Style Recognition in Music with Context Free Grammars and Kolmogorov Complexity

by

Tiasa Mondol

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Sciece

Waterloo, Ontario, Canada, 2020

© Tiasa Mondol 2020

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The Kolmogorov Complexity of an object is incomputable. But built in its structure is a way to specify description methods of an object that is computable in some sense. Such a description method then can be exploited to quantify the bits of information needed to generate the object from scratch. We show that Context-Free Grammars form such a viable description method to specify an object and the size of the grammar can be used to estimate the Kolmogorov Complexity. We use such estimation in approximating the Information Distance between two musical strings. We also show that such distance measure in music can be used to recognize the genre, composer and style and also for music classification.

Acknowledgements

In the first place, I would like to thank wholeheartedly my advisor Professor Daniel G. Brown for his insights, guidance, candor and awesomeness. Although an enormous amount of patience is required to supervise me as a graduate student, Professor Brown had not lost his cool and was always quick to offer help whenever I needed it. The weekly discussions I had with him always inspired me to study more in depth and do solid research. I can not imagine nor probably could muster up the courage to generate this thesis without his assistance and encouragement.

I am thankful to Professor Ming Li, Professor Bin Ma and Professor Jeffrey Shallit for graciously letting me inundate them with questions and helping me find the answers. I am grateful to the readers of my thesis: Professor Lap Chi Lau and Professor Ming Li for generously offering their time and energy to help me graduate.

I also want to express my gratitude to Jeff Ens (Simon Fraser University, Canada), Philippe Pasquier (Simon Fraser University, Canada) and David Meredith (Aalborg University, Denmark) for kindly providing me with the resources from their research which helped me gain insights about what I should do in mine.

Finally I am thankful to my family for their constant love and support; my brother for taking care of me when I fell ill, my parents for supplying me with food and shelter and also for allowing me to attend classical concerts at the Kitchener-Waterloo Symphony Orchestra: the consequence of which has left me more musically informed and appreciative of this great form of art.

Dedication

Dedicated to my late grand-father, who loved singing and listening to Bengali folk songs.

Table of Contents

List of Tables x

List of Figures xi

List of Algorithms xiii

1 Introduction 1

2 Theory 3

2.1 Preliminaries 3

2.1.1 Kolmogorov Complexity 3

2.1.2 $\mathbf{K}(\mathbf{x}|\mathbf{y})$ vs. $\mathbf{K}(\mathbf{x}|\mathbf{y}^*)$ vs. $\mathbf{K}(\mathbf{x},\mathbf{y})$ vs. $\mathbf{K}(\mathbf{xy})$ 3

2.1.2.1 Semi-computability 4

2.1.3 Mutual Information 4

2.1.4 Algorithmic Information Distance 5

2.1.4.1 Derivation from Reversible Computation 5

2.1.5 Symmetry of Algorithmic Information 6

2.1.5.1 Derivation from Mutual Information 7

2.1.6 Normalized Information Distance 7

2.1.7 $d(x, y)$ as a Distance Metric 7

2.1.8 Density Properties of $d(x, y)$ 10

2.2 Approximating $K(x)$ with a Context Free Grammar of x 14
2.2.1 Preliminaries 15
2.2.1.1 Context Free Grammar 15
2.2.1.2 NP-Hardness 16
2.2.1.3 Global Context Free Grammars 16
2.2.2 Algorithms for Global Context Free Grammars 16
2.2.2.1 Most-Frequent Algorithm 17
2.2.2.2 Longest-First Algorithm 18
2.2.2.3 Greedy Algorithm 18
2.2.3 $|G(x)|$ as $K(x)$ 19
2.2.3.1 ϕ as a Decoder 19
2.3 Constrained Best-Fit Model Selection 21
2.3.1 Two-Part Code 21
2.3.2 "Best-fit" Structure Function 22
2.3.2.1 Randomness Deficiency 22
2.3.3 Kolmogorov's Original Structure Function 23
2.3.4 Minimum Description Length Structure Function 23
2.3.5 Model Class of Context Free Grammar 23
2.3.5.1 Constructing G 24
2.3.5.2 Constrained Model Selection for Context Free Grammars . 24

3 Related Work **26**
3.1 Algorithmic Information Distance 26
3.1.1 Roots in Bioinformatics 26
3.1.2 Application in other fields 27
3.2 Approaches to Music Representation and Compression 28
3.2.1 Lossless Data Compressors 28
3.2.2 Geometric Compressors 28
3.2.3 Grammar Based Compressors 31

4 Experiments and Observations **33**
4.1 Music As Low-Entropy Strings . . . 33
4.1.1 Representing Symbolic Music as Strings . . . 33
4.1.1.1 Manual Transcription of Vocal Melody of Rock Songs . . . 34
4.1.1.2 Automatic Transcription . . . 35
4.2 Approximating the Smallest CFG . . . 37
4.2.1 An Overview . . . 37
4.2.2 Implementation of the Global Algorithms . . . 37
4.2.2.1 Counting Non-overlapping Occurrences . . . 38
4.2.2.2 Greedy Algorithm . . . 39
4.2.2.3 Most Frequent Algorithm . . . 40
4.2.2.4 Longest First . . . 41
4.3 Approximating Conditional Kolmogorov Complexity, $K(x|y^*)$ . . . 42
4.3.1 Compressing x with y^* . . . 42
4.4 Approximating Combined Kolmogorov Complexity, $K(x, y)$ . . . 43
4.5 Runtime Analysis . . . 49
4.6 Visualization and Emergent Properties of the Generated Context Free Grammars . . . 51
4.6.1 Parse Tree of x (without compressing it with y^*) . . . 51
4.6.2 Compression Ratio . . . 53
4.6.3 Parse Tree of x (after compressing it with y^*) . . . 54
4.6.4 Inter and Intra-Corpora Normalized Distance for the Three Genres 56
4.7 2-Dimensional Scaling and Support Vector Classification . . . 57
4.7.1 Genre Specific Style Recognition . . . 58
4.7.2 Composer (Same Genre) Specific Style Recognition . . . 62
4.7.3 Composition (Same Composer) Specific Style Recognition . . . 65
4.8 Best-Fitting Composer Selection . . . 68
4.8.1 Method . . . 68
4.8.2 Effectiveness of the Generated Model . . . 70
4.8.3 Composer Classification . . . 72

5 Future Work **74**

5.1 Improving the runtime of Context Free Grammar Generation 74

5.2 Generalized Context Free Grammar and Generative Models 75

5.3 Applications in other Low-entropy Strings 76

6 Conclusion **77**

References **78**

APPENDICES **85**

A Auxiliary Proofs **86**

B Trees **89**

B.1 Suffix Tree 89

B.1.1 Definition 89

B.1.2 Pattern Searching 90

B.1.2.1 Pattern Occurrences 90

B.2 Generalized Suffix Tree or Patricia Tree 92

B.2.1 Pattern Searching and Counting Occurrences 92

C Full List of Music Pieces Used in the Project **93**

C.1 Rock Pieces 93

C.2 Classical Pieces 93

C.3 Jazz Pieces 97

List of Tables

2.1 Approximation Ratios of algorithms generating Global CFG 19

4.1 Compression Ratio for the three Genres 53
4.2 Intra-corpora Distance of three Genres 56
4.3 inter-corpora Distance of three Genres 56
4.4 Best Fit Model Selection Success Rate 73

C.1 Classical Pieces Used 93
C.2 Jazz Pieces Used 97

List of Figures

3.1 Visualization of COSIATEC on Chopin's Op. 28 Prelude No.1 29
3.2 Density Properties of General Purpose and Specialized Compressors 31

4.1 Automatic Transcription of "Hey Jude" by "The Beatles" 36
4.2 Comparing Two Approximations of $K(x, y)$ 45
4.3 Comparing the sizes of the CFGs involved in the Two Approximations of $K(x, y)$ 49
4.4 Parse Tree for "Hey Jude" by "The Beatles" 52
4.5 Conditional Compression of "Eleanor Rigby" with "I Want to Hold Your Hand" 55
4.6 Conditional Compression of "I Want to Hold Your Hand" with "Eleanor Rigby" 55
4.7 2-Dimensional Scaling of three Composers 57
4.8 Support Vector Classification with Linear Kernel of three Composers . . . 58
4.9 2-Dimensional Scaling and Linear Classification of rock and classical music 59
4.10 2-Dimensional Scaling and Linear Classification of rock and jazz music . . 60
4.11 2-Dimensional Scaling and Linear Classification of rock, jazz and classical music 61
4.12 2-Dimensional Scaling and Linear Classification of three classical composers 63
4.13 2-Dimensional Scaling and Linear Classification of two rock composers . . . 64
4.14 2-Dimensional Scaling and Linear Classification of Two Styles of Compositions of Chopin: Études and Preludes 66

4.15 2-Dimensional Scaling and Linear Classification of Two Styles of Compositions of Chopin: Nocturnes and Preludes 67

B.1 Pattern searching in suffix tree 90

List of Algorithms

1 Algorithm for Global CFG 17
2 ϕ-General Decoder 20
3 Algorithm for Global Grammars 21
4 Cosiatec Pseudocode 29
5 Patricia Tree 37
6 CFGGlobalAlgorithm 38
7 Removing Overlap 39
8 Greedy Pattern Selection 40
9 Maximally Repeated Pattern Selection 41
10 Longest Pattern Selection 41
11 Ordering of the Non-Terminals Γ_y 42
12 Compressing x with D_{Γ_y} 43
13 Approximating G_i 69
14 Best Fitting Model/ Composer 69
15 CFGGlobalAlgorithmImproved 75
16 Finding Pattern Occurences 91
17 Node Class: Finding the Positions of occurrences 91

Chapter 1

Introduction

> "If there are alternative explanations for a phenomenon, then, all other things being equal, we should select the simplest one."
>
> Occam's Razor Principle,
> William of Ockham
> (c.1285-1347)

The Kolmogorov Complexity is an absolute measure of information in an object. Such measurement of information has the advantage that it refers to individual objects and not to objects treated as members of a set of objects with a probability distribution given on it (A.N. Kolmogorov). It is also desirable to have a similar absolute notion for the *information distance* between two objects. Such a notion is universal in the sense that it covers all other notions of computable informational distances [40]. Such a notion should also be asymptotically machine-independent and can serve as an absolute measure of the informational, or cognitive distance between two discrete objects x and y [5]. A universal information distance between two strings satisfying these requirements is the minimal quantity of information required to translate between x and y. However, The universality requirement necessarily makes our information distance not computable [40]. Instead we focus on a weakened notion of Information Distance that is still in a broad sense computable. Such notion also requires a feasible and computable estimate of the Kolmogorov Complexity of objects. The focal point of this thesis, is such a method that approximates

Kolmogorov Complexity: short Context Free Grammars (CFG).
The smallest Context Free Grammar of a string is an NP Hard problem [15] and many algorithms exist to generate only approximations of the smallest CFG. They have been categorized in to two types: online and offline or global [15]. Online and more commonly known algorithms include LZ78 [68], Sequitor [50] etc. While they are specifically designed to achieve higher compressions of strings, its not clear whether they can find internal structures of the corresponding strings as they only examine a sequence in a left to right incremental manner. So in this project we focus on the global CFG algorithms that take strings as a whole and process them globally.

A short CFG of a string generated by the global algorithms of is also useful in identifying important repeated and hierarchical patterns, a notion that is known in the Kolmogorov realm as extracting out the regularities in an object. Besides, these global methods can also be extended to find hierarchical structures in low-entropy objects and capture meaningful information. These properties make offline Context Free Grammars ideal for analyzing music, which is known to be hierarchical in structure [34].

In this project, we implement algorithms to generate short and global Context Free Grammars of a string and use the grammar size to approximate the string's Kolmogorov Complexity $K(x)$. We also demonstrate the use of $\frac{K(x|y^*)+K(y|x^*)}{K(x,y)}$ as a Normalized Distance Metric to approximate pairwise Information Distance between two musical objects. We finally show whether such measurement can be fruitful in recognizing style or structure in music, a task that is known to be complicated and controversial.
In chapter 2, we lay out the theoretical foundation of our thesis. We introduce the Normalized Information Distance (NID) we use for our project and examine its properties.
In chapter 3, we discuss the related work that has been done in this field and how our project differs and improves on the existing ideas.
Finally in chapter 4, we apply the NID in the realm of symbolic music and experiment whether such universal notion is useful for style recognition in music.

Chapter 2

Theory

2.1 Preliminaries

2.1.1 Kolmogorov Complexity

Formally, the Kolmogorov Complexity of a binary string x is the length of the shortest binary program x^* to compute x on a universal computer such as a universal Turing Machine [40]. Thus, $K(x) = |x^*|$. Note since $K(x)$ is the ultimate, lower bound on the length of the shortest description of x, it is not computable [40]. However, it is *upper-semicomputable*:

Definition 1. A real valued function $f : \mathbb{N} \to \mathbb{R}$ is *upper-semicomputable* if there exists a rational valued recursive function $g(x, y, t)$ such that $g(x, y, t + 1) < g(x, y, t)$ and $\lim_{t\to\infty} g(x, y, t) = f(x, y)$ [67]. It is *lower-semicomputable* if $-f(x, y)$ is upper-semicomputable.

In other words, $K(x)$ can be approximated *from above*. So, in the subsequent analysis, we only require x^* to be *approximated* but x to be computed in a lossless manner from its description x^*.

2.1.2 $\mathbf{K(x|y)}$ vs. $\mathbf{K(x|y^*)}$ vs. $\mathbf{K(x,y)}$ vs. $\mathbf{K(xy)}$

Definition 2. $K(x|y)$ is the conditional Kolmogorov Complexity of x relative to y. It is defined similarly as the length of the shortest program to compute x when y is provided as an auxiliary input to the computation.

Definition 3. In contrast, $K(x|y^*)$ is defined to be the length of the shortest program to compute x when a shortest program of y, y^*, is provided as an auxiliary input. Conditional complexity in the form of $K(\cdot|\cdot^*)$ was first introduced by G.J. Chaitin in [13].

Definition 4. $K(x, y)$ is defined as the length of the shortest Turing Machine that outputs x and y in a way so that it's possible to tell them apart. A very useful consequence of [28] shows that:

$$K(x, y) = K(x) + K(y|x^*) = K(y) + K(x|y^*) \tag{2.1}$$

Definition 5. Finally, $K(xy)$ is the length of the shortest program to compute the concatenation of x and y without the output-reader having the ability to differentiate them [15].

2.1.2.1 Semi-computability

It is easy to see that $K(x|y)$, $K(x, y)$ and $K(xy)$ are upper-semicomputable as functions of x and y [37, 40]. However, $K(x|y^*)$ behaves differently.
$K(x|y^*)$ is not upper-semicomputable as a *function of x and y* unless $K(K(y)|y) = O(1)$ [40], but y with $K(K(y)|y) \gg O(1)$ exists [28]. For a more complete proof the readers are referred to [40]. $K(x|y^*)$, however, is upper-semicomputable with respect to x and y^* [37]. It is this property that we shall use in this thesis.
In general, $K(x|y^*)$ is an alternative but less natural definition of conditional information than $K(x|y)$, but it has many useful algorithmic properties that are presented in [40, 13, 14] and as we shall also see moderately in the subsequent sections.

2.1.3 Mutual Information

The information about x contained in y is defined as [15]:

$$I(y : x) = K(x) - K(x|y^*) \tag{2.2}$$

Intuitively $K(x|y^*)$ captures the features of x that can not be described when the shortest description of y, y^*, is provided. By subtracting this *information* from the ultimate description of x, $K(x)$ we get the information about x contained in y. Notice from (2.1) that up to an additive $O(1)$ term the following holds:

$$K(x) - K(x|y^*) = K(y) - K(y|x^*)$$
$$I(y : x) = I(x : y)$$

Because of this symmetric nature: $I(y : x)$ or equivalently $I(x : y)$ is known as the *mutual algorithmic information* [37].

2.1.4 Algorithmic Information Distance

Ideally, the *information distance* between two strings x and y is the length of the shortest binary program that computes x from y and vice versa [5]. Being shortest, this program will take advantage of any overlap between the information needed to transform x into y and the information for transforming y into x [40]. In [5], it was shown that the *Algorithmic Information Distance* equals, upto an additive $O(\log \max\{K(x|y), K(y|x)\})$ term to the maximum of conditional Kolmogorov Complexities of the two strings:

$$E(x, y) = \max\{K(x|y), K(y|x)\} \tag{2.3}$$

Although the shortest program $p : K(x|y) = |p|$ that computes x from input y is not the same as the shortest program $q : K(y|x) = |q|$ that computes y from input x, in some simple cases, these programs can be the same when *complete overlap* of information $K(x|y) = K(y|x)$ is achieved [5]. Generally, either $K(y|x) > K(x|y)$ or $K(x|y) > K(y|x)$ can happen.

Without the loss of generality, let $E(x, y) = K(y|x)$. But $K(y|x)$ itself is unsuitable as the optimal information distance because of non-trivial asymmetry between $K(y|x)$ and $K(x|y)$ [40]. This can be remedied by defining $E(x, y)$ as the following *sum distance*:

$$E(x, y) = K(x|y) + K(y|x) \tag{2.4}$$

which holds up to $O(\log(K(x|y)+K(y|x)))$ additive error. The resulting metric will overestimate the information for x to y transformation in case there is some redundancy between the information required to get from x to y and the information required to get from y to x [5]. But when there is minimal overlap, such metric will constitute a more accurate distance between x and y.

2.1.4.1 Derivation from Reversible Computation

The above definition of *Algorithmic Information Distance* in (2.4) was proposed by Zurek in his 1989 Nature paper [69] as a method to capture the *total amount* of *information* needed to translate between x and y both ways. Choosing $K(x|y)$ when $K(x|y) = \max\{K(x|y), K(y|x)\}$ gives us the amount of information needed for an one-way transformation from y to x [39]. We are *irreversibly* losing information about going from x

to y unless of course we keep a history of the operations that take place in the transformation. However, such history-keeping will increasingly fill up the computer memory and eventually this information has to be irreversibly erased to free the memory [6]. This situation can be avoided by supplying the machine with x and a program p that is used to transform x into y and output a program q. When y and q are supplied to the machine, x and p are produced as outputs. So, with $|p| + |q|$ extra bits of information we can achieve $x \to y$ and $y \to x$ both way transformations. Notice: programs p and q are such that $K(y|x) = |p|$ and $K(x|y) = |q|$ hold up to a logarithmic additive error. Hence the total amount of information needed to transform x into y and then in a reversible manner y into x is $K(x|y) + K(y|x)$, which is where the definition of *Algorithmic Information Distance* stems from in 2.4 [69].

2.1.5 Symmetry of Algorithmic Information

We further improve the definition of $E(x, y)$ in (2.4) by taking note of the fact that: to obtain a sharper analogue between the optimal information distance $E(x, y)$ and $K(x|y) + K(y|x)$, a conditional term $K(\bullet|x)$ needs to be replaced by $K(\bullet|x^*)$. We necessarily require the extra information in $K(x)$ because of the possible high-complexity of $K(K(x)|x)$ for some strings [40]:

Recall from the previous section that the sum distance $K(x|y)+K(y|x)$ between x and y is the total amount of information needed to convert x to y and vice versa. Suppose, $|x| = n$ and we supply a Universal Turing Machine U with x and a shortest self-delimiting program $p : |p| = K(y|x)$ in a way such that p transforms x to y besides producing a shortest self-delimiting program q. Program p being self delimited, U needs extra $O(\log K(x))$ bits to tell p and x apart. Because if x has some redundancies, it takes no more than $O(K(x))$ bits to specify the randomness and $O(\log(K(x)))$ bits to delimit it. However, for some strings x with length n, $K(K(x)|x)$ is of high complexity: $K(K(x)|x) \geq \log n - \log\log n + O(1)$ [28]. Hence, U needs the extra information in $K(x)$ apart from x. This need for extra bits can be supplemented if we supply U with x^*, instead of x, such that $|x^*| = K(x)$, is self-delimited (by definition) and capable of generating x on U. The same argument applies to q and y.

Thus rewriting (2.4) we arrive at the final approximation of Algorithmic Information Distance for this thesis [40]:

$$E(x, y) = K(x|y^*) + K(y|x^*) \tag{2.5}$$

2.1.5.1 Derivation from Mutual Information

(2.5) has an intuitive derivation from the idea of *mutual algorithmic information*. The mutual information of two finite binary strings x and y is the amount of information that is shared between them (section 2.1.3). $K(x, y)$, on the other hand, can be thought of an amalgamation of the shared information between x and y (since $K(x, y)$ is the length of the shortest program p from which a universal Turing machine can compute both x and y, it should take care of the redundancy or shared information between them), incompressible or random information in x and incompressible or random information in y. Hence, if we subtract $I(x : y)$ from $K(x, y)$ we are left with the total amount of information by which x and y differ from each other. Thus the algorithmic information distance is equal to this amount by a constant additive term [37]:

$$E(x, y) = K(x, y) - I(x : y)$$

From 2.1 and 2.2 get:

$$\begin{aligned} E(x, y) &= K(x) + K(y|x^*) - K(x) + K(x|y^*) \\ &= K(x|y^*) + K(y|x^*) \end{aligned}$$

2.1.6 Normalized Information Distance

$E(x, y)$ is an absolute measure of distance. But to have a practical distance that expresses similarity or dissimilarity, we are interested in a relative version of $E(x, y)$. For example: if two strings of $1,000,000$ bits differ by 1000 bits of information distance, then we are inclined to think that those two strings are relatively similar. But if two strings of 1000 bits differ by 1000 bits then we find them very different.
So we normalize $E(x, y)$ in (2.5) to obtain a universal similarity metric.

Definition 6. The *Normalized Information Distance*(NID) between two binary sequences x and y is defined as [40]:

$$NID(x, y) = d(x, y) = \frac{K(x|y^*) + K(y|x^*)}{K(x, y)} \tag{2.6}$$

2.1.7 $d(x, y)$ as a Distance Metric

Definition 7. Let S be a nonempty set of objects and R^+ be the set of non-negative real numbers. A *Distance Metric* is a function $D : S \times S \rightarrow R^+$ satisfying the following (in)equalities [18]:

- $D(x,y) = 0$ iff $x = y$(the identity axiom)
- $D(x,y) = D(y,x)$(the symmetry axiom), and
- $D(x,y) \leq D(x,z) + D(z,y)$ (triangle inequality)

Definition 8. Given two sequences x and y, recall the function $d(x,y)$ by:

$$d(x,y) = \frac{K(x|y^*) + K(y|x^*)}{K(x,y)}$$

Lemma 1. *$d(x,y)$ satisfies the metric (in)equalities up to an additive precision $O(\frac{1}{K})$ where K is the maximum of the Kolmogorov Complexities of the objects involved in the (in)equality.*

Claim 1. $d(x,y)$ is symmetrical up to $O(1)$ additive constant term.

Proof.

$$\begin{aligned} d(x,y) &= \frac{K(x|y^*) + K(y|x^*)}{K(x,y)} \\ &= \frac{K(y|x^*) + K(x|y^*)}{K(y,x)} \\ &= d(y,x) + O(1) \end{aligned}$$

following the fact that $K(x,y) = K(y,x)$ up to an additive constant term [40]. □

Claim 2. $d(x,y)$ satisfies the identity axiom up to precision $O(\frac{1}{K})$

Proof. Note that, since x^* is the shortest program that computes x, from x^* we can compute $\langle x, K(x)\rangle$. Conversely, given $\langle x, K(x)\rangle$, we can enumerate all shortest programs for x and the first program that halts with output x on an universal Turing machine U is denoted by x^*. Hence, x^* and $\langle x, K(x)\rangle$ contain the same information although they are not identical objects. Below we can replace x^* by $\langle x, K(x)\rangle$.

$$\begin{aligned} d(x,x) &= \frac{K(x|x^*) + K(x|x^*)}{K(x,x)} \\ &= \frac{2K(x|\langle x, K(x)\rangle) + O(1)}{K(x,x)} \end{aligned}$$

[28] shows that up to an fixed additive constant $O(1)$, independent of x, y the following holds:

$$K(x, y) = K(x) + K(y|\langle x, K(x)\rangle)$$

Setting $y = x$:

$$\begin{aligned} d(x, x) &= \frac{2K(x|\langle x, K(x)\rangle) + O(1)}{K(x) + K(x|\langle x, K(x)\rangle) + O(1)} \\ &= O(\frac{1}{K(x)}) \end{aligned}$$

□

To show that $d(x, y)$ is a metric up to the required precision, it remains to prove the triangle inequality.

Claim 3. $d(x, y)$ satisfies the triangle inequality $d(x, y) \le d(x, z) + d(z, y)$ up to an additive error term of $O(\frac{1}{K})$.

Proof. To show that, $d(x, y)$ satisfies the triangle inequality, we need to show that:

$$\frac{K(x|y^*) + K(y|x^*)}{K(x, y)} \le \frac{K(x|z^*) + K(z|x^*)}{K(x, z)} + \frac{K(z|y^*) + K(y|z^*)}{K(z, y)}$$

Notice that it is sufficient to show the following two inequalities [36]:

$$\frac{K(x|y^*)}{K(x, y)} \le \frac{K(x|z^*)}{K(x, z)} + \frac{K(z|y^*)}{K(z, y)}$$

$$\frac{K(y|x^*)}{K(x, y)} \le \frac{K(z|x^*)}{K(x, z)} + \frac{K(y|z^*)}{K(z, y)}$$

The two inequalities being symmetric, proving the first one will be sufficient.
For all x, y, z, the following *Directed Triangle Inequality* holds up to an additive constant term [40]:

$$K(x|y^*) \le K(x, z|y^*) + O(1) \le K(z|y^*) + K(x|z^*) + O(1) \text{ and,}$$

$$K(y|x^*) \le K(y, z|x^*) + O(1) \le K(z|x^*) + K(y|z^*)$$

Let,

$$\Delta_1 = K(z|y^*) + K(x|z^*) - K(x|y^*)$$

$$\Delta_2 = K(z|x^*) + K(y|z^*) - K(y|x^*)$$

Then, using the symmetry of algorithmic information theory $K(x,y) = K(y) + K(x|y^*) = K(x) + K(y|x^*)$ from [40]:

$$\begin{aligned}
\frac{K(x|y^*)}{K(x,y)} &\leq \frac{K(z|y^*) + K(x|z^*) + \Delta_1}{K(x,y)} \\
&\leq \frac{K(z|y^*)}{K(x,y)} + \frac{K(x|z^*)}{K(x,y)} \\
&\leq \frac{K(z|y^*)}{K(y) + K(x|y^*)} + \frac{K(x|z^*)}{K(x) + K(y|x^*)} \\
&\leq \frac{K(z|y^*)}{K(y) + K(z|y^*) + K(x|z^*) - \Delta_1} + \frac{K(x|z^*)}{K(x) + K(z|x^*) + K(y|z^*) - \Delta_2} \\
&\leq \frac{K(z|y^*)}{K(z,y) + K(x|z^*) - \Delta_1} + \frac{K(x|z^*)}{K(x,z) + K(y|z^*) - \Delta_2} \\
&\leq \frac{K(z|y^*)}{K(z,y) + K(x|z^*)} + \frac{K(x|z^*)}{K(x,z) + K(y|z^*)} \\
&\leq \frac{K(z|y^*)}{K(z,y)} + \frac{K(x|z^*)}{K(x,z)}
\end{aligned}$$

This proves the first inequality. The second inequality is proved symmetrically and both are satisfied up to an additive error term $O(\frac{1}{K})$ □

Clearly, $d(x,y)$ takes values in the range $[0, 1 + O(\frac{1}{K})]$. To show that $d(x,y)$ is an admissible metric, we now move to the next part.

2.1.8 Density Properties of $d(x,y)$

Definition 9. An *admissible distance* $D(x,y)$ is a total, upper-semicomputable, non-negative normalized (it should give a distance 0 when objects are maximally similar and distance 1 when they are maximally dissimilar [40]) function on the pairs x, y of binary strings that have all the metric properties and satisfies the following density conditions [37]: For each $x \in \{0,1\}^*$ and every constant $e \in [0,1]$

$$|\{y : D(x,y) \leq e \leq 1\}| < 2^{eK(x)+1} \tag{2.7}$$

We have already proved that $d(x,y)$ satisfies the metric inequalities. But the function $d(x,y)$ itself being a ratio between two upper-semicomputable functions may itself not be

semi-computable [63]. So we only require that: $d(x, y)$ can be approximated with limited space and time and with reasonable accuracy, the approximating computation is feasible and attainable and also applicable in practice. With that in mind we move on to showing that $d(x, y)$ fulfils the density conditions.

Lemma 2. *The function $d(x, y)$ satisfies the density condition in (2.7).*

Proof. If $d(x, y) \leq e$, then:

$$K(x|y^*) + K(y|x^*) \leq 2\max\{K(x|y^*), K(y|x^*)\} \leq eK(x, y)$$
$$\max\{K(x|y^*), K(y|x^*)\} \leq e\frac{K(x, y)}{2}$$
$$\max\{K(x|y^*), K(y|x^*)\} \leq e\frac{K(x) + K(y)}{2}$$

The last inequality follows from the sub-additive property of K. Now assume that, $K(y) \leq K(x)$. We get:

$$\max\{K(x|y^*), K(y|x^*)\} \leq K(x|y^*) \leq e\frac{K(x) + K(x)}{2} \leq eK(x)$$
$$K(x|y^*) + K(y) - K(x) \leq eK(x) + K(y) - K(x)$$
$$K(y|x^*) \leq eK(x)$$

The last inequality follows from 2.1.
There are at most $\sum_{i=0}^{eK(x)} 2^i < 2^{eK(x)+1}$ binary programs of length $\leq eK(x)$. Therefore for fixed x there are $< 2^{eK(x)+1}$ objects y satisfying $K(y|x^*) \leq eK(x)$. Notice that the proof is symmetric when $K(x) < K(y)$. □

However, we are interested in a deeper density property of our chosen definition of $d(x, y)$: how many objects there are within a distance d of a given object x. Fewer objects would mean that the distance metric is adhering to the Kraft Inequality [40] and the underlying description method is assigning more unique encodings to the objects that it describes. To make our calculation easier we consider the non-normalized version of $d(x, y)$, $E(x, y) = d(x, y)K(x, y) = K(y|x^*) + K(x|y^*)$. Then if $E(x, y) = d$, it tells us how many objects y there are within d irreversible bit operations of a given object x [5].

Definition 10. For a binary string x of length n, a non-negative number d, assume the following definitions:

$$n(d, x) = |\{y : y \neq x, E(x, y) = d\}|$$

$$N(d,x) = \sum_{i=1}^{d} n(i,x)$$

$N(d,x)$ can be approximated by observing $K(y|x^*)$. An important theorem from [40] gives an expression for the maximal complexity of a string of length n can have: *For each* n*,* $\max\{K(x) : l(x) = n\} = n + K(n) + O(1)$. It's easy to see the maximum length $K(y|x^*)$ can have within a distance of d from x, is $d - K(d|x^*)$, an analogous version of the above theorem. It is necessary to subtract $K(d|x^*)$ from d because, $K(y|x^*)$ being self-delimited, needs $K(d|x^*)$ extra bits to delimit the program that produces y given x^* or otherwise $K(y|x^*)$ will have a larger length than d and go beyond the allowed distance. The following lemma 3 finalizes this property. A similar proof for an alternate definition of $E(x,y)$: $K(y|x) + K(x|y) + O(\log(K(x|y) + K(y|x)))$ has been provided in [5]. We provide a complete proof for our definition of $E(x,y)$ in Appendix A.

Lemma 3. *Let x be a binary string of length n. The number of binary strings y with $E(x,y) \leq d$ satisfies up to a constant additive error:*

$$\log N(d,x) = d - K(d|x^*) \tag{2.8}$$

Notice that, the above property does not necessarily impose any restriction on the complexities of x and y and is a general statement on the number of y's in the balls of size d around x. Had we defined $d(x,y)$ as $\frac{max\{K(x|y^*),K(y|x^*)\}}{max\{K(x),K(y)\}}$ and $E(x,y)$ as $\max\{K(x|y^*), K(y|x^*)\} \leq d$, the proof for $\log N(d,x) = d - K(d|x^*) + O(1)$ would be similar to the above. That is: the number of objects y in balls of size d for both definitions will roughly be the same. However, if we impose a complexity restriction on the strings y to be of the similar complexity as x, we see that the number of such strings y in balls of size d is much less for $E(x,y) = K(y|x^*) + K(x|y^*)$.

Since this project specifically focuses on low-entropy strings, we are interested in counting the number of strings y in balls of d around x, where $K(x)$ and $K(y)$ are both "small". Since we use context-free grammars as our description method and the smallest context free grammar has *size* (defined later) $\Omega(\log n)$ [15], we try to count the number of strings y in balls of size d around x when $K(x)$ and $K(y)$ both are $\approx \log n$.

Notice that, here too the maximum length that $K(y|x^*)$ can witness with the given constraints is limited. We fix $K(y)$ to be $\approx \log n$. About $K(x)$ bits of $\log n$ is shared information with x, so $K(y|x^*)$ is at least $\delta(n) = \log n - K(x)$. Note that $\delta(n)$ can be as small as $O(1)$ (x and y have about $K(x)$ bits of shared information). So there can be

strings y of larger $K(y|x^*) \approx \delta(n) + l$ which satisfy $K(y) \approx \log n$ and $E(x, y) \leq d$. The following lemma 4 proves that l can be at most $\frac{d-\delta(n)}{2}$, while $d \geq \log n - K(x)$, where the factor of $\frac{1}{2}$ originates from the fact that about $\frac{d-\delta(n)}{2}$ bits of d have to be allocated for $K(x|y^*)$, as $E(x, y) = K(y|x^*) + K(x|y^*) \leq d$.

Denote inequality to within an additive $O(\log)$ by $\overset{log}{<}$ and $\overset{log}{>}$. Denote both $\overset{log}{<}$ and $\overset{log}{>}$ by $\overset{log}{=}$.

Let:

$$n'(d, n, x) = |y : y \neq x, K(y) \overset{log}{=} \log n, E(x, y) = d|$$

$$N_{low}(d, n, x) = \sum_{i=1}^{d} n(i, n, x)$$

Lemma 4. *For each x of length n we have:*

$$\log N_{low}(d, n, x) \overset{log}{=} \frac{\log n + d - K(x)}{2}$$

while $\log n - K(x) \leq d$. *For* $\log n - K(x) > d$ *we have* $\log N_{low}(d, x, n) \overset{log}{=} d$

Proof. For this we use a similar proof technique from [5]. Let, $K(x) \overset{log}{=} \log n - \delta(n)$

($\overset{log}{>}$): Let $y^* = x^* z$, with $|z| = \delta(n)$, $|y^*| = \log n$, $K(x) \overset{log}{=} |x^*|$ and $K(y) \overset{log}{=} |y^*|$.
x^* can be found by dove-tailing all computations on programs of length less than $\log n$ and taking the first program that generates x.
We can retrieve z from y^* by providing at most $O(\log \log n)$ bits.
Since, $|z| = \delta(n)$, there are $2^{\delta(n)}$ different such y^*'s. For each such y^* we have, $K(x|y^*) \overset{log}{=} O(1)$ since x can be retrieved from y^* using x^*.
Now suppose, for a fixed i (value to be determined later) we replace the fixed first $\frac{i}{2}$ bits of y^* by an arbitrary $u \in \{0, 1\}^{\frac{i}{2}}$. Then the total number of such y^* increases to $2^{\delta(n)+\frac{i}{2}}$. These choices of i and y^* must also satisfy $E(x, y) \leq d$. Clearly:$K(x|y^*) \overset{log}{<} \frac{i}{2}$, since we can retrieve x by providing at most $\frac{i}{2}$ bits, and $K(y|x^*) \overset{log}{<} \delta(n) + \frac{i}{2}$. Therefore:

$$K(x|y^*) + K(y|x^*) \leq d \overset{log}{<} \frac{i}{2} + \delta(n) + \frac{i}{2}$$

$$\Rightarrow \text{The largest } i \text{ can be } \overset{log}{=} d - \delta(n) + O(1)$$

$$\Rightarrow K(y|x^*) \overset{log}{<} \delta(n) + \frac{d - \delta(n)}{2} \overset{log}{<} \frac{\log n + d - K(x)}{2}$$

This shows that the number $N_{low}(d, n, x)$ of y's such that $E(x, y) \leq d$ satisfies:

$$\log N_{low}(d, n, x) \overset{log}{>} \frac{\log n + d - K(x)}{2}$$

($\overset{log}{<}$): Assume on the contrary, for a fixed c (value to be determined later) there are at least $2^{\frac{d+\delta(n)}{2}+c}$ elements y of length n such that $E(x, y) \leq d$ holds. Then for some y:

$$K(y|x^*) \overset{log}{>} \frac{d + \delta(n)}{2} + c$$

By assumption:

$$K(x) \overset{log}{=} \log n - \delta(n)$$

$$K(y) \overset{log}{=} \log n$$

We know that from the symmetry of algorithmic information property up to an additive constant:

$$K(x) + K(y|x^*) = K(y) + K(x|y^*)$$

$$\Rightarrow \log n - \delta(n) + \frac{d + \delta(n)}{2} + c \overset{log}{<} \log n + K(x|y^*)$$

$$\Rightarrow K(x|y^*) \overset{log}{>} \frac{d - \delta(n)}{2} + c$$

However, then for a large enough $c = O(\log n)$, $K(y|x^*) + K(x|y^*) \leq d$ can no longer hold. Hence we reach a contradiction. So,

$$\log N_{low}(d, n, x) \overset{log}{<} \frac{\log n + d - K(x)}{2}$$

□

2.2 Approximating $K(x)$ with a Context Free Grammar of x

The Turing Machine model, used in the previous section, for representing the shortest description of a string x is, in a practical sense, too powerful to be exploited effectively.

Furthermore, it is incomputable. However, weakening the description model from Turing machines to Context-Free Grammars reduces the complexity of the problem from the realm of undecidability to mere intractability [16].

2.2.1 Preliminaries

2.2.1.1 Context Free Grammar

Definition 11. A *Context Free Grammar* is a 4-tuple $\{\Sigma, \Gamma, S, \Delta\}$ in which Σ is a finite alphabet containing all the terminals, Γ is a set containing all the non-terminals such that $\Sigma \cap \Gamma = \emptyset$. All the elements of $\Sigma \cup \Gamma$ are called *symbols*. $S \in \Gamma$ is a non-terminal reserved for representing the *start symbol*. Δ is a set of rules of the form $T \to \alpha$, where $T \in \Gamma$ is a non-terminal and $\alpha \in (\Sigma \cup \Gamma)^*$ is a string of *symbols* and referred to as the *definition of T* [15].
A *Context Free Grammar G* that generates exactly one finite-length string has the following properties [16]:

- There exists exactly one rule in Δ defining each non-terminal $\in \Gamma$.
- G is **acyclic**, that is there exists an ordering of the non-terminals in Γ such that each non-terminal precedes all non-terminals in its definition.

Definition 12. The *size* of a *grammar G*, generating a single finite-length string x, is the total number of symbols in all *definitions*:

$$|G(x)| = \sum_{T \to \alpha \in \Delta} |\alpha|$$

where $|\alpha|$ denotes the number of symbols in the string α.

Definition 13. The *expansion* of a string is obtained iteratively by replacing each non-terminal by its definition until only terminals remain [15]. The expansion of a string α is denoted by $\langle \alpha \rangle$.

Definition 14. Let A be an algorithm that, for any input string x in Σ^*, generates a grammar $G(x, A)$. Then the approximation ratio of A is a function of a given string length n, and is defined as $a(n) = \max_{x \in \Sigma^n} \frac{|G(x,A)|}{|G^*(x)|}$, where $G^*(x)$ is the shortest grammar for x [15].

Lemma 5. *The smallest Context-Free Grammar for a string of length n has size* $\Omega(\log n)$ *[15].*

2.2.1.2 NP-Hardness

The *smallest Context-Free Grammar* generating a given string x is hard to approximate within a small constant factor. In fact, the following theorem was proven in [15]:

Theorem. *There is no polynomial-time algorithm for the smallest Context-Free Grammar problem with approximation ratio less than $\frac{8569}{8568}$ unless $P = NP$.*

The above theorem can be proved by reducing the Smallest CFG to finding a Vertex Cover in a graph. For that matter, finding an algorithm that produces CFG with an approximation ratio of $o(\frac{\log n}{\log \log n})$ would require a progress on an apparently difficult algebraic problem in a well-studied area: *the general addition chain problem* [15, 16].

2.2.1.3 Global Context Free Grammars

We focus on a single special class of offline algorithms for generating context free grammars for our analysis, and we refer to the generated grammars as *Global CFGs*. Every such *Global CFG* is *Irreducible* with the following *nice* properties [30]:

- For $\alpha, \beta \in \Sigma \cup \Gamma$ and $\alpha \neq \beta$, the pair $\alpha\beta$ appears only once on the right sides of the grammar when they do not overlap.
- For $T_1, T_2 \in \Gamma$ and $T_1 \neq T_2$,$\langle T_1 \rangle \neq \langle T_2 \rangle$
- For every T, such that $T \in \Gamma \backslash S$, T appears *at least* twice in the definitions of the non-terminals in the grammar.

Any algorithm that generates *Global CFGs* with this property set, has an upper bound of $O((\frac{n}{\log n})^{\frac{2}{3}})$ on the approximation ratio [15] and captures the features of *hierarchically structured* sequences (comprising of patterns, their repetitions and the bigger patterns that consist of them) better than other online algorithms [15]: SEQUENTIAL [50], LZ78 [68].

2.2.2 Algorithms for Global Context Free Grammars

Broadly, all the *Global CFG* generating algorithms use the following procedure [15]:

Algorithm 1 Algorithm for Global CFG

Input: string σ

1: $G \leftarrow \emptyset$
2: $G.insertRule(S \rightarrow \sigma)$
3: **while** σ contains a maximal string γ and replacing it strictly reduces the size of grammar **do**
4: $\quad G.insertRule(T \rightarrow \gamma)$
5: $\quad$ Apply rule $T \leftarrow \gamma$ to all the right sides of all other rules: replace each occurrence of γ with T

A maximal string γ selected in line 3 has three properties:

- $|\gamma| \geq 2$
- There are at least two non-terminals $T_1, T_2 \in \Gamma$ such that T_1, T_2 have γ in their definitions without overlap.
- Since γ is a maximal string, no string $|\beta| > |\gamma|$ appears at least as many times as $|\gamma|$.

The entire process of finding a maximal string is repeated until replacing it with a non-terminal strictly yields a smaller grammar. If the γ occurs $Occ(\gamma)$ non-overlapping times in the entire grammar, then replacing it with a non-terminal T, will reduce the grammar-size by $(|\gamma| - 1) \times Occ(\gamma)$. Adding the rule $T \rightarrow \gamma$ will add $|\gamma| + 1$ to the grammar size. This means that the loop in line 3 in the above algorithm runs as long as $(|\gamma|-1)(Occ(\gamma)-1)-2$ is strictly positive.
Algorithms for global grammars only differ in the way they select the maximal string γ. We apply the algorithms to a reference string $\sigma = abcabcabcabcabb$ here.

2.2.2.1 Most-Frequent Algorithm

The Most-Frequent algorithm, at each step, chooses the maximal string γ that maximizes $Occ(\gamma)$ and has length at least two. It was proposed by Larsson and Moffat in [32] and they called it the *Re-Pair Algorithm*.
Initially the grammar $G(x)$ has only one rule $S \rightarrow abcabcabcabcabb$. The maximal string γ that has the highest occurrence is, in this case, ab and a new rule $T \rightarrow ab$ is introduced. The grammar becomes:

$$S \rightarrow TcTcTcTcTb$$

$$T \to ab$$

Now γ is Tc. Then a new rule $V \to Tc$ is added, yielding the final grammar:

$$S \to VVTb$$
$$T \to ab$$
$$V \to Tc$$

2.2.2.2 Longest-First Algorithm

Kieffer and Yang [30] proposed the Longest-First algorithm which finds the Longest maximal string which occurs at least twice and selects the one that occurrs first in the string. In our example: this algorithm will select *abcabc* as the maximal string and so the grammar will look like:

$$S \to TTabb$$

$$T \to abcabc$$

2.2.2.3 Greedy Algorithm

Another variation is the Greedy algorithm (Apostolico and Lonardi [3]), which selects maximal string γ that yields a grammar of the shortest size possible *at each step* by maximizing the amount $(|\gamma| - 1) \times (Occ(\gamma) - 1) - 2$. In the above example it will choose $T \to abc$ generating a grammar:

$$S \to TTTTaba$$

$$T \to abc$$

Carrascosa et al. [12] reported that until now no other polynomial time algorithm has been proven (theoretically nor empirically) to perform better than the Greedy algorithm. The only algorithm that comes closer to the Greedy, as in generating the smallest size grammar, is the Most Frequent algorithm [11]. Charikar et al. [15] reported the following approximation ratios for the algorithms mentioned above:

Algorithm	Approximation Ratio Upper Bound	Lower Bound
Most Frequent	$O((\frac{n}{\log n})^{\frac{2}{3}})$	$\Omega(\sqrt{\log n})$
Longest First	$O((\frac{n}{\log n})^{\frac{2}{3}})$	$\Omega(\log \log n)$
Greedy	$O((\frac{n}{\log n})^{\frac{2}{3}})$	$> \frac{5 \log 3}{3 \log 5}$

Table 2.1: Approximation Ratios of algorithms generating Global CFG

2.2.3 $|G(x)|$ as $K(x)$

Now, we adapt CFGs to our compression-based approach, as a universal description method. The description of a string x will be a context-free grammar which uniquely generates x with an agreed-upon mechanism or decoder ϕ to construct x from this description. Then the unconditional prefix Kolmogorov Complexity of x with respect to ϕ is defined as:

$$K_\phi(x) = min\{l(p) : \phi(p, \epsilon) = x, l(p) = |EncodedCFG(x)|\}$$

where p is an encoded description of an approximation of the smallest contest-free grammar of x. Note here that the value $K_\phi(x)$ is an upper bound on the true Kolmogorov complexity $K(x)$, restricting to descriptions in the form of short CFGs.
The conditional prefix Kolmogorov complexity is defined as:

$$K_\phi(x|y^*) = min\{l(p) : \phi(p, y^*) = x, |y^*| = |EncodedCFG(y)|\}$$

Here p denotes the encoded description of the CFG of string x after it has been compressed as much as possible with y^*.

2.2.3.1 ϕ as a Decoder

In order not to hide too much information in the decoder, we want it to be a simple function:

Algorithm 2 ϕ-General Decoder

Input: $p = EncodedCFG(x), y^* =$ empty string,ϵ or $EncodedCFG(y)$

1: $Dictionary \leftarrow []$
2: $S \leftarrow extract_Start_Rule(p)$
3: **while** $p \neq \emptyset$ **do**
4: $\quad extract_Rule(p, \quad T \leftarrow \alpha)$
5: $\quad Dictionary.insert_Key - Value_pair(T, \alpha)$
6: **while** $y^* \neq \emptyset$ **do**
7: $\quad extract_Rule(y^*, \quad T \leftarrow \alpha)$
8: $\quad Dictionary.insert_Key - Value_pair(T, \alpha)$
9: **while** S contains a non-terminal T **do**
10: $\quad \alpha \leftarrow find_Value(Dictionary, T)$
11: $\quad$ Replace T with string α
return S

The definition of the Kolmogorov complexity of a string x relies on the existence of a Universal Turing Machine U such that $K(x) = \min\{l(p) : U(p) = x\}$. Since, we have weakened the restriction on the description method from being Universal to being a Context Free Grammar, it suffices that we have a function or decoder ϕ that reconstructs x given the CFG description, p.
The decoder ϕ is an r-ary decoder ϕ_r, where r depends on the alphabet of the Context-free grammar encoded in p. It is a partial (partial because ϕ only accepts input of a certain form) recursive function which outputs x when given as input, a CFG description p and an auxiliary or conditional input y^*.
The input p encodes a context-free grammar (in our case, the smallest of Most Frequent, Greedy and Longest-First) that describes x in a certain way. To be precise, consider the Greedy grammar $G(x) = \{\Sigma, \Gamma, S, \Delta\}$ of the string $x = abcabcabcabcabb$:

$$S \rightarrow TTTTaba$$

$$T \rightarrow abc$$

Then p is imagined in the following way, where we use the symbols $|$ as a separator between rules and $-$ as a separator between Non-Terminals and their definitions:

$$S - UUaba|U - TT|T - abc$$

Hence we approximate the $K(x)$ (ignoring the separators) as:

$$K(x) \approx K_\phi(x) = \log|\Gamma \cup \Sigma| \times (|\Gamma| + \sum_{T\rightarrow\alpha\in\Delta} |\alpha|) \tag{2.9}$$

When denoting p as an encoded Context Free Grammar of x, we do not specify which of our three CFG generating algorithms (Most Frequent, Greedy or Longest-First) we use to generate the grammar. The standard way of encoding this information would be: Suppose, there are r methods $f_1, f_2, \ldots\ldots, f_r$ to generate Context Free Grammars of x. Then we have to reserve $\log r$ bits of p to specify which method was used for the particular CFG that we are encoding.
Our approximation of $K(x|y^*)$ is constructed as follows: with y^* encoding an efficient Context Free Grammar of y, we try to compress x as much as possible with the rules in y^*, after which we try to compute the grammar $G(x')$ of the compressed x', over the alphabet that now includes the non-terminals in y^* that were used in x'. Hence, $K(x|y^*)$ will be approximated as:

$$K(x|y^*) \approx K_\phi(x|y^*) = \log|\Gamma' \cup \Sigma'| \times (|\Gamma'| + \sum_{T \to \alpha \in \Delta'} |\alpha|) \tag{2.10}$$

Our modified Grammar generation process can be described with the following pseudocode:

Algorithm 3 Algorithm for Global Grammars

Input: string x, $y^* = \emptyset | CFG_y$

1: $x \leftarrow CompressWithGrammar(x, CFG_y)$
2: $G \leftarrow \emptyset$
3: $G.insertRule(S \to x)$
4: **while** x contains a maximal string γ **do**
5: Replace each occurrence of γ with T
6: $G.insertRule(T \to \gamma)$

2.3 Constrained Best-Fit Model Selection

2.3.1 Two-Part Code

It is an elegant fact that the length of the shortest effective description of x can be expressed in terms of the length a two-part code, the length $K(T)$ of the first part describing an appropriate Turing machine T and the length $K(x|T)$ of the second part describing the left-out irregularities or random aspects of x after T *squeezes out* the regularities of x [40]:

$$K(x) = min\{K(T) + K(x|T) : T \in \{T_1, T_2, \ldots\ldots\}\} + O(1) \tag{2.11}$$

The best model T encapsulates the amount of useful or compressible information in x, while minimizing the total description length $K(T) + K(x|T)$. However, it remains to decide which T satisfies these requirements. Following the Occam's Razor principle, we opt for a T that is not too complex [65]. So, to analyze the process of finding such T, we first put a complexity limit on it such that $K(T) \leq \alpha$. Under these constraints, there are three approaches, specifically three "structure" functions [64] that we can use to estimate the right model T that best describes x. To make our analysis easier we work with *low-complexity* Finite Set Model, $S = \{x_1, x_2, \ldots\ldots, x_n\}$ such that $K(S) \leq \alpha$. To create an analogue between the Turing Machine model and the Finite Set model, we can assume that T enumerates S and $T(i) = x_i$ where $i \leq |S|$ and $x_i \in S$. When $i > |S|$, $T(i)$ is undefined.

2.3.2 "Best-fit" Structure Function

2.3.2.1 Randomness Deficiency

The complexity of the finite set S denoted by $K(S)$, is the length of the shortest binary program S^* from which the reference Universal Turing Machine U computes a listing of the elements of S and then halts. The conditional complexity $K(x|S)$ is defined as the length of the shortest program that computes x from input S. For every finite set S containing x we have [65]:

$$K(x|S) \leq \log |S| + O(1)$$

If the elements of S are lexicographically ordered and $x \in S$, then at most $\lceil \log |S| \rceil$ bits are required to specify x's position in S. This is called the data-to-model code length [65]. However, consider the ordering of S's elements when x is the first element of S; then $K(x|S)$ is much less than $\lceil \log |S| \rceil$.

Definition 15. The amount by which $K(x|S)$ falls short of $\lceil \log |S| \rceil$ is called *randomness deficiency* of x with respect to S [64]:

$$\delta(x|S) = \log |S| - K(x|S)$$

for $x \in S$ and ∞ otherwise

The smaller $\delta(x|S)$ is, the more x can be considered as a typical member of S. This means that the model that incurs minimum randomness deficiency with respect to a threshold α is a "best-fitting" model for x in that model class of contemplated sets:

$$\beta_x(\alpha) = \min_S \{\delta(x|S) : x \in S, K(S) \leq \alpha\} \tag{2.12}$$

$\beta_x(\alpha)$ can be viewed as a constrained best-fit estimator [64].

2.3.3 Kolmogorov's Original Structure Function

The original Kolmogorov *structure function* for x is defined as [64]:

$$h_x(\alpha) = \min_S \{\log |S| : x \in S, K(S) \leq \alpha\} \tag{2.13}$$

The model, S, witnessing $h_x(\alpha)$ for a certain α, is called an *optimal set* for x and $K(S)$ is called a sufficient statistic for x [64], and we have:

$$K(x) + O(1) = K(S) + \log |S|$$

$$K(x|S) = \log |S| + O(1)$$

2.3.4 Minimum Description Length Structure Function

The length of the two-part code for x comprising of the model cost $K(S)$ and the length of the index of x in S is denoted by $\Lambda(S) = K(S) + \log |S| \geq K(x) + O(1)$. The minimal $\Lambda(S)$ such that $K(S) \leq \alpha$ is given by the Minimum Description Length (MDL) function [66]:

$$\lambda_x(\alpha) = \min_S \{\Lambda(S) : x \in S, K(S) \leq \alpha\} \tag{2.14}$$

A principal result of [64] shows that: *every set S that witnesses $h_x(\alpha)$(2.13) or $\lambda_x(\alpha)$(2.14) also witnesses $\beta_x(\alpha)$(2.12).* This result is specially useful because the best-fit structure function is neither upper nor lower semicomputable [64]. However, it is easy to see that both $h_x(\alpha)$ and $\lambda_x(\alpha)$ are upper-semicomputable [64]. So, monotonically approximating $h_x(\alpha)$ or $\lambda_x(\alpha)$ to some significant precision will implicitly approximate the best-fitting model. We now use this idea to extend this notion of model selection to *low-complexity* Context Free Grammar model, G instead of Finite Set.

2.3.5 Model Class of Context Free Grammar

In the model-class of CFG, the cost of two-part code for describing x consists of the model cost $K(G)$ and the conditional complexity $K(x|G)$:

$$K(x) = \min\{K(G) + K(x|G) : G \in \{G_1, G_2, \ldots\ldots\}\}$$

Here G is a 4-tuple $\{\Sigma, \Gamma, S, \Delta\}$ Context Free Grammar. We require that it exhibits all the properties of Global CFG and hence is *irreducible* (section 2.2.1.3). Thus we treat G

as its own shortest description and $K(G)$ does not exceed the size of the encoded binary representation of G and is approximated similarly as (2.9):

$$K(G) \approx \log |\Gamma \cup \Sigma| \times (|\Gamma| + \sum_{T \to \alpha \in \Delta} |\alpha|)$$

$K(x|G)$ denotes the length of the encoded description of an efficient CFG of x after it has been compressed as much as possible by G and is approximated similarly as (2.10).
But as we want G to be a model that is an analogue of the Finite Set or Turing Machine model, we want G to describe a set of strings rather than one string. So, G should be rich enough while still subject to the constraint $K(G) \leq \alpha$. Such G can be constructed in the following way.

2.3.5.1 Constructing G

Consider a set of finite binary strings $\{x_1, x_2, \ldots\ldots, x_n\}$ for which we want to construct a CFG model G under the constraint $K(G) \leq \alpha$. Extending a result in [15] that proves the existence of a CFG with size $|G_{x_1}| + |G_{x_2}| + 2$ that generates the string $x_1 x_2$, we see that there exists a CFG of size $n + \sum_{i=1}^{n} |G_{x_i}|$ where $G_{x_1} = \{\Sigma_1, \Gamma_1, S_1, \Delta_1\}, G_{x_2} = \{\Sigma_2, \Gamma_2, S_2, \Delta_2\}, \ldots\ldots, G_{x_n} = \{\Sigma_n, \Gamma_n, S_n, \Delta_n\}$ that produces the concatenation of the strings in the set above. We could model G to be $\{\cup_{i=1}^{n} \Sigma_i, \cup_{i=1}^{n} \Gamma_i, S = S_1 S_2 \ldots S_n, \cup_{i=1}^{n} \Delta_i\}$, then G will have a size of $\sum_{i=1}^{n} |G_{x_i}|$. In this setup, $K(G)$ can be quite large according to our approximation above and the provided α can be much less than $K(G)$. Then G needs to be much smaller while still capturing the essence of the strings.
For this, we allow $G = \{\Sigma, \Gamma, S, \Delta\}$ to only contain those rules $T \to \gamma$ such that the pattern γ is present in at least $\beta\%$ of the set of the strings. As well as adjusting Δ to meet this restriction, we also adjust Σ and Γ to only those terminals and non-terminals that are present in the new rule set Δ. β is inversely related to α. For a fixed α, we can find the optimal value of β by starting with $G = \{\cup_{i=1}^{n} \Sigma_i, \cup_{i=1}^{n} \Gamma_i, S = S_1 S_2 \ldots S_n, \cup_{i=1}^{n} \Delta_i\}$, iterating β's value from t =1 to 100, trimming G as above and approximating $K^t(G)$ and taking the first t such that $K^{t+1}(G) > \alpha$.

2.3.5.2 Constrained Model Selection for Context Free Grammars

Given a class of CFG models $G_1, G_2, \ldots\ldots, G_n$ where $K(G_i) \leq \alpha$ for $i = 1, \ldots\ldots, n$ and a string x, we now want to select a model $G \in \{G_1, G_2, \ldots\ldots, G_n\}$ that is "best-fitting" for x. As discussed before, we attempt to do this by trying to approximate Kolmogorov's Original

Structure Function, $h_x(\alpha)$. We slightly modify the definition of $h_x(\alpha) = \min_S\{\log|S| : x \in S, K(S) \leq \alpha\}$ that is Finite Set model specific to fit our Context Free Grammar model. We propose the following definition:

$$h_x(\alpha) = \min_G\{K(x|G) : K(G) \leq \alpha\} \tag{2.15}$$

We model the code length within the new model by $K(x|G)$, omitting the requirement analogous to $x \in S$ that was present in the original definition of the structure function, which mandated the inclusion of the object x in the model S. We could essentially use the notion of generalized CFG which produces a language of strings and include the requirement $x \in L(G)$ in our definition of $h_x(\alpha)$ where $L(G)$ denotes the set of strings that G produces. It would also allow us to replace $K(x|G)$ with $\log|L(G)|$, in which case our formulation of $h_x(\alpha) = \min_G\{\log|L(G)| : x \in L(G), K(G) \leq \alpha\}$ becomes a direct analogue of the finite set model class. Such structure function may entail a better approximation of the "best-fitting" model for x. However, generalized CFG is beyond the scope of this thesis as we are interested in the conditional compression length $K(x|G)$ of x by G when G acts as a dictionary of important regularities present in a set of strings. We are also interested in the case when x is *not* a part of the above set. In this settings, the model G that witnesses the above $h_x(\alpha)$ (2.15), best describes the regular or meaningful information in x with $K(G) \leq \alpha$, and thus is the "best-fitting" model in our constrained model class.

Chapter 3

Related Work

3.1 Algorithmic Information Distance

The theoretical foundation of using Kolmogorov Complexity in measuring Algorithmic Information Distance between strings was predominately developed by Paul Vitányi, Ming Li, Péter Gács and Charles H. Bennett et al. [35, 40, 5, 28]. To apply such distance measures on real-world sequences, they introduced Normalized Information Distances (NID) $d_1(x,y) = \frac{\max\{K(x|y),K(y|x)\}}{max\{K(x),K(y)\}}$ and $d_2(x,y) = \frac{K(x|y)+K(y|x)}{K(x,y)}$ and used various (standard and specialized) compressors to approximate $K(x)$. As a result, the NIDs' are also called Normalized Compression Distance (NCD) [37, 40].

3.1.1 Roots in Bioinformatics

Li et al. [37, 36] created evolutionary phylogenetic tree of mammals by estimating the NID $d_2(x,y)$ between each pair of mitochondrial DNA x and y, using *GenCompress* (a specialized compression algorithm to compress DNA sequences) to heuristically approximate $K(x|y)$, $K(x)$ and $K(x,y)$. Chen et al. [17] also did a similar experiment with *GenCompress* to compress rRNA sequences of several bacteria and virus species and used an approximation of $\frac{K(x)-K(x|y)}{K(xy)}$ to measure the relatedness between each pair of sequences x and y to create an evolutionary tree. Cilibrasi and Vitányi [18] used general-purpose compressors like gzip, bzip2 to compress x and used the compressed size $C(x)$ to approximate $K(x)$ and clustered mtDNA sequences using a quartet method [8]. Since its not clear how to compute the conditional $K(x|y)$ with a standard compressor, $K(x|y)$ was approximated with $K(x|y) \approx K(xy) - K(y)$.

3.1.2 Application in other fields

Application of Normalized Compression Distance in other fields usually involves approximating $K(x)$ with the length of the compressed x by standard compression software like gzip, bzip2 and $K(x|y)$ with $K(xy) - K(y)$). Cilibrasi et al. [19, 20] used the quartet method to cluster different genres of music (rock, jazz and classical) and classical music from different composers (such as Chopin, Bach, Beethoven). The authors reported satisfactory hierarchical clustering success using NID and the quartet method when objects belonged to different genres, that is objects were highly dissimilar. However, the method seemed to underperform when objects belonged to the same genre but different composers. For example: when clustering classical music belonging to Bach, Chopin and Debussy, it failed to cluster Chopin's music by putting them closer in the quartet tree. Moreover, the experimental results in general, were decidedly worse when the sample size was large.
Cilibrasi, Vitányi, Li and others clustered Russian literature in different author groups [18], 52 different linguistic versions of the Universal Declaration of Human Rights in various language clusters [18, 37]. Li and Sleep [38] classified 771 melodic contours (pitch sequences without timing details) into 4 groups: Beethoven, Haydn, Chinese music and Jazz using k Nearest Neighbour classifier with information distances calculated from both $d_1(x, y)$ and $d_2(x, y)$. For their study they used LZ78 to compress the melodic contour files and the compressed file's size to approximate $K(x)$. They showed $d_2(x, y)$ (a version of sum distance that we use in this thesis) outperforms $d_1(x, y)$ with $k = 1$.
However, using standard compression software to approximate $K(x)$ which are specifically designed to *compress* rather than to extract *meaningful information* from the data, has its immediate drawback: it assigns almost similar distance between two objects that are substantially dissimilar from a human perspective (for example compositions of Bach and the Beatles) and two objects that are comparatively less dissimilar (for example compositions of Bach and Chopin). It also does not normalize the NID very well, generating distance values in the range $[0.8, 1.5]$ [18]. Clustering methods like k-Nearest Neighbor and Quartet Tree as used above, are sensitive to small differences in the distances, so they seem to work reasonably well when the sample size is small. But the results start to quickly degrade as we increase the sample size (more clustering errors [19]) or just increment k from 1 to a number larger than 1 in the k nearest neighbour method [41].

3.2 Approaches to Music Representation and Compression

The existing approaches we are about to present, all deal with lossless compression/ descriptions of *symbolic music* in one form or another. To our knowledge: elaborate research on *lossy* compression of symbolic music and its use in similarity estimation is yet to be conducted.

3.2.1 Lossless Data Compressors

Ens and Pasquier [25] converted music (in MIDI format) files into a sequence of integers indicating pitch onset, offset and duration and compressed them with real-world lossless compressors. They approximated $K(x)$ with the compressed file's size and computed a distance matrix using $d_{approx}(x,y) = \frac{K(xy) - \min\{K(x),K(y)\}}{\max\{K(x),K(y)\}}$ between each pair of objects x and y. The matrix was then used with statistical significance testing to calculate the probability of two separate corpora of music (in MIDI format) being "similar" or "different" in their "style". However, in our trial [1] of this system: we found that this is not a robust method as it was predicting that two corpora are the same even when they were from different composers or genre. It only succeeded in distinguishing two corpora if the file types are vastly different: for example the system could differentiate plain text files from music files. Takamoto et al. [62] represented a music score with a sequence of 88-dimensional bit vector obtained every semiquaver. Each entry is either 1 or 0: 1 for key-on and 0 for key-off events. The bit-vectors were then converted into strings of 0's and 1's, compressed with real-world compressor to approximate $K(x)$ and a compression based dissimilarity measure $\frac{K(xy)}{K(x)+K(y)}$ was used with K-nearest neighbour to estimate the composer of each piece of music. They showed that BZIP2 outperforms the other compressors but the overall result is not satisfactory: only 41 out of 75 musical pieces were assigned to the correct composers. Koopmans et al. [31] took a similar approach to music representation and compression as in [25] and hierarchically clustered Domenico Scarlatti's 555 piano sonatas.

3.2.2 Geometric Compressors

Meredith [46] represented music with a set of 2-dimensional points $\langle t, p \rangle$ where t is the onset time in tatums and p is the *morphetic pitch* [48] of a note or a sequence of tied

[1] We obtained the programs the authors used in their experiments from [24, 26]

notes in a score. The compression works by selecting a pattern (a set of points) that is maximally repeated, the vectors that translate it to the repeated positions, removing all such occurrences from the dataset and repeating the process until no such maximal pattern exists [44]:

Algorithm 4 Cosiatec Pseudocode

Input: string σ

1: $Compressed \leftarrow []$
2: **while** a pattern p exist such that it is most recurring **do**
3: $\quad I_p = \{(x, y) : p \text{ occurs at } (x, y)\}$
4: $\quad Dictionary.insertEntry(p, I_p)$
5: $\quad$ Remove all the occurrences of p
return $Compressed$

Visually the COSIATEC algorithm [44] on a Chopin's composition (MIDI obtained from https://www.classicalarchives.com) looks like the following[2]:

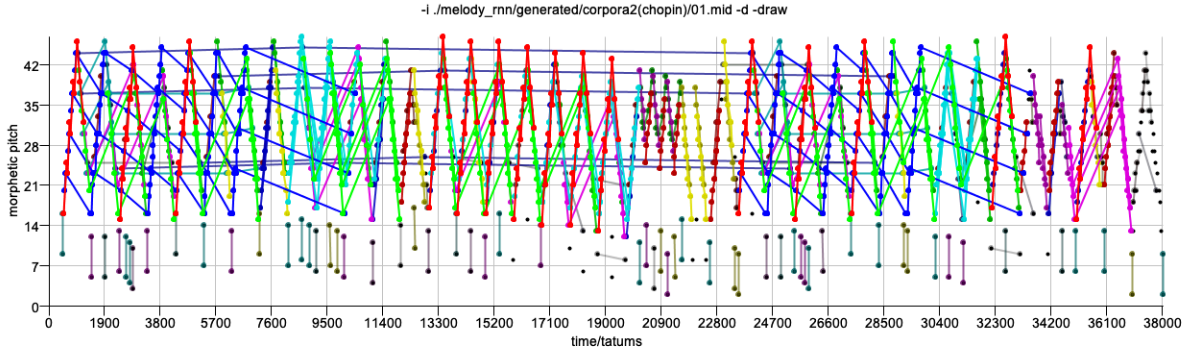


Figure 3.1: Visualization of COSIATEC on Chopin's Op. 28 Prelude No.1

The similarly colored polygonal chains depict patterns that are repeated and translatable by a vector. Thus an encoded version $E(x)$ of x, represented with multi-dimensional data points, in this system would be a sequence of $\langle pattern(P), translating\ vector\ list(V)\rangle$. Meredith [45] approximated $K(x)$ as follows [47]:

$$K(x) = \sum_{\langle P,V\rangle \in E(x)} (|P| + |V|) \tag{3.1}$$

[2]The COSIATEC software can be obtained from [43]

In [41] Louboutin and Meredith compared general-purpose compressors: LZ78, LZ77 and BZIP2 with Meredith's COSIATEC in their classification success rate of Dutch folk song melodies[3] into 26 tune families. For COSIATEC they used the two-dimensional geometric representation of music and approximated $K(x)$ as (3.1). For the general-purpose compressors they represented the melodies with a sequence of (onset interval,pitch) and approximated $K(x)$ with the compressed file size. They used the NID and k nearest neighbour method to classify a tune and showed that COSIATEC (84% classification success rate) outperformed general-purpose compressors (12.5% classification success rate) in melody classification.

They also reported a "weak, insignificant, negative correlation" between compression ratio and classification success rate. But this is counter-intuitive as a higher compression ratio should give a better approximation of $K(x)$ and in turn $d_{approx}(x,y) = \frac{K(xy)-min\{K(x),K(y)\}}{max\{K(x),K(y)\}}$ and should theoretically allude to a better classification rate. We think such low-classification success rate of normal world compressors can be attributed to the fact that they do not satisfy the density property of the Normalized Information Distance as well as specialized compressors like COSIATEC.

To show this we selected a reference object (Chopin's Op.28 Prelude No.1) x and used Google Magenta's Long Short Term Memory based Recurrent Neural Network [29], which had been trained on 17,000 western Classical music pieces, to generate monophonic melodies that were similar in size as x (the size of the music is determined by the number of sixteenth notes in the melody). These were then grouped into various corpora where each corpus contained more objects than the last one, as we want to see how the density property changes with increasing number of objects. We then converted the music into two dimensional points to be used with COSIATEC and also into a sequence of MIDI pitches using python's MIDI library to be used with LZ78. We approximated $K(x)$ for both compressors (as done in [41]) and computed $d_{approx}(x, y)$ with the approximations where x is fixed (Chopin's Op. 28 Prelude No. 1) and y is an object in a corpus. Then we calculated the following for each corpus:

$$|\{y : d_{approx}(x, y) \leq e = 1, C(y) \leq C(x)\}|$$

In other words, how many objects y is within a normalized distance 1 from the reference object x. We chose 1 as both the compressors when used with $d_{approx}(x, y)$ outputs distance values within the range $[0.8, 1.5]$.

[3] The Dutch folk song melodies can be obtained from http://www.liederenbank.nl.

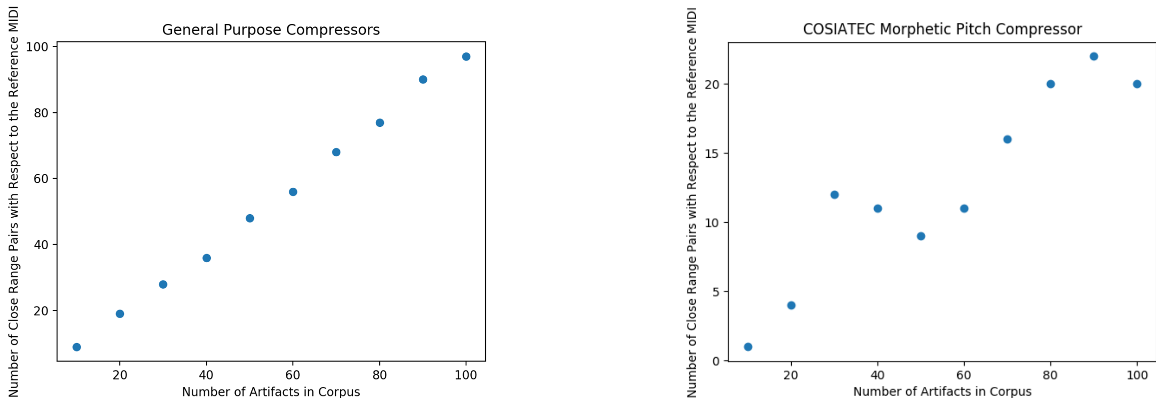


Figure 3.2: Density Properties of General Purpose and Specialized Compressors

As can be seen from the above plots, when approximating $K(x)$ with a general-purpose compressor, the number of y's that are e distance away from the reference x, is much higher (in fact all of the objects in the corpus are within the e distance in the above example) than when $K(x)$ is approximated with a specialized compressor. This is why the k nearest neighbour fails to identify the neighbours that are similar to x, as there are dissimilar neighbours within the same range as well.

3.2.3 Grammar Based Compressors

Sidorov et al. [57] demonstrated the use of Context Free Grammars (CFG) in compressing the melodic information of music that can be decomposed into multiple monophonic voices. At each step in constructing the grammar, they chose patterns that has *maximal repeats* (these patterns can be overlapping). Carrascosa et al. [11] showed that choosing such maximally repeating but possibly overlapping patterns as the constituents of CFG of genomic sequences and whole genomes can result in a smaller grammar than before [10, 12]. The general idea of using CFG as a compressor or a description of an object x and approximating $K(x)$ with the CFG's "size" was explored by Charikar et al. [15, 16]. The size of the CFG is defined as the total number of symbols of in the definitions of the rule [33, 15, 16]. Recall that for the grammar $G = \{\Sigma, \Gamma, S, \Delta\}$, its size would be defined as:

$$|G| = \sum_{T \to \alpha \in \Delta} |\alpha|$$

But we think that this size alone is not a good approximation (under-estimation) of the Kolmogorov Complexity of an object x. In this thesis we modify and extend this idea to take account of the alphabet needed to specify the CFG to better approximate $K(x)$. In general, to our knowledge this thesis is the first extensive exploration of using Context Free Grammars as a description method for Kolmogorov Complexity.

Chapter 4

Experiments and Observations

4.1 Music As Low-Entropy Strings

Before proceeding, it is useful to discuss the definitions of “style” and “genre” in the context of music. Although there are no universally accepted definitions, Franco Fabbri [27, 42] has usefully defined genre as “*a kind of music, as it is acknowledged by a community for any reason or purpose or criteria, i.e., a set of musical events whose course is governed by rules (of any kind) accepted by a community*” and style as “*a recurring arrangement of features in musical events which is typical of an individual (composer, performer), a group of musicians, a genre, a place, a period of time*”.
Genre can thus be broader and more nebulous than style from a content-based perspective, and may be more strongly characterized by cultural features [42].
In this project, we focus on music belonging to three such genres: classical, rock and jazz, use context free grammars to approximate their Kolmogorov complexities, build distance matrices from such complexity-based similarity measurement and observe whether such measurement can capture style in music genre or composition. For a complete list of the songs used in our experiments of these three genres the reader is referred to Appendix C.

4.1.1 Representing Symbolic Music as Strings

In this project we exclusively focus on representing music as a sequence of MIDI (Musical Instrument Digital Interface) pitch numbers, as the MIDI formats are easily obtainable descriptors of the symbolic music data. MIDI formats do not contain any vocal or lyrical

information about the song. However, for the rock songs used in this project, we were able to get the vocal melodic information from De Clercq and Temperley's manual transcriptions of these songs [23, 21]. In general, all the music pieces in our project were programmatically transcribed by converting their MIDI files into a list of MIDI pitch numbers using the following two methods. For the sources from which we obtained these MIDI files, the reader is referred to Appendix C.

4.1.1.1 Manual Transcription of Vocal Melody of Rock Songs

Trevor De Clercq and David Temperley [23, 21] created a corpus of rock music based on Rolling Stone magazine's list of "500 Greatest Songs of All Time"(RS 500) [54]. This list is somewhat skewed to having more songs of earlier decades, so DeClercq and Temperley [21] took top 20 songs in the RS 500 list from the decades 1950s, 1960s, 1970s and 1990s and 19 songs from the 1980s to create a 99-song list. The list was then expanded to include the next unique highest-ranked 101 songs from the RS 500 list creating the "RS 200" corpus which has a total of 200 songs. In this paper we are interested in the authors' melodic transcription of each song in RS 200.

The melodic transcription of a song is a sequence of encoded vocal melodic notes representing the relationship between the note and the local key center (scale degree). Since only the onset of the note is taken into consideration such transcription does not contain any information about the length of the note being played. However, since scale degrees vary from song to song, in order to have a consistent representation of vocal melody we used De Clerq and Temperley's program "process_mel5.pl" [23] to convert the "melodic transcription" to a list of MIDI pitches. Like the original transcription this list only tracks the onset of pitches and dispenses with any information like how long the pitch is played.

For example: the first few notes from “Hey Jude” by the Beatles looks like the following:

60	57	57	60	62	55
Hey	*Jude*	*don't*	*make*	*it*	*bad.*

55	57	58	65	65	64	60	62
Take	*a*	*sad*	*song*	*and*	*make*	*it*	*better.*

4.1.1.2 Automatic Transcription

To convert MIDI files automatically into a sequence of MIDI pitches, we used a program “mftext” (written by Tim Thompson and modified by Sleator and Temperley [61] and then updated by the author of this thesis) that Sleator and Temperley used for representing music to use with their MELISMA program. The program takes a MIDI file as an input and outputs a “notelist”. Each entry of this list contains the note-on and note-off time along with the MIDI pitch. If there are multiple channels in the MIDI file this program assembles the notes in the channels chronologically. In other words, the program is somewhat capable of turning a polyphonic music into a monophonic one. However, to our knowledge it does not keep track of the key of the song. Since are we only interested in a sequence of MIDI pitches, we only extracted the pitch information from this list. It is worth mentioning that, Ens and Pasquier [25] have also used similar representation, although they did not use Sleator and Temperley's program but a Python MIDI library to accomplish a similar task. They also included the MIDI onset, offset and time delta (duration of the pitch) information in their representation [25].
To demonstrate this transcription, consider the same song as above, “Hey Jude” by the Beatles. The first few notes of this song are:

84 81 69 65 60 41 48 60 69 65 53 60 69 65 41 81 48 84 69 60 65 86 53 79 64 60 36 43

Visually this would look like following:

Figure 4.1: The first few bars of "Hey Jude". This sheet music was created with Noteflight [7] based on the "Hey Jude" midi that was used to create the above notelist.

Notice that this representation is also not concerned about the length or duration of the notes/ chords like the vocal-melodic transcription of rock songs mentioned previously, as the longer notes are not repeated in the sequence.
Since the vocal-melodic information is only available for the rock songs, in our experiments, unless otherwise specified, all the music are transcribed automatically to convert them into a sequence of numbers (ranging from 0 to 127) delimited by a space, to ensure consistency. Automatic transcription by nature captures more information (it, in a way, encodes information about the order in which instruments are played and polyphony in a music piece) than the monophonic transcription of the vocal melody in songs. Thus to differentiate this from the previous vocal melodic transcription, we call the sequence generated by the automatic transcription for a music string, its *instrumental transcription*. Note that, for the genre of rock songs we use both the vocal melodic and instrumental transcriptions in our experiments. For the other two genres we use only the instrumental transcriptions.
Since both the transcriptions are non-random sequences of integers representing a hierarchical structure of the respective music, the process of generating Context Free Grammars for them is the same. All the codes for this project are available at https://github.com/Tiasa/StyleRecognitioninMusic.git

4.2 Approximating the Smallest CFG

4.2.1 An Overview

As mentioned in Chapter 3 we use three global algorithms Greedy, Most Frequent and Longest First to generate an approximation of the smallest CFG for a music string x. However, for the smallest grammar, we return the grammar with the smallest approximated Kolmogorov Complexity among the above three.

The three algorithms only differ in how they define the objective function. "Longest First" corresponds to $f(\alpha, \Delta) = |\alpha|$. Choosing the "Most Frequent" repeats, corresponds to $f(\alpha, \Delta) = occ_{\Delta}(\alpha)$ where $occ_{\Delta}(\alpha)$ is the count of non-overlapping occurrences of α in the right-hand sides of Δ. The objective function for "Greedy" algorithm is $f(\alpha, \Delta) = (|\alpha| - 1) \times (occ_{\Delta}(\alpha) - 1) - 2$, which is the effect on the total grammar size. However, the implementations of these algorithms are not immediately intuitive and different approaches are available. Next we describe our approach to this problem.

4.2.2 Implementation of the Global Algorithms

Although there are multiple algorithms to find the constituents or repeating patterns in a string, we use suffix tree structures for this purpose. Normal suffix trees are efficient for storing one string and give insights about its structure. However, when searching for the smallest grammar, one has to consider finding a recurring word α in all the strings on the right sides of the rules in Δ. One way of achieving this is to build a generalized suffix tree or Patricia tree that stores all the right-hand strings that are currently in the grammar. For an in-depth description of how we build suffix and Patricia trees and use them for pattern searching in strings see Appendix B.

Algorithm 5 Patricia Tree

```
function BUILDTREE(g: Grammar)
    concatenated ← ∅
    for each rule in g do
        concatenated ← concatenated + rule.RHS + uniqueDelimeter(rule.LHS)
    buildSuffixTree(concatenated)
```

Notice that when searching for pattern occurrences, we need to know which rule has this pattern on its right hand side, so that it can be replaced with a non-terminal. For this, we encode the left hand side of the rule, in a unique way to be used as a delimiter for the Patricia tree. So, the suffixes in the tree can report which rule the pattern belongs to. We continue this pattern search and replacement scheme until the size of the resultant grammar is strictly smaller than before. However, since after each replacement, the Patricia tree needs to be updated substantially (for example: deletion of the replaced substring, insertion of a new non-terminal) along with new pattern statistics: occurrences, length etc, we build a tree at each step from the updated CFG. A more efficient method of achieving this is discussed in Chapter 5.

Algorithm 6 CFGGlobalAlgorithm

```
 1: function COMPRESS(x, algorithm: "Greedy","Most Frequent" and "Longest First")
 2:     Grammar : CFG_x ← StartRule → x
 3:     moreReductionPossible ← True
 4:     NTnumber ← 1
 5:     while moreReductionPossible do
 6:         patriciaTree ← buildPatriciaTree(CFG_x)
 7:         pattern ← patriciaTree.getPattern(algorithm)
 8:         totalReductionInGrammarSize ← pattern.length × pattern.frequency
                −pattern.length − pattern.frequency
 9:         if totalReductionInGrammarSize ≤ 0 then
10:             moreReductionPossible ← False
11:         else
12:             CFG_x.replace(pattern, T_NTnumber)
13:             CFG_x.addRule(T_NTnumber → pattern)
14:             NTnumber ← NTnumber + 1
15:     return CFG_x
```

4.2.2.1 Counting Non-overlapping Occurrences

Notice that occurrences of a pattern reported by the suffix tree (see Appendix C) can overlap. In the construction of a CFG, since these will be replaced by a non-terminal, when occurrences overlap, its necessary to specify which non-overlapping occurrences have to be replaced [12]. The problem of computing the non-overlapping occurrences of a pattern in a string is known as the *String Statistics Problem*. One solution to this is to choose the occurrences in the (sorted by their indices) in a greedy left to right way. That is, all

occurrences overlapping with the first selected/left-most occurrence are not considered. This ensures that a maximal number of occurrences will be replaced.
For this we augment the suffix tree by counting the occurrences at each internal node (Appendix B), removing overlaps from them and then storing the information at each node. It allows to find the non-overlapping occurrences of α in $|\alpha|$ time.

Algorithm 7 Removing Overlap

Input: $prefixLength$: the path-label to this node
 ▷ Note that *node.occurrences* are already sorted in ascending order of their indices.
 $node.nonOverlapOcc \leftarrow []$
 $leftMostOcc$: Leftmost Occurrences Added
 for Index i: node.occurrences **do**
  if $prevAdded + prefixLength <= i$ **then** ▷ If the current occurrence does not collide with the previous one, add this occurrence
   $node.nonOverlapOcc.add(i)$
   $leftMostOcc \leftarrow i$

4.2.2.2 Greedy Algorithm

The Greedy Algorithm tries to reduce the grammar size as much as possible by selecting a pattern P such that $(P.length - 1)(P.occurrences - 1) - 2$ is maximized. To find such pattern in the suffix tree, we traverse the entire tree, as it is not clear what a good stopping criteria would be in such cases. Notice that, because of the nature of the suffix tree, as we travel down from the root, the pattern length becomes larger and occurrences become rarer. However, since the objective function has both the length and occurrences as its parameter and the occurrences have to be non-overlapping, we chose to traverse the whole tree, even though it might be expensive. Whether there exists a linear-time implementation of the greedy algorithm is still an open question [12].

Algorithm 8 Greedy Pattern Selection

1: **function** GREEDYPATTERN($suffixTreeRoot_x$)
2: $bestPattern \leftarrow \emptyset$
3: $bestCompressionSize \leftarrow -\infty$
4: **for** each edge of $suffixTreeRoot_x$ **do**
5: $tempPattern \leftarrow$ GREEDYPATTERN(edge.endNode)
6: $tempCompressionSize \leftarrow (tempPattern.length - 1) \times (tempPattern.Occurences - 1) - 2$
7: **if** $tempCompressionSize \geq bestCompressionSize$ **then**
8: $bestPattern \leftarrow tempPattern$
9: $bestCompressionSize \leftarrow tempCompressionSize$
10: $curNodePattern \leftarrow suffixTreeRoot_x.prefix$
11: $curNodeCompressionSize \leftarrow (curNodePattern.length - 1) \times (curNodePattern.Occurences - 1) - 2$
12: **if** $curNodeCompressionSize \geq bestCompressionSize$ **then**
13: $bestPattern \leftarrow curNodePattern$
14: $bestCompressionSize \leftarrow tempCompressionSize$
return $bestPattern$

4.2.2.3 Most Frequent Algorithm

Using the idea that smaller patterns are more frequent and as we traverse down from the root patterns are guaranteed to get larger and less frequent, we stop the search as soon as we find a pattern of length at least 2, in any path from the root. Notice that, if two most repeated patterns are found, the longer pattern is selected:

Algorithm 9 Maximally Repeated Pattern Selection

```
function MaxRepeatPattern(suffixTreeRoot_x)
    if suffixTreeRoot_x.prefix.length < 2 then
        for each edge of suffixTreeRoot_x do
            bestPattern ← ∅
            tempPattern ← MaxRepeatPattern(edge.endNode)
            if tempPattern.Occurences > bestPattern.Occurences
              and tempPattern.length ≥ 2 then
                bestPattern ← tempPattern

        return bestPattern
    else
        return suffixTreeRoot_x.prefix
```

4.2.2.4 Longest First

We traverse the whole tree for the longest pattern occurring at least twice. If two longest patterns are found, the one with higher frequency is chosen. Another way of achieving this would be to initiate the pattern searching from the leaves. In this way the searching can be stopped as soon as a pattern is found with non-overlapping occurrence count of at least 2. However we did not implement this because it does not significantly improve the overall runtime, as it involves traversing the whole tree to reach the leaves in the first place.

Algorithm 10 Longest Pattern Selection

```
function LongestPattern(suffixTreeRoot_x)
    bestPattern ← ∅
    for each edge of suffixTreeRoot_x do
        tempPattern ← LongestPattern(edge.endNode)
        if tempPattern.length ≥ bestPattern.length
              and tempPattern.Occurences ≥ 2 then
            bestPattern ← tempPattern
    curNodePattern ← suffixTreeRoot_x.prefix
    if curNodePattern.length ≥ bestPattern.length then
        bestPattern ← curNodePattern
    return bestPattern
```

4.3 Approximating Conditional Kolmogorov Complexity, $K(x|y^*)$

We approximate $K(x|y^*)$ in the following steps:
First, we generate an approximation of the smallest CFG, $G_y = \{\Sigma_y, \Gamma_y, S_y, \Delta_y\}$ of y, following the method described in the previous section.
After that, we create an ordering $D_{\Gamma_y} = \{T_1, T_2, \ldots, T_k\}$ of the non-terminals Γ_y such that each non-terminal T_i succeeds all the non-terminals in its definition. This is always possible as G_y is acyclic [15] and there is exactly one rule $T \rightarrow \alpha$ in Δ_y for each $T \in \Gamma_y$.
Following the ordering $D_{\Gamma_y} = \{T_1, T_2, \ldots, T_k\}$, we look for the pattern α_i in x such that $T_i \rightarrow \alpha_i$ and all the non-overlapping occurrences can be replaced with T_i. Once the list is exhausted, for every definition α_i of $T_i \in \Gamma_y$, α_i appears nowhere else in the compressed x. We denote this processed x, as $x|y^*$.
Finally, we generate an approximation of the smallest CFG of $x|y^*$ following the method described in the previous section.

4.3.1 Compressing x with y^*

Algorithm 11 Ordering of the Non-Terminals Γ_y

```
1: function buildOrdering(ordering, inputGrammar, LHS)
2:     RHS ← inputGrammar.getRule(LHS)
3:     for each symbol in RHS do
4:         if symbol.isNonTerminal then
5:             buildOrdering(ordering,inputGrammar,symbol)
6:     ordering.add(LHS)
```

Algorithm 12 Compressing x with D_{Γ_y}

Input: x, $G_y = \{\Sigma_y, \Gamma_y, S_y, \Delta_y\}$

1: $D_{\Gamma_y} \leftarrow []$
2: BUILDORDERING(D_{Γ_y},, G_y, S_y)
3: $x|y^* \leftarrow x$
4: **while** D_{Γ_y}.hasNext **do**
5: $\quad LHS \leftarrow D_{\Gamma_y}$.next
6: $\quad RHS \leftarrow G_y.getRule(LHS)$
7: $\quad tree \leftarrow buildSuffixTree(x|y^*)$
8: $\quad pattern \leftarrow tree.findOccurrences(RHS)$
9: $\quad$ REPLACE($x|y^*$,$pattern$, LHS)

Notice that even for the same grammar there can be different valid orderings D_{Γ_y} and different conditional descriptions $x|y^*$. For example: consider the following grammar G_y for $y = abcdabcdabcdfdeghdeghdegh$:

$$S_y \rightarrow T_1T_1T_1fT_2T_2T_2$$

$$T_1 \rightarrow abcd$$

$$T_2 \rightarrow degh$$

There are two valid orderings for this grammar $D_1 = \{T_1, T_2, S_y\}$ and $D_2 = \{T_2, T_1, S_y\}$. If $x = abcdegh$ and we try to compress x with y^*,we get two different conditional descriptions: $x|y^* = T_1egh$ when the ordering D_1 is used and $x|y^* = abcT_2$ when D_2 is used. This might lead to different estimates of conditional Kolmogorov Complexities. Although in our project we do not try to find the ordering that produces the smallest estimate of conditional Kolmogorov Complexity for x, it is certainly worth investigating how different orderings may lead to better approximations of Kolmogorov Complexity and thus better similarity measures.

4.4 Approximating Combined Kolmogorov Complexity, $K(x, y)$

Recall that $K(x, y)$ denotes the length of the shortest program p from which a Universal Turing Machine U computes x, y and a way to tell them apart. So, we approximated $K(x, y)$ with $K(x\#y)$ where $x\#y$ is a single string and $\#$ is a placeholder for a separator

that does not occur anywhere else in x and y. This way when the Universal Turing Machine U produces the string $x\#y$, we can still tell the strings apart. In our case, $K(x\#y)$ is approximated by the size of an efficient CFG that produces the string $x\#y$.
We could make use of the sub-additive property of $K(x,y)$ [40]: $K(x,y) \leq K(x)+K(y)$ and *Lemma 2* from [15] (it shows that there exists a grammar of size $G(x)+G(y)+2$ that generates the string xy) to approximate $K(x,y)$ with $K(x)+K(y)$. These approximations of $K(x,y)$: $K(x\#y)$ and $K(x)+K(y)$ do not differ as much when the strings have high entropy. The differences start to stack up when the strings are of low entropy with $K(x)+K(y) \geq K(x\#y)$.
To show this we make use of a Long-Short Term Memory Based Recurrent Neural Network[1] proposed by Simon and Oore [59] to generate polyphonic musical sequences. The model is trained on 1400 piano compositions from the *Yamaha e-piano Competition dataset* [1] and generates sequence by predicting the next musical event (note on-off events) by learning the probability distribution over the event space. The entropy or randomness of these sequences can be controlled by a parameter called *temperature* that dictates to what extent patterns should be repeated. Values less than 1.0 outputs sequences with more repetitions and values greater than 1.0 creates a more random sequence. Examples generated with different temperature values can be listened to on the authors' Magenta blog [58]. For each of the two temperature values 0.8 and 1.5 corresponding to low and high entropy respectively, we varied the sequence size (the size of the sequence is determined by the number of sixteenth notes in the melody) from 1400 to 2400 with interval 100 and generated two sequences, x and y, for each size. Since the output files are in MIDI format, we converted them into a sequence of MIDI pitches with "mftext" [61] and estimated $K(x,y)$ for each x,y pair of each size with the two approximations: $K(x\#y)$ and $K(x)+K(y)$. For $K(x\#y)$, we joined x and y with the number 100000 as a separator in between because since both x and y are a sequence of MIDI pitches that range from 0 to 127, the number 100000 occurs nowhere in them. For $K(x)+K(y)$, we simply treated the strings separately, estimated their Kolmogorov Complexities with the size of their encoded CFG and added them. We found that when the music sequences are of high entropy (temperature 1.5), the difference between the two approximations of $K(x,y)$ is negligible. But the difference is prominent when the sequences are of low entropy (temperature 0.8), with $K(x)+K(y)$ being much smaller than $K(x\#y)$:

[1] The code for the music generating Neural Network can be found at https://github.com/tensorflow/magenta

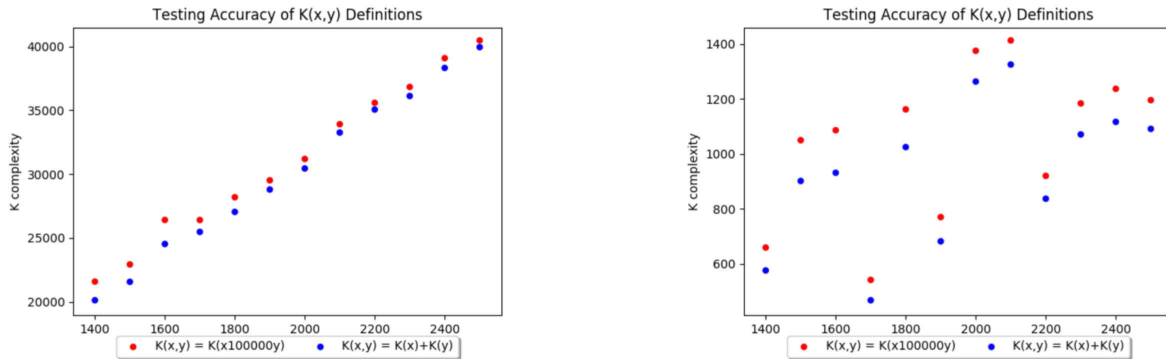


Figure 4.2: The figure on the left shows that there is little difference between the two approximations of $K(x, y)$ when the sequences are of high-entropy and mostly incompressible. The figure on the right shows that the difference between them increases when sequences have low-entropy and are more compressible.

One explanation for this could be that: when x and y individually have low Kolmogorov complexities but share little or no information with each other (differences in alphabet size, structure of patterns etc.), their combination $x\#y$ becomes more complex. Following is an example of such x and y (generated by the recurrent neural network with temperature 0.8 and with unequal sizes), for which our approximation of $K(x\#y)$ is greater than the approximation of $K(x) + K(y)$. The individual short CFGs for x and y are:

$$
\begin{array}{l}
T1 \rightarrow T7\ T4\ 55\ T5\ 55\ T5\ T3\ 60\ 62 \\
\quad 60\ T4\ 62\ T3\ T6\ T5\ 55\ T2\ 59 \\
\quad 64\ 64\ T10\ T3\ T3\ T4\ T2\ T3 \\
\quad T10\ 67\ 67\ T7\ T6\ T8\ T7\ 57 \\
\quad 57\ T10\ T4\ 55\ 62\ 55\ 62\ T6 \\
\quad 64\ T7\ T6\ T6\ T2\ T4\ T9\ T3\ T3\ T5 \\
\quad T5\ T7\ T4\ T4\ T4\ 67\ T3\ T4\ T6\ T2 \\
\quad T4\ 57\ T4\ T9\ T8\ T8\ T8\ T2\ 60\ T5 \\
\quad 55\ T2\ 60\ T9\ 55\ T8\ 65\ 65\ 65\ T8 \\
\quad T4\ 62\ T8\ T2\ 60 \\
T2 \rightarrow T7\ T7 \\
T3 \rightarrow T6\ T7 \\
T4 \rightarrow 62\ 62 \\
T5 \rightarrow 60\ 55\ 60 \\
T6 \rightarrow 55\ 55 \\
T7 \rightarrow 60\ 60 \\
T8 \rightarrow T2\ T2 \\
T9 \rightarrow T4\ T4\ T6 \\
T10 \rightarrow 57\ 57\ T4
\end{array}
$$

$$
\begin{array}{l}
T1 \rightarrow T8\ T4\ T4\ T4\ T4\ T4\ T2\ T5\ 69 \\
\quad T9\ 72\ 71\ 71\ T9\ 71\ T7\ T6\ T6 \\
\quad T6\ T6\ T10\ T10\ 62 \\
T2 \rightarrow 60\ T5\ 67\ T8 \\
T3 \rightarrow 64\ T7 \\
T4 \rightarrow T2\ T2\ T2\ T2\ T2\ T2 \\
T5 \rightarrow T10\ T11\ T11\ 67 \\
T6 \rightarrow T3\ T3\ T3\ T3\ T3 \\
T7 \rightarrow T10\ T10\ T10\ T11\ T10\ 64 \\
T8 \rightarrow 60\ 60\ 60\ 60\ T5\ 67\ T11\ 60 \\
T9 \rightarrow 69\ 69\ 69\ 69\ 69\ 71\ 71\ 72 \\
T10 \rightarrow 62\ 62 \\
T11 \rightarrow 64\ 64
\end{array}
$$

From these CFGs, we approximate $K(x) + K(y)$ to be about 830 bits. If we join x and y with "100000" as a separator, the approximation of $K(x\#y)$ increases to about

941 bits:

$$T1 \rightarrow T13\ T5\ 55\ T10\ 55\ T10\ T6\ 60\ 62\ 60\ T5\ 62\ T6\ T11\ T10\ 55\ T4\ 59\ T16\ T18\ T6\ T6\ T5\ T4\ T6\ T18\ 67\ 67\ T13\ T11\ T15\ T13\ 57\ 57\ T18\ T5\ 55\ 62\ 55\ 62\ T11\ 64\ T13\ T11\ T11\ T4\ T12\ T11\ T6\ T6\ T10\ T10\ T13\ T12\ 67\ T6\ T5\ T11\ T4\ T5\ 57\ T12\ T11\ T15\ T15\ T15\ T4\ 60\ T10\ 55\ T4\ 60\ T5\ T5\ T11\ 55\ T15\ 65\ 65\ 65\ T15\ T5\ 62\ T15\ T4\ 60\ 100000\ T17\ T7\ T7\ T7\ T7\ T7\ T2\ T8\ 69\ T14\ 72\ 71\ 71\ T14\ 71\ T19\ T9\ T9\ T9\ T9\ T5\ T5\ 62$$

$$T2 \rightarrow 60\ T8\ 67\ T17$$

$$T3 \rightarrow 64\ T19$$

$$T4 \rightarrow T13\ T13$$

$$T5 \rightarrow 62\ 62$$

$$T6 \rightarrow T11\ T13$$

$$T7 \rightarrow T2\ T2\ T2\ T2\ T2\ T2$$

$$T8 \rightarrow T5\ T16\ T16\ 67$$

$$T9 \rightarrow T3\ T3\ T3\ T3\ T3$$

$$T10 \rightarrow 60\ 55\ 60$$

$$T11 \rightarrow 55\ 55$$

$$T12 \rightarrow T5\ T5\ T5$$

$$T13 \rightarrow 60\ 60$$

$$T14 \rightarrow 69\ 69\ 69\ 69\ 69\ 71\ 71\ 72$$

$$T15 \rightarrow T4\ T4$$

$$T16 \rightarrow 64\ 64$$

$$T17 \rightarrow T4\ T8\ 67\ T16\ 60$$

$$T18 \rightarrow 57\ 57\ T5$$

$$T19 \rightarrow T12\ T16\ T5\ 64$$

However, when x and y are similar to each other, our approximation of $K(x\#y)$ stays below $K(x) + K(y)$. Following is an example of x and y where $y = x$ (sharing the maximum amount of information) and $K(x\#y) \approx K(x)$:

$T1 \to T7\ T4\ 55\ T5\ 55\ T5\ T3\ 60$
$62\ 60\ T4\ 62\ T3\ T6\ T5$
$55\ T2\ 59\ 64\ 64\ T10\ T3\ T3\ T4\ T2\ T3$
$T10\ 67\ 67\ T7\ T6\ T8\ T7\ 57\ 57$
$T10\ T4\ 55\ 62\ 55\ 62\ T6\ 64\ T7\ T6$
$T6\ T2\ T4\ T9\ T3\ T3\ T5\ T5\ T7\ T4\ T4$
$T4\ 67\ T3\ T4\ T6\ T2\ T4\ 57\ T4$
$T9\ T8\ T8\ T8\ T2\ 60\ T5\ 55\ T2\ 60\ T9$
$55\ T8\ 65\ 65\ 65\ T8$
$T4\ 62\ T8\ T2\ 60$
$T2 \to T7\ T7$
$T3 \to T6\ T7$
$T4 \to 62\ 62$
$T5 \to 60\ 55\ 60$
$T6 \to 55\ 55$
$T7 \to 60\ 60$
$T8 \to T2\ T2$
$T9 \to T4\ T4\ T6$
$T10 \to 57\ 57\ T4$

$T1 \to T2\ 100000\ T2$
$T2 \to T8\ T5\ 55\ T6\ 55\ T6\ T4\ 60$
$62\ 60\ T5\ 62\ T4\ T7\ T6\ 55$
$T3\ 59\ 64\ 64\ T11\ T4\ T4\ T5\ T3\ T4\ T11$
$67\ 67\ T8\ T7\ T9\ T8\ 57\ 57\ T11\ T5$
$55\ 62\ 55\ 62\ T7\ 64\ T8\ T7\ T7\ T3$
$T5\ T10\ T4\ T4\ T6\ T6\ T8\ T5\ T5\ T5\ 67$
$T4\ T5\ T7\ T3\ T5\ 57\ T5\ T10\ T9$
$T9\ T9\ T3\ 60\ T6\ 55\ T3\ 60\ T10\ 55\ T9$
$65\ 65\ 65\ T9\ T5\ 62\ T9\ T3\ 60$
$T3 \to T8\ T8$
$T4 \to T7\ T8$
$T5 \to 62\ 62$
$T6 \to 60\ 55\ 60$
$T7 \to 55\ 55$
$T8 \to 60\ 60$
$T9 \to T3\ T3$
$T10 \to T5\ T5\ T7$
$T11 \to 57\ 57\ T5$

The individual short CFG for x on the left gives an approximated Kolmogorov complexity of about 492 bits for x, while the CFG on the right gives an approximated Kolmogorov complexity of about 527 bits for the concatenated $x\#x$. Thus, $K(x\#x) \approx K(x)$.

However, the *size* of the generated CFG of $x\#y$ (defined by the total number of symbols in the definitions of the non-terminals) remains smaller than the cumulative *size* of the individual CFGs of x and y. Following is a comparison of the sizes of the CFGs involved in approximating $K(x\#y)$ and $K(x) + K(y)$ for the same set of music strings used for figure 4.2:

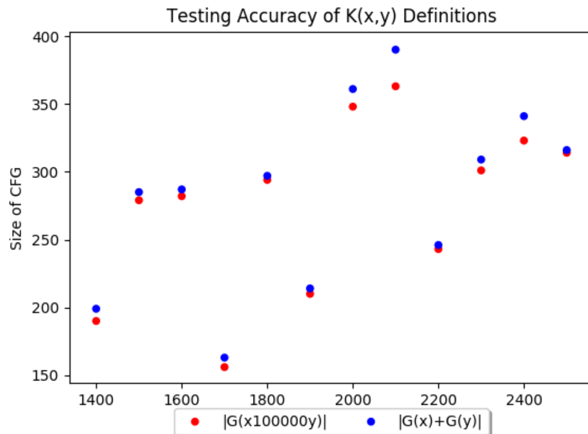


Figure 4.3: The size of CFG of $x100000y$ is smaller than the cumulative size of the individual CFGs of x and y

Despite having a smaller CFG size, our approximation of $K(x\#y)$ is greater than that of $K(x) + K(y)$ as it takes more bits to encode the CFG for $x\#y$ (due to the larger alphabet needed for describing and separating x and y in the description). However, these very properties of $K(x\#y)$ are helpful for us to contain the value of the Normalized Information Distance: $\frac{K(x|y^*+K(y|x^*)}{K(x\#y)}$ within $[0,1]$. When x and y are individually of low-entropy without sharing much information between them, $\frac{K(x|y^*+K(y|x^*)}{K(x)+K(y)}$ runs the risk of being greater than one, which does not happen when we use $K(x\#y)$. Besides, we have not seen any improvement in clustering and classification of objects when switching between these two definitions. So we approximate $K(x,y)$ with $K(x\#y)$.

4.5 Runtime Analysis

We analyse the runtime of `CFGGlobalAlgorithm` presented in algorithm 6. Notice that since the size of the grammar is strictly decreasing at each iteration, an upper bound on the number of iterations is $O(n)$.
At each iteration, we create a generalized suffix tree of the grammar using Ukkonen's $O(n)$ algorithm and during the tree building process we count the non-overlapping occurrences of each of its internal nodes. Since there are $O(n)$ leaves in tree rooted at each internal

node, it takes $O(n)$ time to count the non-overlapping occurrences at each internal node (refer to Appendix B). The number of internal nodes in a suffix tree is also $O(n)$. So creation of the generalized suffix tree and augmenting it with necessary string statistics takes $O(n^2)$.
Searching time for "Greedy", "Most Frequent" or "Longest" pattern and subsequent replacements of these patterns with a non-terminal is upper bounded by $O(n)$, since these operations only involve traversing the tree and iterating over the grammar. Hence, the total runtime of the algorithm is bounded by $O(n^3) + O(n^2) = O(n^3)$.
The most expensive operation carried out in this algorithm is augmenting the suffix tree with necessary non-overlapping occurrence information. This can be made faster by building a *Minimal Augmented Suffix Tree*(MAST) [4, 11] which permits computation of the non-overlapping occurrences of a pattern α in $O(|\alpha|)$ time. The best known algorithm to construct a MAST has a time complexity of $O(n \log n)$ [9]. This way the total runtime for the `CFGGlobalAlgorithm` could be improved to $O(n^2 \log n)$. We did not implement this because the algorithm requires the effort to augmenting the suffix tree with extra information at each internal node for faster retrieval of non-overlapping occurrences, which is beyond the scope of this thesis.
We ourselves have been very pessimistic about the runtime of our algorithm being $O(n^3)$. Although the upper bound of the outer loop at line 5 of `CFGGlobalAlgorithm` is $O(n)$, in practice it will never reach n, as at each step a new rule is added and its never the case that there are n rules in the CFG for a string of length n: this would imply that there is at least one non-terminal in the grammar that has only one symbol in its definition.
However, a tighter bound for the number of iterations is also not easy to prove: recall that Global Grammars have an approximation ratio for the grammar size of $O((\frac{n}{\log n})^{\frac{2}{3}})$. So:

$$\frac{GlobalGrammarSize}{SmallestGrammarSize} = O((\frac{n}{\log n})^{\frac{2}{3}})$$

$$GlobalGrammarSize = O((\frac{n}{\log n})^{\frac{2}{3}}) \times SmallestGrammarSize$$

The size of the grammar is initially n and when the loop terminates it becomes $GlobalGrammarSize$. In the worst case, the grammar size reduces by atmost one at each step. So the outer loop runs $n - GlobalGrammarSize$ times. This term is maximum when $GlobalGrammarSize$ in turn $SmallestGrammarSize$ is smallest. The lower bound of the size of the smallest grammar is $\Omega(\log n)$ [15]. Suppose, for the worst case analysis, $SmallestGrammarSize = \log n$. Then the number of times the loop runs is $O(n - (n^2 \log n)^{\frac{1}{3}})$, which does not improve significantly over the previous bound.

4.6 Visualization and Emergent Properties of the Generated Context Free Grammars

In this section, we present some examples of Context Free Grammars generated by our algorithms.

4.6.1 Parse Tree of x (without compressing it with y^*)

The best approximation of the smallest CFG for the vocal melody of "Hey Jude" by The Beatles is found by the "Greedy" algorithm and its as follows:

$$\begin{aligned}
T1 &\rightarrow 60\ T3\ 60\ T3\ T4\ T4\ T6\ T3\ T4\ 57\ T10\ 70\ T10\ 67\ T12\ T12\ 67\ T7 \\
&\quad 55\ T8\ T6\ 65\ 60\ T9\ T3\ 55\ 57\ 59\ 60\ 64\ 65\ 68\ 69\ 71\ 72\ 77\ T5\ T5\ T5 \\
T2 &\rightarrow 53\ 57\ 60\ 67\ T10\ T10\ T10\ 65\ 63\ T9 \\
T3 &\rightarrow 57\ T11\ 55\ 55\ 57\ 58\ 65\ 65\ 64\ 60\ T7\ T11\ 62\ 62\ 67\ 65\ 64\ 65\ T9\ 53 \\
&\quad 55\ 57\ T9\ 60\ 58\ 57\ 52\ 53 \\
T4 &\rightarrow 53\ T12\ T9\ 60\ 58\ 62\ T12\ 65\ 58\ T8 \\
T5 &\rightarrow T2\ T2\ T2\ T2\ T2\ T2 \\
T6 &\rightarrow T11\ 63\ 63\ 64\ T10\ 67\ 60\ 62 \\
T7 &\rightarrow T9\ 58 \\
T8 &\rightarrow 65\ T7\ 60\ T7\ 57\ 55\ 53 \\
T9 &\rightarrow 62\ 60 \\
T10 &\rightarrow 65\ 67 \\
T11 &\rightarrow 57\ 60\ 62 \\
T12 &\rightarrow 65\ 62
\end{aligned}$$

To better visualize the structure we present a partial parse tree of the grammar created by Python's Natural Language Processing Library [51]:

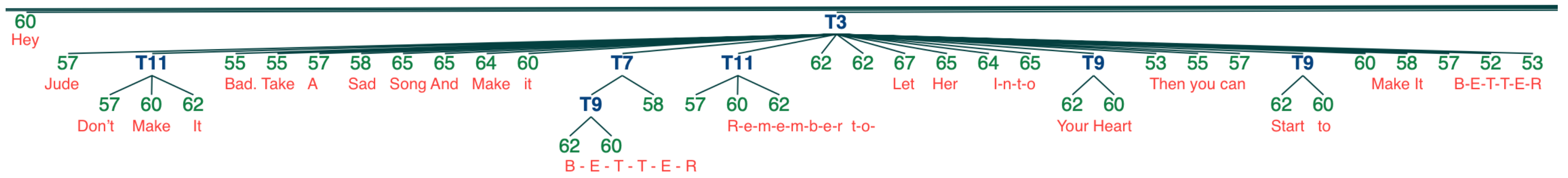


Figure 4.4: Partial parse tree for "Hey Jude". Lyrics are added with the help of De Clercq and Temperley's encodings of the lyrical information of the melodies [23, 21].

4.6.2 Compression Ratio

To determine the compression ratio of a concise description of a symbolic music string x, we consider:

$$\frac{\text{number of bits in the description of } x}{\text{number of bits in } x}$$

In our case, the numerator of the ratio is determined by the number of bits needed to encode an efficient CFG of x such that the decoder ϕ can reconstruct x from the encoding. In other words, we consider the approximation of $K(x)$ by an efficient CFG.
Since each symbolic music string x originally is a sequence of integers from 0 to 127, we determine the uncompressed length of the string by:

$$|x| = \log(\Sigma) \times n$$

where Σ is the number of unique symbols/ integers in the string and n is the number of total symbols in x. Hence, our approximation of the compression ratio of x becomes:

$$\frac{\text{Approximation of } K(x)}{|x|}$$

We used the above approximation to determine the average compression ratio of the three genres of music we use in our thesis. For each genre we randomly selected twelve pieces that belong to them from the list presented in Appendix C. We calculated the compression ratio using the above approximation for each of the twelve pieces in each genre and retrieved the average compression ratio for each genre. We repeated the process five times to eliminate sampling bias as much as possible and averaged the findings. For the genre of rock, along with the vocal melodic transcriptions, we also calculated the average compression ratio of the instrumental transcriptions:

Genre	Average Compression Ratio
Rock (Vocal Melodic Transcription)	65.77%
Rock (Instrumental Transcription)	68.50%
Classical	74.23%
Jazz	93.64%

Table 4.1: Compression Ratio for the three Genres

As expected, rock songs are more compressible than the classical and jazz songs.

4.6.3 Parse Tree of x (after compressing it with y^*)

Now we want to demonstrate how the system builds an efficient CFG of a string x after compressing it with the CFG of y. The individual smallest grammars for the vocal melodies of "Eleanor Rigby" (Object 1) and "I Want to Hold Your Hand" (Object 2) by "The Beatles" are as following respectively:

$T1 \rightarrow T3\ T3$ 57 $T2$ 55 $T2\ T3\ T3$ 55 $T2$
$T2 \rightarrow T4\ T4\ T7$ 64 $T6$ 52 57 $T7$ 67 64 $T6$
$T3 \rightarrow$ 64 66 67 69 67 66 64 $T6$
$T4 \rightarrow$ 57 59 55 52 $T5$ 62 61 59 61 59 57 $T6$ 57 $T5$ 60 59 57
$T5 \rightarrow$ 55 57 59
$T6 \rightarrow$ 59 57 55
$T7 \rightarrow T5$ 55 52 52

$T1 \rightarrow$ 64 $T2\ T8\ T2\ T3\ T3$ 60 59 $T6\ T8$
$T2 \rightarrow T4$ 54 $T4$ 66 $T6$ 60 $T5\ T8\ T6$
$T3 \rightarrow$ 57 55 $T7$ 55 55 55 57 $T7\ T9\ T9\ T9\ T2$
$T4 \rightarrow$ 64 62 60 59 59 $T5$ 59 59 59 59 59
$T5 \rightarrow$ 62 60 59 57
$T6 \rightarrow$ 67 66 64 62
$T7 \rightarrow$ 57 60 64 $T5$ 55
$T8 \rightarrow$ 59 57 55
$T9 \rightarrow$ 60 60 62

When *Object 1* is compressed by *Object 2* before computing the CFG of the compressed version, we see that there are uses of the rules of the second object's CFG in the first object's conditional compression:

$T1 \rightarrow T11\ T11$ 57 $T10$ 55 $T10\ T11\ T11$ 55 $T10$
$T10 \rightarrow T12$ 59 57 $T12\ T8\ T13$ 52 64 $T8$ 52 57 55 $T13$ 52 67 64 $T8$
$T11 \rightarrow$ 64 66 67 69 67 66 64 $T8$
$T12 \rightarrow T13$ 55 57 59 62 61 59 61 59 57 $T8$ 57 55 57 59 60
$T13 \rightarrow$ 57 59 55 52

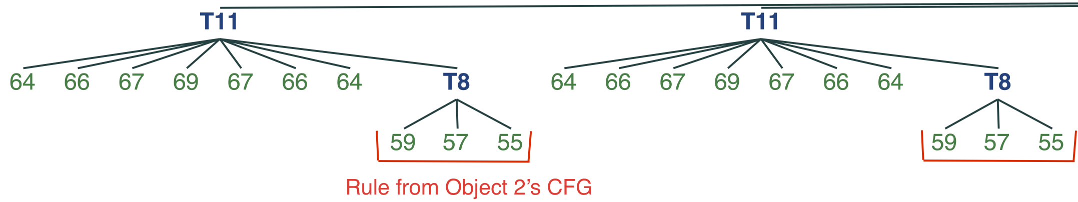


Figure 4.5: Conditional Compression of "Eleanor Rigby" with "I Want to Hold Your Hand"

Similarly, in the conditional compression of *Object 2* with *Object 1* we see the uses of rules from *Object 1*'s CFG:

$$T1 \rightarrow 64\ T8\ T6\ T8\ T9\ T9\ 60\ 59\ T13\ 62\ T6$$
$$T8 \rightarrow T10\ 54\ T10\ 66\ T13\ T11\ T11\ 59\ 57\ T6\ T13\ 62$$
$$T9 \rightarrow 57\ 55\ T12\ 55\ 55\ 55\ 57\ T12\ 60\ 60\ T11\ 60\ T11\ 60\ 62\ T8$$
$$T10 \rightarrow 64\ T11\ 59\ 59\ T11\ 59\ 57\ 59\ 59\ 59\ 59\ 59$$
$$T11 \rightarrow 62\ 60$$
$$T12 \rightarrow 57\ 60\ 64\ T11\ T6$$
$$T13 \rightarrow 67\ 66\ 64$$

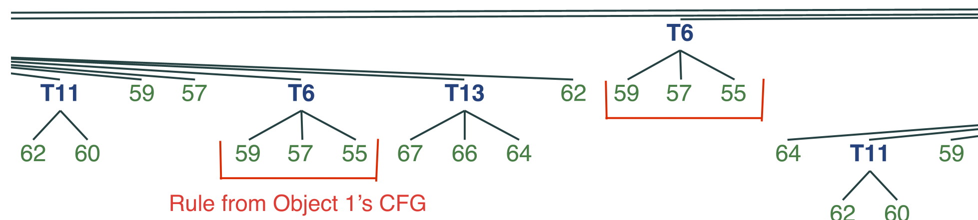


Figure 4.6: Conditional Compression of "I Want to Hold Your Hand" with "Eleanor Rigby"

4.6.4 Inter and Intra-Corpora Normalized Distance for the Three Genres

We consider twelve music pieces belonging to each genre to approximate the inter and intra-corpora distance between them. The twelve pieces are randomly selected from the list of music we use for each genre. We repeated the following processes five times and averaged the results. We use instrumental transcriptions for the all the music pieces in these experiments.
For intra-corpora distance in a genre, G, we approximated the normalized distance $d(x,y) = \frac{K(x|y^*)+K(y|x^*)}{K(x,y)}$ for every $(x,y) : y \neq x, x, y \in G$. We averaged the distance over the number of possible (x,y) pairs, $\frac{n(n-1)}{2}$, where n is the number of objects in G. Notice, we use the symmetric property of $d(x,y)$, so $d(x,y)$ is considered equal to $d(y,x)$.

Genre	Intra-corpora distance
Rock	0.85
Classical	0.82
Jazz	0.84

Table 4.2: Intra-corpora Distance of three Genres

For inter-corpora distance between two genres G_1 and G_2, we approximate $d(x,y)$ for every pair $(x,y) : x \in G_1, y \in G_2$. We average the distance over the number of calculations, $n \times m$ where n and m are the number of music pieces in G_1 and G_2 respectively.

Genre	Inter-corpora distance
Between rock and classical	0.93
Between jazz and rock	0.94
Between classical and jazz	0.90

Table 4.3: inter-corpora Distance of three Genres

As expected the inter-corpora distance is on average larger than the intra-corpora distance.

4.7 2-Dimensional Scaling and Support Vector Classification

We now present some graphical representation of the above results: whether music pieces from the same composer or genre have smaller intra-corpora distances than inter-corpora distances from music pieces of another composer or genre. In other words whether an efficient CFG along with a normalized distance metric $d(x, y)$ can differentiate one composer or genre from another by recognizing style in the music pieces. For that purpose, we project the distances onto a 2-dimensional Euclidean space to visualize whether the objects belonging to the same corpus are closer together than objects belonging to different corpora. We consider N corpora, each of which contain music pieces that belong to one single composer from any of the three genres. If the total number of objects belonging to the N corpuses is n, we approximate the normalized distance $d(x, y)$ between all the $n \times n$ number of (x, y) pairs and create a $n \times n$ distance matrix. We then translate this information into a configuration of n points mapped into an abstract 2-D Cartesian space, in the hope that similar objects are positioned closer than dissimilar objects. We use a Java library for Multi-dimensional Scaling [2] to achieve this 2-Dimensional scaling. An example would be the following where we take the musical pieces of three composers: The Beatles, Bach and Miles Davis:

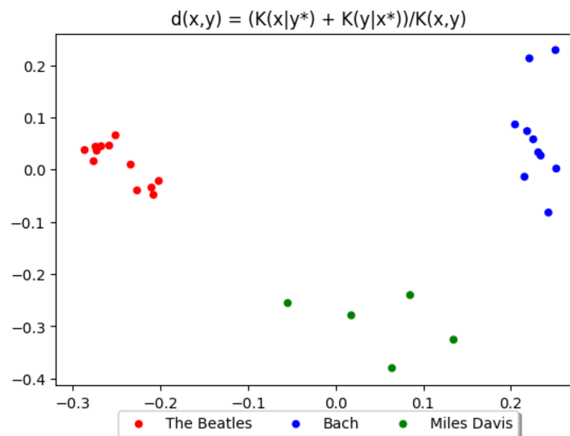


Figure 4.7: The figure is a plane translation of a 27×27 distance matrix to 27 geometric positions on a 2-D Cartesian Plane. It's apparent that musical pieces belonging to the same composers, are closer to each other than those belonging to different composers.

The different categories that the n objects belong to are highlighted with different colors

to visually separate them and understand the efficacy of our system. However, to ensure that our eyes are not doing the clustering, we use Python's Scikit Learn module [51] to run the data through a Support Vector Machine (SVM) with Linear Kernel [56] with their corresponding composers to linearly classify them. We can specify a positive regularization parameter C that tells the SVM optimization how much misclassification of the training samples should be avoided. Larger C values will let the SVM choose a smaller-margin hyperplane if it does a better job of classifying all the training samples correctly. But it runs a risk of over-fitting and the hyperplane can become complex. We thus use a small C value of 1, so that the classifier can generalize better, even if it can misclassify more data points. The linear classification of Figure 4.7 by the SVM is as following:

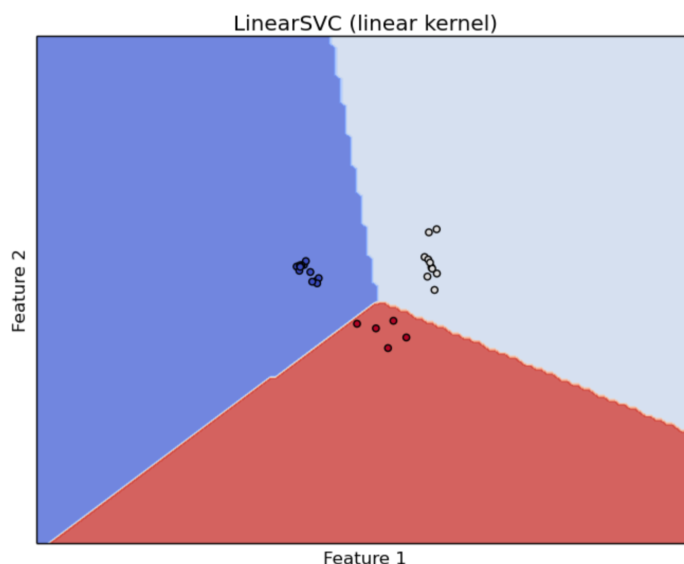
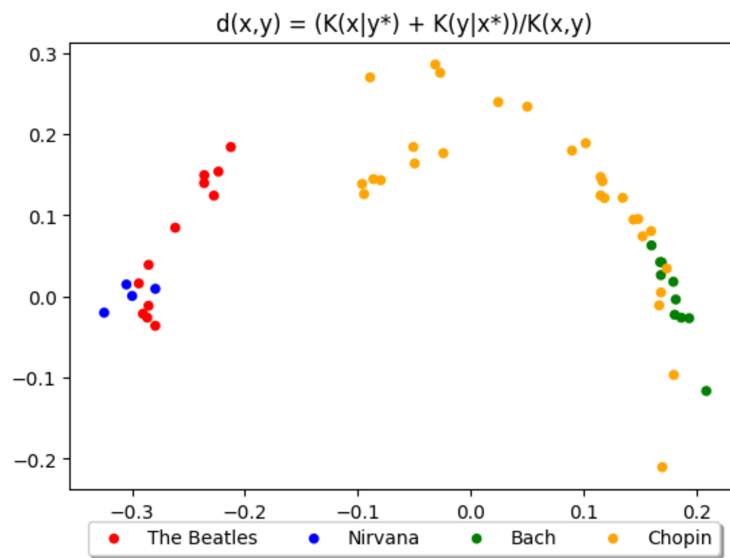


Figure 4.8: SVM with linear kernel classifies the data into three composer categories

4.7.1 Genre Specific Style Recognition

To determine whether our system can recognize style in music, we first experiment whether it can differentiate between the three genres. For this purpose, we select two or three composers specific to each genre and perform two dimensional scaling on the distance matrix of instrumental transcriptions of the music pieces of the composers. Three such experiments were performed and the SVM in each case could identify the different genres:



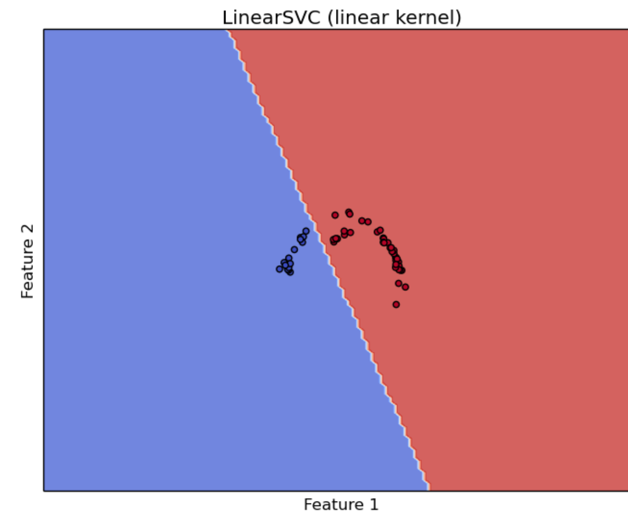
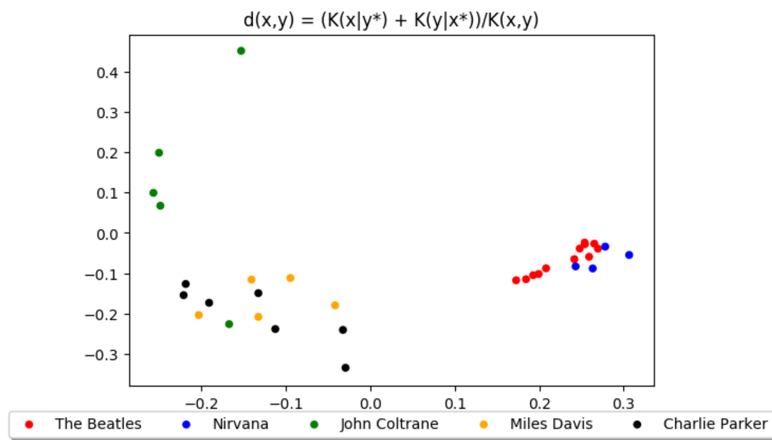


Figure 4.9: 2-dimensional scaling and classification with linear kernel of rock and classical music.



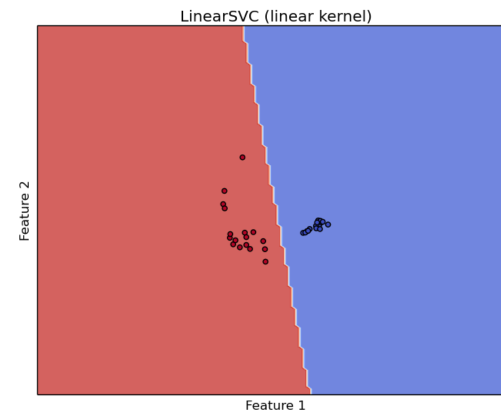


Figure 4.10: 2-dimensional scaling and classification with linear kernel of rock and jazz music.

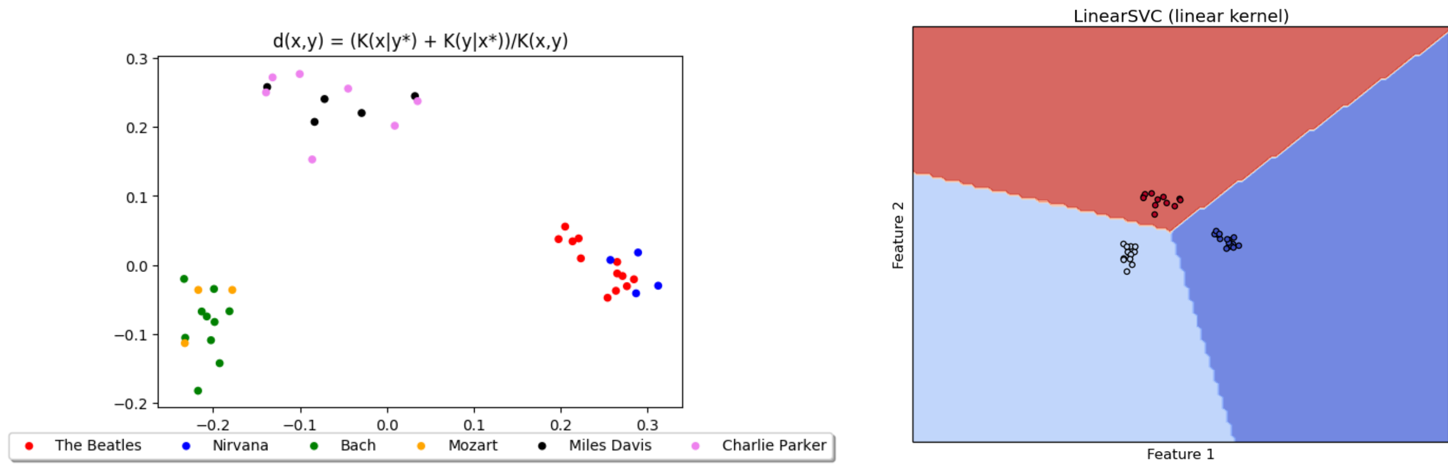
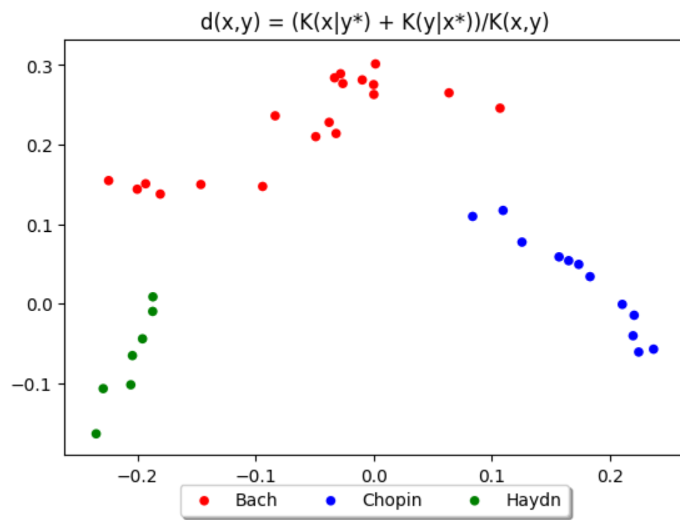


Figure 4.11: 2-dimensional scaling and classification with linear kernel of rock, jazz and classical music.

4.7.2 Composer (Same Genre) Specific Style Recognition

We now select composers from the same genre and experiment whether the system can recognize different composers. Distances between music pieces within the same genre has been shown to be less than those between music pieces of different genres in the previous section. But composers belonging to the same genre exhibit different style in their compositions and the system can correctly distinguish them. We experiment on composers from the classical and the rock genre with the instrumental transcriptions of the music pieces.



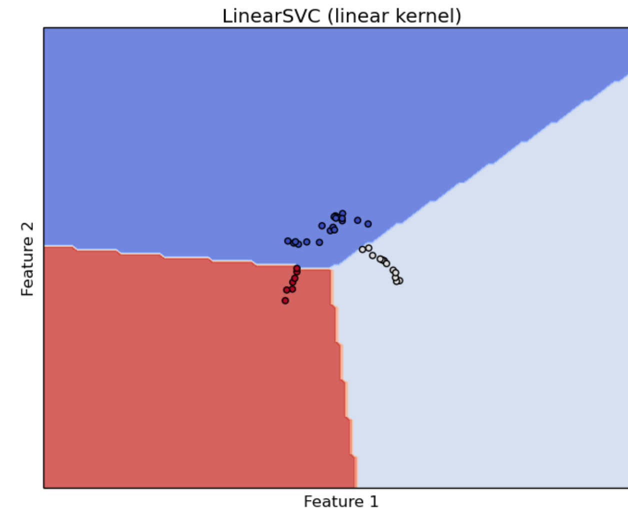
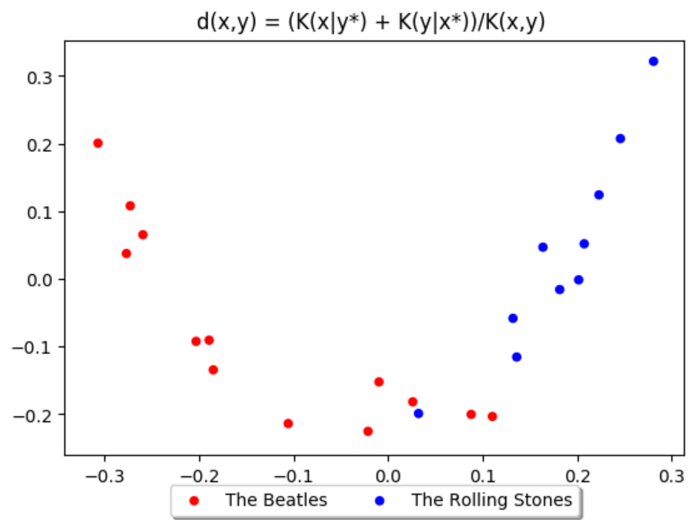


Figure 4.12: 2-Dimensional Scaling and Linear Classification of Chopin, Bach and Haydn. To represent Bach and Chopin in this experiment, we chose Bach's Well-Tempered Claviers, Inventions and Chopin's Études respectively.



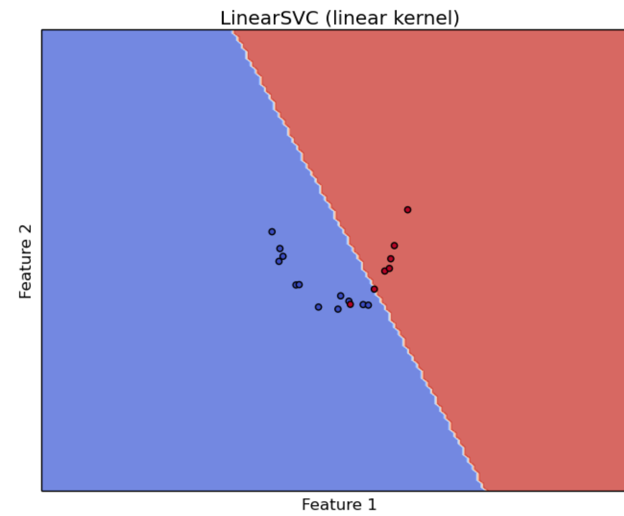
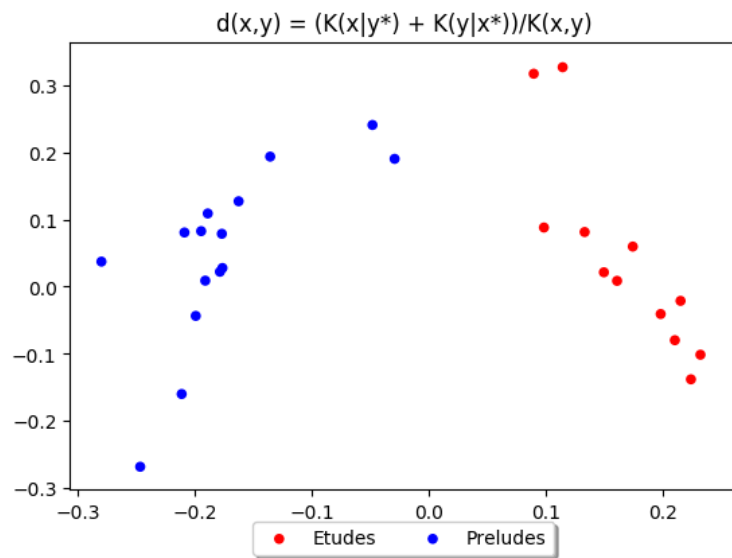


Figure 4.13: 2-Dimensional Scaling and Linear Classification of The Beatles and The Rolling Stones

4.7.3 Composition (Same Composer) Specific Style Recognition

Music pieces from the same composer can exhibit vastly different style in their composition. For example: Chopin's Études (composed for solo pianists and are of considerable difficulty [49]) are quite different from his Preludes (short pieces of piano music [49, 22]). We perform two experiments to see whether the system can identify Chopin's Preludes from his Études and his Nocturnes (inspired by the nature of the night [49, 22])



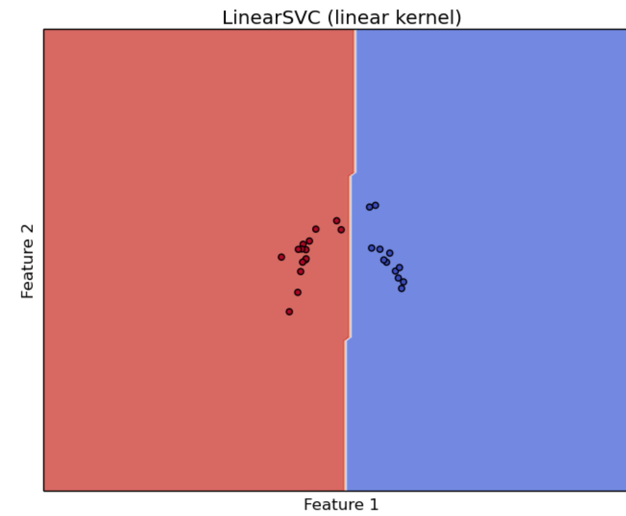
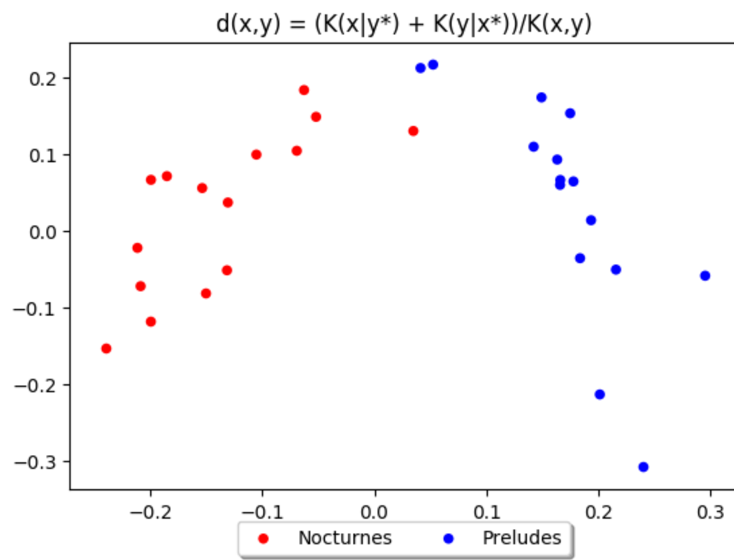


Figure 4.14: 2-Dimensional Scaling and Linear Classification of Chopin's Études and Preludes



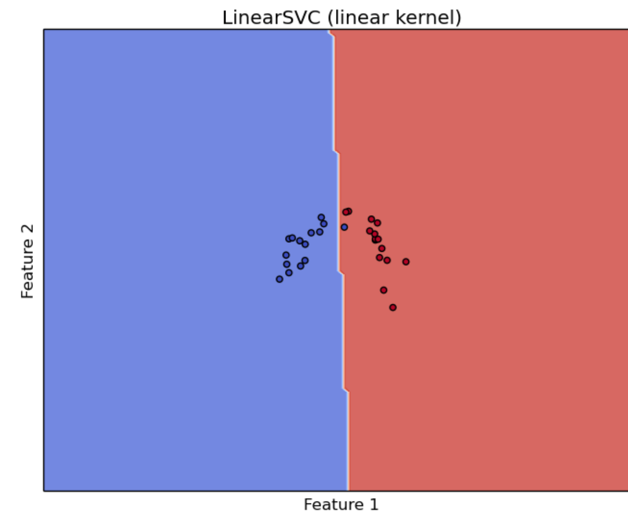


Figure 4.15: 2-Dimensional Scaling and Linear Classification of Chopin's Nocturnes and Preludes

4.8 Best-Fitting Composer Selection

4.8.1 Method

In this section we present a practical application of the Constrained Best-fit model selection discussed in section 2.3: composer identification for an unknown musical string x. For this, we have models G_i for $i = 1, 2, \ldots, n$ each representing a certain composer i. Each model G_i represents a CFG for the musical pieces composed by the composer i such that $K(G_i) \leq \alpha$. Recall that a more suitable restriction parameter for CFG, β dictates the minimum number of musical pieces that a maximal string γ has to be present in so that $T \rightarrow \gamma$ can be included in G_i, and β and α can be calculated from each other. So in order to maintain consistency and simplicity, instead of providing α we provide the parameter β and we allow each model to represent a fixed number of m musical pieces $\{x_i^1, x_i^2, \ldots, x_i^m\}$. With these restrictions in place, each model G_i, notwithstanding the genre and composer specific variations, can only contain a certain number of non-terminals with definitions of certain length that are functions of m and β. Thus for $i, j = 1, 2, \ldots, n$, we can assume $K(G_i) = K(G_j)$ where $i \neq j$ with small additive error. So to find the best-fitting model we try to find the G that is best-able to describe x such that $K(x|G)$ is the smallest and thus witnesses Kolmogorov's structure function $h_x(\alpha)$ in equation (2.15).
To construct G_i we first consider a random ordering $x_i^1, x_i^2, \ldots, x_i^m$. We build G_i incrementally in m steps and in a cascading manner: we construct an efficient CFG for x_i^1, G_i^1. In this first step, all the γ's in the definitions of non-terminals $T \rightarrow \gamma$ in G_i^1 have presence in one string x_i^1. Then we compress x_2^i with G_i^1. We keep track of the non-terminals $T \rightarrow \gamma$ in G_i^1 that were used to compress x_2^i and we increase their occurrence in strings counter by one. After that we compute an efficient CFG of the compressed string, $x_2^i|G_i^1$, add the non-terminal definitions of this CFG to G_i^1 with string occurrence counter 1 and get G_i^2. We repeat this process until x_m^i is compressed with G_i^{m-1} and the non-terminal definitions are added to G_i^{m-1} to get the final grammar G_i^m. Note that this final grammar has all the rules of the efficient CFGs of the m strings. Hence, $K(G_i^m)$ is quite large. We thus select the rules in G_i^m that occur in at least $m \times \frac{\beta}{100}$ number of music pieces and add them to G_i which is the final model of the m strings with restricted complexity. Note that, G_i created this way, does not necessarily produce any string, as for large β, our algorithm seeks the most common patterns that occur in more than one of the strings. Thus the string generating start rules which are unique to their corresponding sequences get omitted. Rather G_i acts as a dictionary with entries that describe the important regularities of the strings that it represents.

Algorithm 13 Approximating G_i

Input: $\{x_1^i, x_2^i, \ldots, x_m^i\}$, β

1: $G_i \leftarrow \emptyset$
2: $cumulativeCFG \leftarrow smallestCFG(x_1^i)$
3: $cumulativeCFG.setRuleOccurrences(1)$ ▷ In the beginning each rule belongs to only one grammar
4: **for** $j = 2$ to m **do**
5:     **for** each rule in $cumulativeCFG$ **do**
6:         $tree \leftarrow buildSuffixTree(x_j^i)$
7:         **if** $tree.containsPattern(rule.RHS)$ **then**
8:             $x_j^i.replace(rule.RHS, rule.LHS)$
9:             $cumulativeCFG.increaseOccurrencebyOne(rule)$
10:     $grammarOfCompressed \leftarrow smallestCFG(x_j^i)$
11:     $grammarOfCompressed.setRuleOccurrences(1)$
12:     $cumulativeCFG.addRules(grammarOfCompressed)$
13: **for** each rule in cumulativeCFG **do**
14:     **if** $rule.Occurrence \geq \frac{\beta \times m}{100}$ **then**
15:         $G_i.insertRule(rule)$
return G_i

For each such model G_i, the optimality measure of G_i for describing x, $K(x|G_i)$ is computed. The model that yields the smallest positive value is selected as the witness for $h_x(\alpha)$ and thus the best-fitting model, or in this case, the best-fitting composer for x.

Algorithm 14 Best Fitting Model/ Composer

Input: Models:$\{G_1, G_2, \ldots, G_n\}$, x

1: $RD \leftarrow []$
2: **for** $i = 1$ to n **do**
3:     $\bar{x} \leftarrow compress(x, G_i)$
4:     $K(x|G_i) \leftarrow smallestCFG(\bar{x}).KolmogorovComplexity$
5:     $RD.insert(K(x|G_i))$
return $\arg\min_{G_i}\{RD\}$

4.8.2 Effectiveness of the Generated Model

To demonstrate the model generation and the subsequent compression of x with the generated model, we present here an example. We first generate a model grammar G with $\beta = 20\%$ for twelve music pieces (vocal melodic transcriptions) by The Beatles, as given below:

$T9 \rightarrow 62\ 60$
$T10 \rightarrow 65\ 67$
$T12 \rightarrow 65\ 62$
$T17 \rightarrow 62\ 62$
$T22 \rightarrow 64\ 66$
$T30 \rightarrow 57\ 59\ 60$
$T32 \rightarrow 61\ 59$
$T37 \rightarrow 59\ 57\ 55$
$T45 \rightarrow 57\ 59$
$T50 \rightarrow 67\ 66\ 64$
$T53 \rightarrow 57\ 57\ 57$
$T70 \rightarrow 64\ 62$
$T79 \rightarrow T70\ 60$

As expected, the definitions of the non-terminals are short and there are limited number of rules in Δ for the model.

We now choose two music sequences (vocal melodic transcriptions) x and y, both of which are composed of about 350 MIDI notes (for consistency and a fair comparison): x from the composer The Beatles ("Let It Be") and y from the composer Elton John ("Your Song") to show that $K(x|G) \ll K(y|G)$. The conditional CFG of x with respect to G has use of

rules from Δ of G as shown below (truncated for easy observation):

$T1 \rightarrow 55\ T91\ T80\ 55\ T90\ T81\ T89\ \ldots\ T79\ T80\ T81 \ldots\ T79\ 62\ T80\ \ldots$
$T80 \rightarrow 64\ 64\ 65\ 64\ T87$
$T81 \rightarrow T79\ T88\ 67\ 67\ T79$
$T82 \rightarrow T86\ T70\ T9\ 60$
$T83 \rightarrow T79\ T88\ 64\ 67\ 67\ T79\ 62\ T89\ 72$
$T84 \rightarrow 55\ 55\ 57$
$T85 \rightarrow 57\ 55\ 64$
$T86 \rightarrow 55\ 55\ 60\ T17\ 64$
$T87 \rightarrow T70\ T70\ T9$
$T88 \rightarrow 64\ 67\ 69$
$T89 \rightarrow T85\ T80$
$T90 \rightarrow T84\ 60\ T82\ T80$
$T91 \rightarrow 55\ T84\ 52\ T82$
$T92 \rightarrow 64\ 64\ T10\ T87$

While on the other hand, the conditional CFG of y with respect to G does not exhibit use of any rule from Δ of G:

$T1 \rightarrow T82\ T82\ T96\ T86\ 56 \ldots\ T91\ 60\ 58\ T82\ T92\ 48\ T93\ T81\ 51$
$T80 \rightarrow 48\ T95\ T93\ T87\ 58\ T89\ T95\ T82\ T87\ T91\ T83\ 63\ T91\ 60\ 58\ 55$
$T81 \rightarrow 51\ 51$
$T82 \rightarrow 55\ 53$
$T83 \rightarrow 58\ 60$
$T84 \rightarrow 51\ 53$
$T85 \rightarrow 53\ T95\ T81$
$T86 \rightarrow T82\ 53\ 58\ 53\ 55$
$T87 \rightarrow 51\ 55$
$T88 \rightarrow T81\ T81$
$T89 \rightarrow T83\ 58\ T82\ T84\ T82\ T95$
$T90 \rightarrow T81\ 53\ 55$
$T91 \rightarrow 60\ T83\ 63\ T83$
$T92 \rightarrow T84\ 48$
$T93 \rightarrow 54\ 53$
$T94 \rightarrow T87\ 58\ 55$
$T95 \rightarrow 51\ 48$
$T96 \rightarrow 55\ 55$
$T97 \rightarrow 55\ 56\ T82$
$T98 \rightarrow T83\ T82\ T92\ T81$

Thus for y, $K(y|G) \approx K(y)$. In this particular experiment, we get $K(x|G) \approx 552$ bits compared to $K(y|G) \approx 1118$ bits, almost twice as $K(x|G)$.

4.8.3 Composer Classification

To extend the above idea to "best-fitting" composer selection for an uncategorized string x, we selected seven composers belonging to the three genres: Bach, Chopin, Haydn, The Beatles, The Rolling Stones, Miles Davis and John Coltrane. m music pieces were randomly selected from the music pieces of each composer where $m = 8, 16, 20$. We then randomly selected 9 music pieces from the music pieces of the seven composers and computed the

fraction of these pieces that were correctly classified by our best-fitting composer selection model. We repeated this process 5 times and averaged the success rate for a fixed β and m.

Number of Models	β	m	Model Selection Success Rate
7	10	8	88%
7	10	16	100%
7	20	16	90%
7	20	20	93%
7	25	20	82%

Table 4.4: Best Fit Model Selection Success Rate

The system did reasonably well in detecting the correct composer for an unknown musical piece x. As m or β increases, the system is more successful in model selection. This happens because the corresponding model G_i becomes in a sense more informed and is able to extract the meaningful information in x more accurately. And understandably, as we restrict $K(G_i)$ to be smaller and smaller, success rate decreases. However it is useful to have the complexity restriction so that G_i does not overfit and have too many redundant rules that correspond to the randomness of individual strings. To compensate for this, we can have G_i represent more strings, that is we can increase m by keeping β fixed to increase the model-selection success rate. However, a more in depth research needs to be done in this respect to investigate how CFG based model selection can have better success rate.

Chapter 5

Future Work

5.1 Improving the runtime of Context Free Grammar Generation

In `CFGGlobalAlgorithm` presented in algorithm 6, at each step we were generating a Patricia tree just to search for patterns and their non-overlapping occurrences. This seems expensive. One immediate solution is to use a Minimal Augmented Suffix Tree to keep track of the non-overlapping occurrences as we build the tree. But instead of building the tree at each step, we can build the tree once and replace patterns with the grammar's new non-terminals in the tree.

Algorithm 15 CFGGlobalAlgorithmImproved

```
1: function Compress(x, algorithm: "Greedy","Most Frequent" and "Longest First")
2:     Grammar : CFG_x ← StartRule → x
3:     moreReductionPossible ← True
4:     NTnumber ← 1
5:     patriciaTree ← buildPatriciaTree(CFG_x)
6:     while moreReductionPossible do
7:         pattern ← PatriciaTree.getPattern(algorithm)
8:         totalReductionRnGrammarSize ← pattern.length × pattern.frequency − pattern.length − pattern.frequency
9:         if totalReductionRnGrammarSize ≤ 0 then
10:            moreReductionPossible ← False
11:        else
12:            CFG_x.replace(pattern, T_NTnumber)
13:            CFG_x.addRule(T_NTnumber → pattern)
14:            patriciaTree.replace(pattern, T_NTnumber)
15:            patriciaTree.maintain()
16:            NTnumber ← NTnumber + 1
17:    return CFG_x
```

The above algorithm requires a lot of changes in the tree at each step, for example: some suffixes would be obsolete, some subtrees would have to collapse because their parents were replaced and new subtrees have to be added. To our knowledge, in-suffix-tree pattern replacement have not been explored extensively, but this is definitely useful in fast generation of context free grammars or other interactions between suffix trees and dynamic compression.

5.2 Generalized Context Free Grammar and Generative Models

Instead of restricting grammars to produce exactly one string, we can think of Generalized Context Free Grammar [60] in which a single non-terminal can have multiple production rules and cyclic dependencies are allowed. Whether these grammars can be generated from a set of string all belonging to the same category and thus can be represented as a model that describes the category is certainly an exciting question. In the realm of music, this

might be used to build generative models that imitate the style of a certain composer. Since generalized CFGs are capable of generating multiple string, we could build such grammar from the musical pieces of a certain composer. This grammar will then hopefully contain all the important characteristic of the composer and can be used to generate musical pieces that preserve those characteristics.
Edit grammars [16] are also interesting: they allow for edit operations like insert, delete and replacement of a character to be performed on a rule. Such grammars can be used to ask the question: if G_1 is the grammar of x and we have another string y, how many edit operations would it take on G_1, so that the modified grammar produces string y. This number of edit operation can be used directly to approximate $K(y|x^*)$.

5.3 Applications in other Low-entropy Strings

We only explored the effectiveness of Kolmogorov Complexity along with Context Free Grammars in music. This exploration can be extended to other artistic realm like literary works of authors, translations of documents in different languages etc. It would also be interesting to see whether CFGs can find structure in properly aligned DNA strings.

Chapter 6

Conclusion

In this project, we used the notion of Context Free Grammar as a description method of strings, used the size of the CFG to approximate the Kolmogorov Complexity and tested whether such combination can be effective in measuring similarity between strings.
In chapter 2, we established the theoretical foundation of Kolmogorov Complexity based Distance Metric. We also discussed the properties of the smallest Context Free Grammars and the existing algorithms to approximate it. In particular, we only focused on offline global algorithms that are very intuitive and achieve better overall approximation ratio on the grammar size.
In chapter 4, we give a detailed explanation of how we built our system and apply it in music, as musical strings are understandably of low-entropy. We estimated pair-wise Kolmogorov Complexity based distance measure between musical strings of various composers and genres and multi-dimensionally scaled the distance matrix that we got. To our satisfaction, the model was able to distinguish between composers, styles and genres. We then tried to use the model for composer classification. For this, we generated constrained model grammar for a corpora of similar strings and tested whether strings belonging to the same corpora compressed well with the model grammar. In this case as well, the model was able to, with high success rate, detect correct composers for given music strings.
Although the system itself is difficult to build, we believe that because Context Free Grammars are capable of finding hierarchical structures in an object, they can prove to be a powerful description method for approximating its Kolmogorov Complexity.

References

[1] Yamaha epiano competition dataset. http://www.piano-e-competition.com/default.asp.

[2] Algorithmics Group, University of Konstanz,Germany. MDSJ: Java library for multidimensional scaling (version 0.2). 2009. http://www.inf.uni-konstanz.de/algo/software/mdsj/.

[3] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, Nov 2000.

[4] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, May 1996.

[5] C. H. Bennett, P. Gacs, Ming Li, P. M. B. Vitanyi, and W. H. Zurek. Information distance. *IEEE Transactions on Information Theory*, 44(4):1407–1423, July 1998.

[6] Charles H. Bennett, Péter Gács, Ming Li, Paul M. B. Vitányi, and Wojciech H. Zurek. Thermodynamics of computation and information distance. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC 93, pages 21-30, New York, NY, USA, 1993. Association for Computing Machinery.

[7] Joe Berkovitz. Noteflight software. https://www.noteflight.com/.

[8] Vincent Berry, David Bryant, Tao Jiang, Paul Kearney, Ming Li, Todd Wareham, and Haoyong Zhang. A practical algorithm for recovering the best supported edges of an evolutionary tree (extended abstract). In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 00, pages 287–296, USA, 2000. Society for Industrial and Applied Mathematics.

[9] Gerth Brodal, Rune Lyngs, Anna stlin, and Christian Pedersen. Solving the string statistics problem in time O(n log n). *International Colloquium on Automata, Languages, and Programming*, 9:728–739, 03 2002.

[10] Rafael Carrascosa, François Coste, Matthias Gallé, and Gabriel G. Infante López. Choosing word occurrences for the smallest grammar problem. In *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*, volume 6031 of *Lecture Notes in Computer Science*, pages 154–165. Springer, 2010.

[11] Rafael Carrascosa, Franois Coste, Matthias Gallé, and Gabriel G. Infante López. Searching for smallest grammars on large sequences and application to DNA. *J. Discrete Algorithms*, 11:62–72, 2012.

[12] Rafael Carrascosa, Franois Coste, Matthias Gall, and Gabriel Infante-Lopez. The smallest grammar problem as constituents choice and minimal grammar parsing. *Algorithms*, 4(4):262–284, Oct 2011.

[13] Gregory J. Chaitin. A theory of program size formally identical to information theory. *J. ACM*, 22(3):329–340, July 1975.

[14] Gregory. J. Chaitin. *Algorithmic Information Theory.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[15] M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, July 2005.

[16] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and Abhi Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 792–801, 01 2002.

[17] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for DNA sequences and its applications in genome comparison. *Genome informatics. Workshop on Genome Informatics*, 10:51–61, 1999.

[18] R. Cilibrasi and P. M. B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[19] Rudi Cilibrasi, Paul M. B. Vitányi, and Ronald de Wolf. Algorithmic clustering of music. *Proceedings of the Fourth International Conference on Web Delivering of Music, 2004. EDELMUSIC 2004.*, pages 110–117, 2003.

[20] Rudi Cilibrasi, Paul M. B. Vitányi, and Ronald de Wolf. Algorithmic clustering of music based on string compression. *Computer Music Journal*, 28:49–67, 2004.

[21] Trevor De Clercq and David Temperley. Rock corpus. http://rockcorpus.midside.com/index.html.

[22] James B. Coover and John C. Franklin. *Dictionaries and encyclopedias of music.* Oxford University Press, 2001.

[23] Trevor de Clercq and David Temperley. A corpus analysis of rock harmony. *Popular Music*, 30(1):47–70, 2011.

[24] Jeffrey Ens and Philippe Pasquier. Statistical test for comparing two corpora. https://goo.gl/ejN1RM.

[25] Jeffrey Ens and Philippe Pasquier. CAEMSI : A cross-domain analytic evaluation methodology for style imitation. In *Proceedings of the Ninth International Conference on Computational Creativity, Salamanca, Spain, June 25-29, 2018*, pages 64–71. Association for Computational Creativity (ACC), 2018.

[26] Jeffrey Ens and Philippe Pasquier. Personal Communication, 2018.

[27] Franco Fabbri. Browsing music spaces: Categories and the musical mind. *In Proceedings of the IASPM (UK) Conference*, 1999. https://www.tagg.org/xpdfs/ffabbri990717.pdf.

[28] Péter Gács. On the symmetry of algorithmic information. *Dokl. Akad. Nauk SSSR*, 218:1265–1267, 1974.

[29] Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Generating music by fine-tuning recurrent neural networks with reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*, 2016.

[30] J. C. Kieffer and En-Hui Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, May 2000.

[31] Jetse Koopmans, Daan van den Berg, and Vadim Zaytsev. Similarity, Data Compression and a Dead Composer. In *Proceedings of the Sixth Student Research Conference (SRC)*, pages 37–40. ScienceWorks, 2015.

[32] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096)*, pages 296–305, March 1999.

[33] Eric Lehman and Abhi Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 02, pages 205–212, USA, 2002. Society for Industrial and Applied Mathematics.

[34] Fred Lerdahl and Ray Jackendoff. An overview of hierarchical structure in music. *Music Perception: An Interdisciplinary Journal*, 1(2):229–252, 1983.

[35] Ming Li. Information distance and its applications. *International Journal of Foundations of Computer Science*, 18(04):669–681, 2007.

[36] Ming Li, Jonathan H. Badger, Xin Chen, Sam Kwong, Paul E. Kearney, and Haoyong Zhang. An information based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17:149–154, 11 2000.

[37] Ming Li, Xin Chen, Xin Li, Bin Ma, and P. M.B. Vitanyi. The Similarity Metric. *IEEE Trans. Inf. Theor.*, 50(12):3250–3264, December 2004.

[38] Ming Li and M. Ronan Sleep. Melody classification using a similarity metric based on Kolmogorov Complexity. *Proceedings of the Sound and Music Computing Conference (SMC'04)*, pages 126–129, October 2004.

[39] Ming Li and Paul M. B. Vitányi. Reversibility and adiabatic computation: trading time and space for energy. volume 452, pages 769 – 789, 1996.

[40] Ming Li and Paul M.B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[41] Corentin Louboutin and David Meredith. Using general-purpose compression algorithms for music analysis. *Journal of New Music Research*, 45:1–16, 02 2016.

[42] Cory McKay and Ichiro Fujinaga. Musical genre classification: Is it worth pursuing and how can it be improved? In *Proceedings of the 7th International Conference on Music Information Retrieval*, pages 101–106. ISMIR, Oct 2006.

[43] David Meredith. Omnisia: Siatec-based pattern discovery algorithms. http://www.titanmusic.com/software.php.

[44] David Meredith. COSIATEC and SIATECCompress: Pattern discovery by geometric compression. In *Music Information Retrieval Evaluation eXchange (MIREX 2013)*. International Society for Music Information Retrieval, 2013.

[45] David Meredith. Music analysis and point-set compression. *Journal of New Music Research*, 44(3):245–270, 9 2015.

[46] David Meredith. Analysing music with point-set compression algorithms. In *Computational Music Analysis*, pages 335–366, Cham, 2016. Springer International Publishing.

[47] David Meredith. Personal Communication, 2019.

[48] David Meredith, Kjell Lemström, and Geraint Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002. Paper id:: 10.1076/jnmr.31.4.321.14162.

[49] Kornel Michaowski and Jim Samson. *Chopin, Fryderyk Franciszek*. Oxford University Press, 2001.

[50] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, September 1997.

[51] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[52] Martin Pfleiderer, Klaus Frieler, Jakob Abeßer, Wolf-Georg Zaddach, and Benjamin Burkhart. The jazzomat research project. University of Music Franz Liszt, Weimar. https://jazzomat.hfm-weimar.de/index.html.

[53] Martin Pfleiderer, Klaus Frieler, Jakob Abeßer, Wolf-Georg Zaddach, and Benjamin Burkhart, editors. *Inside the Jazzomat - New Perspectives for Jazz Research*. Schott Campus, 2017.

[54] RollingStone. 500 greatest songs of all time. http://web.archive.org/web/20080622145429/www.rollingstone.com/news/coverstory/500songs.

[55] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, December 2007.

[56] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA, 2014.

[57] Kirill A Sidorov, Andrew Jones, and A David Marshall. Music analysis as a smallest grammar problem. In *Proceedings of the Fifteenth International Society for Music Information Retrieval Conference (ISMIR)*, pages 301–306, 2014.

[58] Ian Simon and Sageev Oore. Magenta Blog on Performance RNN. https://magenta.tensorflow.org/performance-rnn.

[59] Ian Simon and Sageev Oore. Performance RNN: Generating music with expressive timing and dynamics. https://magenta.tensorflow.org/performance-rnn, 2017.

[60] Payam Siyari and Matthias Gall. The generalized smallest grammar problem. In *Proceedings of The 13th International Conference on Grammatical Inference*, volume 57 of *Proceedings of Machine Learning Research*, pages 79–92, Delft, The Netherlands, 05–07 Oct 2017. PMLR.

[61] Daniel Sleator and David Temperley. The Melisma Music Analyzer. http://www.link.cs.cmu.edu/music-analysis/.

[62] A. Takamoto, M. Umemura, M. Yoshida, and K. Umemura. Improving compression based dissimilarity measure for music score analysis. In *2016 International Conference On Advanced Informatics: Concepts, Theory And Application (ICAICTA)*, pages 1–5, Aug 2016.

[63] Sebastiaan A. Terwijn, Leen Torenvliet, and Paul M.B. Vitnyi. Nonapproximability of the normalized information distance. *Journal of Computer and System Sciences*, 77(4):738 – 742, 2011. JCSS IEEE AINA 2009.

[64] N. K. Vereshchagin and P. M.B. Vitanyi. Kolmogorov's Structure Functions and Model Selection. *IEEE Trans. Inf. Theor.*, 50(12):3265–3290, December 2004.

[65] P. M. Vitanyi. Meaningful information. *IEEE Transactions on Information Theory*, 52(10):4617–4626, Oct 2006.

[66] P. M. B. Vitanyi and Ming Li. Minimum description length induction, Bayesianism, and Kolmogorov complexity. *IEEE Transactions on Information Theory*, 46(2):446–464, March 2000.

[67] Paul M. B. Vitányi, Frank J. Balbach, Rudi L. Cilibrasi, and Ming Li. Normalized information distance. In *Information Theory and Statistical Learning*, pages 45–82, Boston, MA, 2009. Springer US.

[68] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sep. 1978.

[69] Wojciech H. Zurek. Thermodynamic cost of computation, algorithmic complexity and the information metric. *Nature*, 341:119–124, 1989.

APPENDICES

Appendix A

Auxiliary Proofs

Theorem. *Let x be a binary strings of length n. The number of binary strings y with $E(x,y) \leq d$, $N(d,x)$ satisfies up to a constant additive error term $O(1)$:*

$$\log N(d,x) = d - K(d|x^*) \tag{A.1}$$

Proof. First we prove a useful property of $E(x,y)$. For each x:

$$\begin{aligned}\sum_{y:y\neq x} 2^{-E(x,y)} &\leq \sum_{y:y\neq x} 2^{-K(y|x^*)-K(x|y^*)} \\ &\leq \sum_{y:y\neq x} 2^{-K(y|x^*)} \\ &\leq 1\end{aligned}$$

The first inequality is due to the definition of $E(x,y)$. The second inequality is simply for the fact that $-K(y|x^*) - K(x|y^*) \leq -K(y|x^*)$. The last inequality follows from the fact that: the sum $\sum_y 2^{-K(y|x^*)}$ is taken over the programs p for which the universal prefix machine, computes y given x^*. This sum is the probability that U, given x^*, computes y from a program p which is generated bit by bit uniformly at random. Hence the sum is at most 1 [40].
Now we move on to proving equation (A.1).

$\log N(d,x) < d - K(d|x^*)$: For every binary string x, and a positive distance d, the probability of choosing a y such that $E(x,y) \leq d$ is $\frac{N(d,x)}{2^d}$. We want to consider all such

y's as d ranges from 1 to ∞:

$$\sum_{d=1}^{\infty} \frac{N(d,x)}{2^d} = \sum_{d=1}^{\infty} \frac{\sum_{i=1}^{d} n(i,x)}{2^d}$$
$$= \sum_{d=1}^{\infty} \frac{\sum_{i=1}^{d} \frac{2^i n(i,x)}{2^i}}{2^d}$$

Now, $\frac{n(i,x)}{2^i}$ is the probability of choosing a $y \neq x$ such that $E(x,y) = i$ so:

$$\sum_{d=1}^{\infty} \frac{N(d,x)}{2^d} = \sum_{d=1}^{\infty} \sum_{i=1}^{d} 2^{-d+i} \sum_{y:y\neq x, E(x,y)=i} 2^{-E(x,y)}$$

Setting $d - i = j$:

$$\sum_{d=1}^{\infty} \frac{N(d,x)}{2^d} \leq \sum_{j=1}^{\infty} 2^{-j} \sum_{y:y\neq x} 2^{-E(x,y)}$$
$$\leq 1$$

Denote $f(d,x) = \log(\frac{2^d}{N(d,x)})$.
Function $f(d,x)$ is upper-semicomputable if $N(d,x)$ is upper-semicomputable because 2^d is computable. $N(d,x)$ is upper-semicomputable if we are willing to provide the information $x^* : |x^*| = K(x)$ and $y^* : |y^*| = K(y)$ (upper-semicomputability of $N(d,x)$ depends on $E(x,y)$'s upper-semicomputability and $E(x,y)$ is only upper-semicomputable if we provide these information). Then, $N(d,x)$ and in turn $f(d,x)$ becomes upper-semicomputable. It also satisfies $\sum_{d=1}^{\infty} 2^{-f(x,y)} \leq 1$. Now, since, $K(d|x^*)$ is also an upper-semicomputable function and minorizes every other upper-semicomputable function by definition [5], we have $K(d|x^*) \leq f(d,x) = \log(\frac{2^d}{N(d,x)}) = d - \log N(d,x)$

$\log N(d,x) > d - K(d|x^*)$: Consider strings y of the form px where p is a self-delimiting program [5]. For all such programs, $K(x|y^*) = O(1)$. This is possible as we provide the information $y^* : |y^*| = K(y)$
Hence,

$$E(x,y) = K(x|y^*) + K(y|x^*)$$
$$= O(1) + K(px|x^*)$$
$$= K(p|x^*) + O(1) \leq d$$

Recapitulating a previous prefix property of K complexity:

$$\sum_{p:p\neq x} 2^{-(d)} \leq \sum_{p:p\neq x} 2^{-K(p|x^*)} \leq 1$$

So there are atmost 2^d or atleast $2^{d-K(d|x^*))}$ such strings p with $K(p|x^*) \leq d$, yielding $\log N(d,x) > d - K(d|x^*)$.

$\square$

Appendix B

Processing Strings

B.1 Suffix Tree

B.1.1 Definition

The suffix tree of a text $x[1..n]$ is a compressed trie built on all suffixes of x [55]. It has n leaves, each of which corresponds to a suffix of x. The total number of nodes in such a compressed trie is also bounded by $O(n)$. Each edge is labeled by a string, called edge-label. The concatenation of labels on a path from the root to a node is called the path-label of the node and it's length is called the prefix-length. The path-label of each leaf coincides with a suffix. The path-label for each non-leaf internal node depicts the prefix of all the suffixes (leaves) of the sub-tree rooted at that node. For each internal node, the edges to its children are sorted in the alphabetic order of the first characters of edge-labels. For implementation purposes, this can be done by keeping a dictionary of character-edge pairs at each node. Figure B.1 shows the suffix tree for a string “banana$”.

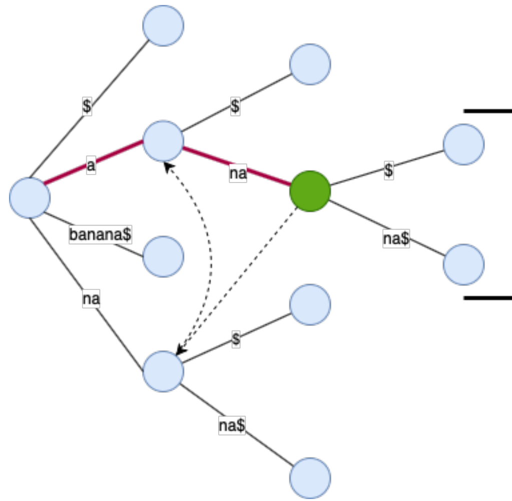


Figure B.1: Demonstration of suffix tree of the string "banana". The pattern "ana" is found ending at a node which has two leaves in the subtree below it. So "ana" occurs twice in the string "banana".

B.1.2 Pattern Searching

Any pattern $P[0 \ldots m-1]$ in the string x will be a prefix of some suffix (see Figure B.1). Hence, to determine whether or not the pattern occurs in the string, we need to find the end of the path from the root with the path-label $P[0 \ldots m-1]$. If no such path exists (there is a mismatch between an edge-label and a substring of P), then the pattern does not occur.

B.1.2.1 Pattern Occurrences

To find the occurrences of an existing pattern P in the string, we must find the path as above and then subtract length of the suffixes that have P as their prefixes from the total string length $x.length$ to find the individual occurrences of P in x:

Algorithm 16 Finding Pattern Occurences

1: **function** FPO(Pattern:$P[i \dots m-1]$, Root of subtree: $root$)
2: $edge \leftarrow root.getEdge(P[i])$
3: **if** $edge = null$ **then** ▷ There is no edge with the first character, so pattern doesn't exist
4: **return** null
5: **if** $edge.label.length \geq m - i$ **then**
6: **if** $edge.label[0 \dots m-i] = P[i \dots m-1]$ **then** ▷ We found the pattern. Now get the occurrences
7: **return** $edge.endNode.occurrences$
8: **else**
9: **return** $null$
10: **else**
11: **if** $edge.label = P[i \dots i + edge.label.length]$ **then**
12: **return** FPO($P[i + edge.label.length + 1 \dots m-1]$, $edge.endNode$)
13: **else**
14: **return** $null$
15:

Algorithm 17 Node Class: Finding the Positions of occurrences

1: **function** setOccurences(prefixLength)
2: $this.occurrences \leftarrow []$
3: **if** this.isLeaf **then**
4: this.occurrences.add($x.length$-prefixLength)
5: **else**
6: **for** each $edge$: this **do**
7: $edge.endNode$.setOccurrences($edge.label.length + prefixLength$)
8: $this.occurrences.sortedMerge(edge.endNode.occurrences)$

We are keeping the list of occurrences sorted in ascending order of their indices, so that identify the overlapping occurrences in this list in $O(n)$ time.

B.2 Generalized Suffix Tree or Patricia Tree

B.2.1 Pattern Searching and Counting Occurrences

Pattern searching in a generalized suffix tree is same as in a normal suffix tree. To find out in which string α_i a pattern occurs, we have to just look at the sentinel character $\$_i$ the suffix ends with and to find the occurrence index of the pattern, in the same way as above, we have subtract the prefix length or path label of the leaf from the length of that string α_i.
If we need to count how many times a pattern occurs in all of the strings, the procedure is same as a single-string suffix tree. The number of leaves in the sub-tree that is rooted at the node which has the pattern as its prefix, will correspond to the number of occurrences of the pattern.

Appendix C

Full List of Music Pieces Used in the Project

C.1 Rock Pieces

The complete list of the Rock songs used in this thesis can be found in the RS200 project of DeClercq and Temperley [21]. However, since they only provide vocal melodic transcription, for instrumental transcriptions of the music pieces from the composers: the Beatles, the Rolling Stones and Nirvana, we obtained MIDI files from the following websites which provide freely accessible contents:

1. http://beatlesnumber9.com
2. https://www.midiworld.com

C.2 Classical Pieces

Table C.1: Classical Pieces Used

Song	Artist
Mazurka Op17No4	Chopin
Nocturnes Op62No1	Chopin
Mazurka Op33No4	Chopin

Nocturnes Op37No1	Chopin
Nocturnes Op62No2	Chopin
Nocturnes Op9No2	Chopin
Nocturnes Op37No2	Chopin
Mazurka Op17No2	Chopin
Nocturnes Op27No2	Chopin
Mazurka Op33No2	Chopin
Nocturnes Op48No2	Chopin
Mazurka Op17No3	Chopin
Mazurka Op17No1	Chopin
Nocturnes Op27No1	Chopin
Mazurka Op6No4	Chopin
Nocturnes Op15No3	Chopin
Nocturnes Op15No2	Chopin
Nocturnes Op15No1	Chopin
Mazurka Op6No2	Chopin
Mazurka Op6No3	Chopin
Mazurka Op6No1	Chopin
Nocturnes Op55No2	Chopin
Mazurka Op30No2	Chopin
Nocturnes Op55No1	Chopin
Mazurka Op7No5	Chopin
Mazurka Op7No4	Chopin
Nocturnes Op32No2	Chopin
Mazurka Op7No1	Chopin
Mazurka Op7No3	Chopin
Nocturnes Op32No1	Chopin
Mazurka Op7No2	Chopin
Prelude No 1	Chopin
Prelude No 2	Chopin
Prelude No 6	Chopin
Prelude No 7	Chopin
4 Seasons	Tchaikovsky
Serious	Schumann
Pleading	Schumann
Piano Sonata 14 Movement 2	Beethoven

Etudes No 1	Chopin
Etudes No 2	Chopin
Etudes No 3	Chopin
Etudes No 4	Chopin
Etudes No 5	Chopin
Etudes No 6	Chopin
Etudes No 7	Chopin
Etudes No 8	Chopin
Etudes No 9	Chopin
Etudes No 10	Chopin
Etudes No 11	Chopin
Etudes No 12	Chopin
Invention No 1	Bach
Invention No 2	Bach
Invention No 3	Bach
Invention No 4	Bach
Invention No 5	Bach
Invention No 6	Bach
Invention No 7	Bach
Invention No 8	Bach
Violin Sonata 1 Movement 1	Bach
Violin Sonata 1 Movement 3	Bach
Violin Sonata 2 Movement 1	Bach
Violin Sonata 2 Movement 2	Bach
Violin Sonata 2 Movement 3	Bach
Violin Sonata 2 Movement 4	Bach
Violin Sonata 3 Movement 1	Bach
Violin Sonata 3 Movement 2	Bach
Violin Sonata 3 Movement 3	Bach
Violin Sonata 3 Movement 4	Bach
Well-Tempered Clavier 2 Fugue 1	Bach
Well-Tempered Clavier 2 Fugue 2	Bach
Well-Tempered Clavier 2 Prelude 1	Bach
Well-Tempered Clavier 2 Prelude 2	Bach
Well-Tempered Clavier 1 Prelude 2	Bach
Well-Tempered Clavier 1 Prelude 3	Bach

PSHoboken8 Movement 2	Haydn
PSHoboken8 Movement 3	Haydn
PSHoboken8 Movement 4	Haydn
Prelude No 3	Chopin
Prelude No 4	Chopin
Prelude No 5	Chopin
Prelude No 9	Chopin
Well-Tempered Clavier 2 Prelude 3	Bach
Well-Tempered Clavier 2 Fugue 3	Bach
Well-Tempered Clavier 1 Fugue 1	Bach
Well-Tempered Clavier 1 Fugue 2	Bach
Piano Sonata 16 Movement 3	Mozart
Piano Sonata 12 Movement 2	Mozart
Prelude No 10	Chopin
Prelude No 11	Chopin
Prelude No 14	Chopin
Prelude No 18	Chopin
Prelude No 20	Chopin
Prelude No 22	Chopin
Prelude No 23	Chopin
Piano Sonata 21 Movement 2	Beethoven
Piano Sonata 26 Movement 2	Beethoven
PSHoboken7 Movement 1	Haydn
PSHoboken7 Movement 2	Haydn
PSHoboken7 Movement 3	Haydn
PSHoboken9 Movement 3	Haydn
Well-Tempered Clavier 1 Fugue 3	Bach
Piano Sonata 8 Movement 2	Mozart
Piano Sonata 14 Movement 1	Beethoven
Fur Elise	Beethoven

The MIDI files of the music pieces in the classical genre were obtained from the following websites which provide free access to their content:

1. https://www.classicalarchives.com/midi/
2. https://www.midiworld.com
3. http://www.piano-midi.de
4. http://www.kunstderfuge.co
5. http://www.bachcentral.com

C.3 Jazz Pieces

Table C.2: Jazz Pieces Used

Song	Artist
Giant Steps	John Coltrane
Blue Train	John Coltrane
Impression	John Coltrane
Nature Boy	John Coltrane
Soultrane	John Coltrane
Bessies Blues	John Coltrane
Blues By Five	John Coltrane
Body And Soul	John Coltrane
My Favorite Things	John Coltrane
Nutty	John Coltrane
Oleo	John Coltrane
Agitation	Miles Davis
So What	Miles Davis
Blues by Five	Miles Davis
Dolores	Miles Davis
Tune Up	Miles Davis
Airegin	Miles Davis
Bitches Brew	Miles Davis
ESP	Miles Davis
Orbits	Miles Davis
Vierd Blues	Miles Davis
Anthropology	Dizzy Gillespie

Cognac Blues	Dizzy Gillespie
Hot House	Dizzy Gillespie
Groovin High	Dizzy Gillespie
I Fall in Love Too Easily	Chet Baker
Just Friends	Chet Baker
Long Ago and Far Away	Chet Baker
There Will Never Be Another You	Chet Baker
Blues for Alice	Charlie Parker
Donna Lee	Charlie Parker
Embraceable You	Charlie Parker
How Deep Is the Ocean	Charlie Parker
Ornithology	Charlie Parker
Out of Nowhere	Charlie Parker
Yardbird Suite	Charlie Parker

The MIDI files for the jazz genre were obtained from the Weimar Jazz Database [53] which is a collection of *jazz solo transcriptions* produced by the Jazzomat Research Project of University of Music Franz Liszt, Weimar [52].