I Knew I Was in Trouble: A Simulation-Based Approach to Recreating Taylor's Hits

Alex Wang[1]
[1]Department of Taylology, Faculty of Swiftness,
University of _____

March 16, 2024

1 Introduction

One of my favorite hobbies is singing (I'm in the Stanford Mendicants!) and playing guitar (I perform at CoHo in my free-time, come watch!). As such I've been inspired to talk about something at the intersection of these interests: Acoustic Songs. As of recent, due to popular demand from my dorm-mates and friends (Look what you made me do...) I've found myself playing more and more Taylor Swift. This is where this idea came from! Inspired initially by the Infinite Monkey Theorem, a thought experiment on randomness, but applied in the Shakespeare of our generation: Taylor Swift. Today, we seek to answer a question that millions of girls across America are asking amongst themselves:

How long would it take for me to become Taylor Swift?

For this problem, we need some clear parameters. "Becoming Taylor Swift" in this case would to write her 15 most popular songs, and all of her No. 1 hits as defined by the *Billboard Hot 100* scale. For more information about which songs were included, check the appendix. However, for brevity's sake it will not be included here. Further, to make sure that we are sufficiently Stanford-oriented, this group of teens will now be musically-talented Stanford Students, however, sadly they don't have perfect pitch and are subject to constraints of using their well-honed relative pitch listening skills. But, because they are Stanford students, they have developed a strategy!

The sampling mechanism is guided by a few parameters for this hypothetical situation. The imagined situation will be as such:

A human who is knowledgeable about all the guitar chords (or can easily learn these guitar chords), will play a series of guitar chords in the length of her song (3:57 minutes on average with a standard deviation of 37 seconds). We want to estimate the total amount of time it would take to create all of Taylor's No.1 hits and assortment of other famous songs with sampling methods. But, we're smarter than monkeys (I think), so we definitely won't be randomly sampling.

2 Chords

An obvious place to begin would be chord progressions for her music. Luckily for her fans, most of her music can be played on guitar with just 4 chords! To increase the likelihood that we will find the chords that we are looking for, we will not sample randomly from the nearly infinite number of chords that exist (especially if we begin to include diminished, augmented, inversions and more!), but from the most popular chords in a dataset of chord analysis from the most popular chords in the most popular songs.

For simplicity sake, I began by testing a simulation-based system for only a single chord progression. Table 1 illustrates a sample chord progression for her song "Delicate". Representative of many of her other songs, the same chord progressions are repeated throughout the song with potentially a small change present in the last or second to last bridge.

However, something interesting to note for all my Swifties out there is that, as a general rule, Taylor repeats many of her favorite chord progressions throughout a majority of her songs, even within the same album. When we account for this, she uses her most popular chord progression 21 times (I V vi IV), the second 20 times (I V ii IV), the third 17 times (I vi IV V), and her fourth (vi IV I V) and fifth (IV I V vi) most popular chord progressions show up 9 times respectively. This is great news for our Swifties to expedite the song discovery process.

Table 1: Structure of 'Delicate' by Taylor Swift

Part	Artist	Chord Progression
Verse	Taylor Swift	I-ii-vi-IV
Chorus	Taylor Swift	I-ii-vi-IV
Post-Chorus	Taylor Swift	I-ii-vi-IV
Bridge	Taylor Swift	iii-vi-V
Outro	Taylor Swift	I-ii-vi-IV

3 Genetic Algorithms

A genetic algorithm (GA) was chosen as the optimizing sampling mechanism because (it seemed cool) the problem can likely be simplified into a sampling/optimization problem from a 5m-dimensional subspace (where m is the number of songs): the first dimension is the key that was sought after, the other four have each dimension as a chord progression). Key features of the GA used in this project were a function for calculating the population fitness, a mechanism for selecting the mating pool, crossover of "chords", and a mechanism for mutation.

Structure of the genes were thus as follows: an alphabetical character from A - X and a section of four genes representing the four chosen chords from that key. For the first gene, or the alphabetical character A-X, represents how there are 12 notes in western music with each having both a major and minor key. Although not intuitive to non-musicians, the 1^{st} letter A represents the C major, the letter B represents C#/D major, C represents D major, D represents D#/ E major, and so on. Then starting on the 13^{th} letter or the letter M represents a A minor key, the letter N represents a A/Bminor and so on. Then for the last four genes, I assigned each of the 12 primary chords, a value from 1-12, though the music theory experts amongst us may only acknowledge that i also included the non diatonic chords. Thus, the tonic or I would be 1, the subdominant or IV is 4, and the dominant or V is 5 for example, for the non-diatonics which follow after the vii° or 7 are labeled beginning with the bII (Neopolitan in the key of C) as 9 and so on. Then the genes of the GA could sample from the discrete set $n \in N$ and $1 \leq n \leq 12$. While this may be an oversimplification of the nuances of music theory chord progressions, this allows actual calculations to be much more stable, removing to develop cumbersome and extensive edge cases.

Thus a sample gene would should be of the form A-1-3-2-7, and would represent a chord progression in the key of C major beginning with the I or the tonic or C major chord, followed by the iii chord (E minor), ii chord (D minor), and vii° chord which would be a B diminished chord.

Now for the optimization. To demonstrate this, we will take "Delicate" to be the example. Delicate is in the key of A minor and has a primary chord progression of "I-ii-vi-IV" so it would represented as M-1-2-6-4 (portrayed as a vector to be optimized for as [13, 1, 2, 6, 4]). For the optimization goal or fitness equation, we take the negative squared value of the "distance" measured as how far the chord is on the scale, from these desired chords to represent their fitness respective goal. The fitness function is then defined thereafter.

Given that m represents the number of songs selected, both C_i and X_i are components of a $5m$-dimensional row vector, where is the desired key and chord progression while is a sample gene. This can be represented as follows:

$$\vec{C} = \begin{bmatrix} C_1 & C_2 & \cdots & C_{5m} \end{bmatrix} \quad \text{and} \quad \vec{X} = \begin{bmatrix} X_1 & X_2 & \cdots & X_{5m} \end{bmatrix}$$

where $\vec{C}$ and $\vec{X}$ are row vectors in R^{5m}.

Let X_i be a chord in the gene sequence, and C_i be the corresponding desired chord. The fitness F which is subject to maximization can be calculated as the negative sum of the squared error/distances of the chords from the desired chords:

$$F = -\sum_{i=1}^{n}(X_i - C_i)^2$$

where:

- x_i is the observed chord at position i in the gene sequence, c_i is the desired chord at position i, n is the number of chords in the gene sequence.

For the example of "Delicate", optimization with the genetic algorithm began with a matrix of shape 20 by 5 with each row or 5-dimensional vector representing a random combination of numbers with the first column sample from a range of 1 through 24, representing the potential keys or the letters A-X respectively, and the last four columns sampled randomly from a range of 1-12, representing the twelve possible chords for the key (code is listed in Appendix Section 6.3 in Population Creation and Optimization Parameters). In other words, each Stanford student selects a random key, and then plays a random progression of four chords following that key.

Next, the top 5 Stanford Students, or those with the greatest fitness functions, were selected to exchange chords or keys. Genetically, this is called crossover. For each generation, the "best" or closest result was taken for a measure of progression. However, as these Stanford students are CS majors, they are not that great at communication, so their "offspring" or their next generation or group of attempts always has a single mistake in a arbitrary location. By this logic, each generation or attempt takes 3.57*(# of song) minutes. This is where mutation and randomness is introduced in the gene pool where an arbitrary index has is instead replaced by a random number sampled uniformly from a range of 1-12 Finally, the new population is then generated and the process repeats for another generation or set of attempts to recreate the Taylor Songs. The control sequence and functions are in Appendix 6.2.2 and Appendix 6.2.3 for brevity's sake.

Thus, when we take this approach for a GA to create an optimal sequence to eventually converge for the song "Delicate" we arrive at Figure 1 as the fitness of the function over time and the generation or the attempt time. As there are drastic modifications in fitness of the best result in each generation, the convergence is working! For a simple visualization of the movement of these genes within higher-dimensional space, Uniform Manifold Approximation Projection (UMAP) was chosen for its ability to balance grouping local structure and maintaining global-structure. Why not other techniques? Check Appendix 6.4. In Figure 2 the UMAP projection of the initial population (in yellow) and the final population (in dark blue) are dimensionally-reduced from 5-dimensional space to 2-dimensional space.

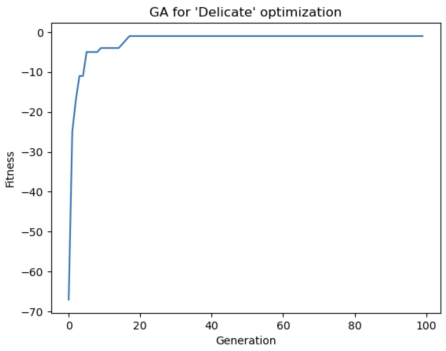

Figure 1: Graph for the optimal solution obtained for "Delicate" by a GA with the following parameters: population size of 20, mating pool of 5, and 1 mutation per offspring.

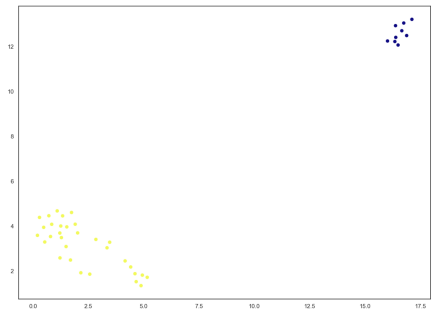

Figure 2: UMAP projection from five to two dimensions illustrating of the movement of the populations with yellow representing the initial sample population and the black represents the population after 100 generations.

4 Modeling Taylor's Hits

For the sequence of Taylor's 15 "hit" songs for the calculation, we create a 5(15)-dimensional or 75-dimensional vector that we seek to converge for. With the same population parameters from before, where we have a group of 20 Stanford students beginning off by randomly choosing 15 keys and 15 chords. As expected, UMAP projections of this present that this is system is distributed relatively uniformly randomly in the 75-dimensional subspace, when projected down to 2-dimensions represented in the Appendix 5.2 Figure 6. However, similar to before, a convergence towards the "optimal" solution occurs once more illustrated in Figure 4 and we are thus able to reach the solution that we are looking for in about 600 generations, with each generation taking 59.55 minutes, illustrated in Figure 3. This curve is unsurprisingly similar to an exponential as the parameters of the exponential are quite similar to a genetic algorithm.

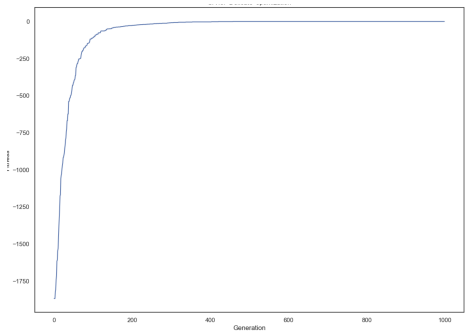

Figure 3: Graph for the optimal solution obtained for all 15 "hit" songs by a GA with the following parameters: population size of 20, mating pool of 5, and 1 mutation per offspring.

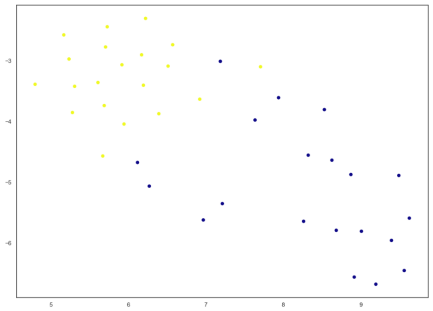

Figure 4: UMAP projection from 75 (5*15) to 2 dimensions illustrating of the movement of the populations with yellow representing the initial sample population and the black represents the population after 100 0generations.

However, in order to truly determine how this variable is distributed, a frequentist mindset was followed. The GA with parameters of 20 students, 1 "mutation" or changed chord/key per following generation, 25% selective "breeding" or exchange, and crossover at the middle position, was sampled for 2000 times to see how the expected value for this system. The resulting distribution was best classified as log-normal illustrated in Figure 6.

From this, I learned that GAs tend to exhibit log-normal distributions of fitness values due to the multiplicative growth that arises from selection and reproduction dynamics, preferential attachment mechanisms, the central limit theorem applied to multiplicative processes, and the interaction of heterogeneous genetic variability. In GAs, beneficial traits can combine multiplicatively across generations, akin to compound interest accumulating small advantages exponentially over time. The selection process favors fitter individuals to contribute more offspring, potentially skewing the distribution as a few achieve very high fitness through preferential accumulation of advantages, analogous to "the rich get richer" in social networks. Furthermore, when many independent multiplicative factors influence an individual's growth rate, the logarithms of final fitness values tend towards a normal distribution by the central limit theorem, implying the original fitness is log-normally distributed. Finally, the diverse genetic variability introduced through crossover and mutation interacts with selection in a way that can compound advantages, also contributing to this characteristic log-normal shape.

The following were the values that were eventually calculated assuming a log-normal distribution from this data with the code in the Appendix:

Mean (Log-Normal): 599.7491
Mode (Log-Normal): 565.5729
Standard Deviation (Log-Normal): 23.9239

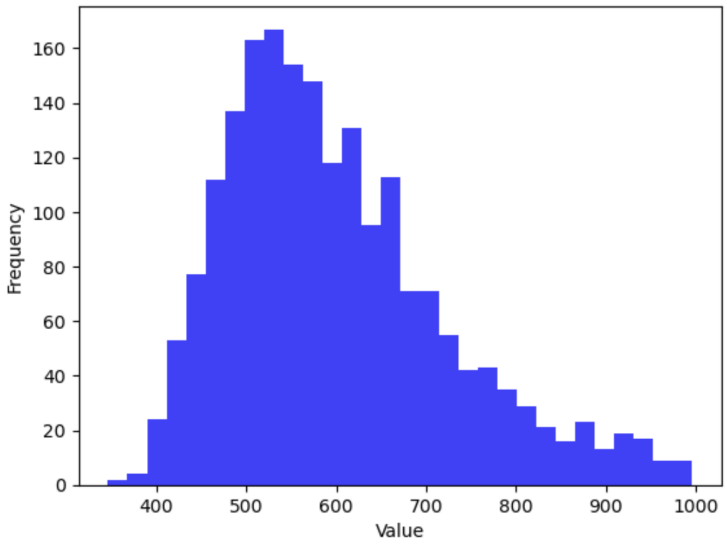

Figure 5: Resulting distribution following 2,000 sampling events from the GA for predicting how long it would take for the first student to sample all of Taylor Swift's hits.

5 So You Wanna be Taylor?

If your **wildest dreams** are to become Taylor Swift, you're in luck, I have an answer for you. Assuming the following parameters of course:

1. Your group of friends has 20 people.
2. You start off playing completely random chords in a random key (*random initialization*).
3. You learn how to play all chords on the guitar.
4. The top 25% of friends share their results and exchange with each other (*Crossover/Mating*).
5. You are good, but not perfect at communicating what you just played so there are mistakes during communication (*Mutations*).

With this optimized strategy, it should take, on average, **35535 minutes** or **592.25 hours** or **24.677 days** continuously for one person to become Taylor Swift (they can share afterwards!).

At the end of the day. *Shake it off*! You'll (probably) be fine even if you don't become Taylor.

6 Appendix 1

6.1 Chosen Songs, Release Date, and Associated Chord Progression

For this, the following songs were considered in order of time spent on the *Billboard Hot 100* followed by their release date:
1. Anti-Hero (11/05/22) - F# major, I(F#) - V(C#) - Vi(D#m) - IV (B): G-1-5-6-4
2. Shake it Off (09/06/14) - G major, I (G) - V (D) - vi (Em) - IV (C): H-1-5-6-4
3. Cruel Summer (09/07/19) - C# minor, (C#m) - IV (F#) - vi (Am) - V (G#): S-1-4-6-5
4. Blank Space (11/15/14) - F major, I (F) - vi (Dm) - ii (Gm) - V (C): F-1-6-2-5
5. Bad Blood (11/15/14) - G major, vi (Em) - V (D) - I (G) - IV (C): H-6-5-1-4
6. We are Never Ever Getting Back Together (08/25/12) - G major, I (G) - V (D) - vi (Em) - IV (C): H-1-5-6-4
7. Look What You Made Me Do (09/09/17) - A minor, vi (Am) - V (G) - IV (F) - I (C): M-6-5-4-1
8. Willow (12/26/20) - E minor, vi (C) - IV (A) - I (E) - V (B): Q-6-4-1-5
9. All Too Well (Taylor's Version: 11/27/21) - C major, I (C) - V (G) - vi (Am) - IV (F): A-1-5-6-4
10. Is it Over Now? (Taylor's Version: 11/11/23) - C major, I (C) - IV (F) - vi (Am) - IV (F): A-1-4-6-4
11. Cardigan (08/08/20) - Bb minor, i (Bbm) - VI (Gb) - III (Db) - v (Fm): X-1-6-3-5

For completeness, other songs were also added that are classics. Although they did not hit No. 1 status, it was only due to being overtaken by their fellow hits.

12. Delicate - A minor, I (A) -ii (Bm) - vi (Fm) - IV (D): M-1-2-6-4
13. You Belong With Me - F# major, I (F) - IV (B) - vi (Dm) - IV: G-1-4-6-4
14. Love Story - D major, I (D) - V (A) - vi (Bm) - IV (G): C-1-5-6-4
15. I Knew You Were Trouble - C major, i (C#m), II (D) - III (E) - iv (F#m): A-7-2-3-11

6.2 Additional Figures

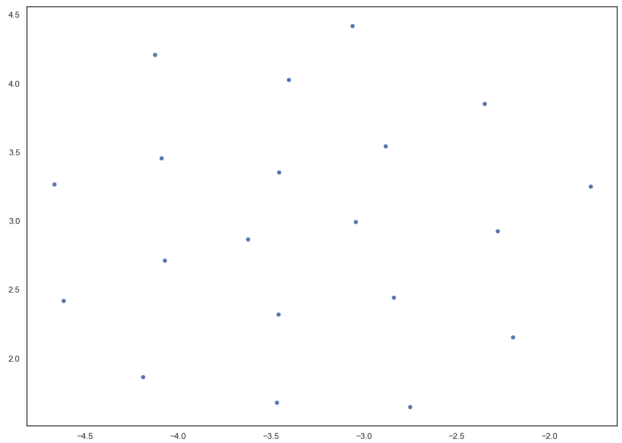

Figure 6: Compression of the twenty 75-dimensional vectors into a 2-dimensional space, relatively well uniformly distributed as expected.

6.3 Major Code Components in GA

6.3.1 Population Creation and Optimization Parameters for "Delicate"

```
Optimization_goal = [13,1,2,6,4]

num_weights = 5

population_size = 20
mating_size = 5

pop_size = (population_size,num_weights)
new_population_gene1 = numpy.random.randint(low=0, high=24, size=(
    population_size, 1))
new_population_gene2 = numpy.random.randint(low=0, high=12, size=(
    population_size, 4))
new_population_combined = numpy.hstack((new_population_gene1,
    new_population_gene2))
```

6.3.2 Full Control Sequence

```
for generation in range(generations):
    print("Generation : ", generation)
    # Measuring the fitness of each chromosome in the population.
    fitness = cal_pop_fitness(Optimization_goal, new_population_combined)
    #print("Fitness")
    #print(fitness)

    best_outputs.append(numpy.max(fitness))

    print("Best result : ", numpy.max(fitness))

    parents = select_mating_pool(new_population_combined, fitness,
                                 mating_size)
    #print("Parents")
    #print(parents)

    offspring_X = crossover(parents,
                            offspring_size=(pop_size[0]-parents.
    shape[0], num_weights))
    #print("Crossover")
    #print(offspring_X)

    offspring_mutation = mutation(offspring_X, num_mutations=1)
    print("Mutation")
    print(offspring_mutation)

    # Creating the new population based on the parents and offspring.
    new_population_combined[0:parents.shape[0], :] = parents
    new_population_combined[parents.shape[0]:, :] = offspring_mutation
```

6.3.3 GA Core Functions

3.1 Fitness Function

```
def cal_pop_fitness(Optimization_goal, population):
```

```
    # Calculating the fitness value of each solution in the current
    population.
    fitness = numpy.sum(-(population - Optimization_goal)**2, axis=1)
    return fitness
```

3.2 Function for selecting the most Fit for reproduction

```

def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents
    for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        most_fit_index = numpy.where(fitness == numpy.max(fitness))
        most_fit_index = most_fit_index[0][0]
        parents[parent_num, :] = pop[most_fit_index, :]
        fitness[most_fit_index] = -100000
    return parents
```

3.3 Function for Finding most Fit Individuals

```
def select_mating_pool(pop, fitness, num_parents):
    # Selecting the best individuals in the current generation as parents
    for producing the offspring of the next generation.
    parents = numpy.empty((num_parents, pop.shape[1]))
    for parent_num in range(num_parents):
        most_fit_index = numpy.where(fitness == numpy.max(fitness))
        most_fit_index = most_fit_index[0][0]
        parents[parent_num, :] = pop[most_fit_index, :]
        fitness[mostb_fit_index] = -100000
    return parents
```

3.4 Function for Crossover Event

```
def crossover(parents, offspring_size):
    offspring = numpy.empty(offspring_size)
    X_point = numpy.uint8(offspring_size[1]/2)

    for k in range(offspring_size[0]):
        parent1_index = k%parents.shape[0]
        parent2_index = (k+1)%parents.shape[0]
        offspring[k, 0:X_point] = parents[parent1_index, 0:X_point]
        offspring[k, X_point:] = parents[parent2_index, X_point:]
    return offspring
```

3.5 Function for Mutation or Changing Chords/Keys

```
def mutation(offspring_X, num_mutations=1):
    for index in range(offspring_X.shape[0]):
        # Randomly choosing the indices for mutation
        mutation_indices = numpy.random.randint(low=0, high=offspring_X.
    shape[1], size=num_mutations)

        for gene_index in mutation_indices:
            # Check if the mutation index is 0 mod 4
            if gene_index % 5 == 0:
                # The random value can be from 0 to 24
                random_value = numpy.random.randint(0, 25, 1)
```

```
                # Mutate the gene using the new random value
                offspring_X[index, gene_index] = random_value
            else:
                # The random value to be added to the gene.
                random_value = numpy.random.randint(0, 13, 1)
                # Make sure the mutation results in a valid value if there are boundaries
                offspring_X[index, gene_index] = (offspring_X[index, gene_index] + random_value) % 12
    return offspring_X
```

3.6 Function for the Optimization of the 15 Hits

```
Optimization_goal = [7, 1, 5, 6, 4,
                     8, 1, 5, 6, 4,
                     19, 1, 4, 6, 5,
                     6, 1, 6, 2, 5,
                     8, 6, 5, 1, 4,
                     8, 1, 5, 6, 4,
                     13, 6, 5, 4, 1,
                     17, 6, 4, 1, 5,
                     1, 1, 5, 6, 4,
                     1, 1, 4, 6, 4,
                     24, 1, 6, 3, 5,
                     13, 1, 2, 6, 4,
                     7, 1, 4, 6, 4,
                     3, 1, 5, 6, 4,
                     1, 7, 2, 3, 11]


num_weights = 75

population_size = 20
mating_size = 5

pop_size = (population_size, num_weights)
new_population_gene1 = numpy.random.randint(low=0, high=24, size=(population_size, 1))
new_population_gene2 = numpy.random.randint(low=0, high=12, size=(population_size, 4))
new_population_combined = numpy.hstack((new_population_gene1, new_population_gene2))

# Loop 15 times to add 70 more columns, 5 columns at a time
for i in range(14):
    new_population_gene1a = numpy.random.randint(low=0, high=24, size=(population_size, 1))
    new_population_gene2a = numpy.random.randint(low=0, high=12, size=(population_size, 4))
    additional_genes = numpy.hstack((new_population_gene1a, new_population_gene2a))
    new_population_combined = numpy.hstack((new_population_combined, additional_genes))
```

3.7 Functions for Calculating the Log-Normal Parameters

```
data_np = numpy.array(num_generations)

# Log-transform the data
log_data = numpy.log(data_np)

# Calculate mean and standard deviation of the log-transformed data
mu = numpy.mean(log_data)
sigma = numpy.std(log_data)

# Calculate mean, mode, and standard deviation of the original data
    assuming a log-normal distribution
mean_log_normal = numpy.exp(mu + sigma**2 / 2)
mode_log_normal = numpy.exp(mu - sigma**2)
# For the standard deviation, we use the formula for a log-normal
    distribution
std_dev_log_normal = (numpy.exp(sigma**2) - 1) * numpy.exp(2*mu + sigma
    **2)**0.5

print(f"Mean (Log-Normal): {mean_log_normal}")
print(f"Mode (Log-Normal): {mode_log_normal}")
print(f"Standard Deviation (Log-Normal): {std_dev_log_normal}")
```

6.3.4 Why not other Dim-Red Techniques?

Something can be said for other linear (PCA) and non-linear (t-SNE) dimensionality reduction techniques, but none as visually clear as UMAP, for the sake of brevity they will be excluded from the report.

In the context of existing research on Uniform Manifold Approximation and Projection (UMAP) and dimensionality reduction techniques, it is common to employ a scaling mechanism. This practice, however, was deliberately omitted in this instance. The decision was based on the observation that the normalization of value ranges was not essential for the desired outcomes. Specifically, the variance within the key parameters was observed to be twice as significant as that within the chords. This distinction underscores the greater importance of the key, or initial value, in accurately determining a song, compared to the chords. Therefore, assigning a higher priority to the key was deemed justifiable and satisfactory for the purposes of this research. Some clustering with k-means was also performed.