Text Ranking and Classification using Data Compression

Anonymous Author(s)
Affiliation
Address
email

Abstract

A well-known but rarely used approach to text categorization uses conditional entropy estimates computed using data compression tools. Text affinity scores derived from compressed sizes can be used for classification and ranking tasks, but their success depends on the compression tools used. We use the `Zstandard` compressor and strengthen these ideas in several ways, calling the resulting language-agnostic technique `Zest`. In applications, this approach simplifies configuration, avoiding careful feature extraction and large ML models. Our ablation studies confirm the value of individual enhancements we introduce. We show that `Zest` complements and can compete with language-specific multidimensional content embeddings in production, but cannot outperform other counting methods on public datasets.

1 Motivation and Overview

The idea of comparing texts using off-the-shelf lossless data compression tools goes back to [1], which linked entropy estimation and using gzip on text with text similarity metrics. Given two strings X and Y, one compresses each of them individually and also the string $X + Y$. Similar strings compress better after being concatenated. An affinity score for X and Y is computed from the three resulting bytesizes. This computation is simple, requires little infrastructure, works for any language, and naturally handles similar words, word forms, typos, etc. It can be easily applied to multi-class classification (e.g., binning news articles by category) and ranking relative to known examples.

On the flip side, performance may be affected by deficiencies of specific compression tools, along with various logistical details of how data compression works in practice. Compression block size and word size (8/12/16 bits), level of effort, and other settings affect the results, whereas the compression headers and dictionaries embedded in each compressed file spoil entropy estimates, especially for small files. In fact, a controversy ensued after the publication of [1] because apparently Naive Bayes worked better than zipping for the language affinity application explored in the paper.

We revisit compression-based text affinity scores [1] because modern compressors are faster than tools from 2002 and produce better entropy estimates. The open-source `zstandard` lossless compressor [2, 3], developed by Yann Collet at Facebook around 2016, produces results not far from those of (slow) arithmetic coding that are considered close to the Shannon bound. The `zstandard` compressor offers a *dictionary interface*, which allows us to improve the approach of [1] and make it more practical, especially for small and medium-sized strings. `Zstandard` can build a compression dictionary and use it to compress many small files, to avoids the overhead of a separate dictionaries. We use the dictionary interface to compress texts to be classified or ranked. This sharpens entropy estimates and improves speed versus compressing concatenated files (where the same file would be concatenated with multiple other files). The resulting text affinity scores can be used as inputs to (multimodal) classifiers and rankers. The simplicity of this language-agnostic method is attractive when building ML platforms, especially for product engineers without ML background.

Under review at ICBINB Workshop @ NeurIPS 2021. Do not distribute.

2 Applications to text ranking and classification

We leverage text affinity estimation in text ranking and classification. For example, given positive and negative examples for 2-class classification, we first build compression dictionaries for each class. As an option, the texts can be normalized by removing punctuation within sentences (but not spaces) and lowercasing the remaining letters (for languages without upper/lower cases, this is a no-op). `Zstandard` can build dictionaries from a set of files without the need to concatenate them, which is useful when working with many small examples.

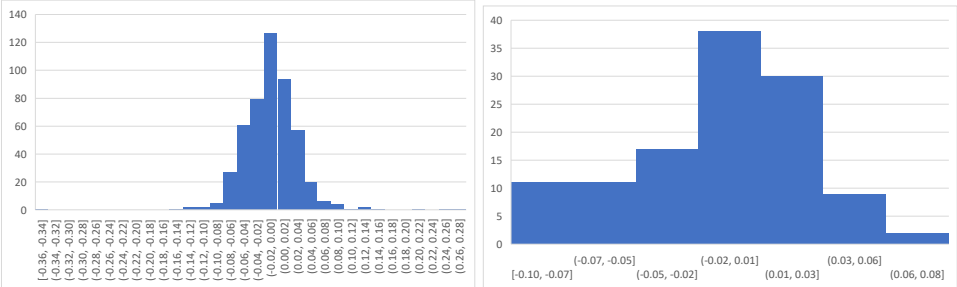
The original approach has several major weaknesses that we are able to address. To illustrate them, consider classifying news articles in topics — Politics, Celebrities, and Sports. Some important words and phrases appear in multiple topics, but with different frequencies, for example, "Arnold Schwarzenegger". For a sufficiently large set of examples, such words compress equally well for each class, and do not contribute useful information. This is particularly detrimental when classifying or ranking short texts. Downsampling the examples would help with common words and phrases, but undermine the handling of rare words and phrases (which would not be compressed for any class). To address this challenge, we use *a set of dictionaries of telescoping sizes* — this way, common words are differentiated by smaller dictionaries and rare words are differentiated by larger dictionaries. Another challenge is the heavier impact of longer words on compression ratios. We address it by *word padding* to fixed length, e.g., "hello" padded to 10 characters becomes "hellohello". We configure `zstandard` to minimize headers in compressed files and, furthermore, subtract the compressed size of an empty string from compressed sizes of evaluated pieces of text. For each evaluated text, for each classification class, we average the byte compression ratio over multiple dictionaries. Subtracting this number from 1.0 produces an affinity score, for which "greater is better". In particular, a sentence that was seen in some class examples may return a value close to 1.0, whereas a sentence in a different script (e.g., Greek vs Cyrillic) would not compress well, resulting in a value close to 0.0. For multiclass classification, we subtract the min class score from all scores. This handles words present in many classes. For ranking applications, affinity scores can be sorted on, to produce an ordering.

3 Implementation and Empirical Evaluation

Our `PyTorch` implementation is based on an untrained `PyTorch` transformer for `Zest`. The Python `zstandard` library was not not supported with `TorchScript`, so we implemented the `Zstandard` interface in C++ as a `TorchScript` operator using the original `C` library. For comparision with other linear models and public text datasets, we used the Python `zstd` library directly. The `Zest` transformer takes in lists of text containing the examples per class to train separate dictionaries, as well as a list of text features for evaluation. It produces affinity scores per class, which can be used separately or combined (e.g., by subtraction) into one score.

Deployment in a production ML Platform. We onboarded `Zest` to a company-internal ML platform that hosts hundreds of ML models for prediction, classification, etc. Using production data, we evaluate the use of `Zest` text affinity scores as additional features to these models, where we can judge features by their importance values in the context of various other features. The specific application discussed in this paper uses text features to rank search results in an internal tool (posts) produced for a given a search query. Some model features provide context: `user_id`, `group_id`, and post `owner_id`. The most useful features in the deployed ML model include the number of characters, query proximity to the post text, and components of a multi-dimensional content embedding. In this study, we check if compression-based affinity scores are comparable in their utility to language-specific content embeddings, which require additional training and cannot handle text in many languages, or mixed-language text. In contrast, `Zest` can be useful where content embeddings are not easily available.

To prepare input for text affinity computation, we feature-engineer the positive and negative post examples to evaluate the post text. For each user on the internal tool, we fetch the 12 most recent posts from the 5 groups with which the user interacted most recently. We split these examples between positive and negative by whether the user viewed them more than once. As most posts end up being negative examples for a user, we added posts that are trending on the internal tool as positive examples. On average, a user has 39 negative examples with 851 characters each and 20 positive examples with 1659 characters each. This way, the number of characters passed into the positive and negative compression dictionaries balances out.



(a) `Zest` score distribution for True examples. (b) `Zest` score distribution for False examples.

Empirical evaluation within a larger model

We ran a baseline GBDT workflow with no additional features, a GBDT workflow with `Zest` scores as features, and a GBDT workflow with 3-gram affinity scores (percentage of matched 3-grams) as features. The `Zest` transformer took roughly 50% longer to compute affinity scores in our workflows compared to the n-gram transformer (both implemented by us). However, the `Zest` classifier code ran much faster with the Python implementation, taking 200-300 seconds to build dictionaries on 25k examples, whereas the transformer would take 40-50 minutes for a few thousands of examples. This could be to due to the memory limits placed on Pytorch transformers when running in production. Feature importance of the top Zest feature was #16 (compared to #201 for the n-gram models), ahead of hundreds of content-embedding dimensions and behind of only 7 of them.

We then tested removing embedding features to see how well `Zest` can fill in the missing information. Based on random removal of embedding dimensions, the new features allow the model to drop 150 dimensions of both the post and query embeddings, while improving normalized entropy (NE) by 7% compared to the baseline model. With half of the embedding features removed, the top `Zest` feature was at #8 with only 4 embedding features ahead. An important distinction between `Zest` and the embedding features is that the Texas SIF embeddings [4, 5] used in the use case can only support English and Spanish, whereas the `Zest` transformer is language-agnostic and can handle mixed-language text. We also checked feature quality of the `Zest` transformer by removing features representing an ID, given that the ID features had high feature importance and could be affecting model quality. After removing the ID features, feature importance jumped from #16 to #9, again being outperformed by only 4 embedding dimensions.

Table 1 compares different workflow runs with different configurations along with the offline results for comparison. There is a distinction in the score distribution between the True and False examples, with most True examples around 0.03 - 0.11, and most False examples around 0-0.07, indicating that the feature values can be strong enough to help with classification on its own.

Standalone comparisons versus linear models

As demonstrated on production data, Zest features can be useful when passed into a model along with other features. To compare `Zest` to other text transformers while avoiding non-text features, to perform ablation studies, and experiment with ensembling, we worked with various public text classification and sentiment analysis datasets. We ran Logistic Regression (LR) trained on count vectorizers as a feature, LR trained on Facebook's `InferSent` sentence embeddings [6], and the multi-class version of `Zest` on a dataset with various categories of news headlines [7] and various sentiment datasets (Stanford Movie sentiment [8] and IMDB movie review sentiment [9]). We ran `Zest` with 1, 2, and 4 telescoping dictionaries each to check performance on the news headlines dataset. The `Zest` model with 4 telescoping dictionaries performed significantly better than the rest. Word padded generally improved accuracy by 0.5-1% based on the dataset. Compared to sophisticated language models — BERT and such, — `Zest` has a much smaller resource footprint and is easier to work with, yet customizable.

The ensemble (averaged) prediction of `Zest` and LR with a count vectorizer performs the best on the news headline categories dataset. However, count vectorizer as a feature worked well with Logistic Regression — it ran faster and outperformed standalone `Zest` by 1-5% in accuracy, depending on the dataset. The sentiment analysis datasets both had an accuracy of ≈87% with LR and CV versus ≈78-80% for Zest, no matter how it was ensembled.

Table 1: Comparisons in a production setting

model	post emb.	query emb.	feat. imp.	NE
Baseline	All	All	—	0.196
`Zest`	All	All	0.29%	0.212
3-gram	All	All	0.06%	0.216
`Zest`	None	None	**3.57%**	0.448
`Zest`	150 dims	150 dims	1.54%	0.189
Baseline	All	All	—	0.173
Baseline	None	All	—	0.161
`Zest`	None	All	**1.14%**	**0.151**
3-gram	None	All	0.17%	0.151

Table 2: Comparisons for `news headlines` [7]

model	size (MB)	train sec.	acc.
NB/CV	3.34	0.170	0.926
LR/CV	1.67	26.53	**0.947**
LR/InferSent	777	**822.3**	0.874
`Zest` 4D	3.04	74.43	0.924
`Zest` 4D/LR-CV	4.71	100.9	**0.951**
`Zest` 2D	1.02	60.49	0.886
`Zest` 2D/LR-CV	2.69	87.02	0.946
`Zest` 1D	0.02	49.83	0.741
`Zest` 1D/LR-CV	1.69	76.36	0.938

The baseline model is a gradient-boosted decision tree (GBDTs). Alternatively, we pass either `Zest` or the 3-gram counter in as additional features to the baseline model. Word-level Count Vectorizers (CV) are also compared to `Zest` and also passed in as features, to a Logistic Regression (LR) or Naive Bayes (NB). `Zest` is ablated with 1, 2 and 4 telescoping dictionaries. Additional sentiment analysis datasets (not shown) exhibit similar trends.

4 Conclusions and Future Work

We have demonstrated model-free language-agnostic text features based on data compression that can be useful to text rankers and classifiers. In addition to using a modern data compression tool, our implementation goes beyond the ideas in [1] by leveraging the *dictionary mode* in `zstandard`, using *telescoping dictionaries* and performing *word padding*. Empirical performance of `PyTorch` transformer on a production ML platform is competitive with that of content-embedding features. However, for some simpler datasets with clear distinction between text classes, count vectorizer shows better ML performance, while being simple and fast.

Evidence from Section 3 allows us to conclude the following:

1. Compression-based methods can be significantly improved by telescoping dictionaries and word padding.
2. Counting methods can achieve strong performance in some cases.
3. Ability to handle synonyms via embeddings offers no advantage on some practical datasets.
4. Different datasets and classification types favor different models.
5. Zest is competitive with trained sentence embeddings in production settings.
6. Zest outperforms other counting methods such as n-grams in production datasets.

Although `Zest` performance is strong and can be an adequate replacement for content embeddings in a production environment, count vectorizers as features consistently performed the best on all public datasets that we used to evaluate `Zest`. Cursory analysis suggests that these datasets allow identifying each class by a small set of words, making explicit counting more accurate than compression-based methods and trained word/content embeddings. However, in a production environment as the one described in Section 3, frequent class-specific words are less common, allowing methods like `Zest` to be on par with competition but with no training. We also evaluated (Markov-chain) language models that estimate the probability that a given text was generated by a given language [10]. In our experiments (not shown), they produce slightly more accurate classifiers than `Zest`, but tend to be more complex and consume greater resources.

Our findings are useful when designing ML platforms that need to deal with text features without asking product engineers to write ML code. They suggest maintaining several lightweight, language-agnostic text features including compression-based ones, and letting the model-training process choose which features are helpful. In many applications, such low-hanging features provide performance that is close to or better than more sophisticated word/content embeddings, while using a much smaller resource and latency footprint. Unlike more sophisticated methods, lightweight methods tend to be language agnostic and can be implemented without language detection.

References

[1] D. Benedetto, E. Caglioti, V. Loreto, “Language Trees and Zipping,” Phys. Rev. Lett., 2002.

[2] Zstandard, *Wikipedia*, `https://en.wikipedia.org/wiki/Zstandard`

[3] Y. Collet and C. Turner, "Smaller and faster data compression with Zstandard," *Facebook Engineering Blog*, 2016, `https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/`

[4] S. Arora, Y. Liang, T. Ma, “A Simple but Tough-to-beat Baseline for Sentence Embeddings,” *ICLR* 2017.

[5] S. Arora, M. Khodak, N. Saunshi, K. Vodrahalli, “A Compressed Sensing View of Unsupervised Text Embeddings, Bags-of-n-grams, and LSTMs", *ICLR* 2018.

[6] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, A. Bordes, “Supervised Learning of Universal Sentence Representations from Natural Language Inference Data," *EMNLP* 2017.

[7] M. Lichman, “UCI Machine Learning Repository," *Irvine, CA: University of California, School of Information and Computer Science* 2013.

[8] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, C. Potts, “Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank", *EMNLP* 2013.

[9] A. Maas, R. Daly, P. Pham, D. Huang, A. Ng, C. Potts, “Learning Word Vectors for Sentiment Analysis", *ACL* 2011.

[10] D. Jurafsky and J. Martin, "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition," *Prentice Hall* 2000.