Available Article

Author's final: This draft is prior to submission for publication, and the subsequent edits in the published version. If quoting or citing, please refer to the proper citation of the published version below to check accuracy and pagination.

Cite as: Bergman, M. K. Testing and Best Practices. in *A Knowledge Representation Practionary: Guidelines Based on Charles Sanders Peirce* (ed. Bergman, M. K.) 295–316 (Springer International Publishing, 2018). doi:10.1007/978-3-319-98092-8_14

Official site: https://link.springer.com/book/10.1007/978-3-319-98092-8

Full-text: http://www.mkbergman.com/publications/akrp/chapter-14.pdf

Abstract: KM poses a couple of differences from traditional scientific hypothesis testing. We offer best practices learned from client deployments in areas such as data treatment and dataset management, creating and using knowledge structures, and testing, analysis, and documentation. Modularity in knowledge graphs, or consistent attention to UTF-8 encoding in data structures, or emphasis on 'semi-automatic' approaches, or use of literate programming and notebooks to record tests and procedures are just a few of the examples where lines blur between standard and best practices.

14

TESTING AND BEST PRACTICES

Builds, in a knowledge environment, should be responsive to the nature of knowledge, open and always changing. This knowledge environment includes the knowledge graph, plus its knowledge bases, and its management and analysis tools. To maintain the integrity of this structure going forward, we must test the structure for consistency and coherence after every batch of updates or changes. This constant requirement demands that the entire structure be re-compiled and tested quickly and frequently. Constant revision is the only correct mindset, subject to user input and scrutiny, for which we need tools and guidance to do so in an intelligent way. As we wrap up our discussion on building a KM system, we need to give equal weight to the practical activities that keep our knowledge structures relevant.

We will start the discussion by introducing two straightforward metrics, from which all of our statistical tests flow.[1] From these we derive many useful and common statistics that are good to know, and easy to calculate. Our approach leverages the knowledge aspects, including good populations of type instances, to continue to improve the quality of the domain representation. Enhanced domain representations improve the subsequent ability to test new candidate representations, all in a virtuous circle. To make these efforts practical, we need scripts for both building the structure and testing its integrity. We want the control of these skills to continue to migrate to knowledge workers. Knowledge is best captured by those discovering it. These guidances then lead us to the question of best practices, especially for the build steps covered in *Chapter 13*. As we wrap up this chapter, we also conclude our discussion of building the knowledge representation system. This chapter completes the stage of the why and wherefore of a KR system, enabling us in the next part to tackle the question of applications and potential practical uses.

A PRIMER ON KNOWLEDGE STATISTICS

Semantics is a funny thing. All professionals come to know that communication with their peers and external audiences requires accuracy in how to express things. Even with such attentiveness, communications sometimes go awry. It turns out that background, perspective, and context can all act to switch circuits at the point of in-

teraction. Despite, and probably because of, our predilection as a species to classify and describe things, all from different viewpoints, we can often communicate with terms and language that convey to others something different from what we intended. Alas! This reality is why, I suspect, we have embraced as a species things like dictionaries, thesauri, encyclopedias, specifications, standards, sacred tracts, and such, to help codify what our expressions mean in a given context. So, yes, while sometimes we are sloppy in language and elocution, many misunderstandings between parties are also a result of the difference in perspective.

When we process information to identify relations or extract entities, to type them or classify them, or to fill out their attributes, we need measures to gauge how well our algorithms and tests work, all attentive to providing adequate context and perspective. These very same measures can also tell us whether our attempts to improve them are working or not. We also use these measures, in turn, to establish effective 'gold standards' and create positive and negative training sets for machine learning. Still, despite their importance, it is not always easy to explain these measures. The truth is, sometimes we don't adequately understand these measures.

Two Essential Metrics, Four Possible Values

In our context, we can see a couple of differences from traditional scientific hypothesis testing.[2] The problems we are dealing with in information retrieval (IR), natural language understanding or processing (NLP), and machine learning (ML) are all statistical classification problems, specifically in binary classification.* For example, is a given text token an entity or not? What type amongst a discrete set is it? Does the token belong to a given classification or not? Binary classification makes it considerably easier to posit an alternative hypothesis and the shape of its distribution. What makes it binary is the decision as to whether a given result is correct or not. We now have a different set of distributions and tests from more common normal distributions. The most common scoring methods to gauge the 'accuracy' of natural language or supervised machine learning analysis involves statistical tests based on the ideas of two essential metrics: negatives or positives, true or false. We can measure both of these metrics by scoring correct 'hits' for predictions compared to a 'gold standard' of known results. This gold standard provides a representative sample of what our actual population looks like, one we have characterized in advance. We can use this same gold standard repeatedly to gauge improvements in our test procedures. I talk more about gold standards at the conclusion of this section.

Statistical tests will always involve a trade-off between the level of false positives (in which a non-match is declared a match) and the level of false negatives (in which an actual match is not detected).[3] Let's see if we can simplify our recognition and understanding of these conditions:

1. *TN / True Negative:* case was negative and predicted negative

* I refer here to statistical classification; clearly, language meanings are not binary but nuanced.

2. *TP / True Positive:* case was positive and predicted positive
3. *FN / False Negative:* case was positive but predicted negative
4. *FP / False Positive:* case was negative but predicted positive.

Combining these thoughts leads to a much simpler matrix, sometimes called a confusion matrix, for laying out the true/false, positive/negative characterizations:

Correctness	*Test Assertion*	
	Positive	*Negative*
True	**TP** True Positive	**TN** True Negative
False	**FP** False Positive	**FN** False Negative

Table 14-1: Two Essential Metrics, Four Possible Values

As we can see, 'positive' and 'negative' are simply the assertions (predictions) arising from our test algorithm of whether or not there is a match or a 'hit.' 'True' and 'false' merely indicate whether these assertions proved correct or not as determined by gold standards or training sets. A false positive is a false alarm, a 'crying wolf'; a false negative is a missed result. Thus, all true results are correct; all false results are incorrect. More formally, we can now define these four values as:

- **TP** = test assertion is positive and correct; standard provides labels for instances of the same types as in the target domain; manually scored; test identifies the same entity as in the gold standard;
- **FP** = test assertion is positive but incorrect; manually scored for test runs based on the current configuration; test indicates as positive, but deemed not true; test identifies a different entity than what is in the gold standard (including no entity);
- **TN** = test assertion is negative and correct; standard provides somewhat similar or ambiguous instances from disjoint types labeled as negative; manually scored; test identifies no entity, gold standard has no entity; and
- **FN** = test assertion is negative and incorrect; manually scored for test runs based on the current configuration; test indicates as negative, but deemed not true; test identifies no entity, but gold standard has one.

These measures are sufficient to calculate most of the relevant statistics for our knowledge management and representation purposes.

Conversely, we can relate these two metrics to the branch of statistics known as statistical hypothesis testing. This testing is likely the statistics that you were taught in school. In hypothesis testing, we begin with a hypothesis about what might be going on concerning a problem or issue, but for which we do not know the cause or truth. After reviewing some observations, we formulate a hypothesis that some fac-

tor A is affecting or influencing factor B. We then formulate a mirror-image null hypothesis that specifies that factor A does **not** affect factor B; this is what we test using statistical hypothesis testing. The null hypothesis is what we assume the world in our problem context looks like, absent from our test. If the test of our formulated hypothesis does not affect that assumed distribution, then we reject our alternative (meaning our initial hypothesis fails, and we keep the null explanation).

We make assumptions from our sample about the distribution of the population, which enables us to choose a statistical model that captures the shape of assumed probable results for our measurement sample. These shapes or distributions may be normal (bell-shaped or Gaussian), binomial, power law, or many others. These assumptions about populations and distribution shapes then tell us what kind of statistical test(s) to perform. (Misunderstanding the actual shape of the distribution of a population is one of the major sources of error in statistical analysis.) Different tests may also give us more or less statistical power to test the null hypothesis, which is that chance results will match the assumed distribution. Different tests may give us more than one test statistic to measure variance from the null hypothesis.

We then apply our test and measure and collect our sample from the population, with random or other statistical sampling necessary so as not to skew results, and compare the distribution of these results to our assumed model and test statistic(s). We reject the null hypothesis if we observe significant differences from the expected shape in our sample at a high level of confidence. If we reject the null hypothesis, but in fact it was correct, we call that a Type I error, or a false positive (FP), the same as FP in a binary classification. If we accept the null hypothesis, we reject the alternative hypothesis that some factor A is affecting or influencing factor B. However, if we accept a null hypothesis that is not correct, we term that a Type II error, or a false negative (FN), the same as FN in a binary classification. Statisticians often apply common rules for how differences and level of confidence may lead to rejection of the null hypothesis, thereby leading us to accept the alternative hypothesis that factor A is affecting or influencing factor B.

The binary classification TP *v* FP *v* TN *v* FN approach is better than the statistical hypothesis approach because it explicitly recognizes either the sampling method or our test may be in error. Further, the TP *v* FP *v* TN *v* FN approach is also easier to explain and understand.

Many Useful Statistics

Armed with these four characterizations — true positive, false positive, true negative, false negative — we now can calculate nearly all essential statistical measures. Most of these measures also have exact analogs in standard statistics. The first metric captures the concept of *coverage*. In standard statistics, this measure is called sensitivity; in IR and NLP contexts it is called *recall*. It is the fraction of the documents that are relevant to the query that is successfully retrieved. Recall measures the 'hit' rate for identifying true positives out of all potential positives, and we also call it the true positive rate, or TPR:

$$TPR = TP/P = TP/(TP + FN)$$

A high recall value, expressed as a fraction of 1.00 or a percentage, means the test has a high 'yield' for identifying positive results. We measure it as *true positives* divided by all potential positives in the corpus.

Precision is the complementary measure to recall, in that it is a measure of how efficient the system is to make correct identifications from the positive ones. Precision is the fraction of retrieved documents that are relevant to the query. We measure it as *true positives* divided by all measured positives (true and false):

$$precision = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + false\ positives}$$

High precision indicates a high percentage of true positives compared to all positive results. Precision is something, then, of a *quality* measure, which we express as a fraction of 1.00 or a percentage. It provides a positive predictive value, as defined as the proportion of the true positives against all the positive results (both true positives and false positives). So, we can see that recall gives us a measure as to the breadth of the hits captured, while precision is a statement of whether our hits are correct or not. Note also that false positives are a proper focus of attention in test development because they directly lower precision and the efficiency of the test.

One of the preferred overall measures of IR and NLP statistics is the F-score, which is the adjusted (beta) mean of precision and recall. It recognizes that precision and recall are complementary and linked. The general formula for positive real β is:

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

which we can express for TP, FN, and FP as:

$$F_\beta = \frac{(1 + \beta^2) \cdot true\ positive}{(1 + \beta^2) \cdot true\ positive + \beta^2 \cdot false\ negative + false\ positive}$$

In many cases, the harmonic mean is used, which means a beta of 1, which is called also called the F_1 statistic or the F_1 score:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{number\ of\ true\ positives + false\ positives + false\ negatives}$$

However, F1 displays a tension. Either precision or recall may be improved to achieve an improvement in F_1, but with divergent benefits or effects. What is more highly valued? Yield? Quality? These choices dictate what kinds of tests and areas of improvement need to receive focus. As a result, the weight of beta can be adjusted to fa-

vor either precision or recall. Two other commonly used F measures are the F_2 measure, which weights recall higher than precision, and the $F_{0.5}$ measure, which puts more emphasis on precision than recall.

Accuracy is another metric that can factor into our evaluations, though we use it less in the IR and NLP realm. Accuracy is a statistical measure of how well a binary classification test correctly identifies a condition. We calculate it as the sum of true positives and true negatives divided by the total population (TP + FP + TN + FN):

$$accuracy = \frac{number\ of\ true\ positives + number\ of\ true\ negatives}{total\ population}$$

An accuracy of 100% means that the measured are the same as the given values.

All of the measures above only require the measurement of false and true, positive and negative, as do a variety of predictive values and likelihood ratios. We may also calculate relevance, prevalence, and specificity, which use these same metrics in combination with the total population. By bringing in some other rather simple metrics, we can expand this statistical base to cover such measures as information entropy, statistical inference, pointwise mutual information, variation of information, uncertainty coefficients, information gain, AUCs, and ROCs. All of these still bridge from the basic four values that we need to measure of TP, FP, FN, and TN. We may accommodate these additional tests by keeping track of distributions, calculating confidence intervals, tracking joint or conditional distributions, or summing the area under the distribution curve, in addition to our standard four measures.

We can summarize across all of these basic statistical tests on a single chart, courtesy of a template on Wikipedia,[4] for which I have taken some minor liberties. I show this summary chart of IR and NLP statistical tests in *Table 14-2* on the following page.

Working Toward 'Gold Standards'

Academic researchers in natural language processing (NLP) and machine learning (ML) commonly compare the results of their studies to benchmark, reference standards. A *gold standard* is a reference, benchmark test set where we have already scored results, with a minimum (if not zero) amount of *false positives* or *false negatives.* We should also include *true negative* results in a proper gold standard approximate to the likely ratio expected in the overall population to improve overall accuracy.[5] Gold standards that themselves contain false positives and false negatives, by definition, immediately introduce errors, as we noted for Type I and Type II errors above. A skewed baseline makes it difficult to test and refine existing IR and NLP algorithms. Moreover, because gold standards also often inform training sets, errors there propagate into errors in machine learning. The requirement to compare research results to existing gold standards provides an empirical basis for how the new method compares to existing ones, and by how much. Precision, recall, and the combined F1 score are the most prominent amongst these statistical measures.

		True condition				
	Total population	Condition positive	Condition negative	Prevalence = Σ Condition positive/ Σ Total population	Accuracy (ACC) = Σ True positive + Σ True negative/Σ Total population	
Test condition	Predicted condition positive	True positive (TP) Power	False positive (FN) Type I error	Positive predictive value (PPV), Precision = Σ True positive/ Σ Predicted condition positive	False discovery rate (FDR) = Σ False positive/ Σ Predicted condition positive	
	Predicted condition negative	False negative (FN) Type II error	True negative (TN)	False omission rate (FOR) = Σ False negative/ Σ Predicted condition negative	Negative predictive value (NPV) = Σ True negative/ Σ Predicted condition negative	
		True positive rate (TPR), Recall, Sensitivity = Σ True positive/ Σ Condition positive	False positive rate (FPR), Fall-out = Σ False positive/ Σ Condition negative	Positive likelihood ratio (LR+) = TPR/FPR	Diagnostic odds ratio (DOR) = LR+/LR−	F1 score = 2/ ((1/Recall) + 1/Precision))
		False negative rate (FNR), Miss rate = Σ False negative/ Σ Condition positive	True negative rate (TNR), Specificity (SPC) = Σ True negative/ Σ Condition negative	Negative likelihood ratio (LR−) = FNR/TNR		

Table 14-2. Various Statistical Measures

We often refer to specific standards, such as the NYT Annotated Corpus or the Penn Treebank,[6] as gold standards because they have been in public use for some time, with many errors edited from the systems. Vetted standards such as these may have inter-annotator agreements in the range of 80% to 90%. More typical use cases in biomedical notes[7] and encyclopedic topics[8] tend to show inter-annotator agreements in the range of 75% to 80%. While a claimed accuracy of even, say, 95% sounds impressive, applied to a large knowledge graph such as KBpedia, with its 55,000 concepts, translates into 2,750 concept misassignments (actually, the problem is many orders of magnitude greater than that when we include all assertions). That sounds like a lot, and it is. Misassignments of some nature occur within any standard. When they occur, they are sometimes glaringly obvious, like being out of plumb. It is pretty easy to find most errors in most systems. Still, for the sake of argument, let's accept we have applied a method that has a claimed accuracy of 95%. Remember, this is a measure applied only to the gold standard. If we take the high-end of the inter-annotator agreements for domain standards noted above, namely 80%, then we have this overall accuracy of the system:

$$0.80 \cdot 0.95 = 0.76$$

Whoa! Now, using this expanded perspective, for a candidate knowledge graph the size of KBpedia — that is, about 55,000 items — we could see as many as 13,200 concept misassignments (again, orders of magnitude greater for all assertions). Those numbers now sound huge, and they are. They are unacceptable.

A couple of crucial implications result from this analysis. First, we need to take a holistic view of the error sources across the analysis path, including and most especially the reference standards. (They are, more often than not, the weak link in the analysis path.) Second, we want to get the accuracy of reference standards as high as possible. Thus, we can see many areas by which gold standards may need attention:

1. They may contain false positives;
2. They may contain false negatives;
3. They have variable inter-annotator agreement;
4. They have variable mechanisms, most with none, for editing and updating the labels;
5. They may lack sufficient inclusion of true negatives; or
6. They may derive from an out-of-context domain or circumstance.

You should be aware of these potential sources of error to improve test foundations.

An integral part of any knowledge representation or management effort must be to create gold standards for continuous quality improvements. The domain coverage inevitably requires new entity or relation recognizers, or the mapping of new datasets. The nature of the content at hand may range from tweets to ads to Web

pages or portions or academic papers, with specific tests and recognizers from copyrights to section headings informing new learners. Every engagement requires reference standards. One effort might favor instance records over concepts. Creating gold standards efficiently with a high degree of accuracy is a competitive differentiator.

We may use each type and its instances in KBpedia as a training set for learners. We can continue to improve the accuracy of instance assignments for each type by testing shared attributes or neighbors or type inheritance, plus errors fixed after inspection. One key to growing a consistent knowledge graph over time is to apply these virtuous improvements. Once you create a gold standard, you then run your current test regimes against it when you run your same tests against unknowns. Preferably, of course, the gold standard only includes true positives and true negatives (that is, the gold standard is the basis for judging 'correctness'; see confusion matrix above). If it does not, misassignments, when encountered, must be fixed, preferably as part of existing workflows (see *Chapter 12*).

More accurate standards and training sets lead to improved IR and ML algorithms, feeding the virtuous circle in knowledge-based artificial intelligence (KBAI) (see *Figure 4-2*). Continuing to iterate better knowledge bases and validation datasets is a driving factor in improving both the yield and quality from the KBs. KBAI, then, is a practice based on a curated knowledge base eating its tail, working through cycles of consistency and logic testing to reduce misassignments, while continually seeking to expand structure and coverage. Adding and testing structure or mapping to new structures and datasets continually gets easier, and also produces a network effect. These efforts enable us to partition the knowledge structure efficiently for training specific recognizers, classifiers, and learners, while also providing a logical reference structure for adding new data and structure.

We then use this basic structure — importantly supplemented by the domain concepts and entities relevant to the domain at hand — to create reference structures for training the target recognizers, classifiers, and learners. The process of testing and adding structure identifies previously hidden inconsistencies. As corrected, the overall accuracy of the knowledge structure to act in a reference mode increases. Through straightforward SPARQL queries, we can retrieve both positive and negative training sets for machine learning. Clean, vetted gold standards and training sets are thus a critical component to improve our knowledge bases going forward.[9] We need to give much attention to the practice of creating gold standards and training sets because, without it, we are shooting in the dark when we attempt to improve our learners or language analysis.

BUILDS AND TESTING

The implications of working with knowledge bases are clear. KBs are constantly in flux. Single-event, static processing is dated as soon as we run the procedures. The only way to manage and use KB information comes from a commitment to constant processing and updates. Further, with each processing event, we learn more about the nature of the underlying information that causes the processing scripts and

methods to need tweaking and refinement. Without us documenting what we have done with prior processing, it is impossible to know how to tweak next steps to avoid dead-ends or mistakes of the past. KBAI processing cannot be cost-effective and responsive without a memory. We find literate programming, discussed below, an enabler in this process.

Any knowledge management installation may involve multiple input sources, all moving at different speeds of change. We require numerous steps in processing and updating the input information, the 'systems,' if you will, to achieve our artificial intelligence and data interoperability purposes. The artifacts associated with these activities range from functional code and code scripts; to parameter, configuration and build files; to the documentation of those files and scripts; to the logic of the systems; to the process and steps followed to achieve desired results; and to the documentation of the tests and alternatives investigated at any stage in the process. The kicker is that you will need updates to all of these components. Without a systematic approach, you will not easily remember the script code of what you previously did, leading to costly re-discovery and re-work.

Build Scripts

We seek simplicity in our code scripts through modularity and aggressive use of the OWL API. This API gives us the ability to manipulate the graph connections and structure, using direct triple assertions (often in N3 or Turtle). We seek modularity to segregate code for testing and debugging, and because of the use of simpler data input files, again based on triples. We tightly couple the scripting approach with the platform's Web services design, how we set parameters, and how we ingest or export datasets.

We initially require build scripts for installing the apps on the platform and installing other build scripts. Since we recommend open source configurations for the platform, the multiplicity of tools included with the platform can impose installation challenges. Project build utilities such as Maven or Ant can be very helpful here. For a new build, you may need to create local directory structures, backup prior versions, install input knowledge structures, update metadata and any hardwired script references (which, should, over time, evolve to more sophisticated control structures), log setups, and reboots. Your actual build process may take dozens or hundreds of runs as you test changes, errors get generated from various tests (see next section), and then you resolve them. Throughout the entire process of data ingest and error resolution, we strongly recommend you enforce UTF-8 encoding across all knowledge representations.

In our KBpedia experience, we employ a series of testing scripts during builds (next section) to debug the knowledge structure in its current implementation. We build the base knowledge graph (in the case of KBpedia, this is KKO) first. To that, we add the various typologies used for classifying the structure instances. We perform separate build checks against the typologies, particularly the identification of orphan concepts (types) and fragments of types within the typologies. We make modifica-

tions to the basic input files that guide these scripts, often using a triples format. I overviewed the kinds of build steps requiring scripting in *Chapter 13*. These kinds of iterations are what account for the many runs necessary to produce an integral, connected knowledge structure. Though we could make these modifications via an ontology editor such as Protégé, which is still used during inspection to identify the needed changes, by working with the input file structure, we have a more streamlined basis for full, complete build routines. Modifying the input files keeps the build routines simpler and, therefore, repeatable.

We advise securing sufficient memory for the build process. In the case of KBpedia, we recommend a commodity server with a minimum of 8 GB of RAM. More is preferred, though your domain needs may raise or lower memory needs. Once build issues are worked out, and the graph appears complete and consistent, we advise adding further scripts to generate statistics, which we run at this time. You may want to add standard ontology metrics such as concept and assertion counts at this time. You may also invoke more detailed stats or fragments, including graph-wide statistics, at this point. Some of the statistics runs require a census of the knowledge structure involving multiple, repeated SPARQL queries, which can take quite a bit of time to run.

Only deploy the updated knowledge structure when the build scripts run to completion and all tests pass. You may find small projects at the department level require little in the way of formal deployment; systems in enterprise-wide use may require staging through multiple servers with various approval steps before official deployment. If you have complicated deployments, perhaps involving multiple servers to host key platform components, you may need to give these aspects scripting attention. You may need to conform mature installations with broader enterprise workflow steps and procedures.

We have progressed KBpedia through this growth and renewal process uncounted times. Our automated build scripts mean we can re-generate KBpedia on a commodity machine from scratch in about 45 minutes. If we add all of the logic, consistency and satisfiability checks, we can create a new build in about two hours. One of our recent expansions to KBpedia involved reciprocal mapping (see *Chapter 15*) to Wikipedia, and added about 40% new nodes to KBpedia's then-current structure. Remarkably, using the prior KBpedia as the starting structure, we were able to achieve this expansion with even better logical coherence of the graph in a few hundred hours of effort due to our build philosophy and scripts.

Testing Scripts

We invoke various test scripts as an integral part of the build procedure. We apply scripts against platform tools, for coherency and consistency checks, for reference standards used for placements or machine learning, or for general incremental improvements to the KBpedia structure. Each build invokes some structure tests. Standard tests include 1) 'unsatisfied' classes, which lack characterizations sufficient to standards or degree of connectedness; 2) misassigned classes, where subsumption re-

lationships are contradicted; 3) wrong relations in terms of degree or probability of relationship; 4) wrong SuperTypes; 5) missing, new, or misassigned attributes; 6) general ontology checks such as orphans, fragments, or splits; 7) various graph and connectedness measures; 8) OOPS! consistency and completeness checks; 9) Protégé and add-on checks; and 10) others of your choosing. We try to provide common details and organization to error messages and labels, a consistent approach we also try to extend to Web services and tools. Every error type should have an error code and adequate explanatory text.

Incremental builds (with version numbers, even if not released) are the secret to being able to maintain these knowledge structures. Accumulating too many changes between builds can lead to multiple error sources and greater difficulty to debug. Incremental builds surface errors quickly and fewer at a time. Still, we may require various build runs before we fix all errors. We only release builds that pass all tests, when we assign a new version number.

We can train using 'dirty' training bases that have embedded error no better than the quality of their inputs. If we want to train our knowledge applications with Dick and Jane reader inputs, too often in error to begin with, we will not get beyond the most basic of knowledge levels. We need clean reference standards. On average, we can create a new reference standard for a given new type in 20-40 labor hours. Specifics may vary, but we typically seek, at a minimum, about 500 true positive instances per standard, with 20 or so true negatives. This criterion is a minimum for a reference standard. For machine learning purposes, more is better. We could conceivably lower the requirement for a reference standard below 500 true positive instances as we see the underlying standards improve. We are not seeking definitive statistical test values but a framework for evaluating different parameters and methods. In most cases, we have seen our reference sets grow over time as new wrinkles and perspectives emerge that require testing.

In all cases, our most critical success factor is to engage users, the knowledge workers and managers, in manual review and scoring of the reference standards. We document and train this process so that we may repeat and refine it. User analysts understand and detect patterns that then inform improved methods. We believe clean, vetted training sets and reference standards that move toward 'gold' ones are essential to any KM/KR project.

Literate Programming

The only sane way to tackle knowledge bases at these structural levels is to seek consistent design patterns that are easier to test, maintain and update. Open world systems must embrace repeatable and mostly automated workflow processes, plus a commitment to timely updates, to deal with the constant, underlying change in knowledge. Code and scripts do not reside in isolation. We need to explain the operation of the code to others so they may fix bugs or maintain it. If the software is a processing system, we learn much from testing and refinement, which we should document for subsequent iterations. We must install and deploy our systems. We need to

update libraries and languages frequently for security and performance reasons; we need to update executables and environments as well. When we update systems, we need to run tests for expected performance and accuracy. The severity of some updates may require revision to whole portions of the underlying systems. New employees need tech transfer and training, and managers need to know how to take responsibility for the systems. These are all needs that literate programming can help support.

Literate programming is a style of writing code and documentation first proposed by Donald Knuth. In any aspect of a project that uses code or scripts — tests, configurations, installations, deployments, maintenance, or experiments — the developers or users write narratives and documentation to accompany it. The documentation should be robust by explaining what it is, the logic of it, and what it is doing and how to exercise it. This documentation far exceeds the best-practices of inline code commenting. Literate programming narratives might provide background and thinking about what is being tested or tried, design objectives, workflow steps, recipes, listings of data or discussions of datasets, or whatever. The style and scope of documentation are similar to a scientist's or inventor's lab notebook. Indeed, the breed of emerging electronic notebooks, combined with REPL coding approaches, which allow the embedding of live code demos within notebooks, now enable interactive execution of functions and visualization and manipulation of results, including supporting macros. Thus, we can include working demos and code in-line with our narratives.

Notebook systems that support literate programming, such as Org-mode, can 'tangle' their documents to extract the code portions for compilation and execution. They can also 'weave' their documents to extract all of the documentation in the code now formatted for human readability, including using HTML. Some electronic systems can process multiple programming languages and translate functions. Some electronic systems have built-in spreadsheets and graphing libraries, and most open-source systems can be extended (though with varying degrees of difficulty and in different languages). Some of the systems interact with or publish Web pages.

Leading notebook software includes the iPython Notebook (Jupyter), Org-mode, Wolfram Alpha, Zeppelin, Gorilla, and others. Literate programming requires a focused commitment. The objective of programmers should not be solely to write code but to write systems that can be used and re-used to meet desired purposes at an acceptable cost. Documentation is integral to that objective. Experiments should be documented, codified, and improved. A lines-of-code (LOC) mentality is counter-productive to effective software for knowledge purposes. Literate programming is the most conducive workflow to achieve these ends, with notebooks as the medium for tracking and training work tasks.

One question is what language to use for the literate programming or scripts. Lisp (defined as a *lis*t *p*rocessing language) is one of the older computer languages around, dating back to 1958, and has evolved to become a family of languages. 'Lisp' has many variants, with Common Lisp one of the most prevalent, and many dialects that have extended and evolved from it. Most recently our scripting choice has been Clojure, a modern language based on Lisp, but able to run in the Java virtual machine (JVM),

which makes integrating with existing Java tools much easier. In the context of knowledge management and semantic uses, fully 60% of existing applications can now interoperate with Clojure apps, an instant boon for leveraging many open source capabilities. Java gives us certain advantages, including platform independence and the leverage of debugging and profiling tools, among others. Clojure is a functional programming language, which means it has roots in the lambda calculus and functions are 'first-class citizens.' Functions can pass as arguments to other functions, and return as values, or assign as variables in data structures. These aspects make the language well suited to mathematical manipulations and the building up of more complicated functions from simpler ones. Because of Clojure's REPL (read-event-print-loop) abilities, we can interpret code immediately as we execute instructions at the time of input, leading to a very dynamic and responsive code-development and -testing environment, also well suited to literate programming.

Alternatives like Scheme, Erlang, Haskell, or Scala offer some of the same JVM benefits. Further, tooling for Clojure is still limited, and it requires Java to run and develop. Even with extensions and DSLs, learning Lisp's mindset may be awkward for some. The point here is not to point to a specific language alternative but to enumerate the kinds of evaluation criteria that may go into such a software decision. External factors, too, such as popularity and skills knowledge, certainly can and should enter into language decisions.

As a summary observation, a knowledge management project brings substantial *technical debt*, defined as the overhead and overlooked consequences of adopting a given technological solution, and then needing to develop, stage, manage, and use it. Technical debt is broader still for knowledge management projects because all aspects of the source knowledge are dynamic. Keeping current with changes is a positive thing, and no responsive KM solution would last long without it. Literate programming captures all of these dynamics.

SOME BEST PRACTICES

We have discussed at length build components and practices over the past five or six chapters. While we have not been prescriptive, since techniques and tools are continually improving, we have tried to cover the major steps and background that goes into building a knowledge representation and management platform. We have also discussed the approaches for building the knowledge structures and graphs upon which these systems run. As we wrap up these discussions, let me recount some of the best practices we apply in these steps. We have learned most of these best practices from client deployments in areas such as data treatment and dataset management, creating and using knowledge structures, and in testing, analysis and documentation.

No bright line separates recommended steps and best practices, so we have touched upon many key arguments already in our presentation. Modularity in knowledge graphs, or consistent attention to UTF-8 encoding in data structures, or

the emphasis on 'semi-automatic' approaches, or the use of literate programming and notebooks to record tests and procedures, are just a few of the examples where lines blur between standard and best practices. The key point is that best practices are also an integral part of doing standard tasks right.

Data and Dataset Practices

We have emphasized the importance of using only one or a few internal canonical forms for representing our data, including the importance of testing and ensuring we maintain UTF-8 encoding throughout. UTF-8 is important to maintain multi-language capabilities and the uniform treatment of different language character sets and accents. We have noted the use of basic triple assertions (often in N3 or Turtle) for use in our data transfer protocols. We have also noted the importance of using language tags for all of our labels as one means to promote multi-lingual use and internationalization (sometimes referred to as *i18n*). We want to add on to these points in this section by pointing out best practices in dataset packaging and the use of linked data.

Dataset Best Practices

Datasets are one of the fundamental dimensions for organizing content within our recommended design. Some consideration needs to go into how best to bound these structures. The first consideration relates to the domain, or the scope of the dataset: What is the applicable scope or business purpose of this information? It is best to think of this question from a perspective of access, which is, after all, the most pragmatic way to think of it. We also want to capture the source of the data, and whether it may vary by publisher or source location. For example, provenance or download location or format may be an important distinguishing factor in release or access and may have copyright or royalty implications. That leads to the need to record when we create the data, perhaps adding metadata for whether the data has periodic update or creation times. It may be helpful to distinguish between preliminary data and final data or to segregate data because of workflow or processing considerations. We also need to record all data by type; that is, does the data vary by class or kind? For example, we may find it desirable to keep records about schools separate from records about churches, though at a different level both may be considered buildings. We should be attentive to the data attributes for specific instances, and to use common vocabulary and schema for organizing those characteristics. We may also want to record the completeness of records in regards to attributes or descriptions, since we may want to prefer using better-characterized data in parts of our analysis or may want to flag areas needing future attention. Any of these differences may warrant creating a separate dataset or adding new metadata. Ultimately, these structural considerations of how to organize for the data comes down to possible differences in access rights, both at the record and attribute level. Access differences may warrant altered dataset organization. No limits occur to the number of datasets that may be managed by a given KM instance.

Once you set such boundaries, then think about common attributes or metadata that should be applied. The KBART Recommended Practice is worth review since it suggests a file format and common sense set of metadata fields and formats for transmission of metadata from content providers useful to linked knowledge bases.[10] Still, further, datasets and their records (as all decision or information artifacts in an enterprise) go through natural work stages or progressions. Even the lowliest written document goes through the steps of being drafted, reviewed, characterized, approved, and then possibly revised. Whatever such workflow steps may be, including versioning, may argue to assign some records to a different dataset. Lastly, whatever operational mode you devise, find naming conventions to reflect these variations in your dataset files. These considerations show that datasets are meaningful information artifacts in and of themselves.

Linked Data

Linked data is a set of best practices for publishing and deploying instance and class data using the RDF data model, naming the data objects using uniform resource identifiers (URIs or IRIs), and exposing the data for access via the HTTP protocol, while emphasizing data interconnections, interrelationships and context useful to both humans and machine agents. The challenge is not the mechanics of *linking data*, but the meaning and basis for connecting that data. Connections require logic and rationality sufficient to inform inference and rule-based engines reliably. It also needs to pass the sniff test as we 'follow our nose' by clicking the links exposed by the data.

Most linked data uses a woefully small vocabulary of data relationships, with even a smaller set used for setting linkages *across* existing linked data sets. Linked data techniques are a part of the foundation of overall best practices, but not the whole foundation. We do not, for example, have sufficient and authoritative linking predicates to deal with common 'sort of' conditions. Just as SKOS is a generalized vocabulary for modeling taxonomies and simple knowledge structures, we need a similar vocabulary for predicates that reflect real-world usage for linking data objects and datasets with one another.[11] KBpedia provides this. Practice to date suggests that uncurated, linked datasets in the wild are unlikely useful nor used in combination with other datasets. On the other hand, users desire and readily consume quality linked data. Where you want your KM installation to interact with outside parties, employing linked data is one way to help ensure interoperability.

Knowledge Structures and Management Practices

A central role of ontologies is to describe a 'worldview,' and in specific organizations, this means a shared understanding of the concepts, relations, and terminology to describe the participants' shared domain. In turn, these shared understandings establish the semantics for how to effect communication and understanding within the population of domain users. All of this means that finding ways to identify and agree

upon shared vocabularies and understandings is central to the task of modeling (creating an ontology) for the domain, and it involves practices in collaboration, naming and use of these knowledge structures. Sometimes this perception of shared views is too strictly interpreted as needing to have one and only one understanding of concepts and language. Far from it.

Organizational and Collaborative Best Practices

One of the strengths of ontologies and language modeling within them is we can accommodate multiple terms for the same concept or slight differences in understandings about nearly similar concepts. It is perfectly OK to have differences in terminology and concept understandings so long as those differences are also captured and explicated within the ontology. Embedding collaboration as an implementation best practice is important. We should understand that prior investments in agreed-upon structures and vocabularies deserve respect and we should review them for incorporation. We should capture essential differences, not smudge or obscure them. We want to organize our work teams, and support processes for consensus making, including tools support, so that our teams identify and decide upon terminology, definitions, alternative labels (*semsets*), and relations between concepts. These processes need not be at the formal ontology level, but at the level of the concept graph that underlies the ontology.

Naming and Vocabulary Best Practices

We recommend in our standard build practice to define all concepts and terminology, use semsets to capture alternative ways to name things, and to sometimes treat concepts as either classes or instances. While consensus building and collaboration methods are at the heart of effective ontology building, we should not impose language and concepts by fiat. Try to name all *concepts as single nouns*. Use *CamelCase notation* for these classes (that is, class names should start with a capital letter and not contain any spaces, such as `MyNewConcept`). Name all *properties as verb senses* (so that we may easier read triples); e.g., `hasProperty`. Try to use *mixedCase notation* for naming these predicates (that is, begin with lower case but still capitalize each word after and do not use spaces). Try to use common and *descriptive prefixes and suffixes* for related properties or classes (while they are just labels and their names have no inherent semantic meaning, it is still a useful way for humans to cluster and understand your vocabularies). For examples, properties about languages or tools might contain suffixes such as '`Language`' or '`Tool`' for all related properties. Provide *inverse properties* where it makes sense, and adjust the verb senses in the predicates to accommodate. For example, `<Father> <hasChild> <Janie>` would be expressed inversely as `<Janie> <isChildOf> <Father>`.

Give all concepts and properties a *definition*. We conduct the matching and alignment of things by concepts (not merely labels), which means each concept must be defined.[12] Provide clear definitions (along with the coherency of its structure) to give your ontology its semantics. Remember not to confuse the label for a concept with its

meaning. (This approach also aids multi-linguality). Provide a *preferred label* annotation property that is used for human readable purposes and in user interfaces. KBpedia uses the property of `skos:prefLabel`. Include a class *semset*, robustly harvested and populated, for all concepts and ambiguous entities. Try to assign *logical and short names to namespaces* used for your vocabularies, such as `kbpedia:XXX` or `skos:XXX`, with a maximum of five letters preferred. Enable *multi-lingual capabilities* in all definitions and labels. This language requirement is a somewhat complicated best practice in its own right. For the time being, it means attending to the `xml:lang="en"` (for English, in this case) property for all annotation properties.*

Best Ontology Practices

To my knowledge, the most empirical listing of ontology best practices comes from Simperl and Tempich.[14] In that 2006 paper they examined 34 ontology building efforts and commented on cost, effectiveness and methodology aspects. Various collective ontology efforts also provide listings of principles or best practices. The OBO (*The Open Biological and Biomedical Ontologies*) effort, for example, offers a useful, organized listing of criteria[15] for an exemplary ontology. Their recommendations include their own best practices and to formulate and use a unified methodology. Simperl and Tempich emphasize modularity in their findings, consistent with our standard recommendation. They also recommend metrics for ontology evaluation and tools to extract ontology components from existing data sources, also consistent with our recommended standards.

One best practice we recommend is to embrace a mindset that ontologies can, and should, start small, and may grow incrementally. Another best practice is to keep relationships (predicates) simple at first until you gain fluency. Use simple, well-defined and documented attributes. Aggressively mine and re-use existing knowledge and structure. Knowledge graphs, like knowledge, must be a continuous, dynamic structure, designed for (comparatively easy) updates and automatic builds. Another best practice is to enter items once, and relate them only to direct parents, not more removed upper categories. The upper categories can be inferred, and single, proper placements lead to a cleaner graph structure that is easier to interpret. Be cognizant of the many internal platform needs in workflow management and user interfaces and widgets where administrative ontologies may also contribute. If this mindset if followed, your initial ontology development need not be comprehensive nor expensive. You may grow efforts as you realize benefits. You can adopt pragmatic approaches to testing and then building out a knowledge management system. Starting with a stable structure like KBpedia is likely your most efficient path.

Testing, Analysis and Documentation Practices

The usefulness of a KM platform depends on its accuracy, consistency, and con-

* The Protégé manual[13] is also a source of good tips, especially with regard to naming conventions and the use of the editor.

currency for the domain. We need to ensure these factors remain true as we extend our use of the knowledge and grow the domain further. We need to identify, characterize, and vet concepts. We need to teach this process, and to master tools. We need to assign responsibilities and manage the practice. We find testing and documentation central to this process.

Testing Best Practices

You should embed testing for functionality and testing your knowledge structures for consistency and coherency in all phases and steps of your KM platform. Testing is best when it is incremental and best as part of any build process. You should assign domains and ranges to properties, and invoke reasoners and other tools during update efforts to find inconsistencies. You should test all new concepts and properties at the time of introduction, which you may batch so long as you can manage the increments. Test *external class* assignments because they work to 'explode the domain'[16] and surface other inconsistencies. Use already vetted knowledge bases as reference testbeds when testing the coherence of concepts in a new domain ontology; if the domain ontology describes concepts quite differently than standard practice, or if relationships between concepts are at variance, then you likely have coherency problems. As you work with the system, continue to evolve ontology specifications to include *necessary and sufficient conditions* for complete reasoner testing.

Analytical Best Practices

The two core opportunities of a KM system in data interoperability and knowledge-based artificial intelligence place a premium on analysis, principally in natural language understanding and machine learning. External search engines also fit into this category. In all cases, these analytic tools or learners are third-party applications, with varying degrees of ease-of-use and documentation. Three areas of best practices apply to these external tools. First, it is important to discover, test and select the tools. Second, employ documentation and support structures, such as input data files or run-control specifications, to help make analytic runs repeatable. Third, be cognizant of the technical debt that each adopted tool may bring.

Every new analytical task should begin with a survey of available tools. You should include standard search, plus a search of major code repositories such as GitHub, plus monitoring of technical publications sources (blogs, RSS feeds, arXiv.org, etc.), in your initial investigations. As you research the tools, you may need to migrate from simple spreadsheet listings to detailed characterization of the alternatives. You should download leading candidates, install them, and initially test. This research is a good place to use the notebook paradigm. The choice of tools is fundamental to a KM system built from multiple parts from multiple parties with multiple purposes. Performance and scalability may rapidly become concerns as a KM system grows within larger enterprises.

As tools progress from candidates to provisional, we need to integrate them into the existing platform. Though you likely used support for the internal canonical data

forms as an initial screening criterion, you now need to stage and test data exchange for the tool. As in other areas, you should document steps thoroughly for broader training and adoption. In the evaluation process, every new analytic area, even more so than the specific tools involved, will also incur some degree of technical debt. Scully *et al*. provide one example for a machine learning installation of how to think about various categories of technical debt potentially arising from new tools.[17] Andrew Ng also provides a concise listing of practical machine learning tips.[18]

In varied guises, other analytical tools impose similar or related overhead costs. Search, as we discussed in *Chapter 12*, also poses debt and changes to work procedures. We always have hard-to-quantify benefits and the costs at the more intangible ends of the spectrum when using tools. Our benefits might be qualitative; our costs may hide. If we are to include intangible benefits in the positive column, then we must also be expansive in how we think of the costs of adoption as well.

Documentation Best Practices

Let me emphasize strongly we need to bake documentation into the cake. We need to document every step of our efforts, like is done for good science notebooks but leveraging today's modern electronic versions. We must adequately comment and annotate our ontologies. We should document the entire *ontology vocabulary* via a dedicated system that allows finding, selecting and editing of ontology terms and their properties. We should document ontology maintenance and construction methodologies, including naming, selection, completeness and other criteria. We find wiki documentation useful for training purposes , which is easily updated and maintained. Try to accommodate both standard wikitext and WYSIWYG editors; users have split preferences. Supporting the output of notebook files to wikis or Web pages is a best practice. Also, find large-scale graph and visualization tools so that you can prepare and distribute navigable versions of your knowledge graphs. You may also find other diagrams and flowcharts, including UML diagrams, useful for documenting and training workflows or defining use cases for tools.

While it is not yet seamlessly achievable, try to move toward single-source publishing, where one can author once and then publish selected portions in a variety of formats (HTML, PDF, doc, csv). We want wiki-like environments where multiple authors may contribute, and we have easy collaboration and rollback of versions. Simple import and export versions, such as XHTML or XML, helps facilitate this, though it is still difficult to theme or layout content easily for multiple publication venues. We also want to adopt single-source publishing environments that enable us to characterize and label workflow steps as part of our natural interaction with the content. These systems should allow user-defined steps and labels and rules.

Best practices, like worldviews, depend on the circumstance and the players. We may need to modify broad guidelines that work in general for the specific. In all cases, KR and KM systems tailored to particular needs, and scoped to specific domains, will have their own set of capabilities and configurations. Today's requirements will evolve to different ones tomorrow. The work environment in which you

need to embed these systems and their workflows will vary greatly. That is why it is practical to consider standard and best practices as guidelines, and not prescriptions.

Chapter Notes

1. Some material in this chapter was drawn from the author's prior articles at the *AI3:::Adaptive Information* blog: "Listening to the Enterprise: Total Open Solutions, Part 1" (May 2010); "Using Wikis as Pre-Packaged Knowledge Bases" (Jul 2010); "A Reference Guide to Ontology Best Practices" (Sep 2010); "The Conditional Costs of Free" (Feb 2012); "Why Clojure?" (Dec 2014); "A Primer on Knowledge Statistics" (May 2015); "Literate Programming for an Open World" (Jun 2016); "Gold Standards in Enterprise Knowledge Projects" (Jul 2016).
2. The *Open Semantic Framework* wiki is a contributor to content in this chapter, particularly "NLP and Knowledge Statistics" (http://wiki.opensemanticframework.org/index.php/NLP_and_Knowledge_Statistics) and "Ontology Best Practices" (http://wiki.opensemanticframework.org/index.php/Ontology_Best_Practices).
3. See http://en.wikipedia.org/wiki/Type_I_and_type_II_errors.
4. See http://en.wikipedia.org/wiki/Template:DiagnosticTesting_Diagram.
5. Hripcsak, G., and Rothschild, A. S., "Agreement, the F-measure, and Reliability in Information Retrieval," *Journal of the American Medical Informatics Association*, vol. 12, 2005, pp. 296–298.
6. Miltsakaki, E., Prasad, R., Joshi, A. K., and Webber, and B. L., *The Penn Discourse Treebank*, 2004.
7. Ogren, P. V., Savova, G. K., and Chute, and C. G., "Constructing Evaluation Corpora for Automated Clinical Named Entity Recognition," *Medinfo 2007: Proceedings of the 12th World Congress on Health (Medical) Informatics; Building Sustainable Health Systems*, IOS Press, 2007, p. 2325.
8. Stoyanov, V., and Cardie, C., "Topic Identification for Fine-Grained Opinion Analysis," *Proceedings of the 22nd International Conference on Computational Linguistics-Volume*, 2008, pp. 817–824.
9. Dellschaft, K., and Staab, and S., "On How to Perform a Gold Standard Based Evaluation of Ontology Learning," *The Semantic Web-ISWC*, Heidelberg: Springer Berlin, 2006, pp. 228–241.
10. KBART Phase II Working Group, *KBART: Knowledge Bases and Related Tools Recommended Practice*, NISO, 2014.
11. A vocabulary of linking predicates would capture the variety and degrees to which individuals, instances, classes, and concepts are similar or related to objects in other datasets. This purpose is different than, say, voiD (Vocabulary of Interlinked Datasets), which has as its purpose providing descriptive metadata about the nature of particular datasets.
12. As another commentary on the importance of definitions, see http://ontologyblog.blogspot.com/2010/09/physician-decries-lack-of-definitions.html.
13. Horridge, M., Jupp, S., Moulton, G., Rector, A., Stevens, R., and Wroe, C., *A Practical Guide To Building OWL Ontologies Using Protégé and CO-ODE Tools*, University of Manchester, 2007.
14. Simperl, E. P. B., and Tempich, C., "Ontology Engineering: A Reality Check," *On the Move to Meaningful Internet Systems*, Springer, 2006, pp. 836–854.
15. See http://obofoundry.org/wiki/index.php/OBO_Foundry_Principles.
16. Giasson, F., "Exploding the Domain," *Frederick Giasson*, Apr. 2008.
17. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D., "Hidden Technical Debt in Machine Learning Systems," *Advances in Neural Information Processing Systems*, 2015, pp. 2503–2511.
18. Jalan, K., "How to Improve Machine Learning Performance? Lessons from Andrew Ng" Available: https://www.kdnuggets.com/2017/12/improve-machine-learning-performance-lessons-andrew-ng.html.