Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

PyLZJD: An Easy to Use Tool for Machine Learning

Edward Raff[§‡*], Joe Aurelio[‡§], Charles Nicholas[‡]

Abstract—As Machine Learning (ML) becomes more widely known and popular, so too does the desire for new users from other backgrounds to apply ML techniques to their own domains. A difficult prerequisite that often confounds new users is the feature creation and engineering process. This is especially true when users attempt to apply ML to domains that have not historically received attention from the ML community (e.g., outside of text, images, and audio). The Lempel Ziv Jaccard Distance (LZJD) is a compression based technique that can be used for many machine learning tasks. Because of its compression background, users do not need to specify any feature extraction, making it easy to apply to new domains. We introduce PyLZJD, a library that implements LZJD in a manner meant to be easy to use and apply for novice practitioners. We will discuss the intuition and high-level mechanics behind LZJD, followed by examples of how to use it on problems of disparate data types.

Index Terms—compression, complex data, machine learning

Introduction

Machine Learning (ML) has become an increasingly popular tool, with libraries like Scikit-Learn [PVG+11] and others [CG16], [Raf17], [MBY+16], [HFH+09] making ML algorithms available to a wide audience of potential users. However, ML can be daunting for new and amateur users to pick up and use. Before even considering what algorithm should be used for a given problem, feature creation and engineering is a prerequisite step that is not easy to perform, nor is it easy to automate.

In normal use, we as ML practitioners would describe our data as a matrix $\boldsymbol{X}$ that has n rows and d columns. Each of the n rows corresponds to one of our data points (i.e., an example), and each of the d columns corresponds to one of our features. Using cars as an example, we may want to know what color a car is, how old it is, or its odometer mileage, as features. We want to have these features in every row n of our matrix so that we have the information for every car. Once done, we might train a model $m(\cdot)$ to perform a classification problem (e.g., is the car an SUV or sedan?), or use some distance measure $d(\cdot,\cdot)$ to help us find similar or related examples (e.g., which used car that has been sold is most like my own?).

The question becomes, how do we determine what to use as our features? One could begin enumerating every property a car might have, but that would be time consuming, and not all of the features would be relevant to all tasks. If we had an image of a car, we might use a Neural Network to help us extract information or find similar looking images. But if one does not have prior experience with machine learning, these tasks can be daunting. For some types of complex data, feature engineering can be challenging even for experts.

To help new users avoid this difficult task, we have developed the PyLZJD library. PyLZJD makes it easy to get started with ML algorithms and retrieval tasks without needing any kind of feature specification, selection, or engineering from the user. Instead, a user represents their data as a file (i.e., one file for every data point, for n total files). PyLZJD will automatically process the file and can be used with Scikit-Learn to tackle many common tasks. While PyLZJD will likely not be the best method to use for most problems, it provides an avenue for new users to begin using machine learning with minimal effort and time.

The Lempel Ziv Jaccard Distance

LZJD stands for "Lempel Ziv Jaccard Distance" [RN17a] and is the algorithm implemented in PyLZJD. LZJD takes a byte or character sequence x (i.e., a "string"), converts it to a set of substrings, and then converts the set into a *digest*. This digest is a fixed-length summary of the input sequence, which requires a total of k integers to represent. We can then measure the similarity of digests using a distance function, and we can trade accuracy for speed and compactness by decreasing k. We can optionally convert this digest into a vector in Euclidean space, allowing greater flexibility to use LZJD with other machine learning algorithms.

The inspiration and high-level understanding of LZJD comes from compression algorithms. Let $C(\cdot)$ represent your favorite compression algorithm (e.g., zip or bz2), which takes an input x and produces a compressed version $C(x)$. Using a decompressor, you can recover the original object or file x from $C(x)$. The purpose of this compression is to reduce the size of the file stored on disk. So if $|x|$ represents how many bytes it takes to represent the file x, the goal is that $|C(x)| < |x|$.

What if we wanted to compare the similarity of two files, x and y? We can use compression to help us do that. Consider two files x and y, with absolutely no shared content. Then we would expect that if we concatenated x and y together to make one larger file, $x\|y$, then compressing the concatenated version of the files should be about the same size as the files compressed separately, $|C(x\|y)| = |C(x)| + |C(y)|$. But what if $|C(x\|y)| < |C(x)| + |C(y)|$? For that to be true, there must be some overlapping content between x and y that our compressor $C(\cdot)$ was able to reuse in order to achieve a smaller output. The more similarity between x and y, the greater difference in file size we should see. In which case, we could use the ratio of compressed file lengths to tell us how similar

* Corresponding author: raff_edward@bah.com
§ Booz Allen Hamilton
‡ University of Maryland, Baltimore County

the files are. We could call this a "Compression Distance Metric" [KLR04] as shown in Equation 1, where CDM(x,y) returns a smaller value the more similar x and y are, and a larger value if they are different.

$$\text{CDM}(x,y) = \frac{C(x\|y)}{|C(x)|+|C(y)|} \tag{1}$$

The CDM distance we just described gives the intuition behind LZJD. That we can use compression algorithms to measure the similarity between arbitrary files. CDM has been used to perform time series clustering and classification [KLR04]. A large number of compression based distance measures have been proposed [SB06] and used for tasks such as DNA clustering [LCL+04], image retrieval [Tra07], and malware classification [Bor15].

Mechanics of LZJD

While the above strategy has seen much success, it also suffers from drawbacks. Using a compression algorithm for every similarity comparison makes prior methods slow, and the mechanics of standard compression algorithms are not optimized for machine learning tasks. Equation 1 also does not have the properties of a true distance metric[1], which can lead to confusing behavior and prevents using tools that rely on these properties. LZJD rectifies these issues by converting a specific compression algorithm, LZMA, into a dedicated distance metric [RN17a]. LZJD is fast enough to use for larger datasets and maintains the properties of a true distance metric. LZJD works by first creating the compression dictionary of the Lempel Ziv algorithm [LZ76].

```
def lzset(b): #code for string case only
    s = set()
    start = 0
    end = 1
    while end <= len(b):
        b_s = b[start:end]
        if b_s not in s:
            s.add(b_s)
            start = end
        end += 1
    return s

def sim(A, B): # A & B should be set objects
    return len(A & B) / len(A | B)
```

The `lzset` method shows the Lempel compression dictionary creation process. Since LZJD cares about similarity as a direct goal, we do not put in the extra work or code normally required to make an effective compressor. Instead, we simply create a Python set of many different sub-strings of the input sequence `b`. Because the `lzset` method gives us a set of objects, we use the well-known Jaccard similarity to measure how close the two sets are. This is defined in the `sim` method above, and mathematically in Equation 2.

$$J(A,B) = \frac{|A\cap B|}{|A\cup B|} = \frac{|A\cap B|}{|A|+|B|-|A\cap B|} \tag{2}$$

The distance $d(A,B) = 1-J(A,B)$ is a valid metric, and thus provides all the tools necessary to measure the similarity between arbitrary sequences or files. If a and b represent different sequences, their LZJD is computed as:

```
dist = 1.0-sim(lzset(a),lzset(b))
```

While the procedure above will implement the LZJD algorithm, it does not include the speedups that have been incorporated into PyLZJD. Following [RN17a] we use Min-Hashing [BCFM98] to convert a set A into a more compact representation A', which is of a fixed size k (i.e., $|A'| = k$) but guarantees that $J(A,B) \approx J(A',B')$[2]. [RN18] reduced computational time and memory use further by mapping every sub-sequence to a hash and performing `lzset` construction using a rolling hash function to ensure every byte of input was only processed once. To handle class imbalance scenarios, a stochastic variant of LZJD allows over-sampling to improve accuracy [RN17b]. All of these optimizations were implemented with Cython [BBC+11] in order to make PyLZJD as fast as possible.

Vectorizing Inputs

The LZJD algorithm as discussed so far provides only a distance metric. This is valuable for search and information retrieval problems, many clustering algorithms, and k-nearest-neighbor style classification, but it does not avail ourselves to all the algorithms that would be available in Scikit-Learn. Prior work proposed one method of vectorizing LZSets [RN17b] based on feature hashing [WDL+09], where every item in the set is mapped to a random position in a large and high dimensional input (they used $d = 2^{20}$). For new users, we want to avoid such high dimensional spaces to avoid the *curse of dimensionality* [Bel57], a phenomena that makes obtaining meaningful results in higher dimensions difficult.

Working in such high dimensional spaces often requires greater consideration and expertise. To make PyLZJD easier for novices to use, we have developed a different vectorization strategy. To make this possible, we use a new version of Min-Hashing called "SuperMinHash", [Ert17]. The new SuperMinHash is up to 40% slower compared to the prior method, but enables us to use what is known as b-bit minwise hashing to convert sets to a more compact vectorized representation [LK11]. Since $k \leq 1024$ in most cases, and $b \leq 8$, we arrive at a more modest $d = k \cdot b \leq 8,192$. By keeping the dimension smaller, we make PyLZJD easier to use and a wider selection of algorithms from Scikit-Learn should produce reasonable results.

Over-Sampling Data

Another feature introduced in [RN17b] is the ability to stochastically over-sample data to create artificially larger datasets. This is particularly useful when working with imbalanced datasets. Given a value `false_seen_prob`, their approach modifies the inner if statement of `lzset` to falsely "see" a sub-string that it has not seen before. This is a one line change that looks like the following:

```
if b_s not in s
  and random.uniform() > false_seen_prob:
```

By doing so, the set of sub-strings returned is altered. However, the altered set is still true to the data in that every string in the set is a real and valid sub-string from the corpus. This works because the Lempel Ziv dictionary creation is sensitive to small changes in the input, so a few small alterations can propagate forward and cause a number of differences in the entire process. By making the condition random, we can repeat the process several times and get different results each time. This provides additional example diversity that can help train a model. When `false_seen_prob`

1. The properties of a true distance metric are symmetry, indiscernibility, and the triangle inequality.

2. The bottom-k approach is used by default, where one hash $h(\cdot)$ is applied to every item in the set, and the bottom-k values according to $h(\cdot)$ are selected.

= 0, we get the standard LZJD output. To perform oversampling, we recommend using small values like `false_seen_prob` $\leq$ 0.05.

Using PyLZJD

Now that we have given the intuition and described how LZJD works, we show three examples of how PyLZJD performs machine learning, without having to specify a feature processing pipeline. PyLZJD, along with complete versions of these examples, can be found at https://github.com/EdwardRaff/pyLZJD.

To use PyLZJD, at most three functions need to be imported, as shown below.

```
from pyLZJD import digest, sim, vectorize
```

These three functions work as follows:

- `digest(b, hash_size=1024, mode=None, processes=-1, false_seen_prob=0.0)`: takes in (1) a string as data to convert to a digest or (2) a path to a file and converts the file's content to an LZJD digest. If a list is given as input, each element of the list will be processed to return a list of digests.[3]
- `vectorize(b, hash_size=1024, k=8, processes=-1, false_seen_prob=0.0)`: works the same as digest, but instead of returning a list, returns a numpy array representing a feature vector.
- `sim(A, B)`: takes two LZJD digests, and returns the similarity score between two files. 1.0 indicating they are exactly similar, and 0.0 indicating no similarity.

The above is all that is needed for practitioners to use PyLZJD in their code. Below we will go through three examples of how to use these functions in conjunction with Scikit-Learn to get decent results on these problems. For new users, we recommend considering LZJD as a first-pass easy-to-use algorithm so long as the length of the input data is 200 bytes/characters or more. This recommendation comes from the fact that LZJD is compression based, and it is difficult to compress very short sequences. A quick test of LZJD's appropriateness, is to manually compress your data points (as files) with your favorite compression algorithm. If the files compress well, LZJD may work. If the files do not compress well, LZJD is less likely to work.

T5 Corpus Example

The first example we use is a dataset called T5, which has historically been used for computer forensics [Rou11]. It contains 4,457 files that are of one of 8 different file types: html, pdf, text, doc, ppt, jpg, xls, or gif. As a simple first step to using PyLZJD, we will attempt to classify a file as one of these 8 file types. Our code starts by collecting the paths to each file into a list `X_paths`. Creating a LZJD digest for each of these files is simple; call the `digest` function as shown below:

```
X_hashes = digest(X_paths, processes=-1)
```

The processes argument is optional. By setting it to -1, as many processor cores as are available are used. If set to any positive value n, then n cores will be used. A list of digests will be returned with the same corresponding order as the input. The `digest` function will automatically load every file path from disk, and perform the LZJD process outlined above.

For this first example, we will stick to using LZJD as a similarity tool and distance metric. When you want to use distance based algorithms, you want to use the `digest` and `sim` functions instead of `vectorize`. `vectorize` is less accurate and slower when computing distances.

To use LZJD's digest with Scikit-Learn, we need to massage the files into a form that it expects. Scikit-Learn needs a distance function between data stored as a list of vectors (i.e., a matrix X). However, our digests are not vectors in the way that Scikit-Learn understands them, so Scikit-Learn needs to be told how to properly measure distances when using LZJD. An easy way to do this[4], which is compatible with other specialized distance a user may want to leverage, is to create a 1-D list of vectors. Each vector will store the index of its digest in the created `X_hashes` list. Then we create a distance function which uses the index and returns the correct value. While wordy to explain, it takes only a few lines of code:

```
#This will be the vetor given to Scikit-Learn
X = [ [i] for i in range(len(X_hashes))]

#sklearn will give us two vectors a and b from 'X'
def lzjd_dist(a, b):
    #Each has len(a) = 1, so only one value to grab
    #The stored value tells us which index
    #has 'our' digest
    digest_a = X_hashes[int(a[0])]
    digest_b = X_hashes[int(b[0])]
    #Now that we have the digests, compute a
    #distance measure.
    return 1.0-sim(digest_a, digest_b)
```

This is all we need to use the tools built into Scikit-learn. For example, we can perform k-nearest-neighbor classification with cross-validation to see how accurately we predict a file's type.

```
knn_model = KNeighborsClassifier(n_neighbors=5,
    algorithm='brute', metric=lzjd_dist)

scores = cross_val_score(knn_model, X, Y)
print("Accuracy: %0.2f (+/- %0.2f)"
    % (scores.mean(), scores.std() * 2))
```

The above code returns a value of 91% accuracy, where a majority-vote baseline returns 25%. This was all done without us having to specify anything about the associated file formats, how to parse them, or any feature engineering work. We can also leverage other distance metric based tools that Scikit-Learn provides. For example, we can use the t-SNE [MH08] algorithm to create a 2D embedding of our data that we can visualize with matplotlib. Using Scikit-Learn, this is only one line of code:

```
X_embedded = TSNE(n_components=2, perplexity=5,
    metric=lzjd_dist).fit_transform(X)
```

The resulting plot is shown in Figure 1. We see that the groups are mostly clustered into separate regions, but that there is a significant collection of points that were difficult to organize with their respective groups. While a tutorial on effective t-SNE use is beyond our scope, LZJD allows us to leverage t-SNE for immediate visual feedback and exploration.

3. `mode` controls which version of min-hashing is used. `None` for the standard hash, or `"SuperHash"` to use the approach that is compatible with vectorization.

4. This approach is how the Scikit-learn developers recomend using other non-standard distance metrics. For example, the Scikit-learn FAQ shows how to use this approach for doing edit-distance over strings.

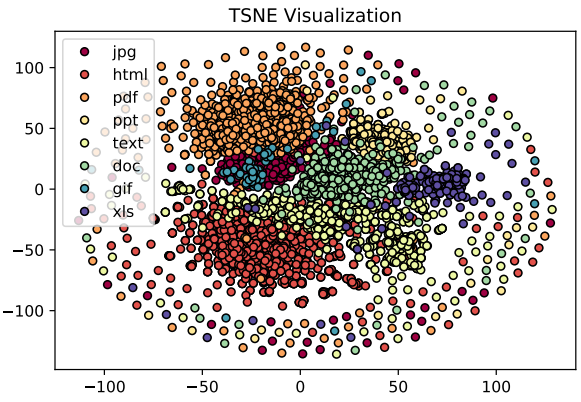

Fig. 1: *Example of t-SNE visualization created using LZJD. Best viewed digitally and in color.*

Spam Image Classification

The prior example used files of varying types, which is similar to the problem domain that LZJD was developed for. In this example, we change the type of data and how we approach the problem. Here, our goal is to predict if an email image attachment is a *spam* image (i.e., undesirable) or a *ham* image (i.e., desirable - or at least, more desirable than spam). This dataset was collected in 2007 [DGEB07], with 3298 spam and 2021 ham images.

Fig. 2: *Example of ham (left) and spam (right) images from the dataset's website.*

We use the `vectorize` function to create feature vectors for each data point. Using `vectorize` instead of `digest` allows us to build models that avoid the nearest neighbor search, which can be slow and cumbersome to deploy. The trade off is we spend more time during the training phase of the algorithm. Doing this with PyLZJD is simple, and the below code snippet handles the work of creating the labels, loading the files, and creating feature vectors, again, without us having to specify anything about the input.

```
spam_paths = glob.glob("personal_image_spam/*")
ham_paths = glob.glob("personal_image_ham/*")

all_paths = spam_paths + ham_paths
yBad = [1 for i in range(len(spam_paths))]
yGood = [0 for i in range(len(ham_paths))]
y = yBad + yGood
X = vectorize(all_paths)
```

Now that we have feature vectors, we can train a Logistic Regression model to predict if a new image is a spam or not. The code to train and evaluate it (by several metrics) is:

```
X_train, X_test, y_train, y_test =
  train_test_split(X, y, test_size=0.2,
    random_state=42)

lgs = LogisticRegression(class_weight='balanced')
lgs.fit(X_train, y_train) #training our model

predicted = lgs.predict(X_test)

fpr, tpr, _ = metrics.roc_curve(y_test,
  (lgs.predict_proba(X_test)[:, 1]))
auc = metrics.auc(fpr, tpr)
print("Accuracy: %f" %
  lgs.score(X_test, y_test))
print("Precision: %f" %
  metrics.precision_score(y_test, predicted))
print("Recall: %f" %
  metrics.recall_score(y_test, predicted))
print("F1-Score: %f" %
  metrics.f1_score(y_test, predicted))
print("AUC: %f" % auc)
```

This produces an accuracy of about 94.6%, and an AUC of 98.7%. In the above code snippet, we included the `class_weight` parameter to address class imbalance in the data. There are more examples of spam images, which can bias a model toward calling most inputs "spam" by default. Using a 'balanced' class weight re-weights the data as if there was an equal number of examples of each class. With PyLZJD, you can perform a special type of over-sampling to help further reduce this impact and improve accuracy. Here is a simple version of this ability:

```
paths_train, paths_test, y_train, y_test =
  train_test_split(all_paths, y,
    test_size=0.2, random_state=42)

X_train_clean = vectorize(paths_train)
X_train_aug = vectorize(paths_train*10,
  false_seen_prob=0.05)
X_test = vectorize(paths_test)
```

In this code, `X_train_clean` constructs the training data in the normal manner. Alternatively, `X_train_aug` has over-sampled both the spam and ham training data 10 times. Normally, this would create 10 copies of the same vectors and have no impact on the solution learned. But, we added the `false_seen_prob` flag, which alters how the `lzset` is constructed: this flag turns on the stochastic component and you get a different result every call. We get a variety of different (but realistic) examples for each datapoint. If we train a new logistic regression model on this data, we get improved results (Table 1).

TABLE 1: *Results on training a Logistic Regression model for spam image detection. Over-sampled scores show results when 'false_seen_prob' is used.*

Metric	Score	Over-sampled Score
Accuracy	0.946	0.957
Precision	0.950	0.954
Recall	0.966	0.979
F1-Score	0.958	0.966
AUC	0.987	0.992

LZJD won't always be effective for images, and convolutional neural networks (CNNs) are a better approach if you need the best possible accuracy. However, this example demonstrates that LZJD can still be useful, and has been used successfully to find slightly altered images [Fj]. This example also shows how to build a more deployable classifier with PyLZJD and tackle class-imbalance situations.

Text Classification

As our last example, we will use a text-classification problem. While other methods will work better, the purpose is to show that LZJD can be used in a wide array of potential applications. For this, we will use the well-known 20 Newsgroups dataset, which is available in Scikit-Learn. We use this dataset because LZJD works best with longer input sequences. For simplicity we will stick with distinguishing between the newsgroup categories of 'alt.atheism' and 'comp.graphics'. An example of an email from the later group is shown below.

> By '8 grey level images' you mean 8 items of 1bit images? It does work(!), but it doesn't work if you have more than 1bit in your screen and if the screen intensity is non-linear.
>
> With 2 bit per pixel; there could be 1*c_1 + 4*c_2 timing, this gives 16 levels, but they are linear if screen intensity is linear. With 1*c_1 + 2*c_2 it works, but we have to find the best compinations -- there's 10 levels, but 16 choises; best 10 must be chosen. Different compinations for the same level, varies a bit, but the levels keeps their order.
>
> Readers should verify what I wrote... :-)

When a string is not a valid path to a file, PyLZJD will processes the string itself to create a digest. This simplifies working with strings, and getting results is as easy as:

```
X_train = vectorize(newsgroups_train.data)
X_test = vectorize(newsgroups_test.data)

clf = LogisticRegression()
clf.fit(X_train, newsgroups_train.target)

pred = clf.predict(X_test)
metrics.f1_score(newsgroups_test.target,
    pred, average='macro')
```

With the above code, we get an F_1 score of 83%. Using Scikit-Learn's TfidfVectorizer achieves an F_1 of 89%. The point here is that with pyLZJD we can get decent results without having to think about what kind of vectorization is being performed, and any string encoded data can be feed directly into the `vectorize` or `digest` functions to get immediate results.

Conclusion

We have shown, by example, how to use PyLZJD on a number of different datasets composed of raw binary files, images, and regular ASCII text. In all cases, we did not have to do any feature engineering or extraction to use PyLZJD, making application simpler and easier. This shortcut is particularly useful when feature specification is hard, such as raw file types, but can also make it easier for people to get into applying Machine Learning.

REFERENCES

[BBC+11] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011. doi:10.1109/MCSE.2010.118.

[BCFM98] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 327–336, New York, NY, USA, 1998. ACM. URL: http://doi.acm.org/10.1145/276698.276781, doi:10.1145/276698.276781.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bor15] Rebecca Schuller Borbely. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques*, pages 1–8, 2015. doi:10.1007/s11416-015-0260-0.

[CG16] Tianqi Chen and Carlos Guestrin. XGBoost: Reliable Large-scale Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. arXiv:1603.02754.

[DGEB07] Mark Dredze, Reuven Gevaryahu, and Ari Elias-Bachrach. Learning fast classifiers for image spam. In *Proceedings of the Conference on Email and Anti-Spam (CEAS)*, 2007.

[Ert17] Otmar Ertl. SuperMinHash – A New Minwise Hashing Algorithm for Jaccard Similarity Estimation. *arXiv*, 2017. arXiv:arXiv:1706.05698v1.

[Fj] João Felipe Pontes Faria-joao. Detecção de Imagens Similares: Aplicabilidade de Ferramentas Software Livre de Hash de Similaridade de Uso Geral. URL: https://www.ipog.edu.br/revista-especialize-online/edicao-16-2018-dez/deteccao-de-imagens-similares-aplicabilidade-de-ferramentas-software-livre-de-hash-de-similaridade-de-uso-geral/.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA Data Mining Software: An Update Mark. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, nov 2009. doi:10.1145/1656274.1656278.

[KLR04] Eamonn Keogh, Stefano Lonardi, and Chotirat Ann Ratanamahatana. Towards Parameter-free Data Mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 206–215, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1014052.1014077, doi:10.1145/1014052.1014077.

[LCL+04] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul M.B. Vitanyi. The Similarity Metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004. arXiv:0111054, doi:10.1109/TIT.2004.838101.

[LK11] Ping Li and Arnd Christian König. Theory and Applications of B-bit Minwise Hashing. *Commun. ACM*, 54(8):101–109, aug 2011. URL: http://doi.acm.org/10.1145/1978542.1978566, doi:10.1145/1978542.1978566.

[LZ76] A. Lempel and J. Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, jan 1976. URL: http://ieeexplore.ieee.org/document/1055501/, arXiv:0009084, doi:10.1109/TIT.1976.1055501.

[MBY+16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D B Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL: http://jmlr.org/papers/v17/15-237.html.

[MH08] Laurens Van Der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[PVG+11] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL: http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

[Raf17] Edward Raff. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *Journal of Machine Learning Research*, 18(23):1–5, 2017. URL: http://jmlr.org/papers/v18/16-131.html.

[RN17a] Edward Raff and Charles Nicholas. An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pages 1007–1015, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3097983.3098111, doi:10.1145/3097983.3098111.

[RN17b] Edward Raff and Charles Nicholas. Malware Classification and Class Imbalance via Stochastic Hashed LZJD. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 111–120, New York, NY, USA,

2017. ACM. URL: http://doi.acm.org/10.1145/3128572.3140446, doi:10.1145/3128572.3140446.

[RN18] Edward Raff and Charles K. Nicholas. Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, feb 2018. URL: https://doi.org/10.1016/j.diin.2017.12.004, arXiv:1708.03346, doi:10.1016/j.diin.2017.12.004.

[Rou11] Vassil Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8:S34–S41, 2011. doi:10.1016/j.diin.2011.05.005.

[SB06] D Sculley and Carla E Brodley. Compression and Machine Learning: A New Perspective on Feature Space Vectors. In *Proceedings of the Data Compression Conference*, DCC '06, page 332, Washington, DC, USA, 2006. IEEE Computer Society. URL: http://dx.doi.org/10.1109/DCC.2006.13, doi:10.1109/DCC.2006.13.

[Tra07] Nicholas Tran. The normalized compression distance and image distinguishability. In Bernice E. Rogowitz, Thrasyvoulos N. Pappas, and Scott J. Daly, editors, *Proc. SPIE 6492, Human Vision and Electronic Imaging XII*, volume 64921D, feb 2007. URL: http://dx.doi.org/10.1117/12.704334, doi:10.1117/12.704334.

[WDL+09] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1113–1120, New York, New York, USA, 2009. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1553374.1553516, doi:10.1145/1553374.1553516.