Tabular Datasets

Social-science data gains utility when stored and shared as a dataset with a well-documented structure. This section focuses on one popular dataset structure: a collection of records, where each record is a collection of named fields. Each record effectively associates field identifiers to values.

Abstractly, an associative array is a natural model for a record. One associative array can represent a single record, with the field identifiers as the keys and the associated data values as values. For example, the following three associations could represent three records in a very simple data set.

In[•]:=
```
assoc01 = <|"first" → "John", "last" → "Doe", "h1" → 50, "h2" → 30|>;
assoc02 = <|"first" → "Jane", "last" → "Doe", "h1" → 30, "h2" → 50|>;
assoc03 = <|"first" → "SR", "last" → "Rossi", "h1" → 46, "h2" → 45|>;
```

Create a small dataset by collecting these three associations into a list named **ds**. This list of records is a very simple example of a dataset. Since the record order is irrelevant and there are no duplicate records, this dataset is effectively a set of associations.

In[•]:=
```
rows = {assoc01, assoc02, assoc03};
```

This dataset has a *tabular format* (or more simply, it is a table) because every record has the same fields. When a dataset has a tabular format, the records are often called rows of the dataset. More concretely, imagine a table with field names at the top and associated values in each row, as in a spreadsheet. This allows a compact representation of the same data, as in the following table.

Out[•]=

Table 1: Dataset Displayed as Table

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
SR	Rossi	46	45

An earlier discussion of functions emphasized that an association is essentially an enumerated function. In this sense, a tabular data can be conceptualized as a collection of enumerated functions. WL provides a variety of commands for accessing and manipulating such a dataset.

Filtering and Recoding

From a given tabular dataset, researchers often extract a smaller dataset comprising only those records that satisfy some criterion. This process corresponds to selective set building, also called *filtering*, or *subsetting*. Criterion-based filtering of datasets is a very common need in empirical social science. Given a list of records, the **Select** command readily performs dataset filtering: we just need an appropriate predicate. This predicate is a boolean-valued function that returns **True** iff a record satisfies our criterion and **False** otherwise.

For example, suppose that from this list of associations we wish to keep only those records where the

h1 and **h2** fields sum to less than 90. Begin this task by implementing an appropriate selector. The following example uses the operator form of **Select**, which takes a predicate as its argument.

In[•]:=
```
select01 = Select[#h1 + #h2 < 90 &];
```

Notice how the predicate directly uses the field names. Recall that when associations that have strings as keys, functions operating on these associations can refer by key name to the values. This is the case with the associations in our simple dataset. Now **select01[rows]** will subset the data, as desired. The result is once again a list of associations. In order to produce more visually organized output, stack the selected associations with **Column**.

Out[•]=

Expression	Result
select01[rows] // Column	⟨|first → John, last → Doe, h1 → 50, h2 → 30|⟩ ⟨|first → Jane, last → Doe, h1 → 30, h2 → 50|⟩

Next, consider substitutive set building, which is sometimes called *transformation* or *recoding*. The **Map** command is typically adequate to this need. One may map over the records of a dataset to produce a new dataset by transforming each record. Typically, dataset transformations retain some kind of unique record identifier, precluding the creation of exact duplicates. Since this simple dataset is a list of records, the list index is a unique identifier. The list order is retained when **Map** transforms it.

Once again, functions that transform associations have direct access to the association keys. For example, the following expression defines a function **sub01** that accesses three fields of an association and then creates a new association.

In[•]:=
```
sub01 = <|"first" → #first, "total" → #h1 + #h2|> &;
```

This facility implies that **Map** can recode tabular datasets by using easily understood transformations. Since this dataset is just a list of associations, **Map[sub01,rows]** maps the substitution function across the dataset and thereby produces a transformed list of associations.

Out[•]=

Expression	Result
Map[sub01,rows] // Column	⟨|first → John, total → 80|⟩ ⟨|first → Jane, total → 80|⟩ ⟨|first → SR, total → 91|⟩

Queries for Filtering and Recoding

In database terminology, a selective set building or a substitutive set building performs a *query*, which retrieves information from a dataset. WL correspondingly provides **Query** for query operations. Applying a query to a dataset subsets it or transforms it (or both). The query syntax is extremely flexible, and queries may be complex.

Consider the filtering and recoding examples from the previous subsection. The query to implement the filtering example is **Query[select01]**, while the query to implement the recoding example is **Query[All,sub01]**. Each of these produces a query operator (i.e., a function) that may be applied

to a list of associations to produce the specified filtering or recoding. Applying the first query to the dataset produces the same effects as applying the selector, while applying the second query to the dataset produces the same result as mapping the transformation over the dataset. In either case, there may seem to be little gain from the new syntax. However, queries are readily combined, as follows. The query **Query[select01,sub01]** selects only the records of interest (rather than all of them) and then transforms each record.

Out[•]=

Expression	Result
Query[select01,sub01][rows] // Column	⟨|first → John, total → 80|⟩ ⟨|first → Jane, total → 80|⟩

As a final example of transforming this particular dataset, consider recoding the data to anonymize the scores. Achieve this by a transformation that drops the two name fields from every record. Given the description, this transformation is perhaps most naturally achieved by means of **KeyDrop**. This can be part of a query, but **KeyDrop** also works directly on a dataset.

Out[•]=

Expression	Result
KeyDrop[{"first","last"}][rows] // Column	⟨|h1 → 50, h2 → 30|⟩ ⟨|h1 → 30, h2 → 50|⟩ ⟨|h1 → 46, h2 → 45|⟩

As an alternative approach, first characterize set of records to process (here, all of them), and then list (by name) the columns to retain: **Query[All,{"h1","h2"}][rows]** . Interestingly, it is even possible to use **Part** to select parts of a list of associations: **rows[[All,{"h1","h2"}]]**. Choose the method that seems most communicative for a given problem.

Underpinnings of Query

Query provides a convenient interface for dataset manipulations, and the use of queries is idiomatic when working with datasets. Nevertheless, it can be helpful to recall that **Query** is just a convenient way to apply functions at various levels of the dataset. To reveal the underlying operations, use the **Normal** command. The results for **Map** use its single-argument operator form. The result for an input of two functions is a **RightComposition**. Ordinarily, the first function is applied to the dataset after the second function is mapped across the records.

Out[•]=

Expression	Result
Normal @ Query[f]	f
Normal @ Query[All,g]	Map[g]
Normal @ Query[f,g]	Map[g] /* f

However, there is a remaining subtlety. A **Query** classifies its dataset operations as descending or ascending. Given a sequence of operations in a query, the descending operations are applied in order (at deeper and deeper levels) and then the ascending operations are applied in reverse order at higher and higher levels. Crucially, user-defined functions are ascending, but selection is considered a descending operation. So a selection is applied before the mapping operation. (See the documenta-

tion of **Query** for more details.) This is desirable, since we generally wish to describe a selection in terms of the original dataset.

Out[•]=

Expression	Result
`Normal @ Query[Select@f,g]`	`Select[f]/*Map[g]`

Dataset Objects

This subsection briefly considers a WL data type that can be very similar to a list of associations: the **Dataset**. It provides the same natural representation of tabular datasets, with additional advantages for transformation, selection, and aggregation. (The Wolfram Language Guide entitled Computation with Structured Datasets provides an extended discussion.)

The **Dataset** command can convert a list of associations into a unified structure that is easily queried and manipulated. In a Mathematica notebook such a dataset has a convenient visual display.

Out[•]=

Expression

```
ds=Dataset[{
    assoc01,
    assoc02,
    assoc03
}]
```

Result

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
SR	Rossi	46	45

Such a dataset object is essentially the list of associations, along with a convenient wrapper. In fact, whenever needed, the **Normal** command can retrieve the list of associations from such a dataset. However, dataset objects support a particularly convenient syntax for common data manipulations.

Most often, the creation of a WL dataset begins with an import of data from an external file. The **SemanticImport** command excels at creating data sets from external data files, such as spreadsheet files. In order to produce a self-contained example, the following illustration uses the closely related **SemanticImportString** command, which operates directly on strings. Even without additional clues, a semantic import determines that the string data is in CSV format.

Out[•]=

Expression

```
SemanticImportString[
"first,last,h1,h2
John,Doe,50,30
Jane,Doe,30,50
Signor,Rossi,46,45"]
```

Result

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50
Signor	Rossi	46	45

Filtering and Recoding Datasets

The queries used above on a list of associations work just as well on this dataset. However, when applied to a dataset the result is a dataset, which in a notebook has a nice display. In addition to the

obvious syntax **select01[ds]** query syntax, the dataset object allows **ds[select01]** as an alternative query syntax.

Out[•]=

Expression: ds[select01]

Result

first	last	h1	h2
John	Doe	50	30
Jane	Doe	30	50

This alternative syntax generalizes to other queries. For example, **Query[All,sub01][ds]** can be more compactly written as **ds[All,sub01]**.

Out[•]=

Expression: ds[All,sub01]

Result

first	total
John	80
Jane	80
SR	91

Of course, as before, these queries can be readily combined.

Out[•]=

Expression: ds[select01,sub01]

Result

first	total
John	80
Jane	80

As another example of recoding a dataset, consider the selection of a subset of the fields from every record. First characterize set of records to process (here, all of them), and then list the desired columns (by name). Here instead of applying **Query[All,{"h1","h2"}]** to the dataset, use **KeyTake**.

Out[•]=

Expression: ds[KeyTake[{"h1","h2"}]]

Result

h1	h2
50	30
30	50
46	45

The same strategy can augment records as needed. In this fashion, substitutive set building can add fields. For example, the following function will append to an associations a new key (**"total"**) with a value that is the sum of the values associated to the **"h1"** and **"h2"** keys.

In[•]:=

```
sub02 = Append[#, "total" → #h1 + #h2] &;
```

It is quite possible to map this function across the **ds** dataset object. When recoding datasets, a more idiomatic alternative is to create an equivalent query. The following example again does this by means of the shorthand query syntax; it use **sub02** to append a **"total"** column to the dataset.

Out[•]=

Expression	Result
ds[All,sub02]	

first	last	h1	h2	total
John	Doe	50	30	80
Jane	Doe	30	50	80
SR	Rossi	46	45	91

At this point, it is clear that **Query** is remarkably flexible, but we have only scratched the surface. For example, like any function, a query can apply **LinearModelFit** to the data in a dataset object, although we must first use **Values** to remove the field names. Applying **Values** produces a headerless rectangular dataset, which essentially wraps a rectangular list of lists rather than a list of associations. A simple query can then fit scores on the second homework to scores on the first homework.

Out[•]=

Expression	Result
data=Values@ds[All,{"h1","h2"}]; data[LinearModelFit[⌗, x, x]&]	FittedModel[75.4 – 0.804 x]

Later discussions treat linear regression in more detail. The Wolfram Language *How To* entitled Perform a Linear Regression provides another useful introduction.

Partitioning and Classification

A *partition* of a set *X* is a collection of non-empty, pairwise-disjoint subsets whose union is *X*. Each subset is a *block*. Recall that a collection of sets is *pairwise-disjoint* if the intersection of any pair of the sets is empty, so different blocks of a partition have no elements in common. The simplest interesting partition is a two-block partition. Two-block partitions are naturally formed by dividing set elements into those that satisfy a certain property and those that do not. Whenever selective set building picks a subset of items that satisfy some predicate, one may produce a corresponding partition by additionally creating the complement of that subset.

Since a list without duplicates nicely represents a finite set, partitioning such a list corresponds to partitioning a set. The following example uses **TakeDrop** and **Partition** to produce some arbitrary blocks from a small list of numbers. The basic form of the **TakeDrop** command takes as arguments a list and an initial sequence length. It partitions the list into two sublists: an initial sequence, and the remainder of the list. The second argument can alternatively be a list of two integers, which specify the start index and stop index of a sublist to take. The **Partition** command offers a different approach to partitioning: it can partition a list into sequential sublists of a common length. The following example uses the two-argument form of **Partition**, where the first argument is a list and the second argument is the length of the sublists. In order to avoid discarding terminal elements

when the list is not evenly divisible, use the **UpTo** command in the second argument.

Out[•]=

Expression	Result
n7=Range[7]	{1, 2, 3, 4, 5, 6, 7}
TakeDrop[n7,3]	{{1, 2, 3}, {4, 5, 6, 7}}
TakeDrop[n7,{3,5}]	{{3, 4, 5}, {1, 2, 6, 7}}
Partition[n7,UpTo[3]]	{{1, 2, 3}, {4, 5, 6}, {7}}

This example shows that **TakeDrop** or **Partition** can produce a list of lists that represents a particular set partition. However, data science more commonly needs a partition that is based on properties of the elements. Such a partition is called a categorization or a *classification* of the elements, where the classification typically derives from a classifier function.

Classification with GroupBy

The simplest two-argument form of the **GroupBy** command accepts as arguments a list to be classified and a classifier function. The result is an association from categories in the range of the classifier function to a list of elements for each category that is present. This is a *one-way classification* of the data. For example, a set of households might be classified by household size.

The following functions serve as classifiers on the integers. The first maps each integer to a single classification, either **"d2"** or **"¬d2"**. The second maps each integer to a single classification, either **"d3"** or **"¬d3"**; this classifies any integer as divisible by 3 or not. The third maps each integer to a single classification, either **"d4"** or **"¬d4"**; this classifies any integer as divisible by 4 or not. Each of these classifiers can partition a set of integers into two blocks. These classifications can overlap in various ways, as discussed below.

In[•]:=
```
div2 = item ↦ If[0 == Mod[item, 2], "d2", "¬d2"];
div3 = item ↦ If[0 == Mod[item, 3], "d3", "¬d3"];
div4 = item ↦ If[0 == Mod[item, 4], "d4", "¬d4"];
```

When applied to a set of elements, categorization with **GroupBy** effectively produces the inverse relation of the classifier function on a set. For example, apply the **div2** classifier to the first seven positive integers, as follows.

Out[•]=

Expression	Result
gb2=GroupBy[n7,div2]	⟨| ¬d2 → {1, 3, 5, 7}, d2 → {2, 4, 6} |⟩

In a Mathematica notebook, applying the **Dataset** command to the resulting association produces a convenient tabular display.

Out[•]=

Expression	Result	
ds2 = Dataset[gb2]	¬d2	{1, 3, 5, 7}
	d2	{2, 4, 6}

The next example is almost identical, but it uses the one argument form of **GroupBy**. This produces a grouping operator, which can then be applied to a list. Recall that the at-sign operator is the prefix

notation for function application, so **GroupBy[div3]@n7** is exactly the same as **GroupBy[div3][n7]**.

Out[•]=

Expression	Result	
Dataset[GroupBy[div3] @ n7]	¬d3	{1, 2, 4, 5, 7}
	d3	{3, 6}

Partition Π' is a refinement of partition Π iff every block of Π' is included in a single block of Π. That is, partition blocks can themselves be partitioned, resulting in a refined partition of the original set. A single classification can partition a dataset, and a second classification can then refine this partition.

It is possible to sequentially refine a partition by mapping a second **GroupBy** operator across the values of an initial grouping. However, **GroupBy** supports a simpler syntax: it accepts a list of classifiers that are sequentially applied. The following example illustrates this nested categorization by producing a *two-way classification*. The result is a multilevel association: an association whose values are in turn associations. Once again, the **Dataset** command can produce a convenient tabular display of the result. The outer association keys are used as row headers. When the inner associations share keys, as they do here, the **Dataset** command typically displays these keys as column headers. (Refined control of the display of datasets is beyond the scope of this book, but see below for a few hints.)

Out[•]=

Expression	Result		
ds23 = Dataset[GroupBy[{div2,div3}] @ n7]		¬d3	d3
	¬d2	{1, 5, 7}	{3}
	d2	{2, 4}	{6}

Block Size as a Dataset Query

Social scientists often care about the sizes of the partition blocks, since these are the frequency counts implied by the categorization. A one-way frequency table displays categorical data as a list of categories and their frequency of occurrence in a dataset. Mapping **Length** across a one-way categorization produced by **GroupBy** will produce the information for a one-way frequency table (as an association from categories to counts). However, when working with data set objects, it is more natural to produce the length of each block by means of a query. When constructing the query, use the following reasoning: for each category in a one-way categorization, determine the total number of items in that category. This suggests the query **Query[All,Length]**. Try this out on the previous even-odd classification dataset. In a Mathematica notebook, the result displays as a simple one-way table.

Out[•]=

Expression	Result	
ds2[All,Length]	¬d2	4
	d2	3

The same reasoning applies to the creation of a two-way frequency table, which shows frequency counts for each possible pair of classifications. Recall that a two-way categorization using **GroupBy** creates a multilevel association, which is an association whose values are also associations. A frequency table reports the number of items for each value of the inner association. This suggests the following query.

In[•]:=
```
cts = Query[All, All, Length];
```

Try this out on the previous two-way classification dataset. In a Mathematica notebook, the result displays the element count for each block of the resulting partition of the dataset. It is a frequency table for data described by two categorical variables, which is often called a two-way table, a two-dimensional cross-tabulation, or a contingency table.

Out[•]=

Expression	Result		
ds23 //cts		¬d3	d3
	¬d2	3	1
	d2	2	1

Display Issues

The example above produces a nice cross-tabulation display with little effort. In the presence of empty categories, a nice display involves a bit more work. For example, pick a different secondary classifier: divisibility by 4 instead of divisibility by 3. The result is still a partition of the set, but now nothing is listed for the empty category (odd and divisible by 4). Using the **Dataset** command to produce a convenient categorical display is still acceptable, but the hierarchical display is not as easy to decipher as a two-way table.

Out[•]=

Expression	Result		
ds24=Dataset[GroupBy[{div2,div4}] @ n7]	¬d2	¬d4	{1, 3, 5, 7}
	d2	¬d4	{2, 6}
		d4	{4}

In this case, it is possible to produce a display comparable to the previous two-way table by first providing a default value for every inner group and then updating it based on the data. For example, adjust the counts by mapping an association providing zero as a default value for the divisible-by-four category. (The default values must come first.) Technically the resulting categorization is no longer a true partition, since one category is empty, but the resulting display is much easier to read. The

following example provides only the single default that this particular case requires.

Out[•]=

Expression	Result

```
ds24[cts][All,<|"d4"->0,#|>&]
```

	d4	¬d4
¬d2	0	4
d2	1	2

To understand how this works, peer inside the dataset wrapper with the **Normal** command. This dataset is essentially an association whose values are also associations. Recall that applying **Map** to an association transforms the values, not the keys. For example, the key **"¬d2"** associates to the value **<|"¬div4"→4|>**. Applying the default to this value produces **<|"div4"→0,"¬div4"→4|>**. The example does this, using the specialized query syntax for datasets.

As of Mathematica version 13, this kind of control of dataset display remains tricky. It is affected by inferred type information, which is beyond the scope of this book. Sometimes the above trick fails to produce a two-way display rather than a less readable hierarchical display. An alternative is to use a dataset for recoding, filtering, and aggregation, and then subsequently extract the results for use with **TableForm** or **Grid**.