{\log(4)}{\log(2)} = 2.00$ bits. There are 4 FirstNames, and receiving 2 bits of information on average about which FirstName is being used could be understood as one bit indicating whether the FirstName is in the first two FirstNames or the second two FirstNames, and the second bit indicating whether the FirstName is the first or the second from the previously chosen group. The attribute Preference has only two equally possible values, and thus has the

```
    Entropy('Preference')
```

which equals 1.00 bits.

Course ID	Student Name	Instructor Name
001	Allan	Dr. No
002	Barbara	Prof. Smith
001	Chuck	Dr. No
003	Debra	Prof. Jones
004	Ed	Prof. Green
002	Fran	Prof. Smith

Table 6.1: Students in courses and their instructors.

6.0.2 Composition of a Database

Consider the data in Table 6.1. This may be represented as

```
SetData(['001','002','001','003','004','002'],'CourseID')
SetData(['Allan','Barbara','Chuck','Debra','Ed','Fran'],
        'StudentName')
SetData(['Dr. No','Prof. Smith','Dr. No','Prof. Jones',
        'Prof. Green','Prof. Smith'], 'InstructorName')
```

The value of each attribute (column) in the database may be computed as the entropy of that attribute, such as the entropy of the StudentName attribute:

```
Entropy('StudentName',numeric=True)
```

which is 2.58 bits. The joint entropy of all three attributes, taken together, is $\frac{\log(6)}{\log(2)}$ or 2.58 bits.

This database might be decomposed into two database, one containing the CourseID and the StudentName attributes, and the other database containing the CourseID and the InstructorName. This decomposition has the the advantage that if a particular student adds or drop the course, this has no impact on the information available about the instructor for the course. Conversely, if the instructor is changed, this has no impact on the students presence or absence from the course. Referred to as *insertion anomolies* and *deletion anomalies*, the addition or deletion of data on one topic (who teaches a course or who is enrolled in a class) should have no impact on other facts. By structuring datasets appropriately, the databases can be said to be normalized, with varying degrees of normalization being discussed in database and other texts [4, ???].

6.0.3 Functional Dependence

Tony Lee has suggested several problems as illustrative of how information theory may be applied to databases and to their decomposition. All are in Lee's classic articles [2], but he, in turn, used some files from other sources. Lee's Table 1 provides the following data:

```
SetData(['Smith','Smith','Jones','Jones'],'Student')
SetData(['Math','Physics','Math','Physics'],'Course')
SetData(['Prof. White','Prof. Green','Prof. White',
     'Prof. Brown'],'Teacher')
```

Given this data, one may compute the entropy of the attribute Teacher as: $\frac{\log(4)}{2\log(2)} + \frac{1}{2} = 1.50$ and the attributes of the Teacher and Course combined as $\frac{\log(4)}{2\log(2)} + \frac{1}{2} = 1.50$ bits.

Both of these entropy calculations produe the same result. When X completely determines Y, denoted as $X \longrightarrow Y$, then $H(X) = H(X,Y)$. This is referrred to as a *functional dependence.* The relationship Teacher $\longrightarrow$ Course is a functional dependence that Course has on Teacher. This occurs because the value of the variable Course is completely dependent on the value of the variable Teacher. Thus, the variable Teacher contains all the information that might exist between Teacher and another variable which it completely controls, such as Course.

It should be noted that the entropy value may be larger for the key than for an attribute that it controls, or the key may have less entropy than the attributes that it controls. Consider, for example, the names of readers of this material as a key with the binary attribute 'HumanYesOrNo'. In this case the names will vary widely, with the key thus having a relatively high entropy value, while the 'HumanYesOrNo' variable will likely have a low entropy, with all or almost all readers being human. In a different case, a family's listing of members's names and each item of clothing that they own will have relatively few family member's names but a much larger number of items of clothing. With the names as the key, the entropy of the key will be much lower than the entropy for the controled clothing attribute.

6.0.4 Decomposition of a Database

One finds the following data can be used to illustrate decomposition of a database:

```
SetData(['Hilbert','Hilbert','Gauss','Gauss','Gauss','Gauss',
       'Pythagoras','Pythagoras'],'Employee')
SetData(['Hubert','Hubert','Gwendolyn','Gwendolyn','Greta','Greta',
       'Peter','Peter'],'Child')
SetData(['35K','40K','40K','50K','40K','50K','15K','20K'],'Salary')
SetData([1975,1976,1975,1976,1975,1976,1075,1976],'Year')
```

The Entropy of the full database, the joint entropy of the attributes Employee, Child, Salary, and Year produces the following: $\frac{\log(8)}{\log(2)} = 3.00$ bits.

Assume that the database was split (decomposed) into two smaller databases, one composed of the Employee and Child attributes, and the other with the Employee, Salary, and Year attributes. Ideally, no information is lost with this decomposition. Such a lossless decomposition results in a higher normal form for the resulting databases, with fewer insertion and deletion anomalies.

The entropy of Employee and Child is $\frac{\log(4)}{\log(2)} = 2.00$ bits. The joint entropy of the Employee, Salary, and Year combined database is $\frac{\log(8)}{2\log(2)} + \frac{\log(4)}{2\log(2)} = 2.50$ bits, while the entropy of the Employee, which occurs in both and thus needs to be removed from one of these when the are combined to produced the Employee, Child, Salary, and Year database is $\frac{\log(4)}{2\log(2)} + \frac{1}{2} = 1.50$ bits. We find that $2 + 2.5 - 1.5 = 3$ bits, the entropy of the full table. This suggests that the if we join the two sub-tables, the 'Employee' and 'Child' sub-table with the sub-table of 'Employee', 'Salary', and 'Year', based on matching 'Employee' values, produces the original table.

Theory of Decomposition of Tables

Further dependencies may be illustrated by noting different ways that groups of attributes may be removed from a table to provide separate tables. *Multivalued dependencies* are one approach, examining *decomposition pairs* as a useful way to view groups of attributes. If there are three attributes, X, Y, and Z, and they are all attributes of a particular database, then if X serves as a key, $I(Y;Z|X) \geq 0$, meaning that given X (the key), then the combined amount of information about Y and Z is greater than or equal to 0. This inequality is equivalent to

$$H(X,Y,Z) \leq H(X,Y) + H(X,Z) - H(X). \tag{6.1}$$

We note that equality holds when X is the key for a database composed of three attributes, X, Y, and Z such that the database can be appropriately decomposed into two databases with X as the key for both. Thus, we find that

$$H(X,Y,Z) = H(X,Y) + H(X,Z) - H(X). \tag{6.2}$$

In this case, the two databases can be joined back together to produce the original database. Using Equation 6.1, we find that the inequality holds,

$$H(X,Y,Z) < H(X,Y) + H(X,Z) - H(X), \tag{6.3}$$

which this suggests that the decomposition is such that the decomposed databases contain more information or uncertainty than the full single database.

If we were to decompose the full database into a different set of databases, we might find that the two combined subdatabases do not have the same amount of information as the original database. The entropy of the full Employee, Child, Salary, and Year database has this amount of data: $\frac{\log(8)}{\log(2)} = 3.00$ bits.

If the key is treated as the salary, then we might decompose the full database into subdatabases with Salary as the key. The entropy of a Salary and Year subdatabase is $\frac{\log(8)}{2\log(2)} + \frac{\log(4)}{2\log(2)} = 2.50$ bits. If we add to this the entropy of Salary, Child, and Employee, $\frac{\log(8)}{\log(2)} = 3.00$ bits and subtract the entropy of the salary (which occurs in both of the subdatabases) $\frac{3\log(8)}{8\log(2)} + \frac{\log(4)}{4\log(2)} + \frac{3\log\left(\frac{8}{3}\right)}{8\log(2)} = 2.16$ bits, one can see that this value exceeds the entropy of the original full database: $\frac{\log(8)}{\log(2)} = 3.00$ bits. Simplifying this, $2.50 + 3.00 - 2.16 = 3.34 > 3.00$ bits. By applying Equation 6.3, we find that the 'Salary' variable is not a suitable key upon which the other factors in the subdatabases are dependent and thus the sub-tables ('Salary' and Year') and ('Salary', 'Child', 'Employee') cannot be joined together using the 'Salary' key.

6.0.5 Pairwise Decomposition

A set of relationships that are informally independent but can be joined together is referred to as a *join dependency.* It may be helpful to view a possible hieraracy of relatioships by noting that they are acyclic, that one does not move from one relationship to another relationship to a third relationship and then back to the original relationship, which would be a set of cyclic relationships. Sets of pairwise dependencies can be found in the following data:

```
SetData(['CS','CS','CS','EE','EE','EE',
       'EE','EE'],'A') # Dept (A)
SetData(['Taylor','Taylor','Taylor','Smith',
      'Smith','Smith','Smith','Smith'], 'B')  # Researcher (B)
SetData(['DB','DB','DB','DB','DB','DB',
      'Sig Proc','Sig Proc'],'C') # Project (C)
SetData(['PDP11','VAX11','VAX11','PDP11',
      'VAX11','VAX11','VAX11','VAX11'], 'D')  #  Computer (D)
SetData(['C','PL1','Fortran','C','PL1','Fortran',
      'PL1','Fortran'],'E') # Lang (E)
```

The entropy of these pairs of variables (B is dependent on A, C on B, D on C, and E on D) may be computed as the joint entropies, minus the single variable entropies that occur twice in the set of joint entropies:

$$\text{Entropy of A,B: } \frac{3\log\left(\frac{8}{3}\right)}{8\log(2)} + \frac{5\log\left(\frac{8}{5}\right)}{8\log(2)} = 0.954$$
$$+$$
$$\text{Entropy of B,C: } \frac{\log(4)}{4\log(2)} + \frac{3\log\left(\frac{8}{3}\right)}{4\log(2)} = 1.56$$
$$+$$
$$\text{Entropy of C,D: } \frac{\log(4)}{2\log(2)} + \frac{1}{2} = 1.50$$
$$+$$
$$\text{Entropy of D,E: } \frac{\log(4)}{4\log(2)} + \frac{3\log\left(\frac{8}{3}\right)}{4\log(2)} = 1.56$$
$$-$$
$$\text{Entropy of B: } \frac{3\log\left(\frac{8}{3}\right)}{8\log(2)} + \frac{5\log\left(\frac{8}{5}\right)}{8\log(2)} = 0.954$$
$$-$$
$$\text{Entropy of C: } \frac{\log(4)}{4\log(2)} + \frac{3\log\left(\frac{4}{3}\right)}{4\log(2)} = 0.811$$
$$-$$
$$\text{Entropy of D: } \frac{\log(4)}{4\log(2)} + \frac{3\log\left(\frac{4}{3}\right)}{4\log(2)} = 0.811$$
$$=$$
$$3.00 \text{ bits}$$

which also equals the joint entropy of the entire database:

Entropy of ABCDE: $\frac{\log(8)}{\log(2)} = 3.00$ bits. Thus each of the pairs described earlier can be joined together to produce the full table.

6.0.6 Failure of Pairs to Form a Decomposition Pair

The following table may be used to example potential problems with decomposition:

```
SetData(['Smith','Smith','Smith','Jones','Jones','Jones',
       'Brown','Brown','Brown','Brown'],'A')  # Agent
SetData(['Ford','Ford','GM','Ford','Ford','Ford','Ford',
       'GM','Ford','Ford'],'C')  # Company
SetData(['car','truck','car','car','car','truck',
       'car','car','car','car'],'P')  # Product
```

Using this data, one can attempt to decompose the table into two smaller tables, both with key C. In this case, the entropies of the components exceed the entropy of the combined table, suggesting that the decomposition is inappropriate:

$$\text{Entropy of AC: } \frac{\log(10)}{5\log(2)} + \frac{\log(5)}{5\log(2)} + \frac{3\log\left(\frac{10}{3}\right)}{5\log(2)} = 2.17$$
$$+$$
$$\text{Entropy of CP: } \frac{2\log(5)}{5\log(2)} + \frac{3\log\left(\frac{5}{3}\right)}{5\log(2)} = 1.37$$
$$-$$
$$\text{Entropy of C: } \frac{\log(5)}{5\log(2)} + \frac{4\log\left(\frac{5}{4}\right)}{5\log(2)} = 0.722$$
$$= 2.82$$
$$>$$
$$\text{Entropy of ACP: } \frac{\log(10)}{2\log(2)} + \frac{\log(5)}{5\log(2)} + \frac{3\log\left(\frac{10}{3}\right)}{10\log(2)} = 2.65$$

Similarly, decomposing with key P results in the components having entropic components that exceeds the joint entroy

$$\text{Entropy of CP: } \frac{2\log(5)}{5\log(2)} + \frac{3\log\left(\frac{5}{3}\right)}{5\log(2)} = 1.37$$
$$+$$
$$\text{Entropy of AP: } \frac{\log(10)}{5\log(2)} + \frac{2\log(5)}{5\log(2)} + \frac{2\log\left(\frac{5}{2}\right)}{5\log(2)} = 2.12$$
$$-$$
$$\text{Entropy of P: } \frac{\log(5)}{5\log(2)} + \frac{4\log\left(\frac{5}{4}\right)}{5\log(2)} = 0.722$$
$$=$$
$$2.77$$
$$>$$
$$\text{Entropy of ACP } \frac{\log(10)}{2\log(2)} + \frac{\log(5)}{5\log(2)} + \frac{3\log\left(\frac{10}{3}\right)}{10\log(2)} = 2.65$$

Thus the pairs mentioned cannot be joined as suggested to produce the full table.

Bibliography

[1] J. Astola and I. Virtanen. A measure of overall statistical dependence based on the entropy concept. In *Proc. Univ. Vaasa. number 91*, 1983.

[2] Tony T. Lee. An information theoretic analysis of relational databases, parts I and II. *IEEE Transactions on Software Engineering*, SE-13(10):1049–1072, October 1987.

[3] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, December 2004.

[4] Robert M. Losee. *Information From Processes*. Springer, New York, 2012. `http://InformationFromProcesses.org`.

[5] Steven Roman. *Coding and Information Theory*. Springer, 1992.

[6] C. Studholme, D. L. G. Hill, and D. J. Hawkes. An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition*, 32:71–86, 1999.

Index