Computational Research 101
Experimental Design - Data Analysis
Software Engineering

Domenico Salvagnin
DEI, University of Padova

IPCO 2023
Madison, WI
Motivation

Computational research is key ingredient in many areas

Deceptively simple:

1. Implement algorithm
2. Compare against existing methods
3. Publish
Not so fast…

Need proper practices for:

- Correctness
- Reproducibility of Results
- Sharing
Outline

Experimental design
Data analysis
Writing code
Conclusions
Experimental Design
Why experiments?

Increasing gap between theory and practice:

❖ Worst-case analysis not always very telling
❖ Average-case analysis often relies on questionable assumptions
❖ Complex algorithms and data structures too hard / impossible to analyze theoretically
❖ Many complex algorithms never implemented at all :-(

Why experiments?

Inherent value in efficient implementations:

- Algorithm engineering can improve runtime by orders of magnitude
- Theories are confirmed and suggested by experimentation
- Empirical science no easier than theoretical one (just different)
Which type of paper?

Different types of papers:

- Application
- Horse Race
- Experimental Analysis
The Scientific Method

1. exploration
2. formulate hypothesis or question
3. design experiment to test its validity
4. data analysis
5. draw conclusions

and reiterate!
Basic Principles

Ask interesting questions

Use appropriate test sets

Use appropriate experimental design

Use reasonable efficient implementation

Ensure reproducibility

Test significance and draw justified conclusions
Interesting Questions

Exploratory experimentation to find good questions

Think before you compute

❖ Which interactions are you planning to investigate?
❖ Are you collecting the right data?
❖ What are the potential outcomes? How would they affect the hypothesis?

Don't spend too much time (on the wrong experiments)
Cannot clearly test on all possible instances…

Need the (finite) subset to be representative of the actual instances of interest

Easier said than done…
Testsets

Large and heterogeneous enough

Right level of difficulty (avoid ceiling/floor effects)

Avoid over/underrepresentation of problem classes

Real-world vs randomly-generated models
Types of Experiments

Manipulation

Observation

Factorial
Pitfalls

Missing control experiment (related: confirmation bias)

Data collection biases (e.g., survival bias)

Overtuning

- Split into training/testing/validation

- Avoid data leakage
Pitfalls

Do not compare algorithms that play a different game:

- Exact methods vs. heuristics
- Exact methods with different stopping criteria (e.g. optimality tolerances!)

You never compare algorithms, but rather their implementation!
Reasonable implementation

Naïve implementation can hide fundamental weaknesses of the proposed approach

Unnecessarily inefficient implementation is a limit factor in how much testing you are able to perform

Do not overdo it! (see profiling tomorrow)
Reproducibility

Keep track of all necessary metadata about the experiments

Provide enough information for another researcher to be able to replicate the experiment

- No need to replicate the same runtime and/or path
- But an equivalent experiment should be consistent and allowing to draw the same conclusions
Reproducibility

Code and paper must match

Horse race papers: instances and/or the code publicly available

Tell the full story:

❖ Do not overly aggregate data (put detailed results online, in a proper format!)
❖ Do not hide/omit anomalous results
❖ Always report running times (even if not the main focus)
If measuring time (*almost always*), need to enforce reliable measures:

- Identical machines
- Exclusive usage
- No background processes
- No Turboboost and Hyperthreading (or similar)
- Avoid unnecessary I/O (disk and network) while benchmarking
What about the timelimit?

Introduces a bias \textit{against} methods that solve more models.

Inherently nondeterministic and nonreproducible.

…but it is a practical necessity: use the highest your resources can afford!

[No, PAR-x is not the answer]
What performance measures?

time to optimality (but remember time limit bias!)

- variant: time to x% gap

number of instances solved (within resource limits)

B&B nodes (only meaningful for instances solved to optimality by all methods under comparison!)

Primal-dual integral (gives more global view on solution process)
What performance measures?

What about heuristics?

- Number of solutions found (success rate)
- Time to first solution
- Time to optimal solution
- Time to solution within X% gap
- Primal integral
Justified Conclusions

Use appropriate data analysis tools

Interpret the data: look for patterns/explanations

Avoid statements not supported by data
Data Analysis
How to aggregate results?
Performance variability
Statistical significance of results
How to aggregate numbers?

First of all: aggregation always leads to information loss

Always look at the whole dataset before aggregation!
Means are mean

Obvious choice: arithmetic mean

- Proportional to total runtime in the real world
- Uns suited for normalized numbers
- Too sensitive to big numbers

Geometric mean: too sensitive to small numbers

Quartiles/medians: too insensitive to measure progress

Common empirical tradeoff: shifted geometric mean
How to split results?

Aggregation on the whole testset too harsh: what about splitting into groups first?

Very natural criterion: split by difficulty of instances

Beware of biased selections :-(
\[ N = 10000 \]

\[ \text{names} = \text{list}('AB') \]

\[ \text{df} = \text{pd.DataFrame}({100*\text{np.random.rand}(N, 2)}, \text{columns}=\text{names}) \]

easy = df[df['A'] < 80]

hard = df[df['A'] >= 80]

easy = df[df.min(axis=1) < 80]

hard = df[df.min(axis=1) >= 80]
Performance Variability

The behaviour of the solver can be significantly influenced by seemingly neutral changes in the environment/input data:

- random seed initialization
- order of constraints and variables in the problem

Runtime of the a given algorithm is basically a random variable even on what is mathematically the same instance!
Fig. 3: Solution times for 100 permutations
Performance Variability

Where does performance variability comes from?

In a nutshell: imperfect tie-breaking

- Solvers take many decisions with limited knowledge

Can we fix it? No

This is the price to pay for trying to be smart and efficient!
Statistical Tests to the Rescue

Mathematically sound approach to computationally evaluate the probability of two sequences of numbers coming from the same distribution (*null hypothesis*)

- numbers are the performance measures on the selected testset by the methods under comparison
- *null hypothesis* is that the methods are equivalent, and the difference we measured is just noise
Many different statistical tests that differ on:

❖ Assumptions
❖ Power
❖ Paired vs unpaired samples
Statistical Tests III

Either way: need a sufficiently large testset!!!

Remember that significant ≠ meaningful
Conclusions (so far)

1. Apply Scientific method
2. Avoid biases in experiment setup/data analysis
3. Deal with Performance Variability
4. Use Statistical tests
Writing (Good) Code
Why good code?

Correctness  Productivity  Sharing
“I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.”

—Bjarne Stroustrup (inventor of C++)
How to write good code?

Writing (good) code is inherently hard:

❖ Complex problems require complex code
❖ Requires (almost inhuman) attention to every details…
❖ …over (many) different levels of abstractions
How to write good code?

It does not come natural, but can be learnt!

❖ Best practices from software engineering
❖ Use the right tool for the job

This is very relevant even for academic code developed by a single person
Use right tool for the job

Pick the right language for job:

- Compiled language where performance is critical
- Scripting language for the rest

Use sufficiently powerful editor / IDE

Don’t debug with print statements: learn to use a debugger

Don’t do version control by hand
Invest time in learning your tools

Did you know that vim supports compiler assisted code completion?

Did you know that gdb can be scripted with Python?

Are you proficient with templates and the STL in C++?

Are you proficient with numpy, pandas and matplotlib in Python?

Don’t reinvent the wheel!!!
There is **no** real excuse for not using version control

- Even for academic code
- Even if developing alone

At a bare minimum:

- Need to sync code back and forth between laptop and workstation
- Ability to revert back bad changes
- And no, sending code by email or `scp` is not an alternative
Git: a game changer

Changes your approach to coding completely:

❖ Time machine for code
❖ Multiverse for code (branches)
❖ Key to reproducibility (tags/commit hashes)

Fundamental to share code with others!

No need to become a wizard: can go long way with just the basics!
Good code: correctness

Correctness trumps everything else…

…but how to make sure your code is correct?

Readability

Testing

Reviews
Good code: readability

Code is read way more often than it is written

Unreadable code is hard to:

❖ Understand (and thus argue correct)
❖ Maintain (modify, fix, improve, extend)

Please be gentle to the next developer (it could be you!)
“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

–Brian Kernighan
Good code: readability

Proper naming (classes, functions, variables)

Use language features whenever appropriate

Split code into relatively short chunks

Single responsibility *(one thing well)* at a single abstraction level

Explicit preconditions, postconditions, invariants *(asserts)*
Proper naming

Intention revealing

```c
int t; // elapsed time in hours
```
bad

```c
int elapsedHours;
```
good

Unambiguous

```c
void copy(char* a1, char* a2);
```
bad

```c
void copy(char* source, char* destination);
```
good

Searchable

Use language from problem domain
## Language features

### Const/access modifiers

**Bad**

```c
void copy(char* source, char* destination);
```

**Good**

```c
void copy(const char* source, char* destination);
```

### Named constants (instead of defines)

**Really bad**

```c
if (context == 128) {...}
```

**Bad**

```c
#define RELAXATION_CONTEXT 128
if (context == RELAXATION_CONTEXT) {...}
```

**Good**

```c
const int RELAXATION_CONTEXT = 128;
if (context == RELAXATION_CONTEXT) {...}
```
Functions

Small

Single responsibility *(one thing well)* at a single abstraction level

Avoid side effects

Explicit preconditions, postconditions, invariants *(asserts)*
How many times the same code needs to be written before we turn it into a function?

Once, just once!
Good code: comments

```java
int colIndex; // column index

// indices 1-based
for (int i = 0; i < n; i++) {
    a[i] = compute_value(i);
}

// Autogenerated, do not edit. All changes will be undone

// http://tools.ietf.org/html/rfc4180 suggests that CSV lines should be terminated by CRLF, hence the \r\n.
csvStringBuilder.append("\r\n");
```
“If the comment and code disagree, both are probably wrong.”
—Bjarne Stroustrup

“Don’t comment bad code—rewrite it!”
—Brian Kernighan & P.J. Plaugher
Good code: testing

There is no substitute for actual testing

Whenever the program misbehaves, write a test

But how do you know the program has a bug?

Well, you should write the tests first!

Testing must not be an afterthought!

Time consuming? Yes, but usually well worth it!
One of the most effective practices
4 eyes are better than 2, but not just that
As authors of the code, we are the most biased in evaluating it
If developing alone, please ask your supervisor to review your code
After all, you would never put your name on a paper without checking its proofs…this is no different!
Code quality measurement: WTFs per minute

Good Code

Bad Code

Code review

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF?

WTF is this?

Dude, WTF?
“Premature optimization is the root of all evil.”
–Sir Tony Hoare

“The full version of the quote is "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." and I agree with this. Its usually not worth spending a lot of time micro-optimizing code before it’s obvious where the performance bottlenecks are. But, conversely, when designing software at a system level, performance issues should always be considered from the beginning.”
–Charles Cook
Good code: profiling

Worry about other issues (good algorithm design and good implementations of those algorithms) before worrying about counting cycles.

How do you spot inefficiencies?

❖ Do not trust your judgement
❖ Use a profiler (valgrind, perf, VTune)
Conclusions

Inherently hard (no easier than theoretical research)

Many challenges:

❖ general lack of formal training
❖ increasingly resource hungry
❖ some fields are more mature

Still great opportunities :-)
One more thing...
Presentation

Don’t spoil your hard work with a mediocre presentation!

Invest time to learn properly display of information in:

❖ Tables
❖ Plots
❖ Slides

Remember: you are a professional communicator!
<table>
<thead>
<tr>
<th>Instance</th>
<th>Time</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.34</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>7.1</td>
<td>123</td>
</tr>
<tr>
<td>C</td>
<td>100.2</td>
<td>40000</td>
</tr>
<tr>
<td>D</td>
<td>3600</td>
<td>1453322</td>
</tr>
<tr>
<td>E</td>
<td>15.22</td>
<td>430</td>
</tr>
<tr>
<td>F</td>
<td>93.333</td>
<td>7023</td>
</tr>
<tr>
<td>G</td>
<td>211.3</td>
<td>50311</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance</th>
<th>Time</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.34</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>7.10</td>
<td>123</td>
</tr>
<tr>
<td>C</td>
<td>100.20</td>
<td>40,000</td>
</tr>
<tr>
<td>D</td>
<td>3,600.00</td>
<td>1,453,322</td>
</tr>
<tr>
<td>E</td>
<td>15.22</td>
<td>430</td>
</tr>
<tr>
<td>F</td>
<td>93.33</td>
<td>7,023</td>
</tr>
<tr>
<td>G</td>
<td>211.30</td>
<td>50,311</td>
</tr>
<tr>
<td>α (%)</td>
<td>Line</td>
<td>Fat</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Delay</td>
</tr>
<tr>
<td>0</td>
<td>BZVR</td>
<td>16.149</td>
</tr>
<tr>
<td>0</td>
<td>BrBO</td>
<td>12.156</td>
</tr>
<tr>
<td>0</td>
<td>MUVR</td>
<td>18.182</td>
</tr>
<tr>
<td>0</td>
<td>PDBO</td>
<td>3.141</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>49,628</td>
</tr>
<tr>
<td>1</td>
<td>BZVR</td>
<td>14,399</td>
</tr>
<tr>
<td>1</td>
<td>BrBO</td>
<td>11,423</td>
</tr>
<tr>
<td>1</td>
<td>MUVR</td>
<td>17,808</td>
</tr>
<tr>
<td>1</td>
<td>PDBO</td>
<td>2,907</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>46,537</td>
</tr>
<tr>
<td>5</td>
<td>BZVR</td>
<td>11,345</td>
</tr>
<tr>
<td>5</td>
<td>BrBO</td>
<td>9,782</td>
</tr>
<tr>
<td>5</td>
<td>MUVR</td>
<td>16,502</td>
</tr>
<tr>
<td>5</td>
<td>PDBO</td>
<td>2,412</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>40,041</td>
</tr>
<tr>
<td>10</td>
<td>BZVR</td>
<td>9,142</td>
</tr>
<tr>
<td>10</td>
<td>BrBO</td>
<td>8,496</td>
</tr>
<tr>
<td>10</td>
<td>MUVR</td>
<td>15,153</td>
</tr>
<tr>
<td>10</td>
<td>PDBO</td>
<td>1,791</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>34,762</td>
</tr>
<tr>
<td>20</td>
<td>BZVR</td>
<td>6,210</td>
</tr>
<tr>
<td>20</td>
<td>BrBO</td>
<td>6,664</td>
</tr>
<tr>
<td>20</td>
<td>MUVR</td>
<td>13,004</td>
</tr>
<tr>
<td>20</td>
<td>PDBO</td>
<td>1,357</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>27,235</td>
</tr>
<tr>
<td>40</td>
<td>BZVR</td>
<td>3,389</td>
</tr>
<tr>
<td>40</td>
<td>BrBO</td>
<td>4,491</td>
</tr>
<tr>
<td>40</td>
<td>MUVR</td>
<td>10,289</td>
</tr>
<tr>
<td>40</td>
<td>PDBO</td>
<td>676</td>
</tr>
<tr>
<td></td>
<td>Tot</td>
<td>18,845</td>
</tr>
</tbody>
</table>
References

❖ G. Law. “Give me 15 minutes & I’ll change your view of GDB”, CppCon 2015 (on youtube)
❖ S. Few. “Show Me the Numbers”, 2012