Easier AI

How to program simpler, smarter, faster, more flexible and understandable analytics.

Tim Menzies + the EZR mob, North Carolina State University

DOI: `https://doi.org/10.5281/zenodo.11183059`
Package: `https://pypi.org/project/ezr/0.1.0/`
Source: `http://github.com/timm/ezr`
Latex: `http://github.com/timm/ezr-tex`

Analytics is how we extract high-quality insights from low quality data. Here, we use a "less is more" approach to create a simple, fast toolkit that can tackle complex problems with just a few data points (using incremental sampling). The tool supports classification, regression, optimization, fairness, explanation, data synthesis, privacy, compression, planning, monitoring, and runtime certification (but not generative tasks). For all these tasks, our minimal data usage simplifies verification.

Our message is that while not everything can be simplified, many things can. When simplicity works, we should embrace it. Who can argue against that?

Audience

We write this book for programmers (or those that teach programmers). Here, we show the most we can do with AI, using the least amount of code.

In our own work, this material is used to teach a one semester graduate class in SE for AI.

Get the code

Our code is written in Python 3.11. To install the code, use pip install ezr. To Test that installation use texttt ezr -h.

About the Authors

This book was written by the EZR mob (students from North Carolina State University, USA) in a two-month hackathon June,July 2024.

That work was coordinated by Tim Menzies, a professor of Computer Science at NC State University (USA). Tim is the Editor-in-Chief of the Automated Software Engineering journal; an IEEE Fellow; and the recipient of over over 13 million in grant money and industrial contracts. In the literature, Google Scholar ranks him as #2 for AI for SE and software cost estimation, #1 for defect prediction, and #3 for software analytics. He has graduated 50 research students-by-thesis (including 20 Ph.D.s). This book is reversed engineered from the work of those students, who have explored applications of analytics for spacecraft control, fairness, explanation, configuration, cloud computing, security, literature reviews, technical debt, vulnerability prediction, defect prediction, effort estimation, and the management of open source software projects.

Profits from this Book

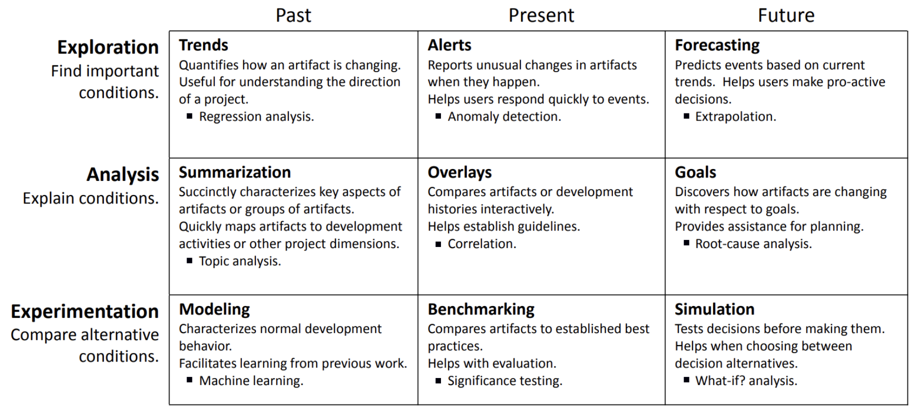
All profits from this book will be donated to the Direct Relief & Direct Relief Foundation (`https://directrelief.org`) to improve the health and lives of people affected by poverty or emergency situations by mobilizing and providing essential medical resources needed for their care.

Contents

1 Introduction: All you Need is Less

Suppose we want to use data to make decisions about what to do, what to avoid, what to do better, etc etc. How to do that?

This process is called *analytics*, i.e. the reduction of large amounts of low-quality data into tiny high-quality statements. Think of it like "finding the diamonds in the dust". For example, in one survey of managers at Microsoft, researchers found many kinds of analytics functions [?]. As shown in the following table, those functions include regression, topic analysis, anomaly detection, what-if analysis, etc:



	Past	Present	Future
Exploration Find important conditions.	**Trends** Quantifies how an artifact is changing. Useful for understanding the direction of a project. ▪ Regression analysis.	**Alerts** Reports unusual changes in artifacts when they happen. Helps users respond quickly to events. ▪ Anomaly detection.	**Forecasting** Predicts events based on current trends. Helps users make pro-active decisions. ▪ Extrapolation.
Analysis Explain conditions.	**Summarization** Succinctly characterizes key aspects of artifacts or groups of artifacts. Quickly maps artifacts to development activities or other project dimensions. ▪ Topic analysis.	**Overlays** Compares artifacts or development histories interactively. Helps establish guidelines. ▪ Correlation.	**Goals** Discovers how artifacts are changing with respect to goals. Provides assistance for planning. ▪ Root-cause analysis.
Experimentation Compare alternative conditions.	**Modeling** Characterizes normal development behavior. Facilitates learning from previous work. ▪ Machine learning.	**Benchmarking** Compares artifacts to established best practices. Helps with evaluation. ▪ Significance testing.	**Simulation** Tests decisions before making them. Helps when choosing between decision alternatives. ▪ What-if? analysis.

But is analytics as complicated as all that? Are all these functions really different or do they share a common core? And if they share a common core, does that mean if we coded up, say, regression then everything could be coded very quickly? More importantly, if we found someway to optimize that core, would that optimization apply to many kinds of analytics?

We think so. We've been working on applications of analytics for decades[1]. And in all that work, one constant has been the *compressability* of the data:

- Many data sets can be pruned down to a surprisingly small set of rows and columns, without loss of signal.
- In that compressed space, modeling becomes more manageable and all our functions algorithms run faster (since there is less to explore).
- Also, data becomes private since we threw away so much in the compression process.
- Further, explanation is easier since this there is less to explain. This means, in turn, that is easier to understand/ audit/critique our solutions.

We are not the first to say these things. For example, many researcher accept that higher dimensional data can often be reduced to a lower dimensional latent manifolds inside that high-dimensional space [?]. As a consequence, many data sets that appear to initially require many variables to describe, can actually be described by a comparatively small number of variables.

That said, this book has two novel features:

- The simplicity of our implementation: just a few hundred lines of Python;
- How much we reduce the data: often we will end up reasoning over just a few dozen key examples.

Why has not someone else published a book showing that many seemingly complex analytics tasks can be reduced to very simple code (that only needs a little bit of data)? Maybe our culture prefers complex solutions:

> *Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better*
> – Edsger W. Dijkstra

By making things harder than they need to be, companies can motivate the sale of intricate tools to clients who wished there was a simpler way.

Well, maybe there is a simpler way.

[1]In that work, we've explored data–driven applications in spacecraft control, fairness, explanation, configuration, cloud computing, security, literature reviews, technical debt, vulnerability prediction, defect prediction, effort estimation, and the management of open source software projects.

2 Before we Begin

Before anything else, we need to cover some preliminaries.

2.1 A Few Imports

Table **??** imports the Python modules used by this code. The line `sys.dont_write_bytecode = True` ⓐ tells Python not to clog the source code with a `__pycahce__` directory containing .pyc files.

2.2 A Few Special Types

In this code, UPPER case words are classes and words starting with a single upper case letter are special types (the special types are just used for documentation purposes and so have no methods). For examples, as shown in Table **??**:

- Missing values are marked with a `DontKnow` symbol; ⓑ
- A `Row` is a list of any atom (floats, ints, booleans, strings). ⓒ A `Row` can also contain `DontKnow`.
- `Rows` hold many `Rows`. ⓓ
- `Classes` are dictionaries whose keys are class names and whose values are `Rows`. ⓔ

ALso, `OBJ` is a handy-dandy super class for everything we do. This class is an easy way to generate an `OBJ`ect with very little ; e.g.

```
person = OBJ(age=10, job="programmer")
```

That new `OBJ`ject gives us access to all its slots for reading and update:

```
person.age += 1
peron.job = "retired"
print(person)
```

```
from __future__ import annotations   # <1> ## types
import sys
sys.dont_write_bytecode = True ⓐ
from collections import Counter
import re,ast,copy,json,math,random
from fileinput import FileInput as file_or_stdin
```

Listing 1: *Row creation. And column header creation (from row1)*

```
from typing import Any,Iterable,Callable

DontKnow = "?"  ⓑ
class Row    : has:list[Any] ⓒ
class Rows   : has:list[Row] ⓓ
class Classes: has:dict[str, Rows] ⓔ

class OBJ:
  def __init__(i,**d)    : i.__dict__.update(d)
  def __repr__(i) -> str : return i.__class__.__name__+show(i.__dict__)
```

Listing 2: *Special types, just for this system.*

2.3 “the” stores the Config

All code has magic settings and this code is no exception. Our config options are help in the `the` variable and is referenced via (e.g.) `the.seed`.

```
ez.py: Active learning, find best/rest seen so far in a Bayes classifier
(c) 2024 Tim Menzies <timm@ieee.org>, BSD-2 license

OPTIONS:
  -s --seed     random number seed    = 1234567891
  -g --go       start up action       = help
  -f --file     data file             = ../data/auto93.csv
  -n --ntiny    a tiny number         = 12
  -N --Nsmall   a small number        = .5

  Discretize:
  -C --Cuts     max number divisions of numerics = 16

  Classify:
  -k --k        low frequency kludge  = 1
  -m --m        low frequency kludge  = 2

  Optimize:
  -i --init     initial eval budget   = 4
  -B --Budget   max eval budget       = 20
  -T --Top      keep top todos        = .8

  Explain:
  -l --leaf     min leaf size         = 2
```

Table 1: *Help text: defines the global options.*

This `the` is generated by parsing the help string of Table **??**:

- The control parameters are the word that follows “–”.
- That parameter’s default value is the last work on the line with “–”.
- That default can be changed on the command line. E.g. our random number “seed” can be set from the Mac or UNIX operating system using `ezr -s $RANDOM`

Listing 3: *Special types, just for this system.*

2.4 Just a Little Maths

2.4.1 Numbers and Symbols

2.4.2 What’s in the middle? What is spread around the middle?

2.4.3 Pseudo-Random Numbers

2.5 Just a Little Stats

2.6 Statistically Distinguishable

2.7 Types

For ease of documentation, this code uses type hints. Most of our types are standard in Python 3.11 but we had to import and define some specials:

3 Core Classes

How can we combining many things into a much smaller number of things? One way is to look for the glue, undere-the-hood, that is shared across all those things.

For analytics, that glue is the data processed by the different alorithms. So the core of this system is four classes: `DATA`, `NUM`, `SYM`, and `COLS`. There are several other classes but these four are always center-stage.

`DATA` is where we store rows of data. Each column of that data is summarized in a `NUM`eric or `SYM`bolic header. And `COLS` is a helper clas that turns a list of column names into the various `NUM`s and `SYM`s.

`DATA` can be loaded in from file of comma-separate values. In these files, the first row contains the column header names. For example:

```
Clndrs  Volume  Model   origin  Lbs-   Acc+  Mpg+
------  -----   ------  ------  -----  ---   ----
4       97      82      2       2130   24.6  40
4       96      72      2       2189   18    30
4       140     74      1       2542   17    30
...     ...     ....    ....    .....  ..... ....
4       119     78      3       2300   14.7  30
8       260     79      1       3420   22.2  20
4       134     78      3       2515   14.8  20
6       231     78      1       3380   15.8  20
8       302     77      1       4295   14.9  20
8       351     71      1       4154   13.5  10
```

Just to explain the column names:

- Names starting with “Uppercase” are `NUM`eric and the other columns are `SYM`bolic.
- Names ending with “-”,“+” or “!” are the *goals* which must be minimized, maximized or predicted. The other columns are the observables or controllables used to reach the goals.

The rows are all examples of some function $Y = F(X)$ where:

- Y are the goals (also called the dependents)
- X are the observables or controllables (also called the independents)
- F is the model we want to generate.

For example, the above table is about motor cars:

- Lighter cars cost less to build since they use less metal. Hence "Lbs-" (minimize weight).
- Faster, fuel efficient cars are easier to sell. Hence "Acc+" (maximize acceleration) and "Mpg+" (maximize miles per gallon).

The rows of this table are sorted by *distance to heaven* i.e the distance of each row to some mythical best car with least weight, most acceleration and miles per hour. Those rows are then divided into a `smallN` best rows (the first three rows) and rest (the other rows).

- `smallN` is our shorthand for $\sqrt{N}$.
- We have another term, `tinyN`, which denotes a dozen ($N = 12$) examples.

I'm not the first to say these things[2]. So it is a little strange that someone else has not offer something like this simpler synthesis. But maybe our culture prefers complex solutions:

> *Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*
> – Edsger W. Dijkstra

By making things harder than they need to be, companies can motivate the sale of intricate tools to clients who wished there was a simpler way. Well, maybe there is.

```
import re,ast
from typing import Any,Iterable,Callable
from fileinput import FileInput as file_or_stdin
#---------- ---------- ---------- ---------- ---------- ---------- ----------
def coerce(s:str) -> Any:
  "s is a int,float,bool, or a string"
  try: return ast.literal_eval(s) #
  except Exception:  return s

def csv(file=None) -> Iterable[Row]:  [a]
  "read from file or standard input"
  with file_or_stdin(file) as src:
    for line in src:
      line = re.sub(r'([\n\t\r ]|#.*)', '', line)
      if line: yield [coerce(s.strip()) for s in line.split(",")]
#---------- ---------- ---------- ---------- ---------- ---------- ----------
class COLS(OBJ):
  """Turns a list of names into NUMs and SYMs columns. All columns are held
  in i.all.  For convenience sake, some are also help in i.x,i.y
  (for independent, dependent cols) as well as i.klass (for the klass goal,
  if it exists)."""
  def __init__(i, names: list[str]):
    i.x, i.y, i.all, i.names, i.klass = [], [], [], names, None
    for at,txt in enumerate(names):
      a,z = txt[0], txt[-1] % first and last letter
      col = (NUM if a.isupper() else SYM)(at=at,txt=txt)
      i.all.append(col)
      if z != "X": # if not ignoring, maybe make then klass,x, or y
        (i.y if z in "!+-" else i.x).append(col)
        if z == "!": i.klass= col

  def add(i,row: Row) -> Row:
    "summarize a row into the NUMs and SYMs"
    [col.add(row[col.at]) for col in i.all if row[col.at] != "?"]
    return row
```

Table 2: *Creating rows and column headers (from row1).*

[2]From Wikipedia: The manifold hypothesis posits that many high-dimensional data sets that occur in the real world actually lie along low-dimensional latent manifolds inside that high-dimensional space. As a consequence of the manifold hypothesis, many data sets that appear to initially require many variables to describe, can actually be described by a comparatively small number of variables, likened to the local coordinate system of the underlying manifold.