Beyond spell checking - what else can we check automatically?

By Tibs / Tony Ibbs (they / he)

A talk to be given at WtD Prague 2022

Contents

Abstract

Writing documentation is hard, and spotting errors in that documentation is harder. Luckily, if we're working in a docs-as-code environment, we can apply some of the techniques that programmers have been using for a long time, in particular *linting*, the automated checking for common errors or stylistic problems.

In this talk, I shall go through some of the types of check it is possible to make, including spell checking (it's still important), suggesting replacements for words/phrases (`multi-cloud` instead of `multicloud`), "if this then that" rules (remember to expand an acronym on first use) and targeting checks more closely using simple NLP (Natural Language Processing)

I'll cover the use of pre-prepared styles, and I'll give some consideration as to how linting can be "plumbed in" to the review process.

At the end I'll demonstrate how to implement some of these techniques, using the example of Aiven's open source developer documentation. I'll make sure to include my favourite check, for the correct usage of ® on product names.

Introduction

> Note
>
> *This started out as the "Who and Why" section of the proposal. Some of it can be re-used as the "Introduction", but it needs a good pruning!*

As a software developer until the end of 2021, I've been used to both automated checking of source code (linting) and also code review. Both are valuable in different ways, neither is a substitute for the other.

I'm sure that all documentarians understand the value of having another person read over their text and make comments. I want to show that there is value in automatic checking as well, beyond spell checking and "grammar checking".

The thing that programmers learnt was that surprisingly simple rules can give useful results. I'll look at a sequence of such relatively simple rules that are applicable to text, and how they can be used to detect common problems.

By the end, at least one of my example use cases should cause each audience member to go "Aha! that sounds useful".

I shall definitely be pointing out that it's not necessarily a requirement to write your own rule sets - that there are typically pre-bundled sets of rules one can adopt, and for different purposes.

While we use Vale as our text linting tool, the talk is not about Vale, but about the general techniques of linting documentation, and the types of check that one might make. Vale will only be relevant torwards the end, when talking about how we use these techniques in our own environment, and specifically in our github review process.

Origins of "linting"

`lint` was the name of a program written in 1978 to find common errors and stylistic problems in C code, and it is indeed named in analogy with pulling bits of fluff off fabric. Classically, linting programs don't actually *understand* the programming language they're analysing - they use a set of heuristics and rules to recognise common patterns that are likely to be mistakes. That same approach can be applied to our documentation, and it can be surprisingly powerful.

In summary, we're after simple checks, that can be fast, and give good results.

Text is *not* code - code has rigorous restrictions that do not apply to text. However, that doesn't mean that we can't take the idea of "simple checks applied to great benefit" - the trick is in working out the limits of "simple checks" and "great benefit".

Types of check

So let's work through what sorts of check we might make with a linter, and think about some of the implications.

This is not meant to be an exhaustive list.

I shall illustrate with example error messages - these are not necessarily real, and I don't necessarily know how to produce them.

> Note
>
> *I've definitely taken inspiration from Vale for much of this, but that's partly because I think it has a reasonable set of tests that we can look at.*

Spelling

Example message:

- `'Arglebargle' does not seem to be a word`

Notes from the other night:

Spelling checkers look up "words" from the document in a "dictionary" and report any that are not recognised.

> Note
>
> We're not interested here in how the dictionary is constructed, for all we care it could actually just be a long list of all correct words.

The checking process probably (if it's going to be useful) ignores some things, like sequences of numbers, and most punctuation (the possible exception of possessives / apostrophes?)

An important principle we can already see here is that checking for spelling can only report *mistakes*, things that are wrong. And worse, it's actually *possible* mistakes, things which *might* be wrong. Which in this case means words that are not in the dictionary, which means I'll have problems with the text `There is no such word as "glurble"` (and of course I'll have worse problems if that's not true!).

This is a general principle that applies to all linting - the linting program does not understand the text, it is just applying rules to look for what might be mistakes.

> ### Note
>
> There is a secondary assumption that any checker that could understand the text would either be approximate at best (machine learning?) and almost certainly too expensive to run locally - see caveats about software that talks to the cloud later on.

More notes from late at night:

Spell checking is a big subject! We're definitely not going to address much of that here.

Capitalisation and spell checking: In english we use capitalise a word at the start of a sentence, even if it would otherwise be lower case (although, to be awkward, consider things like `iPhone`). So either a spell checker (and its dictionary) has to have a way to encode that, or it needs to do something like "the dictionary word `tony` (all lower case) matches `tony` and `Tony`, and the dictionary word `Tony` just matches `Tony`, not `tony`". What one does for capitalisation inside a word is definitely beyond the scope of this text.

People, product and company names: Traditional dictionaries will have some proper names in them. What does one do about people's names (should `Tibs` be in the dictionary?), product names (we would clearly want `PostgreSQL` in there) and business names (we definitely think that `Aiven` is a correct spelling). It's normal for spelling checkers to allow additional dictionaries, but should one seperate these from "ordinary" words?

Ignored words: It can be useful to have a list of words that are techincally not correct but will be ignored (for the purposes of spell checking). It can be especially useful if these can also be *phrases*. (Can I actually come up with a good example of this other than `aiven` because of the problem we have with checking `mailto:` items using Vale?)

> ### Vale `spelling`
>
> Looks up words in one or more Hunspell-compatible dictionaries. Supports filters and a file of words to ignore.
>
> *We use this*
>
> Note: uses the dictionary as a word list, but doesn't support all Hunspell capabilities. For instance, it doesn't support KEEPCASE (and `/K`).

Aside: But I really did mean `arglebargle`!

```
For this, we shall invent the term 'arglebargle'.
```

What to do?

- Put up with it

- Mark it up differently (e.g., as "literal" text)
- Configure in the text (`.. lint: off` / `.. lint: on`)
- Configure to ignore `arglebargle`
- Configure to ignore `arglebargle` *in this file / location*

Earlier text:

The problem of false positives

- Should one mark, in the text, that this is not an error?
- If one does that too much, then surely the rule is not useful
- Possible difficulty of fine-grained "ignore this" markup - not so good if it's paragraph level
- Is one saying "ignore all checks", or "ignore specific checks"

Programming linters don't have so much problem with this - marking up a line to ignore is already fairly fine grained in most programming languages. And the tests are generally hard-coded in the linter, so generally have an id, and it's possible to say "ignore just this specific test".

That's a bit harder if we're using a *framework* to define new tests.

So, marking parts of the text as "do not check" - is this a good idea, a sometimes good idea, a useful compromise, or just awful?

Aside: Why auto-correction is not (generally) a (good) thing

Corollary of only being able to spot (things that might be) errors: we can't do automated correction of text, because we'd have too many false positives. (This might not actually be true in certrain well constrained cases, like the `adn` case, but is still probably not worth doing - that particular problem is better addressed in the text editor.)

Repetition

> **Note**
>
> Not in the slides

Example message:

- `'the' is repeated`

We are probably all familiar with the example of:

> What is wrong with this text:
>
> ```
> The cat
> and the
> the dog
> ```

where it is surprisingly hard to spot the repeated `the`.

So it's natural to consider having a test to spot such repetitions. Unfortunately, it can't be a blanket check for *any* repeated words, because there are legitimate phrases that repeat words (`knock knock`, `there there`)

That means that the rule needs to specify which words to check for.

The question is, how often do you actually see this done in real documents, and thus is it worth actually adding a test for it?

> Vale `repetition`
>
> Looks for repetition of its tokens.

Don't say that

Example message:

- `Consider not using 'it is obvious that'`

Examples might include complaining about use of the words `simply` and `obviously`, and the phrase `it is obvious`.

> Vale `existence`
>
> Look to see if particular tokens exist. Supports exceptions.

Use *this* instead of *that*

Example messages:

- Errors:
  - `Use 'and' instead of 'adn'`
  - `Use 'supersede' instead of 'supercede'`
  - `Use 'Aiven for PostgreSQL' instead of 'Aiven PostgreSQL'`
- Warnings (or perhaps "suggestions" would be a better term):
  - `Consider using 'flink' instead of 'flick'`
  - `Consider using 'for instance' instead of 'e.g.'`

((*Rework the following to indicate the distinction between "wrong" and "maybe".*))

Examples might be `adn` -> `and` (that's a relatively common typo) or `supercede` -> `supersede` (a mistake I know I often make). These are basically N-distance fuzziness or `slop` changes, and are often provided as part of indepdenent spellcheckers.

Slightly more complex, we use the (product name) `Flink`, and know (we've seen this happen) that people sometimes type `flick` instead. We don't expect to ever need to use that word in our documentation, so it's reasonable to have a rule suggesting `flick` -> `flink`.

At a previous employer, where many of the staff were in Japan, I was told that abbreviations like `i.e.` and `e.g.` are not necessarily well recognised by Japanese developers. So it could be useful to have rules for `e.g.` -> `for example` and `i.e.` -> `that is`.

A little more complex: when referring to the services we provide, we must be careful not to imply ownership of the products/projects ((*what's the correct term I want here?*)). So we have rules like `Aiven PostgreSQL` -> `Aiven for PostgreSQL`.

Vale `substitution`

Looks for token A and suggests token B instead. Supports exceptions.

"Consider using 'B' instead of 'A'"

We use this, although all our examples are currently treated as errors, rather than suggestions

Aside: Errors versus warnings

An error must be fixed, the document is wrong

A warning is just a warning - a "suggestion"

What do you do after you get a warning?

Aside: Create tests you need, retire them when they're not

If we're creating our own checks, only create ones that actually help, and consider reviewing them periodically to check if that is still true. If the person who always mistypes `adn` leaves the team, then we probably don't still need the error message telling us that `"adn" should be replaced by "and"`.

Too many / too few

Note

Not in slides

Example message:

- `More than 3 commas in sentence`

Vale `occurrence`

Enforces minimum or maximum times a token appears. Supports scope - e.g., `sentence`

One or the other, not both

Example message:

- `Inconsistent spelling of 'center' and 'centre'`

Vale `consistency`

Ensures key and value do not occur in the same scope.

If *this* is present, then we need *that*

Example messages:

- `WHO has no definition`
- `At least one use of 'PostgreSQL' must be marked as ®`

The example that Vale uses is a rule that says that if a word occurs that is 3 or more capital letters (for instance, `WHO`) then there must also be an occurrence of an explanation of that term (so in this case, it would be `WHO (...)` where `...` is allowed to be arbitrary text).

To clarify: it's possible to do a rule specifically saying "if `WHO` occurs then `WHO (<some text)` must also occur", but it's also possible to make a rule saying "if `word of 3 or more A-Z` occurs, then `that same word (<some text>)` must also occur".

Bonus points if the rule can say:

- there must be just one occurrence of the "explanation"
- the explanation must come first (or last, or don't care - ideally one would have the ability to specify all three possibilities)
- the occurrence of *that* (e.g., the explanation) must occur in a particular *scope* - for instance, in body text, in a heading, in a footnote.

We use this for the ® checks ((*either explain here or late...*))

Vale `conditional`

Ensures that if token A is present, then so it token B. Supports exceptions, scope.

Terminology on this one is a bit confusing.

We use this

Aside: word versus token versus ...

What is the unit of what we are checking?

It's not as simple as words, because sometimes we want to test for a phrase.

And even words aren't simple - they can include spaces (well, one can argue that) and definitely some other sorts of punctuation (`see-saw`, `can't`).

The term often used in programming, when parsing texts, is token, and that's not a bad name.

But often one also wants a *pattern* - something that describes the thing to be matched. Typical patterns incude regular expressions (there's a lot to these, and they can get very complicated, but as a simple example, `Tib+s` matches `Ti` followed by one or more b followed by `s`, so `Tibs`, `Tibbs`, `Tibbbs` and so on) and "globbing expressions", where the only "wildcards" are that `?` matches any single character and `*` matches any zero or more characters.

Aside: scope

The ability to take control in which parts of the document structure a check is applied.

For instance: only in *headings* or *footnotes*.

In the context of our ® check, we actually would like to say:

- `Thing` must be used with ® in the first *title* to use the name
- `Thing` must be used with ® in the first non-title to use the name

- First use of `Thing` *must* be with ®, regardless

We may also want to be able to say that if `Thing®` occurs, then **after that** in the document there must be the text "`Thing® is a registered tradmark of Thing industries.`"

(For our Aiven documentation we generally don't want that, as we gather the acknowledgement texts into a common footer, but even so we may have occasional terms that aren't acknowledged in that common footer, and then we would want to be able to say this per-section.)

> Nice use of `scope` in https://github.com/errata-ai/vale/issues/184, which checks in scope "link" for links that have names like `this`.

Capitalisation

Example message:

- `'Badly Capitalised Heading' should be in sentence case`

While this is very useful, it's hard to think of how to make it well specified, easy to understand, and doing what one wants. There are some external rules on this sort of thing, which can be adopted.

Problems: consider `iPhone prices`, `The importance of NASA`, `Remembering Terry Jones`, which are all correctly formed.

Note: The Vale capitalization metrics are *not* necessarily as simple as one might expect. For instance, `$sentence` isn't just "first word must start with a capital, rest must not". This is a Good Thing in practice, if harder to explain. I think any system implementing this is going to have some apparent oddities.

> ### Vale `capitalization`
>
> Checks that the text in the specified scope is capitalized according to the chosen scheme. Supports exceptions, scope.
>
> *We use this*

Looking at the raw text

Example messages:

- `In reStructuredText, one backtick without a role becomes italics`
- `In markdown, two backticks is redundant - did you mean single backtick?`
- `Use reStructuredText link format, not markdown`

It can sometimes be useful to make a rule apply the original raw text, so that the markup can also be inspected.

This is not *necessarily* a separate type of rule - in the Vale sense it's an option that can be specified for rules (i.e., that they can see the markup).

> We work in reStructuredText and in markdown. If one switches back and forth, it's very easy to use the wrong notation. So useful rules might be:
>
> - using the wrong sort of inline link text - `[text](link)` in reST, for instance
> - using the wrong number of backticks for literal text - reStructuredText wants them paired (and uses single backticks for more specialised purposes)
> - markdown doesn't support list items with alphabetic "numbering" (`a.`), but reStructuredText does
>
> Maybe something on limitations, as well:

- Linting `someone@place.io` and:
  - Vale uses `rst2html.py` to produce what it lints
  - sphinx produces different HTML from the same reStructuredText source

  So debugging why `support@aiven.io` complains that `aiven` should be `Aiven` isn't quite as simple as it might be.

  Regardless, the *solution* probably needs a rule that looks at the raw markup (which I hope is reStructuredText and not HTML!)
- Catch use of markdown style links:

  ```
  [words](url)
  ```

  in a reStructuredText document - suggest:

  ```
  `words <url>`_
  ```

For markdown, which Vale supports directly, I'd expect `raw` mode to expose the markdown syntax.

For reStructuredText, which is first tranaslated to HTML and then the HTML is inspected, it's not clear to me whether `raw` means the reStructuredText source or the HTML. I haven't had time to investigare yet.

((*I should probably find out before finishing this talk - but actually it doesn't really matter, because the concept is the same regardless*))

Checking for absence

Example message:

- `Image is missing alt text`

For instance, checking there is `alt` text on images

Not the same as "is zero length" - we want *structural element* occurs zero times, not "*text* occurs zero times" (which would be Too many / too few).

This definitely feels like a good text, but how is it done?

- Is it a test on the raw markup? (which feels a bit too low level)
- Does it require some plugin code? (ditto)

Note that in Vale, things like `alt` and `title` should be checked by default - see https://github.com/errata-ai/vale/issues/59. This doesn't necessarily address how one spots that they are **missing**, though.

Hmm. Checking for the *absence* of something is perhaps a different sort of check - maybe it deserves its own catergory. Or can it be counted under the Too many / too few section? That's really (as phrased above) about the count of a particular token, and this is about the absence of that entity (or even the absence of a scope within a scope).

> ((*It doesn't help to look for an empty token in a scope if that scope is entirely absent - so this *is* probably requiring a scope be present inside another scope. Which is getting a bit meta, it's not surprising if it's not directly supported...*))

Arbitrary metrics

Example message:

- `Try to keep the Flesch-Kincaid grade level (12) below 8`

This is calculated as something like

```
(0.39 * (words / sentences)) + (11.8 * (syllables / words)) - 15.59
```

May mean hardcoded support for named metrics, or may mean a general mechanism for doing arithemetic on the number of tokens according to their type, scope, etc.

- Counting word length distribution, sentence length distribution, etc.

> Vale `metric`
>
> Calculates one of various arbitrary metrics and reports if it is exceeded.

Sentence analysis

Using NLP (Natural Language Processing) to categorise the words in a sentence, and then make rules about their combination.

Example message:

- `Did you mean "cars are" instead of "car's are"`

  (from a rule for checking that a plural is used before `` `are ``, rather than `'s'`)
- `Don't use "like" as an interjection`

NLP can allow limiting checks to particular parts of speech, etc.

- This is when it might be possible to distinguish `they're` / `their` / `there`
- I find this harder to quantify and think about
- I don't intend to spend much time on this in the talk!

> Vale `sequence`
>
> Allows rules that specify a sequence of NLP tokens that may or may not form (be part of?) a sentence.
>
> Example at https://vale.sh/explorer/apos_are/, Detect extraneous apostrophes before 'are'.

Just let me code: Arbitrary script / plugin

> Vale `script`
>
> Write a rule using arbitrary Go code (well, a Go-like scripting language)

This is, in fact, a sufficient if rather minimal mechanism for doing everything, and the plugin approach (here are some pre-prepared plugins, and otherwise write your own) is thus quite common.

The Vale approach of "here is a set of templates for rules at a high level" is rarer, probably because it's harder to come up with the set of templates (both in what that set should be, and also in working out they should be formed, what the user has to enter to use them).

Pre-built or hand-designed

"How to get started"

There are several options, and their applicability will differ according to the tool chosen:

- Adopt a "canned" style or styles, something that already exists that does what you want. Examples include Microsoft or Google styles, or accessibility styles like Alex.
- Start with nothing and build up ones own rules
- Start with "canned" styles and add new rules as necessary.

  (I'm assuming that, in general, one can't say "ignore rule ABC from this canned style", but it's possible some tools also allow this)

What it checks

> Note
>
> This is an important point, but quite likely beyond the scope of the slides, and possibly beyond the scope of this whole document - it's certainly not something to dwell on.

There are a lot of tools that will check plain text, and this includes a variety that run in the cloud.

To use those, you'd first need to remove all the markup, which I assume will make it harder to match error reports to line numbers in the original.

There are some tools that understand particular markup languages - typically markdown or HTML. Some also cope with reStructuredText, AsciiDoc or XML.

Some tools *directly* understand some markups (for instance markdown and HTML), but need to run a subsidiary tool or process to convert other markups into (typically) HTML, so that they can lint that. For most purposes, this will work well enough - there should only be a few occasions when details of the actual raw markup are relevant to checks (checking for things like "header" and so on are a different matter, and will typically still work).

If the program allows hand-written plugins (in Go, Python or whatever) then these may have access to the original file, and that then allows the plugin to do whatever it may need to do.

Available tools

Just a brief overview...

- Vale
- LTeX and LanguageTool
- alex
- proselint
- RedPen
- textlint

Vale

> Vale supports checking in Markdown, HTML, reStructuredText, AsciiDoc, DITA, XML, Org and code (comments / docstrings).

Rules ("styles") are specified via YAML files that build on existing concepts, or (less often) via code in a Go-like language

Various pre-packaged rulesets are available

LTeX and LanguageTool

LTeX provides offline grammar checking of various markup languages using LanguageTool

BibTeX, ConTeXt, LaTeX, Markdown, Org, reStructuredText, R Sweave, and XHTML

English, French, German, Dutch, Chinese, Russian, etc.

New rules for LanguageTool are stored as XML files

alex

alex is designed to "Catch insensitive, inconsiderate writing" in Markdown documents, and offer alternatives

proselint

proselint runs checks on Markdown files

It comes with its own set of checks built in

New checks are written as plugins using Python

RedPen

RedPen validates texts in Markdown, Textile, AsciiDoc, reStructuredText and LaTeX

It supports multiple languages, including English, German, Japanese and Chinese

There is a catalogue of existing validators to choose from, and custom validators can be written as plugins in Java or JavaScript

textlint

textlint supports Markdown and plain text by default, with plugins for HTML, reStructuredText, AsciiDoc, Re:VIEW and Org-mode

There is a catalogue of existing rules, which are installed using `npm`

New rules are written as plugins using JavaScript

What I'd written before

Not attempting a complete overview of the field

See the For reference: particular tools section for links and notes that may be useful here.

Only really interested in things that have a CLI (command line interface) so we can run them at the terminal, and from CI (continuous integration).

For each:

1. does it come with built-in checks,
2. does it come with loadable checks ("packages"),
3. can one write new rules,
4. and if so how (templating and/or using a programming language)
   - alex
   - Vale
   - textlint
   - proselint
   - RedPen

- LanguageTool and LTeX

Plumbing in to docs-as-code

Local checks

In the editor - display messages as you're typing, or on saving

At the command line - run a command to make the checks

Checks before commit

Don't allow `commit` if there are errors

This may be a bit extreme?

Checks before review

Run checks when change are pushed for review

The reviewers can see the results

Forbid merging if there are errors?

Seems more reasonable

On GitHub, use workflows for this

Checks before deployment

Don't deploy if there are errors

Probably a good idea - **if** the previous stages mean this essentially never happens

Plumbing in to CI (continuous integration)

Run the checks automatically when a review is requested (GitHub: PR) or before deploying the documentation

No errors before deployment...

CI (Continuous Integration) - specifically thinking of checking a github PR or equivalent

This essentially add the following requirements (or at least desirables):

- runs as a command line tool
- has a provided workflow or is easy to run in on
- configuration can be stored in the repository being checked, or specified on the command line
- preferably runs *fast*, and/or can run only on the subset of documents that have been changed.
- mustn't add artefacts to the (filesystem), or if it does they should be ignored by git or whatever (this *might* be logs) - I think this is somewhat undesirable anyway
- doesn't need to talk to the cloud

What have I forgotten?

What we do at Aiven

We use Vale

...

The checks we use

`devportal/.vale.ini`

```
# For more information, see ``.github/vale/README.rst``
#
# vale-action (https://github.com/errata-ai/vale-action) recommends
# keeping the vale styles in the `.github` directory.
# Since we have a README, styles, a dictionary, and some tests, we are
# keeping related directories files in `.github/vale`

StylesPath = ".github/vale/styles"

# We do not want to check the content of the following HTML tags
# The defaults are script, style, pre, figure
SkippedScopes = script, style, pre, figure

[*.rst]
BasedOnStyles = Aiven
```

and:

```
$ ls devportal/.github/vale/styles/Aiven/ -w 50
aiven_spelling.yml
capitalization_headings.yml
common_replacements.yml
first_PostgreSQL_is_registered.yml
```

and a variety of other `first_<thing>_is_registered.yml` rules.

Use at the command line

```
$ make spell
```

Use in CI

We use the provided vale-action, the official GitHub action for Vale.

Our `devportal/.github/workflows/lint.yaml` is something like the following:

```yaml
name: Linting
on:
  push:
    branches:
      - master
    tags:
      - '**'
  pull_request:
```

```yaml
jobs:
  prose:
    runs-on: ubuntu-latest
    continue-on-error: false
    steps:
    - name: Checkout
      uses: actions/checkout@master

    - name: Vale
      uses: errata-ai/vale-action
      with:
        files: '["index.rst", "docs"]'
      env:
        GITHUB_TOKEN: ${{secrets.GITHUB_TOKEN}}
```

From the proposal - About me

I've been a software developer since the 1980s, and some form of documentarian almost as long (albeit without the use of the term). I used to recommend TeX, but have been enthusing about reStructuredText since it was created. I gave a talk on the history of markup languages at WtD Prague 2018.

Since the start of 2022 I've been a Developer Educator at Aiven (https://aiven.io), and one of my first tasks was to learn about and extend our use of Vale (https://vale.sh) which we use for linting our open source developer documentation. A particular challenge was writing the rules for appropriate use of ® marks, as it turned out that there was a bug in the relevant part of Vale, now fixed after my first PR to the project.

Notes that may be incorporated elsewhere

HTML is not the only aim?

We should not really assume that HTML is the only output (<smile>)

Problems / implementation difficulties

How to deal with All the markups

- Render into HTML and check that
- This isn't always able to be perfect:

  reStructuredText -> HTML with `rst2html` (standalone), `docutils` (more hands on), but the problem is that Sphinx has extra roles and directives, which rst2html/docutils doesn't recognise, and one can't run Sphinx on just selected files
- Does one allow looking at the raw markup (reST) *and* the HTML (which is also in some sense "raw" markup if it is what is being checked)
- Vale is a framework that comes with some predefined checks, and the ability to load packages of existing checks, but also allows you to define your own (and maybe release them as a package). So you get all the power of that approach, and also the need to mend it yourself if your self-written checks don't work.

Pros and cons of commercial and open source systems, and so on

Warning: contains vast generalisations!

- Commercial systems tend to come with pre-setup checks, so that they work "out of the box". However, that may come at the expense of flexibility.

  They may also need to send the text to tbe checked out into the cloud (where someone else's computer can do powerful stuff that yours might not be able to), with all the security implications that this implies.

- Open source systems are more likely to come as a toolkit that you have to assemble yourself to get any sophisitication. Although pre-packaged setups may be available. It is, however, more likely that you'll be able to make them do new things that no-one else has tried. It's also likely to be easier to contribute if the tool doesn't do quite what you want (normal open source project caveats apply)

- There must surely be closed source but free options? I suppose the spelling and "grammar" checking you get bundled with things like Word probably sort-of counts, as it's not something you pay extra for.

  And browser tools may even simple stuff for you... (that's getting a bit fuzzy)

Interactive versus later checking

Hmm. Running a checker *after* writing (or in CI) versus having it run as you type. Pros and cons. Certain sorts of check could be very irritating (I'm thinking the ® check, perhaps) if they're run during typing. Not all tools support being run as-you-type if you're using a local editor. If you're in a browser, is it using a local service, or a remote? - see comments on cloud and privacy. Of course, not all tools can necessarily be (easily) run in CI. Running in CI means that not everyone needs to setup the checking - this is actually necessary if you're going to allow people to make contributions via (for instance) the GitHub web interface. And if you're going to run it in CI, then it is really optional whether people run it locally. Although, turn and turn again, that brings us back to the warning/error discussion - what should even *show up* in CI. It also allows domain experts to fix things - this can be important for some things (the ® check again).

Thoughts on writing local rules

Arguably, having to write one's own configuration (beyond basic spelling and maybe some very general rules) is always going to be a requirement - only you can know what sorts of mistake occur within the particular domain, and with the particular people, you're working with.

For instance, for us it's worth having a rule to suggest replacing `flick` with `Flink`, because (a) we're very unlikely to use the word `flick`, (b) we do use the product name `Flink` and (c) we've observed this particular misspelling more than once in practice.

Looking at the various available tools, there's something to think about on whether new checks are written via plugins using a programming language, or whether there's some "higher level" abstraction (also) available. This is I think a good thing about Vale.

Can we write a check for "bad uses of `like`"?

From twitter (7 August 2022):

> **David R. MacIver** @DRMacIver
>
> I think I need a linter for my tweets that says "Do you really want to do that?" every time I use the word "like" not as a verb.
>
> Replying to @DRMacIver **Tibs**
>
> ■ Adds to ideas of potential tests for my talk on linting text... (hurriedly points out, I'm not going to *implement* the test, but I am interested in how it would be done...)
>
> **David R. MacIver** @DRMacIver
>
> I'm not sure what the current state of part of speech tagging is like, but assuming it's pretty good it seems like a relatively easy thing to check what part of speach "like" is being used as and complain if it's not a verb.

> @DRMacIver Replying to @DRMacIver and @much_of_a
>
> Hmm although I guess you want more than that now that I think about it, as "a like" is a perfectly valid noun in internet.
>
> Me
>
> ■

For reference: particular tools

What Vale provides

In the following, "token" means a word, phrase or regular expression.

The documentation (https://vale.sh/docs/topics/styles) doesn't always list all of the Keys that apply to each style, so the following is likely to be incomplete on that.

`existence`

> Look to see if particular tokens exist. Supports exceptions.
>
> "Consider not using 'bad phrase'"

`substitution`

> Looks for token A and suggests token B instead. Supports exceptions.
>
> "Consider using 'B' instead of 'A'"
>
> *We use this*

`occurrence`

> Enforces minimum or maximum times a token appears. Supports scope - e.g., `sentence`
>
> "More than 3 commas in sentence"

`repetition`

> Looks for repetition of its tokens.
>
> "'the' is repeated"

`consistency`

> Ensures key and value do not occur in the same scope.
>
> "Inconsistent spelling of 'center'"

`conditional`

> Ensures that if token A is present, then so it token B. Supports exceptions, scope.
>
> Terminology on this one is a bit confusing.
>
> "WHO has no definition"
>
> "At least one 'PostgreSQL' must be marked as ®"
>
> *We use this*

`capitalization`

> Checks that the text in the specified scope is capitalized according to the chosen scheme. Supports exceptions, scope.
>
> "'Badly Capitalised Heading' should be in sentence case"
> *We use this*

Note: The capitalization metrics are *not* necessarily as simple as one might expect. For instance, `$sentence` isn't just "first word must start with a capital, rest must not". This is a Good Thing in practice, if harder to explain.

`metric`

Calculates one of various arbitrary metrics and reports if it is exceeded.

"Try to keep the Flesch-Kincaid grade level (%s) below 8"

`spelling`

Looks up words in one or more Hunspell-compatible dictionaries. Supports filters and a file of words to ignore.

"'Arglebargle' does not seem to be a word"

We use this

Note: uses the dictionary as a word list, but doesn't support all Hunspell capabilities. For instance, it doesn't support KEEPCASE (and `/K`).

`sequence`

Allows rules that specify a sequence of NLP tokens that may or may not form (be part of?) a sentence.

Currently, Vale uses prose, an NLP (Natural Language Processing) library for Go. Documentation for the POS (part of speech) tags is there - I don't know yet if there's a standard for those?

There's work in progress to look at using spaCy (maybe as an optional extra?), which would allow support for other languages.

`script`

Write a rule using arbitrary Go code (well, a Go-like scripting language)

There's also a parallel accept/reject mechanism, which allows listing tokens to accept (add to the exception lists for all styles above) or reject (just complain about immediately). This *looks* as if it is a good alternative to dictionaries, but actually isn't for "reasons" (mainly that "adds to the exception list for all styles", which is a bit of a broad brush).

Some notes on what Grammarly provides

- Spelling and grammar checking.
    - grammar mistakes
    - suggested spelling corrections
    - suggested punctuation corrections
    - with premium, word choice, tone and more.
- Plagiarism check
- Suggestions for synonyms to give better reading
- Tonal analysis (how your text may "sound" to readers)
- Rules for term usage, company name spelling/presentation, etc.
- Snippet library
- Analytics

I spent a little bit of time looking to see if I could find out how to define rules for use in Grammarly, and couldn't find anything.

https://geediting.com/grammarly-review-how-good-is-it-an-editor-weighs-in/ seems to suggest that there's broad-scope customisation per document (to give a general idea of what kind of feedback is wanted for that document).

Big question - does it understand markup? Since it's basically catching key events (what you type), it doesn't really sound like their sort of thing.

Some notes on what LTeX and LanguageTool provide

LTeX provides offline grammar checking of various markup languages using LanguageTool

It currently supports BibTeX, ConTeXt, LaTeX, Markdown, Org, reStructuredText, R Sweave, and XHTML documents.

`brew install ltex-ls`

https://dev.languagetool.org/development-overview is the documentation on how to write new error detection rules for LanguageTool. They're stored as XML files.

Some notes on alexjs

alexjs is a linter for markdown, which aims to catch "insensitive, inconsiderate writing". The source is at https://github.com/get-alex/alex. It can be run from the command line.

The rules it follows are listed at retext-equality and retext-profanities.

Note, that last document necessarily contains offensive terms. It also has some which may not be, like `breast` and `european`, because it's trying to warn about *possible* problems - the retext-profanities README makes this clearer:

> When should I use this?
>
> You can opt-into this plugin when you’re dealing with your own text and want to check for potential mistakes.

One might reference the Scunthorpe problem and the problem of identifying offensive words without (sufficient) context.

The documentation at https://github.com/get-alex/alex does explain how to disable specific checks for particular cases - having to do this is probably inevitable with this sort of tool.

Interestingly, the "profanity" check has 3 levels, according to how likely the offending word is to be a profanity.

Note: the command line tool can be run on markdown, MDX and HTML (ignoring the markup syntax) as well as on plain text.

For use in CI, they recommend using the `--diff` option, which will only report on lines that are changed in a push.

Finally, there are some nice links at the end of the readme at https://github.com/get-alex/alex

The article https://dev.to/meeshkan/setting-up-the-alex-js-language-linter-in-your-project-3bpl talks one throuh getting alex up and running.

There is a Vale plugin for similar checks

Other alternatives to Vale

The Vale documentation mentions `textlint` and `RedPen` as alternatives that handle markdown and reStructuredText (and other things), and `alex` as just handling markdown. It also benchmarks Vale as being faster than its competitors.

See also https://lwn.net/Articles/822969/ (Tools to improve Englist text) from 2020.

- https://textlint.github.io/ - Rules are written as plugins using JavaScript.
- https://alexjs.com/ - "Catch insensitive, inconsiderate writing". There is a Vale plugin for at least some of the same functionality
- http://proselint.com/ and https://github.com/amperser/proselint - Rules are written as plugins using Python

- https://redpen.cc/ (don't confuse with `redpen.<anything-else>` - for imstance, the `.cc` domain appears to use real people to do checking!) and https://github.com/redpen-cc/redpen/ - Looks as if custom validators can be added as plugins in Java or JavaScript

---

Somewhat related - accessibility checking

There's a much bigger world of checking things beyond the text itself, including colour usage, layout, and so on. It clearly overlaps with what we're interested in here, but is beyond the scope of this article.

For instance, https://www.accessguide.io/ aims to give a friendly and useful guide to the WCAG 2.1 (Web Content Accessibility Guidelines).

And we already mentioned alexjs

Possibly useful links

- https://passo.uno/prose-linters-implement-workplace-howto/
- https://www.kolide.com/blog/is-grammarly-a-keylogger-what-can-you-do-about-it (but also points out how valuable (something like) Grammarly is, and not to forget that. Links to LanguageTool as an alternative that can run using a local server
- https://geediting.com/grammarly-review-how-good-is-it-an-editor-weighs-in/ gives a counterpoint - this author is an enthusiactic user
- LanguageTool open source, by default uses the cloud, but can run using a local server
- https://news.ycombinator.com/item?id=32236608 an interesting discussion of LanguageTool on HackerNews. Includes an example of writing rules for it, where the commentator says "The art is trying to writing a rule without too much false positives."
- I have the impression that people trying to enter this space are going for browser and cloud based solutions, and I can understand why, but it still always means privacy concerns. Plus not being able to work offline(!)
- https://opensource.com/article/20/3/open-source-writing-tools from 2020 has some interesting suggestions for open source alternatives to Grammarly - basically `flyspell` in emacs, LanguageTool via its API integration with editors, and the Python `proselint` package for grammar advice and style checking.

License