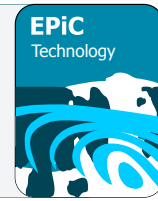




EPiC Series in Technology

Volume 5, 2026, Pages 164–170

Proceedings of 17th International Conference on Current Research Information Systems



Project Highlites: Lightweight Open-Source Predictor for any Classifications, Trained with Data of Choice

Alain Descamps

Wander team

Universiteit Antwerpen, Belgium

alain.descamps@uantwerpen.be

Abstract

Subject classification of publications, projects and other entities is ubiquitous in CRIS systems.

This project arose from practical considerations: how could we easily use AI to facilitate the cris administrator's work? This led to an application that can be deployed in various settings: given a brief textual description (e.g., a title and/or abstract of a publication), the application can generate one or more proposals for subject cataloging. This can be implemented directly in various settings and for different types of classifications. Given sufficient training material, the application can be used with any classification system.

The aim of this article is to demonstrate that the open-source project Highlites is a useful tool for calculating usable classification proposals using one's own limited resources, and in a sufficiently short time.

1 Introduction

The goal of a current research information system (CRIS) is to manage and exchange metadata regarding research activities. Subject indexes form an essential part of making this metadata discoverable. To predict such subject indexes, you can use 'classic' AI tools such as ChatGPT, but in this article, we demonstrate a simpler and more efficient tool that is often sufficient.

2 The Resources

The goal had to be realized with the resources we had available: our production server and the installed Wander software: this is owned by the university, however, relies solely on open-source

packages. The idea was to alter and split up code so that it can be available as an open-source downloadable package. Under the hood, we found several open-source resources that we started working with:

1. The first resource was our database engine, a no-SQL database with which we had over 25 years of experience and which had more than proven its worth: fast, scalable, and with flexible data storage — exactly what we needed. It ticks all the boxes, that a database engine should fulfill: the name of this hidden gem is Yottadb, and it has the perfect choice of open source and support contract availability.
2. The second tool was the programming language used to originally access the database: M, or Mumps, one of few programming languages supported by an ISO standard. Its purpose is again limited, namely controlling the database, but in that respect it excels: all the requirements for a modern database, such as ACID atomicity, consistency, isolation, and durability, transaction processing, database recovery, mirroring, etc., are met. Moreover, it is a highly scalable product, functioning on multi-processor (Linux) servers up to and including several Raspberry Pi models.
3. The third part involved expanding the M database to also be accessible from Lua: this project, now known as YDBLua, was also sponsored by the University of Antwerp and made open-source (<https://gitlab.com/YottaDB/Lang/YDBLua>). This made it possible to develop precisely those things in Lua where Lua is stronger than M, such as processing files in XML format. The M database has also since been integrated into other programming languages, including Python and Go (YottaDB, 2026).
4. The fourth part consisted of software pieces from our Wander repository, which had already proven their worth in our applications and each solved specific problems. For example, there is an algorithm that attempts to intelligently split a sentence into tokens. A nice twist: we also developed an algorithm that converts a sentence to lowercase, and does this for every known Unicode character, but also allows you to choose which Unicode version you want and compare the results. Therefore, this algorithm works with text in any language, as long as encoded in UTF-8.
5. The fifth part was a backoffice application, which allowed us to use certain test setups by our users and to collect initial experiences from them.

3 The Challenge

Developing the algorithm itself was quite an adventure: it had to be lightweight. For practical reasons, guessing a classification had to be done within a reasonable timeframe — say, a second — and it shouldn't consume a lot of computer power, which were shared with all other applications. It quickly became clear that the current prevailing approach with neural networks required too many system resources.

4 The Algorithm

Our algorithm became simple and modest: we assumed from the outset that sometimes no proposal would be calculated in any possible cases. The reason is that, in our opinion, a bad or pointless proposal is likely worse than no proposal.

When training the know-how database, the titles and abstracts are split into a series of tokens, and the resulting classification for each token is stored. No qualifiers are considered.

The tokens are essentially the words from the titles and abstracts, split appropriately and converted to lowercase.

When guessing a classification for a given new title or abstract, the sentences are also split into tokens using the same process. These tokens are ranked according to the number of different classifications they are associated with. Words that lead to many classifications, such as 'about', 'the', etc., are less important; words that lead to few classifications - like 'sociodemographic' - are much more important. In the first round, the classifications to which the 'rarest' word leads are taken. In the second round, you take the intersection with the 'second rarest' word, and you continue this until you have no more classifications left: the classifications you had in the penultimate rounds become your answer. If this exceeds a certain number — for example, 10 — the algorithm does not generate any suggestions.

A few configuration parameters were foreseen:

- A word threshold percentage: if a word leads to more than $x\%$ of all subject codes, it is considered a stop word. The advantage of this is that you do not have to maintain stop words, and that stop words are environment-dependent. For example, 'international' is a stop word in some cases and not in others. Words containing numbers are also ignored.
- A minimum number of suggestions. This is particularly interesting if you use this tool to generate sensible suggestions from which a human can choose. In such cases, this tool truly comes into its own. You can enter a minimum of zero, of course.
- A correction algorithm for 'special' words. If a word leads to only a few subject codes, it can be 'pushed'. The default is that if a code leads to only one or two codes, it is pushed by a factor of 10 or 5.
- The tokenizer. The default tokenizer splits the sentence into words and converts them to lowercase. Special attention is paid to the punctuation marks '*' and '-': for example, 'non-imaginary' is treated as one word. Note: the lowercase conversion works in all languages, as long as the text is provided in UTF-8 encoding. So *text in any language should do the trick*.

Interestingly, this algorithm is generic enough that it can even be used for other purposes than subject classification. A nice example is recognizing the language of a text. After all, one can also view a language as a kind of classification. All you have to do is fill the system with words of which you are 100% certain in what language they are written.

5 The Product

The project is currently available as an open-source beta version at <https://gitlab.com/AlainDsc/highlights/>.

Essentially, two things were added.

The first is a command-line interface that allows the necessary instructions to be executed via the prompt. These instructions consist of a 'training' component — to train the 'know-how' — and a 'predict' component, to calculate a classification proposal. This appears in standard output. The final product of this project is therefore not a ready-made application, but a piece of software that a developer can work with. A Lua module is available, and all ingredients are present to build a module in Python, Go, Javascript or Rust.

The second part consisted of adapting existing code from our own software repository into a cohesive whole that was made available as an open-source version.

Also, installation scripts were added.

Remark: most of this code is hard-boiled; only the wrapper around is new.

The Highlites engine inherits several remarkable properties from the underlying database: for instance, it is genuinely usable in a multi-task and multi-user environment: you can train multiple knowledge systems simultaneously and make use of them, and you can also train the same system via a number of parallel processes. You can train a system on literally thousands of different subject codes.

6 The Use Cases

We went looking for good pilot projects where we could test the Highlites software. Ultimately, we wanted to try to prove that this was usable in practice. It took more effort than expected: your plant's nutrition will determine whether you get a good harvest, and poor nutrition leads to nothing, even with the best algorithm. In this article, let me illustrate two cases: a 'good' and a 'bad' harvest.

6.1 Case 1: Linking UDC Codes to Scientific Publications

When a researcher from our university submits a publication online, he or she must choose from a limited number of 23 UDC (Universal Decimal Classification) codes:

- 02: Documentation and information
- 10: Philosophy
- 15: Psychology
- 20: Religious studies
- 301: Sociology
- 32: Politics
- 33: Economics
- 34: Law
- 37: Educational sciences
- 51: Mathematics
- 53: Physics
- 54: Chemistry
- 57: Biology
- 61: Human medicine
- 615: Pharmacology. Therapy
- 619: Veterinary medicine
- 62: Engineering sciences. Technology
- 659.3: Mass communications
- 681.3: Computer. Automation
- 70: Art
- 800: Linguistics
- 82: Literature
- 93: History

We were able to export 221,848 titles and abstracts, along with their subject indexes.

The following procedure was executed eight times:

- Shuffle the lines in the export file

- Split the export file into a large amount for training, and a small amount for evaluation. Following splits were used: 200,000 + 21,848, 220,000 + 1,848, 221,000 + 848.
- Train a memory system with the training set (reset before training).
- Evaluate with the smaller set.
- Evaluate with a word threshold of 50%, 80% and 90%.
- Evaluate with a minimum number of suggestions of 0, 3 and 5.

The procedures were executed on the available Linux server, which was also performing various other tasks in the meantime. So the execution speed is purely indicative, but it still gives you an idea of what Highlites is capable of.

Each training session took less than an hour: Highlites reached speeds between 61 and 81 trainings per second.

Predicting codes took between 36 and 910 predicts per second. Speed of predicting is primarily determined by the length of the text.

A typical Highlites memory system occupied 50 to 200 MB.

Word thresholds of 90% were noticeably better than 80 % and 50 %. So, we concentrated on the results with the 90% threshold.

A minimum of 1 correct code was predicted in 89.3 % to 97.8 %.

All the correct codes were predicted – among others – in 82.4 % to 94.8 %.

All the correct codes were predicted ‘spot-on’ in only 25 to 29 %.

Predicting codes using already trained text yielded spectacular results, like +98 to 99 %.

Please note that this is a complex process, where you do not know in advance exactly how many codes need to be predicted.

6.2 Case 2: Linking VODS Codes to Projects

Every research institution in Flanders must send project data to the Wewis platform, accompanied by so-called VODS subject codes. VODS is a uniform classification system for research disciplines, based on the international FORD classification system of OECD. VODS has about 2,991 codes in a hierarchical structure, which is quite extensive.

Wewis delivered a file of 76,831 titles and 76,569 abstracts, both in the English language, together with linked VODS codes.

Similar procedures as in case 1 were executed. In short: the first results were disastrous.

So further investigation was necessary, and the following steps were taken:

- Linking was done on different hierarchical levels: example: some titles were linked to VODS 02990103 (Geospatial information systems), and others to the more general 029901 (Geomatic engineering). A new file was generated and used, containing only codes with a length of 6.
- Sometimes code 010399 (Other physical sciences) was assigned, and in other, similar titles got 010303 (Classical physics). A new file was therefore generated, in which codes were ignored if they contained the words 'Other' or 'other' in their description. At the end, it contained 72,898 titles and 72,650 abstracts.

Training and predicting codes were at speeds, very comparable with case 1.

A minimum of 1 correct code was predicted in 44.5 % to 81 %.

All the correct codes were predicted – among others – in 23 % to 67 %.

All the correct codes were predicted ‘spot-on’ in only 6 to 7.5 %.

Predicting codes using already trained text yielded spectacular results, like +96 %.

7 Conclusions

7.1 Surpassing 'Classic' AI Methods in Speed by a Factor

With predicting speeds between 36 and 910 predicts per second in our test, and with the possibility to launch parallel prediction sessions, one cannot help but conclude that, structurally, this is an order of magnitude faster than a 'classic' targeted AI.

7.2 Fast, Economical, and Usable in Real Life

Although we were unable to take exact measurements of power consumption, the impact of the Highlites processes on our servers was no greater than the impact of a standard process: one can therefore cautiously conclude that Highlites is an order of magnitude more consumer-friendly than 'classic' AI processes.

Execution speeds were more than fast enough to support a real-life application.

7.3 Acceptable Prediction Rate, Bad Noise

Although prediction rates are highly dependent on the size and quality of the training data, one can conclude that the Highlites algorithm yields usable results in some cases.

The disappointing part: in addition to predicting correct subject codes, there is a fair amount of noise from false positives.

7.4 *Ceci N'est Pas un Remplaçant*

Highlites is therefore certainly not a replacement for the CRIS administrator.

The contribution of the CRIS administrator should never be underestimated either.

However, it can play an assisting role in generating suggestions, and in reducing hundreds of subject codes to a manageable list from which a researcher can easily choose.

7.5 One Cannot Prepare a Good Meal from Waste

Finally, I would like to briefly point out that predicting subject codes based on titles and abstracts has its limits: a title can be so general or misleading that any prediction using any algorithm fails.

An example of a title: *Obtaining more information from conjoint experiments by best-worst choices*: Highlites found no code that stood out and predicted 8(!) codes, from 61(Human medicine) to 37 (Educational sciences). The correct subject code was not among them (681.3 Computer. Automation).

References

- ECOOM (n.d.). Research disciplines. <https://www.ecoom.be/en/research/research-disciplines>
- Vancauwenbergh, S., and Poelmans, H. (2019). The creation of the Flemish research discipline list, an important step forward in harmonising research information. *Procedia Computer Science*, vol. 146, pp. 265–278.

Yotta, D.B. (2026). Multi-Language Programmer's Guide.
<https://docs.yottadb.com/MultiLangProgGuide/index.html>
Highlites project: <https://gitlab.com/AlainDsc/highlites>