UNIVERSITY OF AMSTERDAM

MASTER'S THESIS

Light-weight tools for clustering and classification by file compression

Author:

Geert KAPTEIJNS

Supervisor:

Dr. Jan VAN EIJCK

Centrum voor Wiskunde en Informatica

February 2016

Abstract

This thesis explores the applications of the normalized compression distance, a feature-free similarity metric based on file compression, in the fields of clustering and classification. Light-weight tools are introduced and used in experiment. In clustering, it is shown that an agglomerative hierarchical clustering algorithm based on a linkage criterion produces good results. In classification, this thesis reports on experiments with feature extraction based on the normalized compression distance. This has, to my knowledge, not been done in other work. Using this method, a 75% precision rate on gender attribution to written text is achieved, versus 80.1% of the best domain-specific method.

Contents

Chapter 1

Introduction

How do we know that Dutch is more similar to German than it is to French? How do we know that Bob Dylan's music is closer to The Beatles' than it is to Bach's?

Does a computer know?

This thesis concerns a method expressing similarity of data that is feature free: it does not use domain knowledge about the data (for example, word origins or grammar rules in the case of languages, or stylometric features in the case of author attribution.) The method is based on file compression and is rooted in Kolmogorov complexity.

The idea is easy to grasp. If a compressor compresses the concatenation of two files better than it compresses the files separately, it must have found some regularities that appear in both files. This compression gain is used to define a similarity metric that aims to capture the similarity of every dominant feature of the data.

Two main areas of machine learning are covered: clustering and classification of data. I introduce very light-weight code, using only standard libraries, to use this technique in experiment.

Chapter 2 explains the mathematical background of this similarity metric.

In the domain of clustering (chapter 3), this thesis shows that with a bare minimum of code, using compression techniques, you get results that are comparable to domain specific approaches, for example in the field of phylogeny [1]. I use the Python binding to the compression algorithm lzma and `hclust`, a built-in function of the R programming language, for hierarchical cluster analysis. I will compare my results to [2], which also uses compression, but a more elaborate clustering algorithm. I show that my results are comparable, and arguably better, since my clustering preserves a sense of distance

between the clusters, and does not assume that data originates from an evolutionary process.

In the domain of classification (chapter 4), I explore a method based on compression that has been hinted at in [3], but which has, to my knowledge, never been used in experiment. To classify a new piece of data, one determines how well it compresses with *anchors*, preselected pieces of data from each class. I show that in gender attribution to written text, this method works almost as well as the state of the art, domain specific approach, which uses stylometric and content-related features. Again, the code is light-weight: I only use methods from the Python machine learning library scikit-learn [4].

Chapter 2

Normalized compression distance

2.1 Foundations in Kolmogorov complexity

In this chapter we will make explicit the idea of similarity based on file compression. But first, we explain the notion of Kolmogorov complexity. For a complete reference, see [5].

The Kolmogorov complexity of a string x, written $K(x)$, is the length of the shortest program that outputs x.

Intuitively, $111\ldots111$, a string of a million ones, is not very complex. It does not contain much information. Indeed, the Kolmogorov complexity of this string is low. A 27-byte Ruby program produces it:

```ruby
1000000.times { print "1" }
```

I do not claim the Kolmogorov complexity of a string of a million ones is 27 bytes. The true Kolmogorov complexity of a string x cannot be computed in the Turing sense. There is no program that, given x, outputs the (length of) the shortest program that produces x (see proof below.)

The string $011\ldots010$, produced by flipping a fair coin a million times, has, with very high probability, a Kolmogorov complexity close to its own length (by counting the number of different bit strings of each length, you can show that the chance to compress a random string by more than c bits is at most 2^{-c}).

So, printing the literal description may be the best we can do:

```
print "011...010"
```

The Kolmogorov complexity of a string x, given a string y, denoted by $K(x|y)$, is the length of the shortest program that outputs x, given y as input.

2.1.1 The Kolmogorov complexity is not computable

To prove that $K(x)$ is not computable, suppose there is a program `kolmogorov_complexity(x)` that computes $K(x)$. Now consider the following program, that outputs a string x with $K(x) \geq n$.

```
def p(n)
  all_strings_of_length(n).each do |x|
    if kolmogorov_complexity(x) >= n
      return x
    end
  end
end
```

Encoding x with $p(n)$ takes only $c + 2log(n)$ bits: a constant number of bits for the programs `kolmogorov_complexity(x)` and `p(n)`, and $2log(n)$ to encode n. Since there exist strings with arbitrarily large Kolmogorov complexity, this is a contradiction, hence $K(x)$ is not computable.

2.1.2 Information distance

The Kolmogorov complexity is a measure of information content in an individual object. In the same way, in [6] the information distance $E(x, y)$ between two strings x and y is defined as the length of the shortest program that converts x to y and y to x. It is shown that

$$E(x, y) = \max\{K(x|y), K(y|x)\}$$

$E(x, y)$ is an absolute distance. But similarity is better expressed relatively. To illustrate: if two binary strings of length 100 have a Hamming distance of 50 (i.e. they have different bits in 50 positions), they are not very alike. If, on the other hand, two strings of length 10^6 have a Hamming distance of 50, they are very much alike.

In [7], the normalized information distance is defined as

$$\text{NID}(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}} \tag{2.1}$$

and it is shown that $\text{NID}(x, y)$ minorizes every distance $d(x, y)$ up to a negligible additive term, where $d(x, y)$ belongs to a wide class of normalized distances that includes everything remotely interesting.

This means that if two strings are similar according to some distance (be it Hamming distance, overlap distance, or any other), they are also similar according to the normalized information distance. This is why $\text{NID}(x, y)$ is also called *the* similarity distance.

2.2 Approximating Kolmogorov complexity with a real-world compressor

The remarkable properties of the normalized information distance come at the price of incomputability. But, the Kolmogorov complexity can be approximated by real-world (lossless) compression programs like zlib or liblzma. The normalized compression distance [2] is defined as

$$\text{NCD} = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \tag{2.2}$$

where xy is the concatenation of x and y, and $C(x)$ is the length of x, after being compressed by compressor C.

The NCD is central to this work. It is the real-world approximation of the normalized information distance (2.1).

The authors of [2] show that most (if not all) real-world compressors guarantee that the NCD is a quasi-universal similarity metric, by satisfying the following properties:

1. *idempotency*: $C(xx) = C(x)$

2. *monotonicity*: $C(xy) \geq C(x)$

3. *symmetry*: $C(xy) = C(yx)$

4. *distributivity*: $C(xy) + C(z) \leq C(xz) + C(yz)$

2.2.1 A compressor does not exploit all regularity

Since K is incomputable, we have no idea how far off the length given by C is. Consider 31415..., the string consisting of the first 10^9 digits of π. The Kolmogorov complexity of this string is low: a simple program, perhaps exploiting a converging series formula, will produce it. Any real-world compressor C, however, fails to compress this string by even a few bits[1].

In the following chapters (and extensively in [2]) it is demonstrated that, while not every regularity is captured by a real-world compressor, the NCD is still adequate for many applications.

---

[1]A textual representation "31415..." can actually be compressed significantly by almost every compressor, but this is an encoding issue. The file consists only of the bytes 0 through 9, which can be more efficiently encoded using, for example, a system where each decimal number is assigned a four bit code (which is still naive.) For clarity, I do not concern myself with encoding in this chapter. Every finite alphabet can be recoded in binary, so it is customary to only think about binary strings in the literature. Conceptually, nothing changes.

Chapter 3

Hierarchical clustering of data

3.1 Introduction

In this chapter, I cluster vastly different types of data using the normalized compression distance. Using only standard libraries, I introduce a lightweight alternative to the CompLearn Toolkit [8], an open source tool for NCD clustering.

Instead of the hierarchical clustering method used in CompLearn [9], I use hierarchical clustering methods that are in the R standard library. R provides functions like `hclust`, which make it very easy to implement and switch between the most common agglomerative hierarchical clustering methods, and has built-in functionality to plot dendrograms. The clustering module of scikit-learn [4] in Python is also a viable option. To calculate compression distances between files, I use the Python binding to the lzma compression algorithm.

I show that using these simple clustering methods, insightful results can be obtained that are comparable to [2]. The authors of [2] focus on constructing a tree that is globally optimal, in an evolutionary sense, with an approximation to an NP-hard problem, while my methods use standard agglomerative clustering methods. The methods in this paper preserve a sense of distance between clusters, which is lost in the method used in [9].

The code [10] is freely available.

3.2 Evolution of placental mammals

To explain the method, I first cluster mitochondrial gene sequences of mammals. I use sequences of the same 24 mammals used in [2], and compare results.

Reconstructing an evolutionary tree has intuitive appeal. It should lend itself well to hierarchical clustering, since species emerge from common ancestors. And, within biology, there is agreement on what the true phylogeny tree is: the brown bear and polar bear are closely related, etc. Only higher up on the tree there is ongoing debate. Do the primates first join with the rodents, or are they more closely connected to the ferungulates?

All materials were taken from the GenBank database [11].

In [1], the authors estimate the likelihood of phylogeny trees based on 12 mitochondrial proteins of 20 placental mammals: rat (*Rattus norvegicus*), house mouse (*Mus musculus*), grey seal (*Halichoerus grypus*), harbor seal (*Phoca vitulina*), cat (*Felis catus*), white rhino (*Ceratotherium simum*), horse (*Equus caballus*), finback whale (*Balaenoptera physalus*), blue whale (*Balaenoptera musculus*), cow (*Bos taurus*), gibbon (*Hylobates lar*), gorilla (*Gorilla gorilla*), human (*Homo sapiens*), chimpanzee (*Pan troglodytes*), pygmy chimpanzee (*Pan paniscus*), orangutan (*Pongo pygmaeus*), Sumatran orangutan (*Pongo abelii*), using opossum (*Didelphis virginiana*), wallaroo (*Macropus robustus*), and the platypus (*Ornithorhynchus anatinus*).

In [2], four more mammals were added: Australian echidna (*Tachyglossus aculeatus*), brown bear (*Ursus arctos*), polar bear (*Ursus maritimus*), and the common carp (*Cyprinus carpio*). The common carp is not a mammal and is used as an outgroup. It should join the phylogeny tree at the very top.

3.2.1 Distance matrix

The distance matrix contains the distances between all pairs of items. Entry i, j is the distance between item i and item j, i.e. $\mathrm{NCD}(i, j)$. The distance matrix for the 24 animals is displayed in table 3.1.

	blueWhale	brownBear	carp	cat	chimpanzee	cow	echidna	finWhale	gibbon	gorilla	graySeal	harborSeal	horse	human	mouse	opossum	orangutan	pigmyChimpanzee	platypus	polarBear	rat	sumatranOrangutan	wallaroo	whiteRhinoceros
blueWhale	0.01	0.72	0.86	0.67	0.78	0.60	0.83	0.24	0.81	0.80	0.67	0.66	0.62	0.78	0.77	0.82	0.81	0.77	0.83	0.70	0.79	0.78	0.83	0.63
brownBear	0.71	0.01	0.89	0.63	0.84	0.72	0.85	0.69	0.84	0.82	0.56	0.56	0.69	0.81	0.80	0.84	0.84	0.82	0.84	0.11	0.82	0.83	0.84	0.64
carp	0.87	0.88	0.01	0.88	0.89	0.87	0.89	0.89	0.89	0.89	0.88	0.88	0.87	0.88	0.86	0.88	0.89	0.89	0.90	0.88	0.88	0.89	0.88	0.88
cat	0.67	0.59	0.87	0.01	0.79	0.68	0.84	0.69	0.79	0.79	0.59	0.57	0.61	0.81	0.77	0.80	0.80	0.80	0.82	0.60	0.75	0.81	0.78	0.59
chimpanzee	0.78	0.83	0.90	0.81	0.01	0.77	0.87	0.82	0.49	0.34	0.80	0.79	0.79	0.29	0.84	0.88	0.47	0.17	0.89	0.83	0.84	0.47	0.86	0.80
cow	0.61	0.73	0.87	0.68	0.78	0.01	0.84	0.61	0.79	0.78	0.66	0.66	0.62	0.81	0.82	0.81	0.77	0.84	0.71	0.76	0.81	0.81	0.81	0.62
echidna	0.84	0.87	0.89	0.85	0.85	0.83	0.01	0.86	0.85	0.86	0.87	0.87	0.84	0.88	0.81	0.81	0.87	0.85	0.55	0.87	0.84	0.86	0.84	0.83
finWhale	0.25	0.72	0.89	0.71	0.81	0.62	0.85	0.01	0.81	0.82	0.70	0.69	0.64	0.81	0.80	0.84	0.80	0.78	0.85	0.72	0.81	0.81	0.85	0.64
gibbon	0.83	0.85	0.90	0.81	0.51	0.81	0.88	0.85	0.01	0.51	0.82	0.83	0.81	0.50	0.85	0.89	0.53	0.50	0.90	0.85	0.85	0.54	0.87	0.79
gorilla	0.79	0.81	0.90	0.82	0.35	0.79	0.88	0.83	0.50	0.01	0.79	0.82	0.83	0.35	0.83	0.89	0.46	0.35	0.87	0.82	0.82	0.47	0.88	0.80
graySeal	0.69	0.56	0.88	0.60	0.79	0.66	0.84	0.69	0.79	0.80	0.01	0.16	0.64	0.80	0.77	0.82	0.80	0.78	0.85	0.55	0.78	0.80	0.80	0.61
harborSeal	0.67	0.54	0.88	0.58	0.78	0.64	0.85	0.69	0.80	0.80	0.16	0.01	0.61	0.79	0.76	0.83	0.77	0.77	0.84	0.54	0.77	0.77	0.80	0.60
horse	0.63	0.65	0.88	0.63	0.76	0.62	0.85	0.64	0.77	0.79	0.62	0.60	0.01	0.79	0.78	0.81	0.78	0.78	0.83	0.66	0.76	0.79	0.80	0.49
human	0.78	0.83	0.88	0.83	0.30	0.79	0.87	0.80	0.50	0.35	0.82	0.82	0.78	0.01	0.82	0.87	0.46	0.30	0.88	0.83	0.82	0.46	0.86	0.78
mouse	0.77	0.79	0.86	0.76	0.81	0.76	0.81	0.80	0.81	0.82	0.77	0.76	0.75	0.83	0.01	0.78	0.83	0.82	0.83	0.78	0.54	0.83	0.77	0.74
opossum	0.82	0.81	0.87	0.80	0.87	0.81	0.84	0.82	0.87	0.88	0.82	0.82	0.81	0.87	0.80	0.01	0.88	0.87	0.84	0.82	0.82	0.88	0.68	0.83
orangutan	0.80	0.83	0.90	0.82	0.46	0.82	0.88	0.80	0.52	0.46	0.82	0.79	0.83	0.45	0.84	0.88	0.01	0.47	0.89	0.83	0.83	0.25	0.86	0.82
pigmyChimpanzee	0.77	0.81	0.90	0.81	0.17	0.77	0.88	0.81	0.49	0.34	0.82	0.79	0.79	0.29	0.83	0.87	0.47	0.01	0.89	0.81	0.83	0.47	0.86	0.78
platypus	0.83	0.85	0.89	0.86	0.89	0.82	0.56	0.84	0.89	0.87	0.84	0.84	0.87	0.87	0.83	0.82	0.88	0.88	0.01	0.85	0.84	0.89	0.84	0.85
polarBear	0.70	0.10	0.89	0.62	0.81	0.70	0.86	0.71	0.84	0.81	0.56	0.56	0.66	0.82	0.79	0.83	0.82	0.81	0.85	0.01	0.82	0.82	0.82	0.65
rat	0.78	0.82	0.89	0.75	0.80	0.76	0.83	0.78	0.81	0.82	0.79	0.77	0.74	0.82	0.55	0.82	0.81	0.83	0.86	0.82	0.01	0.81	0.84	0.75
sumatranOrangutan	0.78	0.82	0.89	0.80	0.47	0.80	0.86	0.79	0.52	0.47	0.80	0.80	0.77	0.44	0.84	0.87	0.24	0.47	0.88	0.82	0.82	0.01	0.87	0.78
wallaroo	0.81	0.82	0.87	0.80	0.85	0.81	0.85	0.83	0.86	0.86	0.80	0.79	0.81	0.85	0.81	0.68	0.86	0.85	0.83	0.83	0.82	0.87	0.01	0.80
whiteRhinoceros	0.64	0.66	0.89	0.65	0.78	0.64	0.86	0.65	0.78	0.79	0.61	0.61	0.51	0.79	0.78	0.84	0.82	0.79	0.86	0.68	0.77	0.81	0.82	0.01

TABLE 3.1: The NCDs were calculated with `lzma`, the Python binding to the LZMA compression algorithm, with the parameters shown in listing 3.1.

```
{
  "id": lzma.FILTER_LZMA2 ,
  "preset": 9 | lzma.PRESET_EXTREME ,
  "dict_size": 5000000 ,
  "lc": 2,
  "pb": 0,
  "nice_len": 273 ,
}
```

LISTING 3.1: Parameters for compression using the LZMA algorithm.

3.2.1.1 Compression parameters

Listing 3.1 shows the filters used in calculating the normalized compression distances between files. The goal is to compress the files as well as possible, since we want to approximate the true Kolmogorov complexity — but without an unreasonable penalty in speed.

I studied the effect of changing the filters by randomly taking two distinct dna sequences, and calculating the compression ratio of the concatenation, averaged over a thousand runs.

`nice_len` denotes the length after which the LZMA algorithm stops looking for a longer match, and it is set to its maximum value. Setting `nice_len = 4`, the highest default in the xz toolkit, produces a significantly worse compression ratio, as expected.

`lc`, the number of literal context bits, is the number is bits of a byte from the input, starting from the first bit, which is assumed to correlate with the next byte. For example, in English language, a uppercase letter (starting with 010 in US-ASCII) is usually followed by a lower case letter (starting with 011). According to the manpage of xz [12], setting `lc = 4` usually gives the best compression, but may also make compression worse. In the case of dna sequences, the default `lc = 3` proved best.

`pb` affects what kind of alignment is assumed in the uncompressed data. Setting `pb = 0`, which is optimized for ascii, gives a minor improvement over the default, `pb = 2`.

3.2.2 Comparison of clustering methods

In hierarchical clustering, the goal is to build a hierarchy of clusters from a distance matrix. The hierarchy can be displayed as a binary tree.

3.2.2.1 Quartet tree method

The authors of [2] use a clustering algorithm described by them in [9]. The algorithm tries to optimize a global criterion, namely the (normalized) summed weights of all consistent quartet topologies (layouts of groups of four items). In a binary tree, only one of three possible pairings of four items ($ab|cd$, $ac|bd$, $ad|bc$) is *consistent*, in the sense that you can connect the two pairs without crossing paths. The sum of the distances (in this case, NCDs) between the items in the consistent pairs is the contribution of quartet *abcd* to the tree score.

This method works especially well if the items you're trying to cluster result from an evolutionary process, since then (without corruption of data) there should exist an evolutionary tree that embeds all the most likely quartet topologies, and, given enough time, the heuristic presented in [9] finds the true tree.

3.2.2.2 Agglomerative (hierarchical) clustering with a linkage criterion

In this paper, I take a lightweight approach by using standard R libraries.

One of the simplest ways to cluster hierarchically is to put every object in its own cluster, and then merge greedily based on a linkage criterion, until all items have been merged into a single cluster.

In single linkage, at each step the two clusters are merged with the smallest pairwise distance. This is also called *nearest neighbor* clustering.

In complete linkage, or *farthest neighbor* clustering, at each step the two clusters with the largest pairwise distance are merged.

Average linkage is a compromise between single and complete linkage. The distance between two clusters is defined as the average between the pairwise distances, i.e.

$$d(A, B) = \frac{1}{|A||B|} \sum_{a \in A} \sum_{b \in B} d(a, b)$$

Assuming the distance matrix is represented as a .csv file, the R code becomes as simple as listing 3.2.

```
distance_matrix <- as.dist( as.matrix( read.csv(distances, row.names=1 ) ) )
clustering <- hclust(distance_matrix, method="average")
plot(clustering)
```

LISTING 3.2: R code for plotting a hierarchical clustering. `method` can be set to `"average"`, `"single"` or `"complete"`, among other options.

3.2.3 Phylogeny tree of 24 mammals

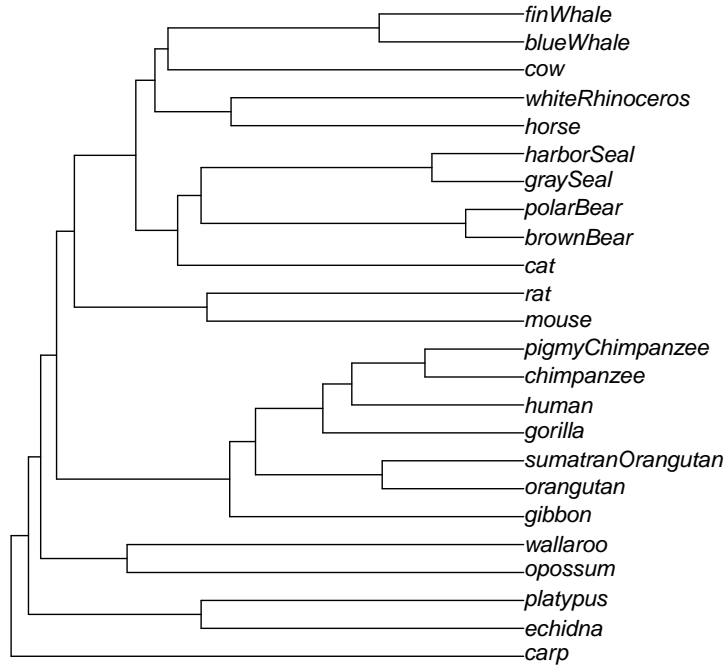Figure 3.1 shows the phylogeny tree obtained by average link clustering.

FIGURE 3.1: Result of average linkage clustering of the distance matrix shown in table 3.1. The height at which clusters join is proportional to the distance between them.

Comparison with [2], which uses CompLearn [8], shows that the dendrogram in this paper differs in two places. Here, ((finWhale, blueWhale), cow) joins (whiteRhinoceros, horse) first, while in the cited paper ((harborSeal, greySeal), cat) first joins (horse, whiteRhinoceros). Also, higher up in the tree, in this paper, the rodents join the ferungulates, and then the primates join, while in the cited paper the primates and ferungulates join first, and then the rodents join. The results of [2] are in line with biological analysis (e.g. [1]), so average linkage is less accurate than the quartet-tree method in this case.

But, this is no surprise, since the quartet-tree method [9] works especially well for evolutionary data, and an exceptionally high global fitness score of $S(T) = 0.996$ is attained in [2]. Other trees, for example containing random correlated data (see comparison below), have fitness scores of around 0.9.

Average linkage gives a sense of distance between clusters, which is lacking in the quartet-tree method. It is easily seen from figure 3.1 that, for example, the Polar Bear and Brown Bear are very close together, and that the primates form a very tight cluster.

3.3 Random correlated data

In this section, we will cluster files containing random bytes, which we have partially correlated, so we know what the clustering should be like. This method of validation was taken from [2]. All byte sequences have been generated with the `random_bytes` function of the Ruby library SecureRandom.

Let *tags* a, b, c be blocks of 1000 random bytes. We create file a in the following way: generate 80.000 random bytes, and at 10 distinct positions (picked from $0 \ldots 79$) replace a 1000 byte block with tag a. To create file ab, we insert tag a at 10 distinct positions, then insert tag b at 10 distinct positions, possibly overwriting some of the earlier insertions of tag a. In this way, we create files a, b, c, ab, ac, bc, and abc. The clustering is shown in 3.2.

I opted for single linkage clustering, i.e. the distance between clusters A and B is $d(A, B) = \min_{a \in A, b \in B} \mathrm{d}(a, b)$, because it shows the fact that ab, ac and bc have about the same distance to abc, while a, b and c are farther away, but also at an even distance, exactly as you would expect.

In figure 3.3, the experiment is repeated with 22 files, mimicking the experiment done in [2]. The result is what you would expect, and you could argue that this clustering is more insightful than the quartet tree method, since that method does not show the degree in which two clusters are related. The clustering here shows that $abcd$ is closer to $abce$ than jk is to any file labeled by more than two tags. It also clearly shows that single tag files have the least correlation, and no correlation among themselves, because they all join the other files at an equal distance of about 1.

3.4 Literature

We proceed to clustering of utf-8 files of books obtained from `www.gutenberg.org`. Here, the result is hard to validate, except for human intuition. The books used in this experiment are (1) A Week on the Concord and Merrimack Rivers, (2) Walden, and On The Duty of Civil Disobedience, both by Thoreau, (3) The Adventures of Sherlock Holmes, (4) The Hound of the Baskervilles, both by Doyle, (5) The Beautiful and the
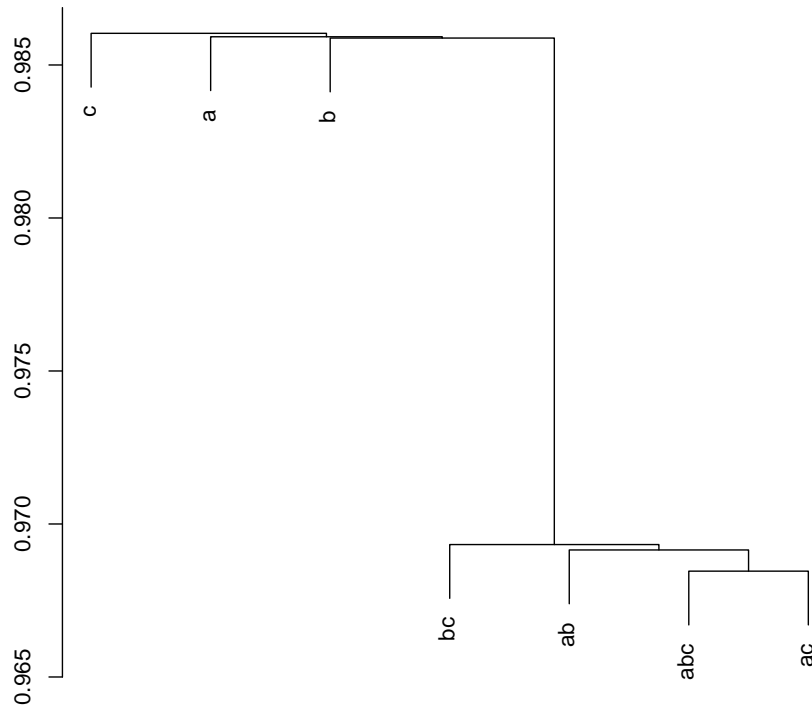
FIGURE 3.2: Single link clustering of seven 80 kB files containing random bytes, which are partially correlated. The height at which clusters join is the distance between them, according to the linkage criterion.

Damned, (6) This Side of Paradise, both by Fitzgerald, (7) Sons and Lovers, (8) White Peacock, both by Lawrence. The result is shown in figure 3.4.

3.5 Source files

We cluster ten files from the Ruby standard library [13] at commit `c722f8ad1d`, ten files from the Clojure standard library [14] at commit `41af6b24dd` and ten files from the Glasgow Haskell Compiler 6.10.1 [15]. All comments and blank lines were stripped. This removes, among other things, copyright notices which were shared by files belonging to the same library. The result is shown in figure 3.5. No preprocessing gives similar results.
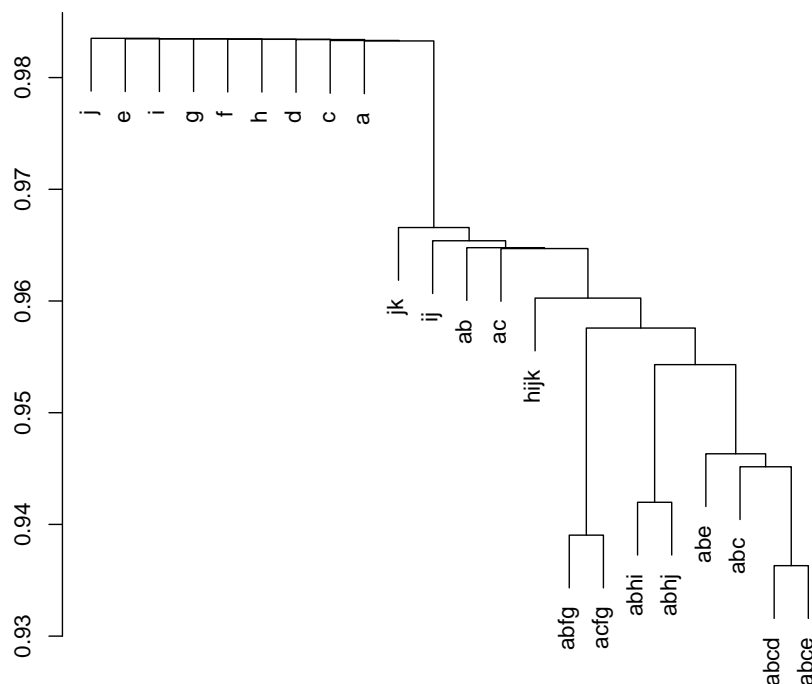
FIGURE 3.3: Single link clustering of twenty-two 80 kB files containing random bytes, which are partially correlated.

3.6   Conclusion

The experiments in this chapter show that with a bare minimum of code, using the Python binding to the lzma compression algorithm, and built-in R functions for hierarchical clustering, results can be obtained that are comparable those in [2], which used [8] for their experiments. The clustering method used in CompLearn, presented in [9], is computationally demanding, and doesn't preserve distance between clusters, but theoretically is able to attain a globally optimal score for evolutionary data. The clustering methods used in this chapter, i.e. single, average and complete linkage agglomerative clustering, are much simpler, are accessible from the R standard library, and preserve a sense of distance between clusters.
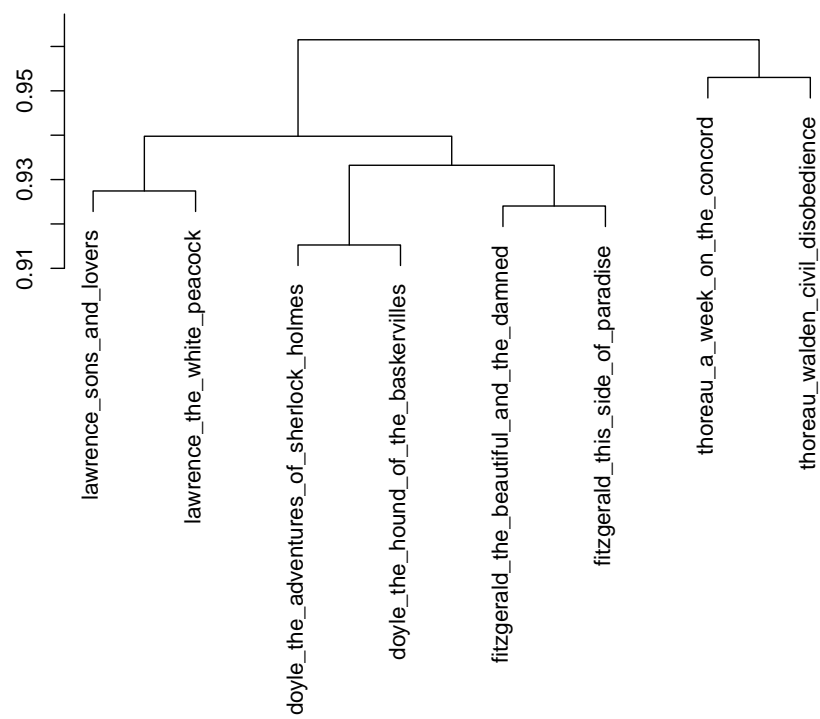
FIGURE 3.4: Average link clustering of eight books by english and american writers in utf-8 format.
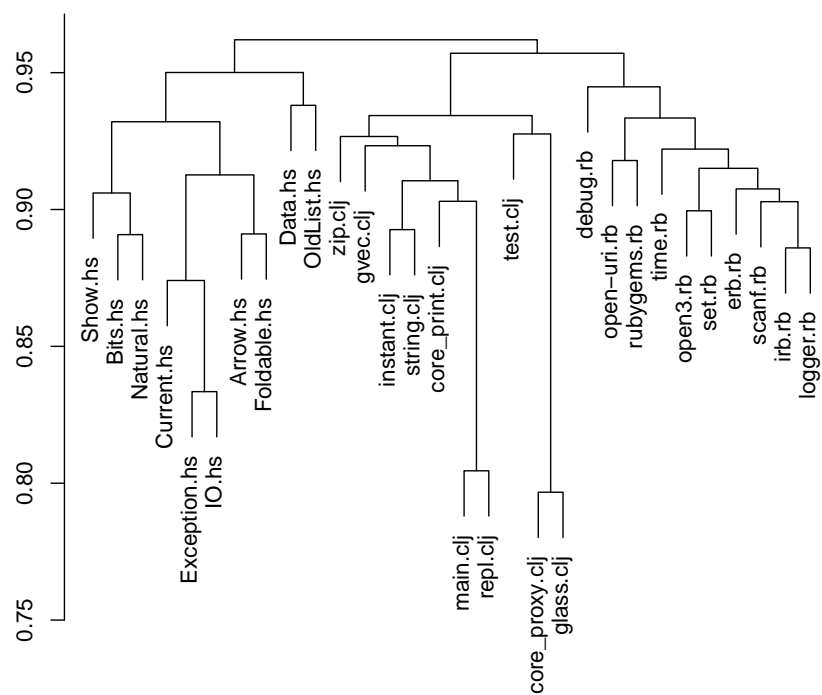
FIGURE 3.5: Average link clustering of 30 source files. The comments and blank lines were stripped using the *cloc* utility.

Chapter 4

Classification

4.1   Introduction

Classification is a fundamental learning task. Given a training set of objects for which the class is known, we want to identify the class of an unknown object.

For example, we want to know if an incoming mail is *spam* (unsolicited message sent in bulk) or *ham* (genuine message.) We need an algorithm that extracts certain features from the incoming message, compares it to features extracted from, say, a million messages labeled as spam and a million labeled as ham, and decides to which class it belongs.

Most real-world applications take a domain specific approach. For example, spam filters count occurences of words and hyperlinks.

In this chapter, the normalized compression distance (2.2) is used to extract features from objects. No domain knowledge is used. This method of feature extraction is hinted at in [3], but has, to my knowledge, never been used in experiment.

To classify a new piece of data, one determines how well it compresses with *anchors*, preselected pieces of data from each class. The distances with these anchors are the features representing an object. With these features, a support vector machine can be trained. This support vector machine can then classify new objects.

First, I classify file fragments of different file types (for example .jpg or .xml), a common forensics task. This is a good sanity check for this method, because fragments should, in general, compress well with fragments of the same file type, and not well with fragments from different file types. I show that the classifier can distinguish very well between uncompressed file types, but does not so well on compressed file types, like .gz and .jpg.

This is because a real-world compressor cannot make an already compressed string much smaller.

Then, I show that in gender attribution to written text, this method works almost as well as the state of the art, domain specific approaches, which use stylometric and content-related features. Again, the code is light-weight: I only use methods from the Python machine learning library scikit-learn [4].

The code [16] is available online.

4.2 Classifying file fragments

In forensics, file reconstruction is a big issue. Corrupted or wiped disks may contain scattered fragments of files without metadata. File fragment classification is a preliminary step in the reconstruction process.

The Govdocs1 dataset [17] is used throughout. It contains nearly one million files of various types, acquired by querying search egines with pseudo-random numbers and words.

Many authors, like [18], [19] and [20], try to classify compound file types (e.g. .doc or .pdf.) Since a .ppt file may contain a .doc file, or a .pdf file may contain a .jpg image, there is no way of telling the filetype of a 512 or 4096 byte block originating from such a file. If you misclassify .pdf as .jpg, did the classifier make a mistake, or did you stumble upon an embedded .jpg withing a .pdf? The answer is unknowable. This oversight is noted in [21].

We will only use filetypes that are not compound, e.g. .html, .csv and .jpg.

4.2.1 Feature extraction

Many statistical learning models, like artificial neural networks or support vector machines, take as a training set fixed dimensional vectors that correspond to a label:

$$D = \{(\vec{x_i}, y_i) \mid \vec{x_i} \in \mathbb{R}^k, i = 1, 2, \ldots, n\} \tag{4.1}$$

The mapping of actual objects onto feature vectors is the crucial step. For example, in [18] byte frequencies are used as features, so $\vec{x_i}$ is effectively a histogram, and y_i is one of .dll, .exe, .pdf, .mp3, .jpg.

.csv	0.91	0.94	0.94	0.86	0.95	1.01	1.00	1.01	1.02	1.02
.jpg	1.01	1.01	1.01	1.01	1.01	1.00	1.00	1.00	1.00	1.00

TABLE 4.1: Example feature vector for a .csv and a .jpg fragment. The first five anchors are .csv fragments, the last five are .jpg fragments. All fragments are randomly drawn from the Govdocs1 corpus.

In [3] a method is described to extract features from objects using the normalized compression distance. For each file type, (randomly) select some *anchors* from the corpus of file fragments. For simplicity, let's assume a binary classification problem of fragments that are either from a .jpg or a .csv file. After picking 5 anchor fragments for each type (totaling 10), we calculate the feature vector $\vec{x}$ for a file f as follows:

$$\vec{x} = (NCD(f, a_1), NCD(f, a_2), \ldots, NCD(f, a_{10})) \tag{4.2}$$

Table 4.1 shows an example feature vector of a fragment for both file types. It becomes clear why a fragment can be characterized this way: it's easy to see that the first fragment is a .csv file, since it is close to the first five anchors (the .csv anchors.) .jpg is a compressed format, so it has a distance of 1 from both the .csv and .jpg anchors.

4.2.2 Support vector machine

A support vector machine is a trainable binary classifier[1], first introduced in its current form in [22]. For an extensive introduction, see [23]. In all experiments, I used the Python library scikit-learn [4], which uses [24] and [25] internally.

In short, we want to separate the data points in the d-dimensional feature space with a hyperplane that maximizes the margin (Euclidian distance) to the closest data points.

New data points fall on either side of the hyperplane and are classified accordingly.

It is not possible to separate every set of points in all possible ways with a linear function (i.e. a hyperplane.) You can show that hyperplanes in $\mathbb{R}^n$ can, at most, separate $n+1$ points in all possible ways. So, if the training set is larger than the number of features plus one, it may be inevitable that points in the training set are misclassified by every hyperplane. The user-specified parameter C weighs the penalty for misclassified points in the training set. A higher value of C results in a hyperplane that misclassifies less training items. (This may actually hurt the classifier's predictive power, a phenomenon known as overfitting.)

---

[1]A classifier for more than two classes can be created from a combination of binary classifiers, for example with a one-vs-rest strategy.

With some mathematical footwork (see [23]), the optimization problem can be formulated in terms of only the dot products $\langle \vec{x_i}, \vec{x_j} \rangle$ of feature vectors.

Nonlinear-classifiers can be created by replacing all dot products with a *kernel function* $k(\vec{x_i}, \vec{x_j})$. This effectively applies a higher-dimensional transformation to the feature space. A hyperplane is then constructed in that space, so that non-separable data may become separable. This introduces extra parameters, is more memory intensive, and is prone to overfitting, but often gives better results.

Two common kernel functions are the polynomial one

$$k(\vec{x_i}, \vec{x_j}) = (\langle \vec{x_i}, \vec{x_j} \rangle + r)^d$$

which introduces parameters r and d, and the radial basis function (RBF)

$$k(\vec{x_i}, \vec{x_j}) = e^{-\gamma||\vec{x_i} - \vec{x_j}||^2}$$

which introduces γ.

4.2.3 Experimental setup

4.2.3.1 Preparing the data set

At the time of writing, it is possible to download samples of the Govdocs1 dataset [17], called threads, of 1000 files each. I used thread 0 through thread 7 as a corpus.

I created two corpora of fragmented files by chopping the files up into 512 byte blocks, throwing away the first and last block of each file. The reasoning is that those blocks often contain header information, which is atypical for the file at large. It also ensures that all blocks are of equal size.

Table 4.2 shows the number of fragments of each file type.

4.2.3.2 Validating the classifier

In each experiment, n_{anchors} anchors per class are randomly selected from the dataset. From the remaining fragments, n_{items} items per class are randomly selected to extract features from. They will serve to train and validate the classifier.

File type	# of fragments
.csv	47782
.jpg	33630
.gz	11507
.log	25625
.xml	9540
.html	19934

TABLE 4.2: Number of 512-byte fragments per file type.

Rescaling the feature vectors As is suggested in the scikit-learn documentation ([4]), after we extract the feature vectors from the file fragments (which contain NCDs mainly between 0.85 and 1.00), we rescale them to have zero mean and unit variance.

The classifier's effectiveness is assessed using k-fold cross validation. The data set is partitioned into k chunks of equal size. $k - 1$ chunks are used as training data and the remaining chunk as test data. In k runs, every chunk is used once as test data. So, in the end, every fragment in the data set is predicted once. For each training partition, a grid search is performed to select the best parameters for the model.

Parameter estimation In [26], the authors describe a practical method for training a support vector machine, and show that a simple grid search on the classifier's parameters can dramatically improve results.

When using a radial basis function kernel, for example, we need to find the best combination of C and γ. First, we try all combinations $\{(C, \gamma) \mid C, \gamma \in 2^n, n = -10, -9, \ldots, 9, 10\}$. Then, after the best C and γ in that grid have been determined, we can perform successively more precise estimations by forming a finer grid around the current optimum.

The fitness of the parameters is assessed by k-fold cross validation. (So k-fold cross validation is used in two places – to partition the whole data set, and to find the best parameters for each of the training partitions.)

4.2.4 Results

4.2.4.1 Classifying .csv and .jpg fragments

As a sanity check, we classify .csv and .jpg fragments. They should be easily distinguishable, since .jpg has high entropy (it is a compressed format) and .csv is very regular. Table 4.3 shows this to be the case. The precision is almost perfect.

anchors/type	.csv precision %	.jpg precision %
2	.9932	.9966
5	.9943	.9982
10	.9970	.9980

TABLE 4.3: The results were acquired by averaging precision rates over five independent runs. In each run, the distinct anchors were drawn randomly from the dataset. Then, 3000 distinct fragments (different from the anchors) per file type were selected. The predictions were validated with five-fold cross validation. The support vector machine with an RBF kernel was trained by doing a grid search for $C \in \{2^{-2}, 2^2, \ldots, 2^8\}$ and $\gamma \in \{2^{-9}, 2^{-7}, \ldots, 2^1\}$, optimized for each of the five training folds with a three-fold cross validation. All fragments are 512 bytes.

4.2.4.2   Classifying .gz and .jpg fragments

Now, we train the classifier on two compressed formats. The recall rate is very bad and .jpg gets many false positives. Training on .gif and .mp3 instead of .gz gives similar results. Increasing the training set from 3000 to 10000 sample fragments per type doesn't improve results, either.

This outcome is to be expected, because compressed file formats can hardly be compressed any further by lzma (or another compressor). They are effectively *random* sequences, although they are, of course, not really random. Here, the difference between a real world compressor and the theoretical notion of the Kolmogorov complexity is especially clear: the (theoretical) information distance between two objects does not change if they are compressed, but the real-world equivalent, the normalized compression distance, does, because compressors cannot find the patterns in the already compressed data.

anchors/type	.gz precision	.gz recall	.jpg precision	.jpg recall
2	.6622	.7151	.7162	.4295
10	.9040	.3306	.5901	.9643

TABLE 4.4: The results were acquired by averaging precision rates over five independent runs. In each run, the distinct anchors were drawn randomly from the dataset. Then, 3000 distinct fragments (different from the anchors) per file type were selected. The predictions were validated with five-fold cross validation. The support vector machine with an RBF kernel was trained by doing a grid search for $C \in \{2^{-2}, 2^2, \ldots, 2^8\}$ and $\gamma \in \{2^{-9}, 2^{-7}, \ldots, 2^1\}$, optimized for each of the five training folds with a three-fold cross validation. All fragments are 512 bytes.

4.2.4.3   Classifying .csv, .html, .jpg and .log

Table 4.5 (two anchors per file type) and table 4.6 (ten anchors per file type) show that classification still works very well on four different file types of low entropy. Note that adding more anchors now significantly improves results.

file type	precision	recall
.csv	.9684	.9520
.html	.9091	.9293
.jpg	.9960	.9889
.log	.9005	.9017

TABLE 4.5: Precision and recall rates for two anchors per file type. The results were acquired by averaging precision rates over five independent runs. In each run, the distinct anchors were drawn randomly from the dataset. Then, 3000 distinct fragments (different from the anchors) per file type were selected. The predictions were validated with five-fold cross validation. The support vector machine with an RBF kernel was trained by doing a grid search for $C \in \{2^{-2}, 2^2, \ldots, 2^8\}$ and $\gamma \in \{2^{-9}, 2^{-7}, \ldots, 2^1\}$, optimized for each of the five training folds with a three-fold cross validation. All fragments are 512 bytes.

file type	precision	recall
.csv	.9932	.9907
.html	.9638	.9681
.jpg	.9965	.9915
.log	.9668	.9699

TABLE 4.6: Precision and recall rates for ten anchors per file type. The results were acquired by averaging precision rates over five independent runs. In each run, the distinct anchors were drawn randomly from the dataset. Then, 3000 distinct fragments (different from the anchors) per file type were selected. The predictions were validated with five-fold cross validation. The support vector machine with an RBF kernel was trained by doing a grid search for $C \in \{2^{-2}, 2^2, \ldots, 2^8\}$ and $\gamma \in \{2^{-9}, 2^{-7}, \ldots, 2^1\}$, optimized for each of the five training folds with a three-fold cross validation. All fragments are 512 bytes.

4.3 Classifying written text by author gender

Men and women's language use differs, generally speaking. The sociolinguistic essay [27], a commentary on women's place in society, is regarded as the first research on this topic. The author postulates there is men's language, like *Shit, you've put the peanut butter in the refridgerator again* and women's language, like *Dear me, did he kidnap the baby?* (examples taken from the article.)

The topic is fuzzy. It is not clear what patterns we are looking for. That means the NCD is a good choice as a distance metric: we trust the compressor to find enough patterns, and do not input any domain-specific information ourselves.

4.3.1 Literature overview

In [28], the authors investigate a large number of blogs in different age and gender brackets. They find big differences in style. Men use more articles (*a, an, the*) and prepositions (words like *after, besides, toward*), while women use more pronouns (*I, you,*

she, ...) and words conveying assent and negation (*never, ever, no, yes,* ...) These findings are supported by the empirical survey [29]. The same features distinguish older bloggers (more articles and prepositions) from younger ones (pronouns, assent/negation.) There are also differences in content. Words like *linux, microsoft, gaming* predict male authorship, while words like *shopping, mom, cried* predict female authorship.

The authors of [28] use 502 stylometric features and 1000 content-related features (i.e. the 1000 words with the highest predictive power) to train a classification algorithm and obtain 80.1% precision with ten-fold cross validation.

In [30], a similar approach (stylometric and feature words) is used, but the author confuses gender attribution with authorship attribution (a task that is much better understood.) The data set consists of 1000 blog posts, by 10 male authors and 10 female authors (50 posts per author). The classifier predicts the right gender 85.4% of the time, but because there are so many posts of the same author in the data set, the results of this experiment cannot be compared to the results of [28] and this paper.

Finally, [31] tries to predict gender based purely on writing style (as opposed to [28] and [30] who also use content-related features.) The author tries to mitigate the gender-bias in topic and genre by hand selecting blogs posts with the same topic, e.g. the television show *How I Met Your Mother.* This seems problematic to me: a human might unconsciously select blog posts that comply to her idea of male or female writing, thereby biasing the experiment. Also, the dataset is very small (280 posts, 140 per gender) and the author tries eight different feature-extraction methods, of which the best one yields 71.3%. This result is hard to compare, because it is the best result of multiple hypotheses, whereas other papers used only one.

4.3.2 Classifying blogs by author gender using the normalized compression distance

4.3.2.1 Cleaning the corpus

I use the Blog Authorship Corpus, assembled by the authors of [28], which contains blogs of 19320 bloggers (averaging 35 posts per person). Blogs are labeled by author gender, age, and genre.

For each blog, I used only the plain text of each post, removing leading and trailing newlines. I tried to replace characters that were not utf-8 with comparable utf-8 alternatives using `utf8_utils` [32], a Ruby library. If this could not be done, I discarded the post. I automatically removed all posts that were not in English, using the Ruby

binding [33] to the Google Chromium Compact Language Detector [34]. The detector also removed posts that only consisted of a hyperlink. I concatenated the remaining posts into a single file.

4.3.2.2 Results

Table 4.7 shows the results achieved by classifying based on the normalized compression distance. A 75% recall rate is quite remarkable, considering no domain features at all were used. Many experiments using domain-specific features do not even reach this number, and the highest valid figure I saw in the literature is 80.1%.

anchors/type	female recall %	male recall %
100	0.745	0.7551

TABLE 4.7: The anchor blogs were randomly drawn from the dataset. Then, 5000 blogs, different from the anchors, were selected for each gender. The predictions were validated with five-fold cross validation. The support vector machine with an RBF kernel was trained by doing a grid search for $C \in \{2^{-2}, 2^2, \ldots, 2^8\}$ and $\gamma \in \{2^{-9}, 2^{-7}, \ldots, 2^1\}$, optimized for each of the five training folds with a three-fold cross validation.

4.4 Conclusion

This chapter shows that the normalized compression distance can be used to extract features from data without using domain-specific knowledge. Those features can then be used to train a classifier (in this paper, a support vector machine) to predict the class of new data items. This method was briefly described in [3], but has, to my knowledge, never been used in experiment. The Python script to run experiments is small, and makes extensive use of the scikit-learn library [4].

File fragment classification is used as a proof of concept, and we show that low entropy formats are classified almost perfectly.

In the field of gender attribution, it is not very clear what patterns to look for, and the NCD shines. In classifying the gender of the authors of blogs, my algorithm has a 75% precision rate, which is very high, considering no domain specific features where extracted, and the highest number in literature is 80.1%, on the same corpus I used.

As follow-up research, one could try to use a combined method: Using (a subset of) the stylometric and content-related features, along with NCDs with fifty or hundred anchors.

Other fields in which NCD could be applied are classification of fiction and non-fiction text, as is done in [35], or other classification by genre, like news stories or financial messages.

Chapter 5

Conclusion

This thesis shows that with a very light-weight approach, using only libraries from the Python and R programming languages, one can harnass the power of the normalized compression distance, the real-world version of the normalized information distance based on Kolmogorov complexity.

Two domains of machine learning are studied: clustering and classification.

In the domain of clustering, I developed a light-weight alternative to CompLearn [8], an open source clustering tool based on NCD. I show that agglomerative hierarchical clustering with a linkage criterion works well for clustering, and compare it to the quartet-tree clustering method used in CompLearn [9]. The quartet-tree method clusters evolutionary data better, but, unlike clustering with a linkage criterion, doesn't preserve a sense of distance between clusters. It also attains less than stellar fitness scores on non-evolutionary data, and is more computationally demanding than clustering with a linkage criterion.

In the domain of classification, I use the normalized compression to extract features from data, and use those features to train a support vector machine. This method is described in [3], but, as far as I know, this paper is the first to report on experiments using this method. This way of feature extraction works very well on the fuzzy problem of gender attribution to written text, scoring a 75% precision rate on blogs, versus 80.1% by the best domain-specific approach.

Bibliography

[1] Ying Cao, Axel Janke, Peter J. Waddell, Michael Westerman, Osamu Takenaka, Shigenori Murata, Norihiro Okada, Svante Pääbo, and Masami Hasegawa. Conflict among individual mitochondrial proteins in resolving the phylogeny of eutherian orders. *Journal of Molecular Evolution*, 47(3):307–322, 1998. ISSN 00222844. doi: 10.1007/PL00006389.

[2] Rudi Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005. ISSN 00189448. doi: 10.1109/TIT. 2005.844059.

[3] Rudi Cilibrasi. Statistical inference through data compression. 2007. URL `http://www.narcis.nl/publication/RecordID/oai:uva.nl:217776`.

[4] Fabian Pedregosa, Ron Weiss, and Matthieu Brucher. Scikit-learn : Machine Learning in Python. 12:2825–2830, 2011.

[5] Ming Li and Paul M B Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2009.

[6] Charles H. Bennett, Péter Gâcs, Ming Li, Paul M. B. Vitânyi, and Wojciech H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4): 1407–1423, 1998. ISSN 00189448. doi: 10.1109/18.681318.

[7] M. Li, X. Chen, X. Li, B. Ma, and P.M.B. Vitanyi. The Similarity Metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004. ISSN 0018-9448. doi: 10.1109/TIT.2004.838101.

[8] Rudi Cilibrasi. CompLearn, 2015. URL `www.complearn.org`.

[9] Rudi L. Cilibrasi and P. M. B. Vitányi. A Fast Quartet tree heuristic for hierarchical clustering. *Pattern Recognition*, 44(3):662–677, 2011. ISSN 00313203. doi: 10.1016/ j.patcog.2010.08.033.

[10] Geert Kapteijns. Clustering by NCD. 2015. URL `https://github.com/Kappie/clustering`.

[11] GenBank. URL `http://www.ncbi.nlm.nih.gov/genbank/`.

[12] XZ Ubuntu manual entry, 2014. URL `http://manpages.ubuntu.com/manpages/raring/man1/xz.1.html`.

[13] ruby @ github.com, . URL `https://github.com/ruby/ruby`.

[14] clojure @ github.com. URL `https://github.com/clojure/clojure`.

[15] The Glasgow Haskell Compiler. URL `https://www.haskell.org/ghc/`.

[16] Geert Kapteijns. Classification by NCD, 2015. URL `https://github.com/Kappie/support_vector_machine`.

[17] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation*, 6 (SUPPL.):2–11, 2009. ISSN 17422876. doi: 10.1016/j.diin.2009.06.016.

[18] Q. Li, A. Ong, P. Suganthan, and V. Thing. A Novel Support Vector Machine Approach to High Entropy Data Fragment Classification. *Proceedings of the South African Information Security Multi-Conference (SAISMC 2010)*, 2010. URL `http://www1.i2r.a-star.edu.sg/~vriz/Publications/WDFIA_SAISMC2010_SVM_High_Entropy_Data_Classification.pdf$\delimiter"026E30F$npapers2://publication/uuid/0A3CD98F-9AA4-4DAF-A2C7-95CB9E080FB4`.

[19] Cor J. Veenman. Statistical disk cluster classification for file carving. *Proceedings - IAS 2007 3rd Internationl Symposium on Information Assurance and Security*, pages 393–398, 2007. doi: 10.1109/IAS.2007.75.

[20] Stefan Axelsson. The normalised compression distance as a file fragment classifier. *Digital Investigation*, 7(SUPPL.):0–7, 2010. ISSN 17422876. doi: 10.1016/j.diin.2010.05.004.

[21] Vassil Roussev and Candice Quates. File fragment encoding classification An empirical approach. 10:69–77, 2013.

[22] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. ISSN 08856125. doi: 10.1007/BF00994018.

[23] Christopher J. C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998. ISSN 13845810. doi: 10.1023/A:1009715923555. URL `http://www.springerlink.com/index/Q878561731Q.pdf`.

[24] Rong-en Fan, Xiang-rui Wang, and Chih-jen Lin. LIBLINEAR : A Library for Large Linear Classification. *Journal of Machine Learning Research*, 9(2008):1871–1874, 2014.

[25] Chih-chung Chang and Chih-jen Lin. LIBSVM : A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2: 1–39, 2011. ISSN 21576904. doi: 10.1145/1961189.1961199.

[26] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A Practical Guide to Support Vector Classification. *BJU international*, 101(1):1396–400, 2008. ISSN 1464-410X. doi: 10.1177/02632760022050997. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[27] Robin Lakoff. Language and Woman ' s Place. *Language in Society*, 2(1):45–80, 1973. ISSN 0047-4045. doi: 10.1017/S0047404500000051.

[28] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James Pennebaker. Effects of Age and Gender on Blogging. *AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs*, pages 199–205, 2005.

[29] a Mulac, J J Bradac, and P Gibbons. Empirical support for the gender-as-culture hypothesis. *Human Communication Research*, 27(1):121–152, 2001. ISSN 03603989. doi: 10.1111/j.1468-2958.2001.tb00778.x. URL `http://dx.doi.org/10.1111/j.1468-2958.2001.tb00778.x`.

[30] George K. Mikros. Authorship Attribution and Gender Identification in Greek Blogs. *Methods and Applications of Quantitative Linguistics*, pages 21–32, 2013.

[31] Ruchita Sarawgi, Kailash Gajulapalli, and Yejin Choi. Gender Attribution : Tracing Stylometric Evidence Beyond Topic and Genre. *Fifteenth Conference on Computational Natural Language Learning*, (June):78–86, 2011.

[32] Norman Clarke. utf8_utils Ruby gem. URL `https://github.com/norman/utf8_utils`.

[33] Compact Language Detector Ruby Gem, . URL `https://github.com/jtoy/cld`.

[34] Dick Sites. Google chromium Compact Language Detector. URL `https://code.google.com/p/cld2/`.

[35] M. Koppel. Automatically Categorizing Written Texts by Author Gender. *Literary and Linguistic Computing*, 17(4):401–412, 2002. ISSN 0268-1145. doi: 10.1093/llc/17.4.401. URL `http://llc.oxfordjournals.org/content/17/4/401.short`.