
Dealing with alternative implementations and approaches

Duncan Temple Lang, University of California at Davis

Table of Contents

Threads, Tasks and Alternatives	1
Alternative Statistical Approaches	4
Threads	4
Tasks	4
Working with Code in a Document	9

Some of this may change slightly as we get more use-cases. Mostly, it will be extended to allow for different scenarios that different people want to express.

Threads, Tasks and Alternatives

Here we talk about different types of code segments that arise when writing a dynamic (and potentially interactive) document. There are several slightly different aspects to consider. The first is that we might have different implementations of the same basic functionality. For example, we might have a naive implementation of an algorithm that we develop first and then a second approach that is more efficient. We may want to present both to the reader, but have the more efficient version be used. We can prohibit the evaluation of the first implementation using `<r:code eval="false">...`, but this is not ideal. We want to allow the code to be available for evaluation, but to explicitly identify the two code segments (or sequence of segments) as being *alternatives* to each other, giving the same outputs. We use the `<altImplementations>` element for this and identify each with its own `<altImplementation>` element. So we might have something like the following:

```
<section>
<title>Implementing random number generators for mixture models</title>

<para>
Here we show how we can implement a random number generator
for a mixture model. We'll show two different approaches.
</para>

<altImplementations>
<para>
We'll focus on a mixture of normals. Each implementation will accept
the number of values to generate, the vector of means and the vector
of SDs of the components, and a vector of probabilities.
(These are bare-bones implementations for illustrating the
<xml:tag>alternative</xml:tag> tag.)
```

</para>

<altImplementation id="A">

The first version handles each value separately,
generating the component from which to sample
and then sampling that.

This is the non-vectorized approach.

```
<r:function><![CDATA[
rmix = rmix1 =
function(n, means, sd = rep(1, length(means)), prob = rep(1, length(means)))
{
  idx = seq(along = means)
  replicate(n, { i = sample(idx, 1, prob = prob)
                 structure(rnorm(1, means[i], sd[i]), names = i)
               })
}
])></r:function>
```

We can test it with

```
<r:test><![CDATA[
x = rmix(10000, c(0, 5, 10))
plot(density(x))
])></r:test>
```

</altImplementation>

<altImplementation>

<para>

A better approach is to sample the components
in one operation and then use the vectorization
of `<r:func>rnorm</r:func>` to sample the values
in a single operation also.

```
<r:function><![CDATA[

rmix = rmix2 =
function(n, means, sd = rep(1, length(means)), prob = rep(1, length(means)))
{
  k = sample(seq(along = means), n, replace = TRUE, prob = prob)

  rnorm(n, means[k], sd[k])
}

])></r:function>
```

Again we can test this using the same code:

```
<r:test><![CDATA[  
x = rmix(10000, c(0, 5, 10))  
plot(density(x))  
]]></r:test>
```

and of course do more extensive tests.

</para>

</altImplementation>

<compare>

<para>

We can compare the results with a Q-Q plot.

```
<r:plot><![CDATA[  
x1 = rmix1(10000, c(0, 5, 10))  
x2 = rmix2(10000, c(0, 5, 10))
```

```
qqplot(x1, x2)
```

```
]]></r:plot>
```

We have to be careful we are comparing values within the same components.

</para>

<para>

We can compare these functions by timing them for different number of observations
We can also look at how this varies with different numbers of components.

```
<r:code><![CDATA[  
  
n = seq(10, length = 50, by = 1000)  
tm1 = sapply(n, function(i) system.time(rmix1(i, means = c(0, 10, 20, 30))))  
tm2 = sapply(n, function(i) system.time(rmix2(i, means = c(0, 10, 20, 30))))  
  
]]></r:code>
```

```
<r:plot><![CDATA[  
matplot(n, cbind(tm1[3,], tm2[3,]))  
]]></r:plot>
```

```
</para>

</compare>

</altImplementations>

</section>
```

Note that we can have arbitrary markup and content within each `<alternativeImplementation>` element and also preceeding, between and following each `<alternativeImplementation>` element. The `<compare>` element allows us to provide some discussion about the alternatives which we can suppress (i.e. discard or omit) in certain views of the document.

Indeed, we can discard different alternatives by specifying which ones to use in the actual "running" of the code within the document. We can give each `<alternativeImplementation>` a tag and specify which to use. By default, we use the last `<alternativeImplementation>` within each `<alternativeImplementations>`.

Alternative Statistical Approaches

In addition to having different implementations of the same programmatic functionality (e.g. implementing a particular algorithmic description or using grid or grz or ggplot2 to create the "same" plot), we also have different approaches to analyzing data. For example, to "fit" a classifier we might use k-nearest neighbors or a classification tree or an SVM. Ideally, what we end up with in each approach is one or more R objects that can be used in subsequent computations, regardless of which approach was used, e.g. via `predict()` and `update()`.

To identify different approaches we use `<altApproach>`. We group these within an `<altApproaches>` element.

Threads

Once a document contains alternatives of any type, we have an issue of identifying which pieces of code across different tasks correspond to different alternatives and which sequence of code we want to run. We use a `thread` attribute on an `<altApproach>` (or potentially on a `<altImplementation>` or `<r:code>`, `<r:plot>`, etc. elements) to connect the different pieces together. This identifier can be thought of as a piece of string that defines a path through the document and when we pull it, only the relevant pieces come from the document. We can have different threads and refer to them by separate names. When we run the code or simply project the document to a particular view, these threads are treated separately, some being discarded or within some viewers, selectable by the reader to explore a different path through the document.

Tasks

Often, a data analysis will have a series of tasks, e.g. access files from a repository and perform transformations and filtering; read the data into R; exploratory data analysis, modeling; presenting results. Within some of these steps we might create a collection of derived variables as a task; the EDA can be divided up

into sub-tasks looking at different aspects of the data. Modeling might involve exploring different families of models and approaches and evaluating these on test data.

Explicitly identifying the tasks (and sub-tasks within these, etc.) in a case study/data analysis into tasks is very useful. It allows the reader to see the connections between the tasks and focus on each task separately. It also identifies different parts of the overall analysis that can be used as incremental exercises across different assignments, or different starting points that an instructor can use by providing the inputs created from earlier tasks.

An additional benefit of identifying tasks is that we can identify their inputs and outputs. We can use this to do minimal evaluation to get to a particular point in the overall computations. We can use these inputs and outputs also to make the document interactive. Specifically, we can provide GUI controls within the rendering of the document to allow the reader to specify these values. With some (programmatic) type inference or specification by the author, we can know the types and possible values for the inputs and provide more tailored GUI controls, e.g. a slider between 0 and 1 rather. The *CodeDepends* package can examine (blocks of) code and potentially identify separate tasks, their connections and also list the inputs and outputs of separate tasks.

The following are displays of the tasks in two case studies: SPAM and wireless geolocation.

Figure 1.

Tasks for the SPAM Case Study

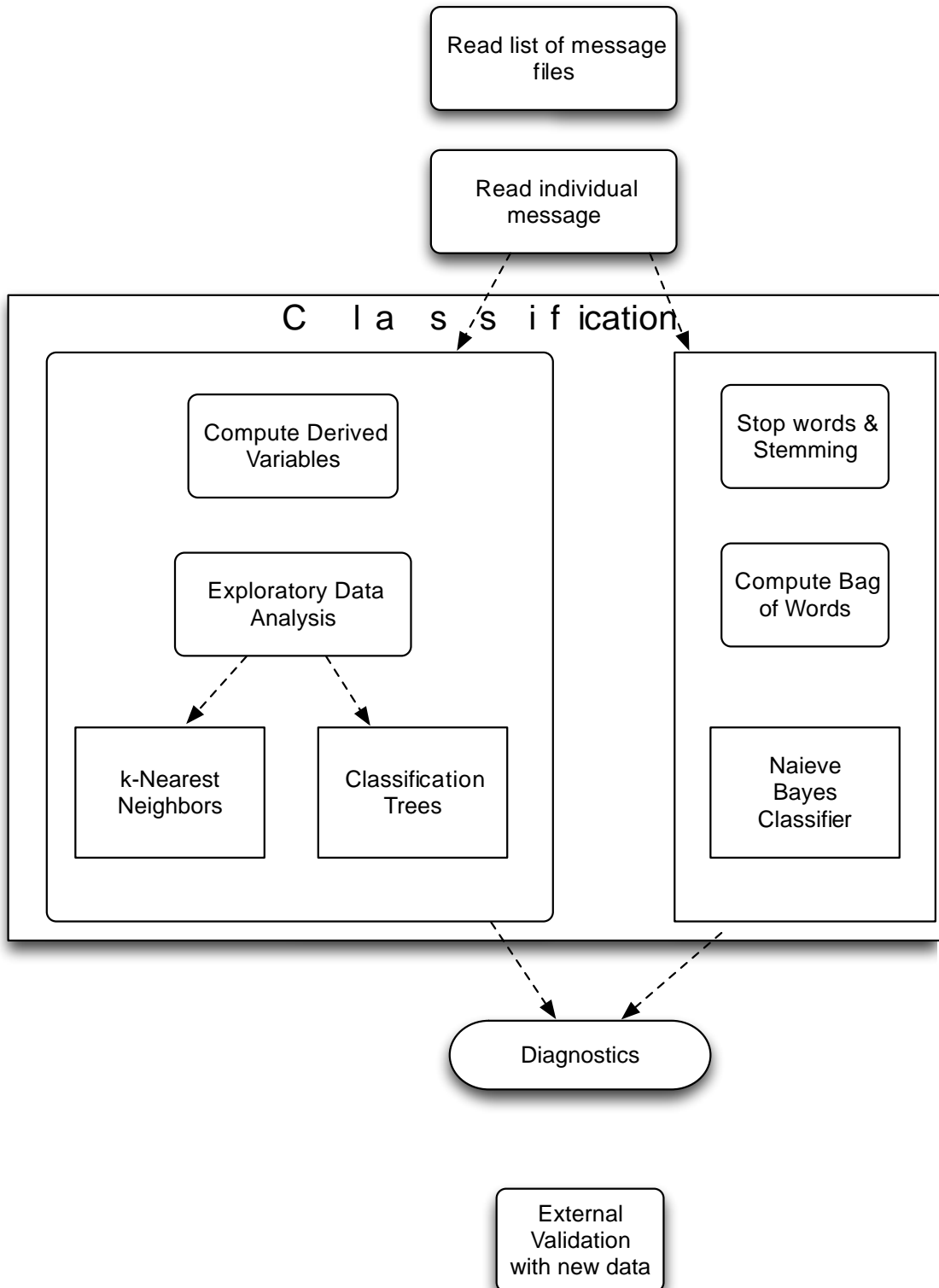
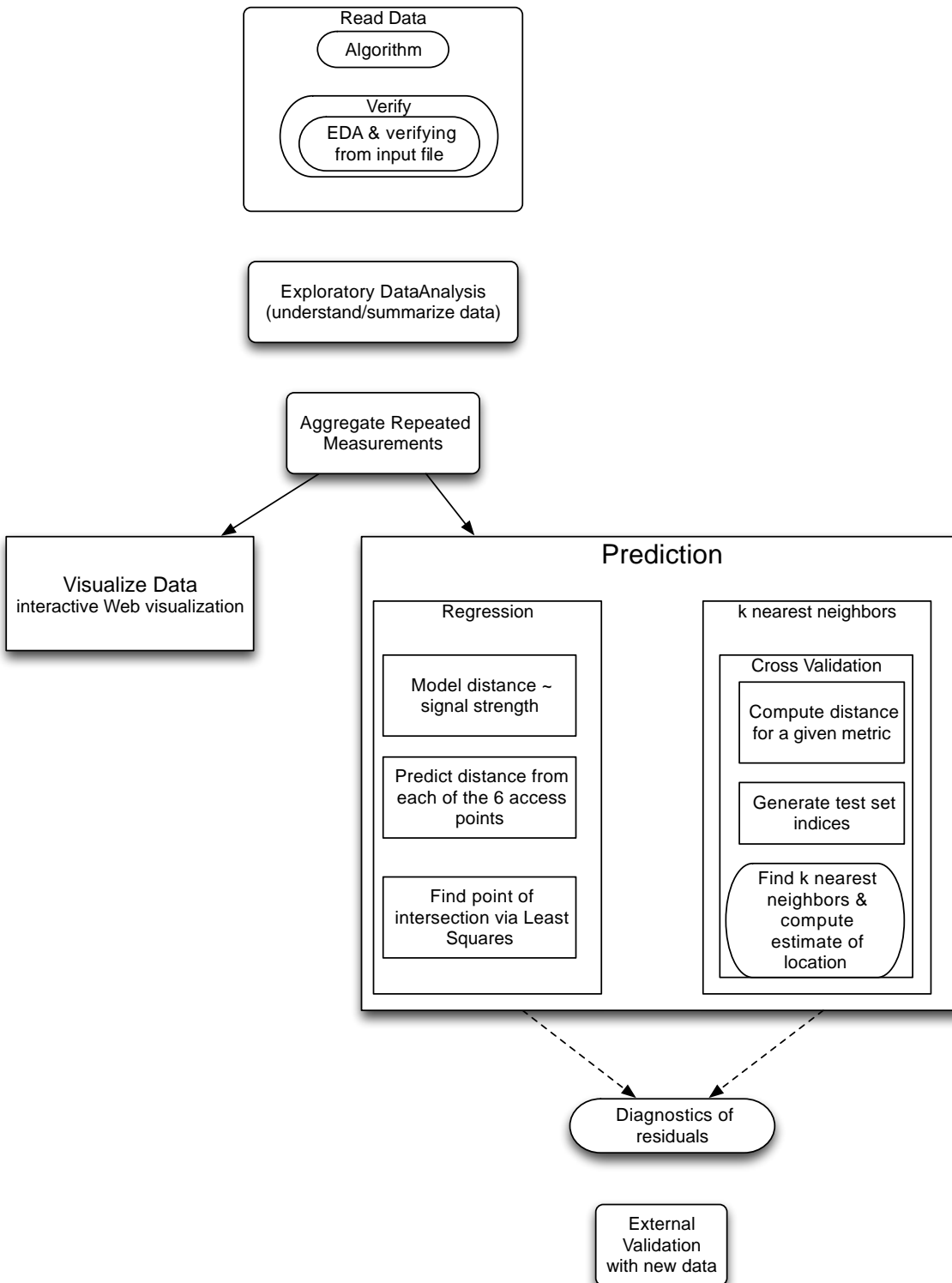


Figure 2.

Tasks for the wireless geo-location case study for the Mannheim experiment



Markup to describe some of these tasks for the SPAM example is in spamTasks.xml

Use the `<task>` element to identify a task. These can be nested. Add a `title` attribute to provide a short description. We expect to be able to use these within programmatically generated figures that display the flow of tasks.

Tasks will often be sequential and be correspondingly introduced within the document, e.g.,

```
<task title="Create Derived Variables"><xml:ns></xml:ns>

</task>

<task title="Build Classifier">
  ...
</task>
```

Within a task however, we may have several non-sequential or parallel tasks. We can identify these with

```
<task title="Create Derived Variables">
  <parallelTasks>

    <task title="Is In-Reply To">

    </task>

    <task title="Is Digitally Signed">

    </task>

    ...

  </parallelTasks>
</task>

<task title="Build Classifier">
  ...
</task>
```

Within a task we can have various `<altImplementation>` and `<altApproach>` elements. For example, we might have a "Build Classifier" task for classifying SPAM messages. Within this, we might use a classification tree, naive Bayes or k-nearest neighbors.

Working with Code in a Document

There are many good things about having code within the actual document. But there can be lots of cutting-and-pasting of the code into R. Emacs, ESS and our extensions to nxml-mode can facilitate this. See

¹These are in very recent versions of the XML package, i.e. $\geq 2.7-0$.

RdocbookEmacs.pdf. There are also facilities within R for evaluating code in such documents. These are *xmlSource()*, *xmlSourceFunctions()*, *xmlSourceSection()* and *xmlSourceThread()*¹