Thomas Bakker

Plagiarism Detection in Source Code

Bachelor thesis, July 10th 2014

Advisors: dr. W.A. Kosters, prof. dr. E.A. Verbitskiy

Mathematisch Instituut

Leiden Institute for Advanced Computer Science

Universiteit Leiden

Contents

1 Introduction

In this bachelor thesis we study plagiarism detection in computer programs using various information-theoretic and probabilistic concepts. We aim to make the detection methods as general as possible by basing them on a minimum amount of domain knowledge. Ideally this would allow us to develop a method in one domain, and then easily apply it in many other domains with minimal adjustments.

In Chapter 2 we introduce the problem of plagiarism in computer source code. In Chapter 3 we give a short overview of prior research in this field. Then in Chapter 4 we describe a new method for plagiarism detection, which we apply in Chapter 6 to two data sets of source code submitted by first year university students for an introductory programming course. We find that the method is able to successfully detect various types of plagiarism. The use of statistical methods to flag suspected regions and calculate p-values is the most important novel part of this thesis.

To prepare for Chapter 5, we describe some information theoretic concepts in detail in the Appendix. We introduce the *Kullback-Leibler divergence* (also called *relative entropy*), and prove some of its most important properties. Then, after introducing another concept called *relative entropy rates*, which generalize the concept of relative entropy from random variables to random processes, we apply this to our data sets in Chapter 6, giving us a way to compare the students who wrote the source code in a data set, and calculate how similar they are.

This thesis was written for a double bachelors degree in Mathematics and Computer Science at Leiden University, under supervision of E. Verbitskiy for the Mathematical Institute, and W. Kosters for the Leiden Institute for Advanced Computer Science.

2 Problem description

We will introduce a method for the detection of plagiarism in source code, and demonstrate its effectiveness on two real-world data sets. Our method will rely on very little domain-specific properties, and could easily be applied to another domain.

The detection of plagiarism is an interesting problem in modern computer science. Plagiarism is uncited copying of material, which can happen in many domains, including literature, computer source code and music. In this thesis we focus on the detection of plagiarism in source code. Such plagiarism might include, as we will see, entire source code files, but also possibly only small pieces of code. For our detection methods we will assume that this plagiarism is malevolent, that is, the plagiarizing authors may attempt to disguise their actions.

Our source data consists of a collection of computer programming assignments handed in by first-year university students in C++, all solving the same task. Some may have copied parts of the assignments from others, and it is our goal to identify exactly what parts might have been copied. We will focus only on the detection of plagiarism *within* the data set, that is, plagiarism where the source document is also part of our data set.

It is possible that the assignment contains some code fragments that every student is allowed to use. Our method should be able to either ignore or automatically detect such sections.

3 Earlier work

Much research has been done regarding plagiarism detection. Given the scope of our data, we focus on methods that work on relatively small populations, between 50 and 100 items. For populations of such size, some categories of detection methods include:

- Bag of words models [Harris, 1970]
- Internal stylometry comparison [Maurer et al., 2006]
- Substring matching [Heintze, 1996]

Bag of words models split documents into words, or combinations thereof. The underlying statistical model assumes that an author writes a document by randomly choosing words (from a "bag of words"), so to detect similarity it compares word frequencies between documents.

Internal stylometry comparison creates stylometric models for many parts of a document, where stylometry is the analysis of writing style. Then, most methods in this category compare those models *within* documents, to find sections that look dissimilar to the rest of the document. The advantage is that this does not require the source of a plagiarized section to be known: it can detect sections of a document that simply *stand out*, that look like they were written by someone else.

Substring matching attempts to find parts of a document that are identical or similar to parts of another document. Since plagiarism is potentially adversarial, the matching should preferably be fuzzy. One way to do this would be to use the Levenshtein distance [Levenshtein, 1966], the number of changes you have to make to get from one string to another, and count all substrings with a distance smaller than some threshold as similar.

A simple form of substring matching would be to take all substrings of length n from a document (usually called n-grams), and compare every substring to all other substrings from all other documents. This can be computationally intensive, especially when performing fuzzy matching. Also, this method by itself only tells us if some substrings match, but it needs extra work to help the user to interpret if those substrings are likely plagiarism, or coincidental matches.

The state of the art depends on the field one wants to detect plagiarism in. The best methods might contain a lot of domain-specific information that other, more universal, methods ignore. One method which is frequently used for source code plagiarism detection is called *winnowing* [Schleimer et al., 2003], which does local substring matching in rolling windows. For every sufficiently long substring in a document, it determines how many n-grams within that longer substring are identical or similar to a part of another document, using an extensive fingerprinting algorithm.

4 Proposed plagiarism detection method

We will now introduce a method that has similarities to substring matching. To detect plagiarism in a given document, we first create variable-order Markov models [Rissanen, 1983] of all known documents, and then for every substring of length n in our document, we calculate the likelihood of that substring based on all other models. Then we calculate p-values for outlying likelihoods, and after Benjamini-Yekutieli multiple testing adjustments [Benjamini and Yekutieli, 2001] we find plagiarized sections by looking for sections with adjusted p-values smaller than 0.05.

Because our method is meant to be used on computer source code, we preprocess all source files with a lexicographic analyzer. This removes information that is not directly related to the workings of the code, such as comments, variable names, etc.

The process can be split up into the following five steps:

1. Preprocessing of source files.
2. Model generation.
3. Local likelihood estimation.
4. Regional likelihood estimation.
5. Outlier detection.

A 2003 publication [Mazeroff et al., 2003] used steps 1 (partially), 2 and 3 for the identification of malicious programming code in Microsoft Office macro files. The authors calculated the likelihood of every symbol being from one of two models: a benign model or a malicious model, and then they determined per symbol which likelihood was higher.

To this concept, we add steps 4 and 5: we introduce regional smoothing of the likelihoods and use statistical methods to calculate outliers, having the advantage that we have between 50 and 100 comparison documents/models, where [Mazeroff et al., 2003] only had 2 models: one benign and one malicious, which is not enough to apply the type of statistical techniques we use.

4.1 Step 1: Preprocessing of source files

To be able to effectively model the source code, we have to preprocess it. We use a custom lexicographic analyzer to tokenize the code, and output one unique character per token. This way, the next steps do not have to understand the grammar of source code. Also, it removes information that is likely changed by a plagiarist to try and avoid detection, such as variable names, literal strings, etc.

This still removes a substantial source of potentially useful information: it loses all indenting and other whitespace, variable naming styles, etc. However, our aim is to make our method as general as possible, so we sacrifice some domain-specific knowledge here. When applying this method for specific practical purposes, it might be useful to include this domain-specific knowledge as well, either by incorporating it directly in the output of this step, or by using different methods separately.

The lexicographic analyzer operates as follows:

1. Replace literal strings and numbers by single-width characters.
2. Remove comments.
3. Replace operators by single-width characters.
4. Replace known literals/constants (`void`, `int`, ...) by some character representing that literal/constant.

5. Replace all leftover identifiers (variable/function/class names) by one single-width character.

6. Remove all whitespacing.

If we feed this fragment of code into this method:

```
  for(int i=0; i<n; i++) {
    if(arr[i])
      cout << arr[i];
    else
        cout << "*";
  }
  cout << endl << endl << "******" << endl;
}

void input_char(char& res) {
  char inp;
  cin.get(inp);
  res = inp;
  while(inp!='\n')
    cin.get(inp);
}
```

we get the following output:

```
f(bI=";I<I;I‘){d(I[I])k!I[I];lk!$;}k!p!p!$!p;}
aI(q&I){qI;j.m(I);I=I;e(I~’)j.m(I);}
```

We apply this process to all source documents in a data set, storing the output as input for the next step.

4.2 Step 2: Model generation

We use variable-length Markov chains with maximum length d as models. To estimate the models, we generate Probabilistic Suffix Trees (PSTs) [Ron et al., 1996] of maximum depth $d + 1$. We build a PST for all source documents in a data set separately.

4.2.1 Probabilistic Suffix Trees

A *PST* is a tree with n nodes corresponding to contexts in the text it is generated on [Mazeroff et al., 2003]. Every node of the tree contains two properties:

1. The string that is represented by the node.

2. The relative frequency vector for the possible next symbols that occur in the text after the string represented by the node. Opposed to most definitions of a PST, we choose to also include these distributions in non-terminal nodes. This allows us to estimate likelihood for characters at the beginning of a document.

The tree with maximum depth d has the properties that:

- The root of the tree represents the empty string.
- Any non-root node represents the string of its parent, with one additional character added in front of it, i.e., with a longer state.
- No node represents a string longer than $d - 1$ characters.
- Fully built, it represents a variable-length Markov chain with maximum memory length $d - 1$.

The building of a PST based on a text is straightforward based on this definition. For example, a document of "`aabbabbaaabb`" results in the PST shown in Figure 1.

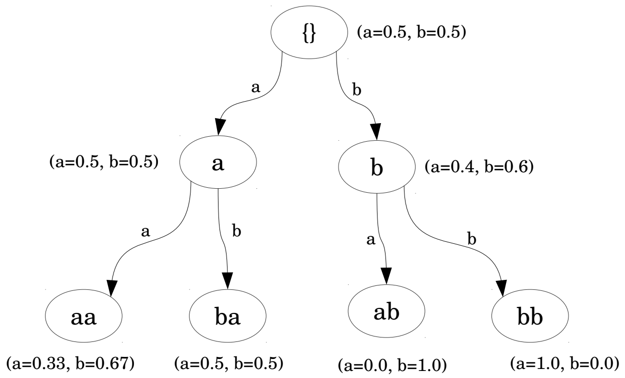

Figure 1: A PST built from the document "`aabbabbaaabb`". Shown with every node is the probability vector of the next character.

4.3 Step 3: Local likelihood estimation

Next, for every document in the population and for all subsections of that document, we calculate the likelihood of that subsection being generated by all calculated models, where "subsection" is defined as shown in the following two steps. For a given PST-model i and document j of size n potentially containing plagiarism, we do this in two steps:

1. Loop over all characters in the document. For every character, find the longest string of characters directly to the left that is contained in the PST. So for the character "`x`" in "`abcxe`" first consider "`abc`", then "`bc`", then "`c`", then "", if none of the former strings were in the PST.
2. Find the relative frequencies (likelihoods) for every string and character found in the previous step, as stored in the PST. Call this lik_k^{ij}, where i represents the model, j represents the document, and k is the position of the character, $0 \leq k < n$.

We could calculate the cumulative likelihood of a document being generated by the given model by multiplying all of its likelihoods found at step 2.

4.4 Step 4: Regional likelihood estimation

After step 3, we have a collection of local likelihoods for every combination of document and model. Those likelihoods are, however, not yet very useful: each lik_k^{ij} is the likelihood of a single character (at position k) given a short (or empty) prefix, while we are interested in the likelihood of larger sections of code.

To get there, we use a moving window of some size m, and multiply all the likelihoods contained in this window. This results in $n-m+1$ calculated likelihoods rl_r^{ij} corresponding to sections of the document. For section r of PST-model i and document j:

$$rl_r^{ij} = \prod_{k,\ r \leq k/m < r+1} lik_k^{ij}$$

For one document, the log-likelihoods per region and per model might now look like Figure 2. We see that in some regions, around 150 and 1150, all source documents are about equally unlikely, meaning that those regions are likely very different from all other documents. In regions $800 - 1000$ we see a wide spread, which might indicate that in some documents code similar to that region is very common, and very uncommon in other documents. We will now need to process these likelihoods to get values we can compare and use.

4.5 Step 5: Outlier detection

Now, for any given document, we have the regional likelihoods based on all other models. We could naively look for the regions that have the highest absolute likelihood and consider those suspicious. However, this ignores the fact that some strings are naturally more common in source code. One example from our data sets would be constructs such as:

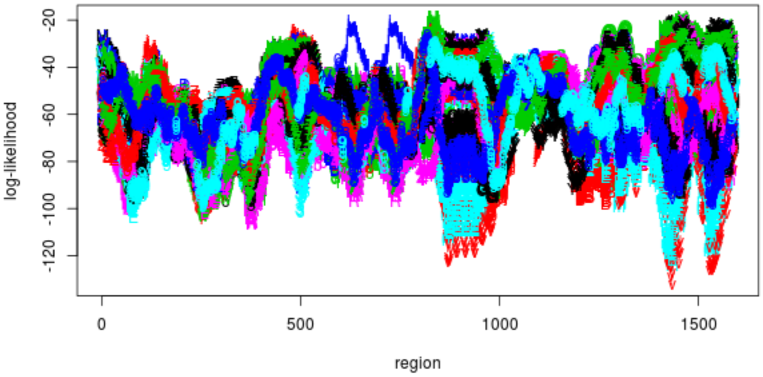

Figure 2: Log-likelihoods per region (k) of one document (i). Every colored line represents one document (j) we compared to. In this example, the region width (m) was chosen to be 50.

```
cout << "string" << endl;
cout << "another string" << endl;
cout << "yet another string" << endl;
```

Such a section has a high likelihood in many models. One can already see this effect in Figure 2 between regions 800 and 1000.[1] It corresponds to a block of `cout`-statements in the source file. The spread in log-likelihood is so high because such blocks are common for some documents, and very uncommon for others, which might use a different format to output multiple lines of text.

To solve this issue, rather than comparing likelihoods of regions horizontally (within a document), we will compare them vertically: between documents.

Let rl_r^{ij} be the regional likelihood of region r in document j for model i. We will now assume that for fixed region and document, the likelihoods estimated by the various models are approximately log-normally distributed with identical distributions:

$$\log\left(rl_r^{ij}\right) \sim \mathcal{N}(\mu_{rj}, \sigma_{rj})$$

This is close enough to be useful in practice. For illustration, a normal QQ-plot as shown in Figure 3 gives some indication of the distribution.[2]

[1] Note that the fact that the likelihoods are similar in these regions between many documents does not mean that all those documents share the same block/structure in that region: it simply means that the contents of that region in *this* document are similar across all models in general.

[2] Note that lik_k^{ij} is not log-normal: it contains many likelihoods of 1.

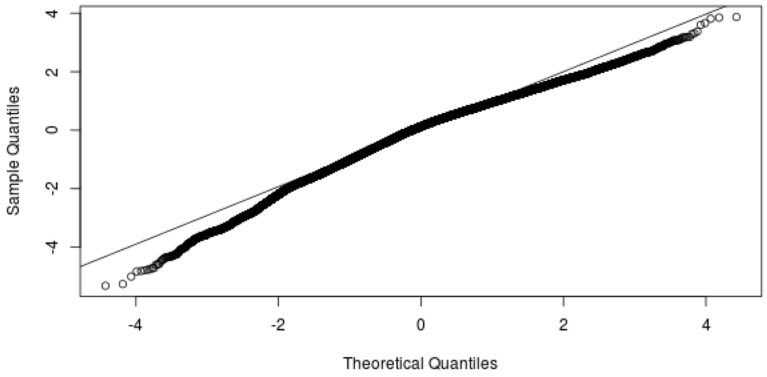

Figure 3: A QQ-plot generated from the logarithm of all regional likelihoods of a typical document. We can see that it is fairly normal.

Furthermore, for every region r within document j, we estimate $\tilde{\mu}_{rj}$ and $\tilde{\sigma}_{rj}$, and then calculate p-values:

$$p_r^{ij} = P\left(X > \log\left(rl_r^{ij}\right)\right) \text{ for } X \sim \mathcal{N}(\tilde{\mu}_{rj}, \tilde{\sigma}_{rj}), \text{ so}$$

$$p_r^{ij} = \Phi^{-1}\left(\frac{\log\left(rl_r^{ij}\right) - \tilde{\mu}_{rj}}{\tilde{\sigma}_{rj}}\right),$$

where Φ represents the distribution function of the standard normal distribution.

Now, we have up to 100 documents and the average document is over 2000 characters long. This means we can easily get $200,000$ such p-values per document. We have drawn the logarithm of the p-values calculated for one document in Figure 4, with a horizontal line at log(0.05).

We see that there are many calculated p-values that are less than 0.05. This is expected, since a p-value of 0.05 indicates a probability of 0.05 of getting a significant result without the hypothesis being true. So, we should expect to get around 5% of p-values as false positives, and because we are calculating 200,000 p-values at once, getting 10,000 false positives should be expected. We can fix this by adjusting for multiple testing.

4.5.1 Multiple testing

To test a hypothesis, we typically formulate a null hypothesis H_0 that is not rejected if there is no definitive evidence otherwise, and an alternative hypothesis

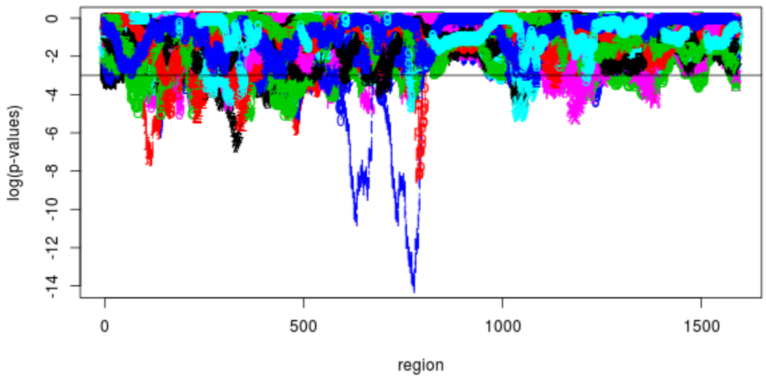

Figure 4: p-values from likelihoods per region (r) of one document (i). Every colored line represents one document (j) we compared to.

H_1 which we want to prove. For some realization $\hat{X}$ of a variable X, we calculate a p-value, which is the probability that a realization of X is at least as extreme as $\hat{X}$, given that the null hypothesis is true. If that probability is very small, it was unlikely that realization $\hat{X}$ came from distribution X, so we reject H_0 in favor of H_1. Usually the threshold of $p < 0.05$ is used to determine significance (i.e., rejection).

This procedure is sound when testing a single hypothesis, but when testing more than one hypothesis at once, we should adjust the threshold: otherwise the number of false positives where $p < 0.05$ by chance will grow as the number of hypotheses grows. Since we might calculate 200,000 p-values for just one input file, we should expect thousands of false positives if we naively use the single p-values with the standard threshold of 0.05.

There are several ways to solve this issue [Hochberg and Tamhane, 1987]. One could assume that all the p-values are independent, and calculate the required threshold such that the probability of a single false positive is smaller than 0.05. Alternatively, again assuming that the p-values are independent, one could calculate the required threshold such that the expected number of false positives is smaller than 0.05. Note that this is a much less restrictive threshold than the previous one. To decide what to do, we will first define some terminology for these two types of adjustments.

Firstly, we could limit the *family-wise error rate* (FWER), which is the probability of making at least one false rejection [Hochberg and Tamhane, 1987]:

$$\text{FWER} = P(V > 0),$$

where V is the number of false rejections.

Alternatively, we could try to control the *false discovery rate* (FDR), which is [Benjamini and Hochberg, 1995]:

$$\mathrm{FDR} = E(V/R),$$

where V is the number of false rejections, R is the number of all rejections.

In this case, limiting the FWER would likely result in too few regions being marked as possible plagiarism: it would guarantee that the probability that *any* of the flagged regions is a false positive would be smaller than some number. Controlling the FDR means that for any flagged region, we know that there is some probability that that specific flag is a false positive. This will result in more false positives, but also in fewer false negatives.

Secondly, we have to consider correlation between the p-values. Because regions overlap horizontally (within files) and vertically (between files), there is a strong correlation.

It turns out that a suitable multiple testing correction to use in this case is the Benjamini-Yekutieli procedure [Benjamini and Yekutieli, 2001]. Some desirable properties of this procedure include:

- The procedure guarantees that $FDR \leq \alpha$ for a chosen value of α;
- It deals well with correlation between p-values.

It can be described in three steps [van de Wiel, 2013]. Given m p-values $p_1, \ldots, p_m$:

1. Without loss of generality, assume the p-values are ordered:
$$p_{(1)} \leq p_{(2)} \leq \cdots \leq p_{(m)}.$$
2. Adjust p-values, where r is the rank from step 1:
$$p'_{(r)} = (m/r)\log(m) \cdot p_{(r)}.$$
3. Fix p-values so that they are ordered (i.e., monotonic):
$$p^{BY}_{(r)} = \min(p'_{(r)}, p'_{(r+1)}, \ldots).$$
4. Use these fixed p-values to reject values lower than some chosen α.

We apply this method to our p-values, setting α to 0.05, and flagging any regions with a lower adjusted p-value. This results in Figure 5, the adjusted version of Figure 4.

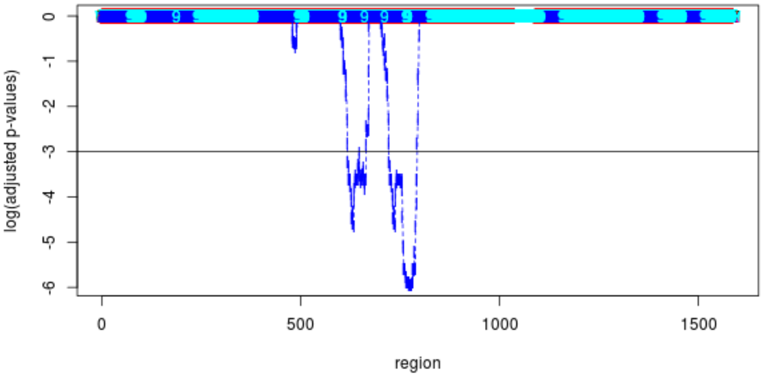

Figure 5: The p-values from likelihoods per region (k) of one document (i), adjusted by Benjamini-Yekutieli. Every colored line represents one document (j) we compared to.

We see that the adjustment has normalized the p-values properly, in that most regions for most documents are now treated as non-significant, and only one section is significantly similar to another document. So, our algorithm indicates that there was plagiarism in this example file, as the region from 600 to 750 has adjusted p-values that cross the line that represents log(0.05), meaning that their adjusted p-values are smaller than 0.05.

We have included two of the offending sections (region $600 - 800$ in the graph) as Algorithm 1, to show an example of what might be found by our method. Inspecting the code in Algorithm 1, it is clear that these sections were indeed likely plagiarized: the code works identical, some strings are literal matches, variables have been renamed to synonyms, and the `cout`-statement on top was switched around with the `char`-definition, likely to avoid suspicion. The whitespace style used as:

```
cin.get ( );
```

was standard in one document, and non-standard in the other. So, we could deduce that the source document was likely the version that has that whitespace style everywhere.

In Chapter 6.1 we apply the method to several datasets, and analyze the results.

Algorithm 1 Two versions of the same (incorrect) function, as submitted in two different submissions.

```
void verwijder ( ) {
  cout << "Geef het nummer van het woord dat je wilt verwijderen."
  << endl;
  char nummer; // char want de functie keuzeinlees...
  keuzeinlees (nummer); // ...werkt met chars
  int welkeweg = 0;
  while (nummer != '\n') {
    if (0 <= nummer && nummer >= 9) {
      welkeweg = welkeweg * 10 + (nummer - '0');
      nummer = cin.get ( );
    }//if
    else cout << "Kies een bestaand nummer." << endl;
  }//while
  welkeweg = welkeweg - 1; // array begint vanaf 0, dus min 1
  if (welkeweg > 20 || welkeweg < 1) {
    welkeweg = 0;
  }//if
  woordenarray [welkeweg] = "";
  welkeweg = 0;
}//verwijder


void verwijder ( ) { //Deze functie verwijdert woorden uit het woordenboek
      char cijfer; // geen int, want functie leesin is een char
      cout << "Geef het cijfer wat voor het woord staat "
      << "dat je wilt verwijderen." << endl;
      cin>>cijfer;
      int verwijderde = 0;
      while (cijfer != '\n') {
        if (0 <= cijfer && cijfer >= 9) {
          verwijderde = verwijderde * 10 + (cijfer - '0');
          cijfer = cin.get ( );
        }//if
        else cout << "Kies een bestaand nummer." << endl;
      }//while
      verwijderde = verwijderde - 1; // een array begint vanaf 0 dus min 1
      if (verwijderde > 20 || verwijderde < 1) {
        verwijderde = 0;
      }//if
      woorden [verwijderde] = "";
      verwijderde = 0;
   }//verwijder
```

5 Comparing full document models

In the previous chapters we described a method to compare regions of documents. It is also possible to compare full estimated author models, giving us distances (dissimilarities) between all the author models corresponding to documents in our data set. This will tell us how similar students are in their programming styles. With regard to plagiarism detection, this will only be useful for full-document plagiarism, but the results might be interesting in other ways.

Our method relies on several concepts from information theory, which are introduced in depth in the Appendix.

5.1 Comparing probabilistic suffix trees

In the previous chapter we introduced probabilistic suffix trees. We will now define *probabilistic suffix automata* [Mazeroff et al., 2003, Ron et al., 1996], which represent the same model, but are more similar in form to Markov processes than the probabilistic suffix trees were.

In Section 4.3, we described a process to calculate the likelihood of a text by using a probabilistic suffix tree. We used an algorithm that looked at the next characters in the text, greedily chose the longest strings that were also contained in the PST, and then found the likelihood of those strings in the PST. We will now see that this process effectively described traversing the corresponding probabilistic suffix automaton.

A probabilistic suffix automaton is an automaton (a graph) consisting of nodes that correspond to context (suffix), and vertices representing characters appearing after that suffix, with their respective probabilities. Because suffices of various lengths all map to unique nodes, this effectively transforms our variable level Markov model to a single level one, on which we can then directly perform calculations.

5.1.1 Construction of probabilistic suffix automata

To construct a PSA from a PST, we start with a new automaton/graph with the root of the PST as non-recurrent state. Then we traverse the PST, adding all its states to the automaton. Then, we iterate over all states in the automaton in order to add transitions. We do this by looking at the probability vector for the next character as stored in the PST. For each character that might follow the state in a new sample (i.e., for every character in the alphabet), we find the corresponding next state in the PSA, and create a transition with the probability from the probability vector. If the next state does not exist in the PST, we remove the first character from the current state, and repeat the procedure.

For example, suppose we have the states {{}, `a`, `b`, `ab`} on an alphabet of {`a`, `b`}. (This means that in the text the model was based on, we never saw the sequence `aa`, for example.) Now, to create the arcs for the `a`-state, we have to consider both of the next possible characters. For the `b`, the next state is `ab`, so we draw an arc between `a` and `ab` with the probability of a `b` after the `a` found in the PST. For the `a` as next character, we first look for an `aa` state. But, since this never occurred in the source document, it is not in the PST. So, we remove the first character from the state, leaving us with an empty string, corresponding to the root node of the PST. From this node there is a probability of the `a` occurring, so we create a looping arc on the `a` state with the probability of an `a` occurring as defined in the root state of the PST. The resulting PSA, minus the probabilities, is illustrated in Figure .

In effect, this algorithm does what we did in Section 4.3: it finds the longest possible context to use for a transition probability, so that there is never a transition with probability 0.

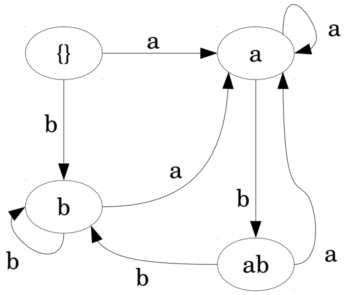

Figure 6: The PSA described in Subsection 5.1.1, without the probabilities.

Having constructed the automaton, we can deduce the corresponding transition matrix. Seeing that this is now an ordinary Markov chain, we can use the techniques introduced in the previous sections to compare models.

5.1.2 Relative entropy rate for PSA models

To calculate the distance between models, we use the *relative entropy rate* between the Markov processes represented by the models [Cai et al., 2006]. The formula for the relative entropy rate for Markov processes P and P^0 is, as shown in the Appendix (A.1.4):

$$h(P||P^0) = \sum_{i,j} \mu(i) P(i,j) \log \frac{P(i,j)}{P^0(i,j)}.$$

There is one issue we need to solve before we are able to calculate the distances between our PST models: we are dividing by the transition probability between two states in the second distribution. This requires that such a transition probability is non-zero, which is not necessarily the case when comparing generated PSA: states that are in one PSA might not be (reachable) in the other. Normally, this would result in an infinite distance. However, since we are not working with true models but with estimated models, we wish to work around this. We can do this by adding all states from the first PST to the second. They are placed where they would normally be placed (i.e. as children of the nodes containing their suffix minus one character), and we copy the probability vector of the parents. Then, we create the PSA based on this modified PST.

6 Experimental results

We performed two types of experiments. The first, in Section 6.1, are to test our plagiarism detection algorithm described in Chapter 4. Then in Section 6.2, we try to compare full document models as described in Chapter 5.

6.1 Plagiarism detection

For this experiment, we have two data sets, labeled 2010−3 and 2009−3. We used the first data set to develop the method and to determine optimal parameters. We then used the second data set to test for efficiency with those parameters. Because cases of plagiarism are not labeled in the source files, we have to check all flagged sections by hand to see if they are correctly flagged as plagiarism by our algorithm. To get an indication of the rate of false negatives (i.e., undetected plagiarism), we manually added some cases of plagiarism. Unfortunately, there is no way to find true false negatives without a different, perfect method.

Based on just the output of our methods it is often impossible to determine the source of plagiarism. A plagiarized section is generally flagged in both the source and the target document. When a section is only flagged in one document, it is likely that that document is the target, since that means that, according to our method, the probability is high that the flagged section was plagiarized.

All data and results shown in this thesis are fully anonymous.

First data set

We used the first data set $(2010-3)$ to develop the method, and to find good parameters. This means that there will likely be some overfitting, but this is not as bad as it often is: since all our inputs are unlabeled, this procedure could theoretically be repeated for every new set of documents: we optimize for results that were the most useful for manual inspection afterwards.

The data set contained 95 submissions for a simple simulation of Conway's Game of Life [Conway, 1970]. After running the algorithm various times with different parameters, we determined the maximum tree depth $k = 4$ and the region width $m = 50$ to be the most useful: these parameters gave us no known false negatives, and a limited amount of false positives. With these parameters, the method identified various types of plagiarism. We have identified some examples of output in Figures 7 - 9.

Figure 7 demonstrates a document that was completely copied from another submission. Many sections are flagged as significant, and almost all sections are somewhat suspicious, as demonstrated by the blue line frequently approaching and crossing the significance line at log(0.05).

Figure 8 is an example of a document where two sections were copied. One long function near the beginning (region 90-200) and a short function in region 1050-1100.

Finally, Figure 9 is an example of the output of the method applied to a document that had no similarities to any other documents. No regions are flagged as plagiarism.

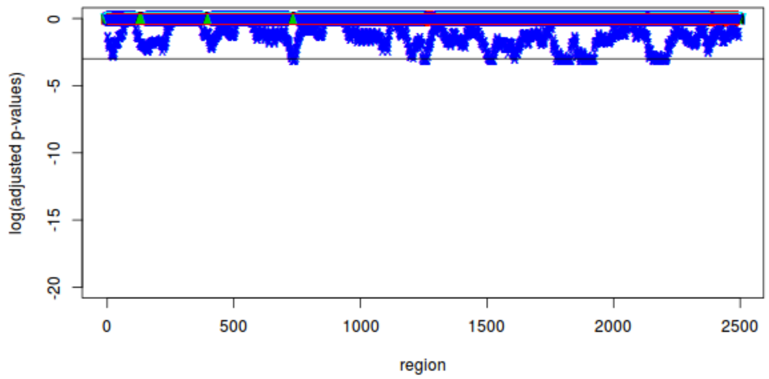

Figure 7: Completely copied code with significant obfuscation.

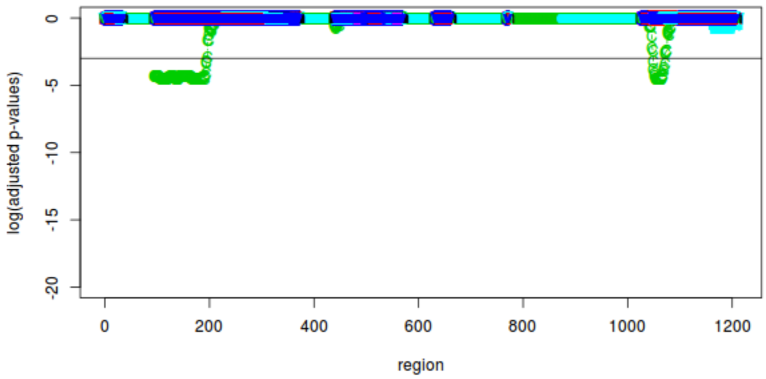

Figure 8: Some functions were copied, but students used some obfuscation.

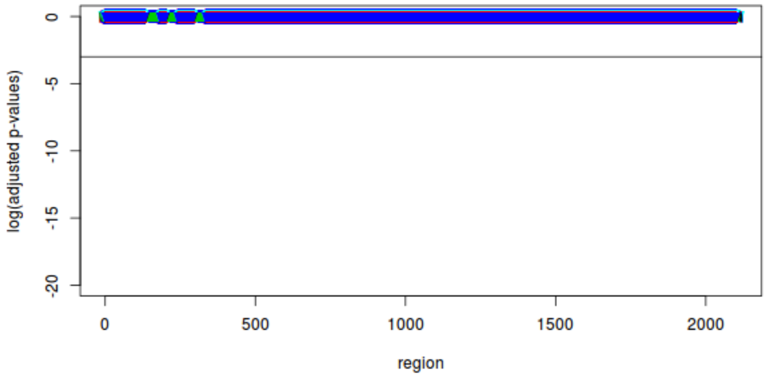

Figure 9: No similarities found.

There were various false positives, which mostly occurred because all 95 submissions implement the same thing. Examples of common false positives within this data set include the function that calculates the sum of neighbors of a point in a matrix and code to copy one matrix to another.

Altogether we found at least six cases of likely plagiarism in this data set, involving 12 files. Closer inspection of all found matches might identify even more. However, we used this data set many times for determining the optimal parameters, so it is better to use a new dataset for validation.

Second data set

Verification of the method and parameters on the second data set (2009-3) gave very promising results, in terms of method performance. Once again, we ran the full method, this time on 70 submissions. We identified 20 unique submissions that had some sections that were, on thorough manual inspection, clearly either based on or the basis for sections in other documents.

There were 8 unique documents with false positives (i.e., identified as containing plagiarism, but containing none) in this data set as well. Most of these occurred when two different authors had sections of similar but very uncommon structure. For example, two authors might include an English-word dictionary directly in their code as array of strings, while everybody else would load them from a text file during run-time. Since string literals are all treated as identical regardless of contents, such dictionaries are flagged as being suspiciously identical.

False negatives are hard to find, since the original data was not labeled. So, to test for false negatives, we used two different methods.

First, to get a rough idea of false positive rates, we ran the algorithm again after creating an extra file that consisted of somewhat obfuscated sections of various lengths, copied from other files. This means that we know that all sections of the document were plagiarized, so any undetected parts would be a false positive.

So, we created a new document in dataset 2009-3 by copying 10 random sections from other submissions. Then we ran the method again on the augmented dataset. Figure 10 shows the results.

There are 8 sections flagged as being significantly likely to be plagiarism. Around region 1000, there is another section recognized (dark blue), but it is not marked as significant. It is possible that, because there were so many other plagiarized sections, the Benjamini-Yekutieli multiple testing adjustment adjusted it down, and that it would be recognized as plagiarism if it occurred on its own. Nevertheless, we will not count it as recognized.

This means that for this particular file, we have a false negative rate of 20%. The code of which no part was detected as plagiarized was at the end of the file. It contained a part of source code that is common in the dataset, and for which there are few unique possible implementations: it removes the leading zeroes from a bignum object.

For a second, more thorough method for determining the false positive rate, we used a program called MOSS [Schleimer et al., 2003] that is commonly used

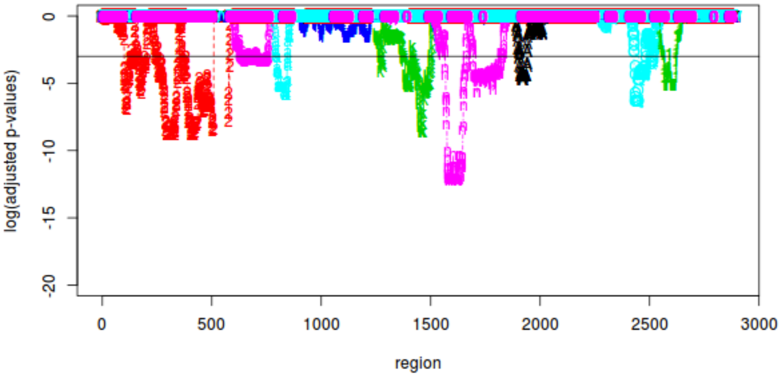

Figure 10: Flagged sections for a custom document consisting of 10 known plagiarized sections. The colored output indicates the locations of the plagiarized sections, with one missing between regions $2000 - 2300$.

	MOSS positive	MOSS negative
True positive	18	2
False negative	4	-

Table 1: Table comparing the results of our method (left) with the results of MOSS.

for plagiarism detection by teachers in this specific domain. It does not output probabilities of plagiarism: instead, it shows all matching sections between documents, ranked by the size of the matching region. This includes many very small matches that are clearly false positives, so we again have to interpret the results manually by looking at most of the matches by hand and evaluating whether they are indeed plagiarism or merely false positives.

The results of the comparison are shown in Table 1. We see that our method identified 20 documents with plagiarism, of which MOSS missed 2. [3]

We can now make a standard 2×2 comparison table for our method, as is shown in Table 2. The sensitivity (recall) of our method is $\frac{20}{20+4} = \frac{5}{6} \approx 0.833$, the specificity is $\frac{38}{38+8} = \frac{19}{23} \approx 0.826$, the precision is $\frac{20}{20+8} = \frac{20}{28} \approx 0.714$, and the F_1 score is $\frac{2 \cdot 20}{2 \cdot 20+8+4} = \frac{40}{52} \approx 0.769$.

[3] Note that MOSS did include those 2 documents in its results, but they were ranked very low, between many false positives. For that reason we had to stop manual inspection much earlier, and count all lower-ranked matches as false positives of MOSS.

	Plagiarism	No plagiarism
Test positive	20	8
Test negative	4	38

Table 2: Table showing the performance of our method.

Some additional observations

In theory, it might be possible to use the calculated likelihoods to determine the direction of plagiarism. Suppose Alice copied a section from Bob. Then we would expect our algorithm to calculate a high likelihood for that section in Alice's document, since it was a part of Bob's work and it was written in Bob's style. Our algorithm would likely also calculate a high likelihood for that section in Bob's document, since it's also part of Alice's model. But, it's not written in Alice's style, so the likelihood calculated for that section in Bob's document might be lower than that for that section in Alice's document.

We were, however, not able to see this effect in practice, presumably because the variance is too high. Another reason could be that the likelihoods are calculated from different models, and therefore not directly comparable.

Similarly, we do not believe the method as-is would be able to detect the difference between normal plagiarism between two documents, and plagiarism by two documents from an identical, unknown, source.

Another limitation of the method is that it is possible to evade it: by changing enough parts of a plagiarized section, it is possible to decrease the likelihood, especially when the changes are spread out. This could be done in many ways. For example, by changing the order of statements, by adding statements or tokens that do nothing, or by moving parts of expressions into intermediary variables.

6.2 Full model comparison experiments

Now, we will try to compare full document models as described in Chapter 5.

To summarize our method, we perform the following steps to compare all documents in a dataset:

1. Create probabilistic suffix trees for all documents.
2. For any combination of two documents, make sure the second PST contains all nodes that the first PST has, by expanding parents, and:
   (a) Create probabilistic suffix automata for both trees.
   (b) Derive the Markov transition matrices P and P^0 from the two PSA.

(c) Calculate the stationary distribution of the first transition matrix P, call it μ.

(d) Calculate

$$\frac{D(\mathbb{P}|\mathbb{P}^0) + D(\mathbb{P}^0|\mathbb{P})}{2} =$$

$$\frac{\sum_{i,j} \mu(i)P(i,j) \log \frac{P(i,j)}{P^0(i,j)} + \sum_{i,j} \mu^0(i)P(i,j) \log \frac{P^0(i,j)}{P(i,j)}}{2}$$

The result of this process consists of distances (dissimilarities) between all documents [Cai et al., 2006].

We applied this method to dataset $2009 - 3$. The resulting dissimilarities are distributed in a way that might indicate that they are interesting (i.e. not near-constant), as shown by a histogram in Figure 11.

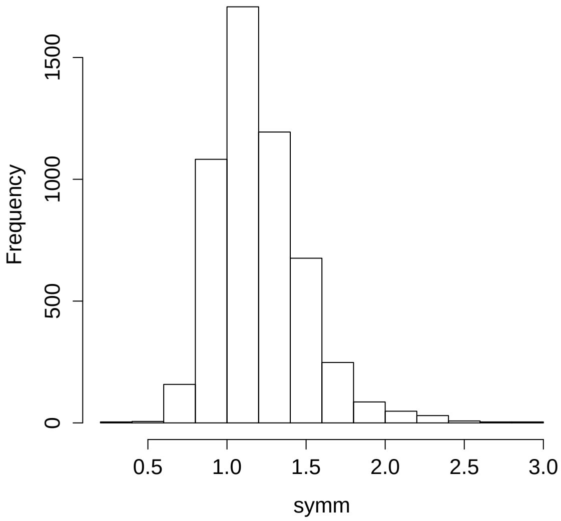

Figure 11: A histogram of dissimilarities between author models from documents.

One thing we can do with these distances is to calculate a phylogenetic tree, i.e., a greedy tree of all documents, to group interesting documents together [Li et al., 2004]. We show such a tree on top of a heatmap of all distances in Figure 12.

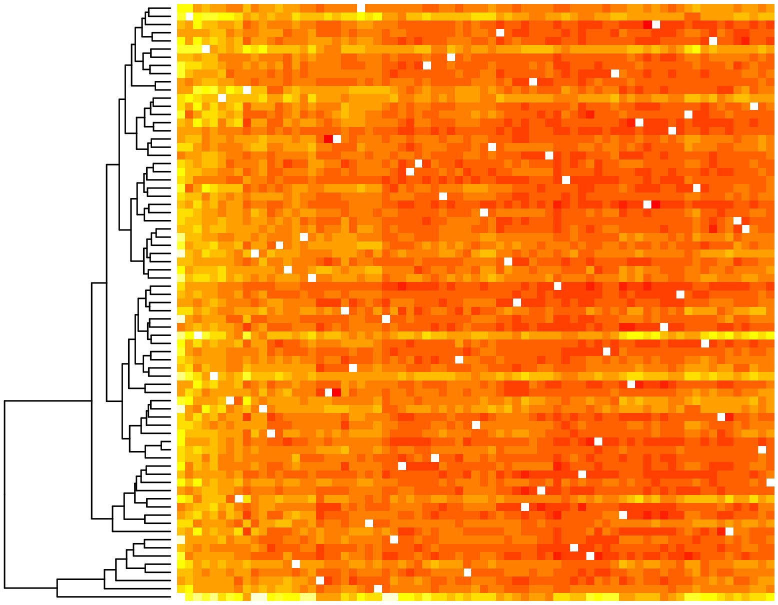

Figure 12: On the left: A phylogenetic tree based on dissimilarities between author models. On the right: dissimilarities, with lighter colors representing lower numbers. On both the x- and y-axis are documents.

Looking at the resulting groupings, we compared some files that were far away in the phylogenetic tree. It seems that the most visible differences between the highest and lowest groups shown include the object syntax used in C++, such as

```
obj->value
```

versus

```
(*obj).value
```

as well as the way to output multi-line strings, such as "<<" on the new line:

```
cout << "text"
     << "more text";
```

versus leaving the "<<" away:

```
cout << "text"
      "more text";
```

Additionally, files that were nearly completely plagiarized show up as very similar: the files we compared in Figure 7 have a distance of 0.276, which is very low, as seen in Figure 11.

Next, we tried classical multidimensional scaling [Borg and Groenen, 2005] to map the documents to two dimensions, in the hopes of finding interesting clusters. As seen in Figure 13, there are no clear clusters. Additionally, the goodness of fit was only 0.23, which makes for a very weak fit.

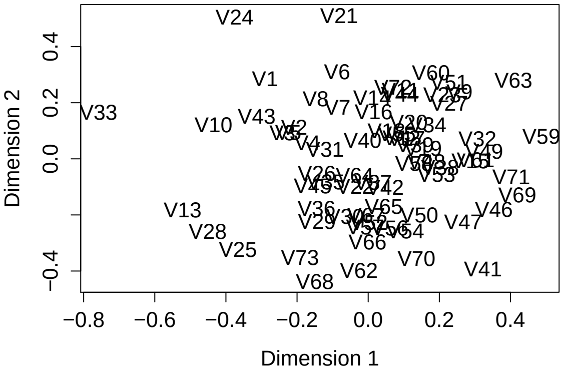

Figure 13: A plot showing the 73 documents V_i from dataset $2009 - 3$ mapped to two dimensions by using classical multidimensional scaling.

Studying the dimensions, we found (by inspecting the source documents) that dimension 1 seems to represent complexity of some sort, as evidenced by the fact that the documents with the highest value on this dimension are often shorter than average, and document 33 is longer. We were unable to determine what the second dimension might represent, as inspecting the highest and lowest documents on this dimension did not result in any insights.

In conclusion, this method successfully calculates distances between models behind documents/document models, and can be used to group documents together to some degree. It is not immediately useful for our goal of detecting plagiarism, given that we already have a robust method that is able to detect plagiarism of short parts of a document. One way to make use of this would be to correlate grades (as given for the documents by a teacher) with the groupings or two-dimensional values found here, to see how well we can predict grades based on the model of a document: students who already have a strong grasp of programming might have a common programming style, whereas students who are new might invent or use very uncommon or non-standard methods.

7 Conclusions

We have introduced a new method for plagiarism detection in program source code. It uses variable-length Markov models to calculate the probabilities of plagiarism of all subsections of computer code. We have shown that it works well on real-world data sets: for a data set of 75 submissions, we found 12 sections of plagiarized code. There were false positives, but those were easily recognized as such when manually looking at the sections. Additionally, we found in Chapter 6.1 that in a file consisting of 10 self-constructed plagiarized sections, we had a false negative rate of 20%. In conclusion, we believe that the method would be useful in practice.

Additionally, we have implemented a way to compare the authors of full source code files. We found that it worked, and were able to create groupings based on the dissimilarities between documents that represented visible differences in style in the underlying documents. Unfortunately, it is unclear what the underlying cause is for these groupings (e.g. experience, study activity, native language). Doing more research into that might be a subject for future research. It might also be worthwhile to find out if similar authors as identified by our full-model comparison technique also received similar grades.

For plagiarism detection, more work can be done to add domain-specific knowledge, for example by including whitespace, variable names and function names in the analysis. The lexicographical analyzer might be replaceable by something that builds syntax trees, and then regularizes them to a format where we can easily compare them.

References

[Benjamini and Hochberg, 1995] Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B*, 57:289–300.

[Benjamini and Yekutieli, 2001] Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *The Annals of Statistics*, 29(4):1165–1188.

[Borg and Groenen, 2005] Borg, I. and Groenen, P. J. (2005). *Modern multidimensional scaling: Theory and applications.* Springer.

[Cai et al., 2006] Cai, H., Kulkarni, S. R., and Verdú, S. (2006). Universal divergence estimation for finite-alphabet sources. *IEEE Transactions on Information Theory*, 52(8):3456–3475.

[Conway, 1970] Conway, J. (1970). The game of Life. *Scientific American*, 223(4):4.

[Cover and Thomas, 2012] Cover, T. M. and Thomas, J. A. (2012). *Elements of information theory.* John Wiley & Sons.

[Harris, 1970] Harris, Z. (1970). Distributional structure. In *Papers in Structural and Transformational Linguistics*, Formal Linguistics Series, pages 775–794. Springer.

[Heintze, 1996] Heintze, N. (1996). Scalable document fingerprinting. In *1996 USENIX Workshop on Electronic Commerce*, volume 3.

[Hochberg and Tamhane, 1987] Hochberg, Y. and Tamhane, A. C. (1987). *Multiple comparison procedures.* John Wiley & Sons, Inc.

[Kesidis and Walrand, 1993] Kesidis, G. and Walrand, J. (1993). Relative entropy between Markov transition rate matrices. *IEEE Transactions on Information Theory*, 39(3):1056–1057.

[Kullback and Leibler, 1951] Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10, page 707.

[Li et al., 2004] Li, M., Chen, X., Li, X., Ma, B., and Vitányi, P. M. (2004). The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264.

[Maurer et al., 2006] Maurer, H., Kappe, F., and Zaka, B. (2006). Plagiarism – A Survey. *Journal of Universal Computer Science*, 12(8):1050–1084.

[Mazeroff et al., 2003] Mazeroff, G., Cerqueira, V. D., Gregor, J., and Thomason, M. G. (2003). Probabilistic trees and automata for application behavior modeling. In *41st ACM Southeast Regional Conference Proceedings*, pages 435–440.

[Rissanen, 1983] Rissanen, J. (1983). A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664.

[Ron et al., 1996] Ron, D., Singer, Y., and Tishby, N. (1996). The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149.

[Schleimer et al., 2003] Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 & 623–656.

[van de Wiel, 2013] van de Wiel, M. (2013). Multiple testing: Introduction & FWER, FDR. *Lecture notes on high-dimensional data analysis.*

A Appendix: Information theory

A.1 Introduction to information theory

In information theory, one primary goal is to quantify the amount of *information* that is contained in a message, in stored data, or in any other medium. For this to be possible we will first try to define what information is for random variables and probabilistic models. We can then apply those definitions to arbitrary messages and data by treating them as realizations of such probabilistic models.

A.1.1 Self-information of an outcome

The first step is to define an information-measure I for an outcome of a random (discrete) variable X over probability space Ω. We want this to have two main properties:

1. If one outcome is less likely than another, it should have a higher information value (consider information value to be a measure of surprisal):
$$\forall A, B \in \Omega : \; P(A) < P(B) \Rightarrow I(A) > I(B).$$
2. This information measure should be additive: the information content of two mutually independent events happening together should be the sum of the information content of those events:
$$\forall A, B \in X : \; P(A)P(B) = P(A \wedge B) \Rightarrow I(A \wedge B) = I(A) + I(B).$$

There is a unique (up to a constant) function I that adheres to both of the above properties:

Definition The *self-information* $I(A)$ of an outcome A is [Shannon, 1948]
$$I(A) = -\log(P(A))$$
where log is the binary logarithm (as it will be throughout this document). We shall say that this measure has the unit *bits*.

Example Let Y be an unfair coin flip, where *heads* has probability $\frac{1}{4}$ and *tails* has probability $\frac{3}{4}$. The self-information of *heads* coming up is now
$$I(heads) = -\log\frac{1}{4} = 2 \text{ bits}.$$
The self-information of *tails* coming up is
$$I(tails) = -\log\frac{3}{4} \approx 0.415 \text{ bits}.$$
We see that the least likely outcome indeed has the highest information-value.

A.1.2 Shannon entropy

Given this definition for the information of an *outcome* of a random variable, we can now do the same for an entire random variable.

Definition The *(Shannon) entropy* of a discrete random variable X is the expected value of the self-information of its outcomes [Shannon, 1948]:

$$H(X) = E(I(X)) = -\sum_{x \in X} P(x) \log P(x).$$

Given that $H(X)$ is the expected value of $I(X)$, it too will have the unit *bits*.

Example Let Z be a fair coin flip having two outcomes, each with probability $\frac{1}{2}$. The entropy of Z is

$$H(Z) = -\frac{1}{2} \log \frac{1}{2} - \frac{1}{2} \log \frac{1}{2} = 1 \text{ bits}.$$

For the unfair coin Y from Section A.1.1, the entropy is:

$$H(Y) = -\frac{1}{4} \log \frac{1}{4} - \frac{3}{4} \log \frac{3}{4} \approx 0.811 \text{ bits}.$$

We can interpret this entropy as the number of bits that are required on average to transmit the outcome of a random variable over a binary stream if we use a smart (i.e., optimally short) encoding, when transmitting many outcomes at once.

A.1.3 Joint and conditional entropy

Similar to the entropy for a single discrete random variable, we can also define the entropy of a pair of random variables given their joint distribution [Shannon, 1948].

Definition The *joint entropy* of two discrete random variables X and Y is the expected value of the self-information of joint outcomes:

$$H(X,Y) = E(I(X,Y)) = -\sum_{x \in X} \sum_{y \in Y} P(x,y) \log P(x,y)$$

Definition The *conditional entropy* of random variable X given Y is defined as the expected value of the self-information of X given Y:

$$H(X|Y) = E(I(X|Y)) = E(-\log P(X|Y)) = -\sum_{x \in X} \sum_{y \in Y} P(x,y) \log P(x|y)$$

A.1.4 Relative entropy

We can use techniques similar to those described above to measure the difference between discrete random variables. This will allow us to compute some distance between different distributions. If we can apply this to empirical distributions, we can possibly use it to compare sources of certain messages and data.

Definition The *relative entropy* or *Kullback-Leibler divergence* between two discrete random variables that both have the same outcome space Ω, with probability density functions $P(x)$ and $Q(x)$, is defined as the expected value of the log-likelihood ratio [Kullback and Leibler, 1951]:

$$D(P||Q) = E\left(\log \frac{P(x)}{Q(x)}\right) = \sum_{x \in \Omega} P(x) \log \frac{P(x)}{Q(x)}.$$

Even though this relative entropy is sometimes called *Kullback-Leibler distance*, it is not a true distance measure: it is not symmetric, nor does it generally satisfy the triangle inequality. This relative entropy measures how many bits we need to code samples from P when using a code based on Q. It can roughly be seen as a measure of complexity of P to someone who is experienced with Q. This makes it intuitive that the measure is not symmetric: if X is highly complex and Y is simple (has a very low entropy), then it is intuitively clear that $D(X||Y)$ is very high, and $D(Y||X)$ is very low.

Example Consider the two coin flip distributions described in Sections A.1.1 and A.1.2. One, distribution Z with probability density function $Z(x)$, was a fair coin with probability $\frac{1}{2}$ of landing heads up. The other, distribution Y with probability density function $Y(x)$, had probability $\frac{1}{4}$ of landing heads up. The relative entropy $D(Z||Y)$ is now:

$$D(Z||Y) = \frac{1}{2} \log \frac{1/2}{1/4} + \frac{1}{2} \log \frac{1/2}{3/4} \approx 0.21 \text{ bits.}$$

Note that the reverse relative entropy $D(Y||Z)$ is:

$$D(Y||Z) = \frac{1}{4} \log \frac{1/4}{1/2} + \frac{3}{4} \log \frac{3/4}{1/2} \approx 0.19 \text{ bits.}$$

So even for these simple distributions, the relative entropy is not symmetric.

Lemma Gibbs' inequality: For any two distributions P and Q over outcome space Ω:

$$D(P||Q) \geq 0.$$

Corollary $D(P||Q) = 0 \Leftrightarrow P = Q$

A.2 Entropy rates for random processes

Earlier we have defined *entropy* for random variables: it was a function of the probabilities of possible outcomes. We will now attempt to do the same for random processes. However, since random processes can have an infinite number of outcomes (paths), the definition is slightly different than it was for random variables.

A.2.1 Entropy rate

Given a random process X, we define its *entropy rate* to be [Shannon, 1948]

$$H(X) = \lim_{n\to\infty} \frac{1}{n} H(X_1, \dots, X_n)$$

if this limit exists, where $H(X_1, ..., X_n)$ is the *joint entropy* of $X_1, ..., X_n$, defined as

$$H(X_1, ..., X_n) = -\sum_{x_1} ... \sum_{x_n} P(x_1, ..., x_n) \log P(x_1, ..., x_n).$$

Example Suppose X is a series of independent fair coinflips. Remember that the entropy of a single coinflip is 1 bit. We can calculate the join entropy of two independent coinflips X_k and X_m where $k \neq m$:

$$H(X_k, X_m) = -\sum_{i=1}^{2}\sum_{j=1}^{2} \frac{1}{4} \log \frac{1}{4} = 2.$$

In fact, since all the coinflips are independent, the join entropy of n coinflips $X_1,\ \dots,\ X_n$ is:

$$H(X_1,\ \dots,\ X_n) = n \cdot H(X_1) = n.$$

This means that the *entropy rate* of our random process X is equal to

$$H(X) = \lim_{n\to\infty} \frac{1}{n} H(X_1, \dots, X_n) = 1.$$

Example Let X be an irreducible aperiodic Markov chain defined on a finite set of states with transition probabilities p_{ij}. Since X is irreducible and aperiodic, it has some stationary distribution $\bar{p}$. For such a process [Cover and Thomas, 2012]

$$H(X) = -\sum_{i,j} \bar{p}_i p_{ij} \log p_{ij}.$$

A.2.2 Relative entropy rate

For random processes P and Q, both over a common probability space X, we now define the *relative entropy rate* to mean

$$h(P||Q) = \lim_{n\to\infty} \frac{1}{n} D(P^n||Q^n),$$

where $P^n$and Q^ndenote finite realizations from processes P and Q of length n, and D is the Kullback-Leibler divergence. We can rewrite this to:

$$h(P||Q) = \lim_{n\to\infty} \frac{1}{n} \sum_{x^n \in X^n} P(x^n) \log \frac{P(x^n)}{Q(x^n)}.$$

As the definition of *entropy rates* extended the definition of *entropy* to random processes, *relative entropy rate* extends the definition of *relative entropy (Kullback-Leibler divergence)* to random processes.

A.2.3 Relative entropy rate for Markov processes

We can apply the concept of relative entropy rates to Markov processes. After a lengthy deduction [Kesidis and Walrand, 1993], we find that the relative entropy between Markov transition probability matrices P and P^0 is:

$$h(P||P^0) = \sum_{i,j} \mu(i) P(i,j) \log \frac{P(i,j)}{P^0(i,j)},$$

where μ is the stationary distribution of Markov model P.