



Project_Module 2

Created	@August 4, 2025 6:51 PM
Class	IIITH
Event	

humans, animals, or even viruses like SARS-CoV-2, are made up of a sequence of four different chemical bases. Think of them as the building blocks of the genetic code. These four bases are:

- **A** (Adenine)
- **T** (Thymine)
- **G** (Guanine)
- **C** (Cytosine)

The order and arrangement of these bases along the genome sequence carry the instructions for building and maintaining an organism (or in the case of a virus, replicating itself). Just like different combinations of letters form different words and sentences, different combinations of these four bases form different genes and genetic information.

Let's break down how the hash function works to convert a k-mer into a single number (its position in the array).

The hash function uses a base-4 system, where each nucleotide is assigned a value:

- A: 0
- G: 1
- T: 2
- C: 3

The position of each nucleotide within the k-mer is also important. The positions are counted from right to left, starting at 0.

The formula is: Hash Value = (Value of last nucleotide * 4^0) + (Value of second to last nucleotide * 4^1) + (Value of third to last nucleotide * 4^2) + ...

Let's re-calculate the example given: **TAG**

- The k-mer is TAG. $k = 3$.
- Positions:
 - G is at position 0
 - A is at position 1
 - T is at position 2
- Nucleotide values:
 - G = 1
 - A = 0
 - T = 2

Now, apply the formula: Hash Value = (Value of G * 4^0) + (Value of A * 4^1) + (Value of T * 4^2)
Hash Value = $(1 * 4^0) + (0 * 4^1) + (2 * 4^2)$
Hash Value = $(1 * 1) + (0 * 4) + (2 * 16)$
Hash Value = $1 + 0 + 32$
Hash Value = 33

Correction: It appears the example calculation in the prompt is incorrect. Let's follow the definition of the hash function strictly. The example states "Multiply the character value by $4^{(\text{position of the character in the kmer})}$ ", and the example calculation shows $(4^0 * 2) + (4^1 * 0) + (4^2 * 1)$ for TAG. If we assume position 0 is the first character, this would correspond to $(4^0 * T) + (4^1 * A) + (4^2 * G)$. Using the values A:0, G:1, T:2, C:3: $(1 * 2) + (4 * 0) + (16 * 1) = 2 + 0 + 16 = 18$. This matches the example output. So, the position seems to be counted from left to right, starting at 0. Let's use this interpretation for the next example.

Let's calculate the hash value for **CATCGAG** with $k = 7$, positions from left to right starting at 0.

- The k-mer is CATCGAG. $k = 7$.
- Positions:

- C is at position 0
- A is at position 1
- T is at position 2
- C is at position 3
- G is at position 4
- A is at position 5
- G is at position 6
- Nucleotide values:
 - C = 3
 - A = 0
 - T = 2
 - G = 1

Now, apply the formula based on the observed pattern in the TAG example: Hash Value = (Value of C * 4⁰) + (Value of A * 4¹) + (Value of T * 4²) + (Value of C * 4³) + (Value of G * 4⁴) + (Value of A * 4⁵) + (Value of G * 4⁶) Hash Value = (3 * 4⁰) + (0 * 4¹) + (2 * 4²) + (3 * 4³) + (1 * 4⁴) + (0 * 4⁵) + (1 * 4⁶) Hash Value = (3 * 1) + (0 * 4) + (2 * 16) + (3 * 64) + (1 * 256) + (0 * 1024) + (1 * 4096) Hash Value = 3 + 0 + 32 + 192 + 256 + 0 + 4096 Hash Value = 4579

So, the hash value for CATCGAG is 4579. This number represents the index in the 16384-sized array where the count for the k-mer CATCGAG would be stored.



indices should be unique right? is there an assurance that these hash values wont match?

That's a great question! With this specific hash function and a fixed k-mer length (like k=7 in this case), the hash values are indeed unique for every *possible* k-mer.

The hash function you're using essentially converts the k-mer from a sequence of nucleotides (which can be thought of as a base-4 representation) into a single integer. Since each unique combination of 7 nucleotides (a 7-mer) will result in a

unique base-4 number, converting that to a base-10 integer will also result in a unique number.

For a fixed k :

- There are exactly 4^k possible unique k -mers.
- The hash function maps each of these 4^k k -mers to a unique integer between 0 and $4^k - 1$.

This type of hash function, where each possible input maps to a unique output within the defined range, is sometimes referred to as a "perfect hash function" for this specific use case (fixed-length strings from a fixed alphabet).

Therefore, for a fixed k , there is an assurance that different k -mers will have different hash values using this method. The size of the `big_matrix` (16384 for $k=7$, which is 4^7) is exactly the number of possible unique 7-mers, and each k -mer maps to a unique index within this range

.

Okay, let's use a small, simplified example to show what `big_matrix` and `sequences_name` would look like.

Imagine we have a `sequences.fasta` file with just two short sequences:

```
>Seq1
ATGC
>Seq2
GCAT
```

Let's use a smaller k -mer size, say $k = 2$.

The total number of possible 2-mers is $4^2 = 16$. Our embedding vector size will be 16, with indices corresponding to the hash values of the 16 possible 2-mers (AA, AT, AG, AC, TA, TT, TG, TC, GA, GT, GG, GC, CA, CT, CG, CC).

Using the hash function (A:0, G:1, T:2, C:3, position from left): AA: $(0*4^0) + (0*4^1) = 0$ AT: $(0*4^0) + (2*4^1) = 8$ AG: $(0*4^0) + (1*4^1) = 4$ AC: $(0*4^0) + (3*4^1) = 12$ TA: $(2*4^0) + (0*4^1) = 2$ TT: $(2*4^0) + (2*4^1) = 10$ TG: $(2*4^0) + (1*4^1) = 6$ TC: $(2*4^0) + (3*4^1) = 14$ GA: $(1*4^0) + (0*4^1) = 1$ GT: $(1*4^0) +$

$(2 \cdot 4^1) = 9$ GG: $(1 \cdot 4^0) + (1 \cdot 4^1) = 5$ GC: $(1 \cdot 4^0) + (3 \cdot 4^1) = 13$ CA: $(3 \cdot 4^0) + (0 \cdot 4^1) = 3$ CT: $(3 \cdot 4^0) + (2 \cdot 4^1) = 11$ CG: $(3 \cdot 4^0) + (1 \cdot 4^1) = 7$ CC: $(3 \cdot 4^0) + (3 \cdot 4^1) = 15$

Now, let's process the sequences:

Sequence 1: ATGC (k=2) K-mers: AT, TG, GC

- AT hash: 8
- TG hash: 6
- GC hash: 13 Embedding vector (size 16): [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0] (1 at indices 6, 8, 13, zeros elsewhere)

Sequence 2: GCAT (k=2) K-mers: GC, CA, AT

- GC hash: 13
- CA hash: 3
- AT hash: 8 Embedding vector (size 16): [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0] (1 at indices 3, 8, 13, zeros elsewhere)

After the code cell runs with this example data:

`sequences_name` would be: `['Seq1', 'Seq2']`

`big_matrix` would be:

```
[
  [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0], # Embedding for Seq1
  [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0] # Embedding for Seq2
]
```

So, `sequences_name` stores the identifiers for each sequence, and `big_matrix` stores the corresponding k-mer count vectors. Each row in `big_matrix` is the numerical representation of a sequence, and each column corresponds to one of the possible unique k-mers (in the real data, there are 16384 columns for k=7). The values in the matrix are the counts of how many times that k-mer appeared in that specific sequence.

Your interpretation about the distance between clusters is also correct in principle. The t-SNE plot attempts to place sequences with similar k-mer profiles closer together and sequences with different profiles further apart. So, if the Gujarat cluster is relatively far from both of the Telangana clusters, it suggests that the sequences from Gujarat have a k-mer composition that is distinct from the k-mer compositions of the sequences in both Telangana clusters. The relative distances between clusters can give you a sense of how genetically different those presumed lineages or variants are from each other based on the k-mer analysis

Yes, the clusters you see in the t-SNE plot, where sequences from the same state group together, likely mean that those sequences share similar **mutation profiles** (when we get to the mutation-based analysis later) or **k-mer compositions** (in this k-mer based analysis).

Think of each cluster as potentially representing a distinct **viral lineage** or **variant** that was circulating in that area during the sampling period. If sequences from a state form multiple distinct clusters (like you're observing for Telangana), it suggests that there might have been at least two different viral lineages present in that state at the time

```
## First we need to get the list of all unique mutations observed in our dataset.
```

```
mutations = []
for i in range(len(data)):

    sample_muts = data["Nuc_muts"][i][1:][: -1].split(",")

    for j in sample_muts:
        mut = j.strip()[1:][: -1]
        if(len(mut)>=2):
            if(mut not in mutations):
                mutations.append(mut)
```

```
print((mutations))
```

This code cell processes the 'Nuc_muts' column in your `data` DataFrame to extract a list of all unique nucleotide mutations observed across all the genome sequences.

Here's a breakdown:

1. `mutations = []` : Initializes an empty list called `mutations` . This list will store the unique mutation strings as they are found.
2. `for i in range(len(data)):` : This loop iterates through each row of your `data` DataFrame. `len(data)` gives the number of rows (which is 685, the number of sequences).
3. `sample_muts = data["Nuc_muts"][i][1:][-1].split(",")` : This line extracts and preprocesses the mutation string for the current row (`i`).
 - `data["Nuc_muts"][i]` : Accesses the string value in the 'Nuc_muts' column for the current row. This string looks something like `['C1281T', 'C4158T', ...]` .
 - `[1:][-1]` : This slices the string to remove the opening square bracket `[` at the beginning and the closing square bracket `]` at the end.
 - `.split(",")` : Splits the resulting string by the comma `,` character, creating a list of individual mutation strings (e.g., `['C1281T', 'C4158T']`).
4. `for j in sample_muts:` : This inner loop iterates through each individual mutation string (`j`) obtained from splitting the `sample_muts` list.
5. `mut = j.strip()[1:][-1]` : This line further cleans the individual mutation string `j` .
 - `j.strip()` : Removes any leading or trailing whitespace from the string (which might happen from the split).
 - `[1:][-1]` : This slicing seems to be intended to remove the single quotes `'` around each mutation string (e.g., turning `'C1281T'` into `C1281T`).
6. `if(len(mut)>=2):` : This checks if the cleaned mutation string has a length of at least 2. This likely filters out any empty strings or invalid entries that might have resulted from the splitting or cleaning process.

7. `if(mut not in mutations): mutations.append(mut)` : This is the core logic for collecting unique mutations. It checks if the current cleaned mutation string (`mut`) is *not already* present in the `mutations` list. If it's a new mutation, it's added to the list.
8. `print(len(mutations))` : After processing all rows and all mutations, this line prints the total number of unique mutations found.

In summary, this code iterates through the list of mutations provided for each sequence in the metadata, cleans up the format of these mutation strings, and builds a comprehensive list of every unique mutation observed across all 685 sequences. This list of unique mutations will then be used to create the binary presence/absence vectors for the mutation-based analysis.

```
samplelist = data["StrainID"].tolist()
df = pd.DataFrame(columns = [*samplelist])
df = df.replace(np.nan,0)

for mutation in mutations:

    dicti = dict(zip(samplelist, np.zeros(len(samplelist), dtype=int)))

    for i in range(len(data)):
        flag=0
        strainid = data["StrainID"][i]
        sample_muts = data["Nuc_muts"][i][1:][-1].split(",")

        for j in sample_muts:
            mut = j.strip()[1:][-1]
            if(len(mut)>=2):
                if(mut == mutation):
                    flag = 1
                    break

        dicti[strainid]=flag

df = pd.concat([df, pd.DataFrame([dicti])], ignore_index=True)
```



```
df = df.T
```

```
df.tail()
```

1. `samplelist = data["StrainID"].tolist()` : Creates a list called `samplelist` containing the Strain IDs of all the sequences from the `data` DataFrame. This list will be used for the column names of the new DataFrame.
2. `df = pd.DataFrame(columns = [*samplelist])` : Initializes an empty pandas DataFrame called `df` . The columns of this DataFrame are set to be the Strain IDs from `samplelist` .
3. `df = df.replace(np.nan,0)` : This line is somewhat redundant given how the DataFrame is populated later, but it would typically replace any initial `NaN` values with 0.
4. `for mutation in mutations:` : This outer loop iterates through each unique mutation in the `mutations` list you generated in the previous cell.
5. `dicti = dict(zip(samplelist, np.zeros(len(samplelist), dtype=int)))` : Inside the outer loop, for each `mutation` , a dictionary called `dicti` is created. This dictionary is initialized with Strain IDs as keys and all values set to 0. This dictionary will temporarily store the presence (1) or absence (0) status of the *current mutation* across all sequences.
6. `for i in range(len(data)):` : This inner loop iterates through each row (each sequence) in the original `data` DataFrame.
7. `flag=0` : Initializes a flag to 0 for the current sequence, indicating that the mutation has not been found yet.
8. `strainid = data["StrainID"][i]` : Gets the Strain ID for the current sequence.
9. `sample_muts = data["Nuc_muts"][i][1:][::-1].split(",")` : Extracts and splits the list of mutations for the current sequence, similar to the previous cell.
10. `for j in sample_muts:` : This loop iterates through each mutation listed for the current sequence.
11. `mut = j.strip()[1:][::-1]` : Cleans the individual mutation string.
12. `if(len(mut)>=2):` : Checks if the cleaned mutation string is valid.

13. `if(mut == mutation):` : This is the key comparison. It checks if the cleaned mutation from the current sequence (`mut`) matches the unique mutation currently being processed by the outer loop (`mutation`).
14. `flag = 1; break` : If a match is found, the `flag` is set to 1 (indicating the presence of the mutation in this sequence), and the inner loop `for j in sample_muts:` is broken because we only need to know if the mutation is present at least once.
15. `dicti[strainid]=flag` : After checking all mutations for the current sequence, the value in the `dicti` dictionary for the current `strainid` is set to the final `flag` value (either 0 or 1).
16. `df = pd.concat([df, pd.DataFrame([dicti]), ignore_index=True)` : After iterating through all sequences for the current `mutation`, the `dicti` dictionary (converted to a single-row DataFrame) is concatenated to the main `df` DataFrame. Each row added this way represents the presence/absence vector for one unique mutation across all sequences.
17. `df = df.T` : After the outer loop finishes (processing all unique mutations), the DataFrame is transposed (`.T`). This swaps rows and columns, so that each row now represents a **sequence** (StrainID) and each column represents a **unique mutation**. This is the desired format for the mutation-based analysis.
18. `df.tail()` : Displays the last few rows of the transposed DataFrame to show its structure.

The resulting `df` DataFrame is a binary matrix with dimensions (number of sequences) x (number of unique mutations), ready for dimensionality reduction and visualization based on mutation profiles.

let's use a simplified example to illustrate how the binary mutation matrix (`df`) is created.

Imagine you have the following simplified `data` DataFrame with 3 sequences and a few mutations:

```
StrainID      Nuc_muts State
0   SeqA  ['A10G', 'C20T'] State1
```

```
1 SeqB ['A10G', 'G30A'] State2
2 SeqC ['C20T', 'G40C'] State1
```

First, the code would find the unique mutations from the `Nuc_muts` column. In this case, the unique mutations are: `['A10G', 'C20T', 'G30A', 'G40C']`.

The `samplelist` would be: `['SeqA', 'SeqB', 'SeqC']`.

The code then iterates through each unique mutation and checks which sequences have that mutation.

Processing 'A10G':

- Create `dicti`: `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}`
- Check SeqA: Has 'A10G'. `dicti` becomes `{'SeqA': 1, 'SeqB': 0, 'SeqC': 0}`.
- Check SeqB: Has 'A10G'. `dicti` becomes `{'SeqA': 1, 'SeqB': 1, 'SeqC': 0}`.
- Check SeqC: Does not have 'A10G'. `dicti` remains `{'SeqA': 1, 'SeqB': 1, 'SeqC': 0}`.
- Concatenate to `df`. `df` is now:

```
SeqA SeqB SeqC
0  1  1  0
```

Processing 'C20T':

- Create `dicti`: `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}`
- Check SeqA: Has 'C20T'. `dicti` becomes `{'SeqA': 1, 'SeqB': 0, 'SeqC': 0}`.
- Check SeqB: Does not have 'C20T'. `dicti` remains `{'SeqA': 1, 'SeqB': 0, 'SeqC': 0}`.
- Check SeqC: Has 'C20T'. `dicti` becomes `{'SeqA': 1, 'SeqB': 0, 'SeqC': 1}`.
- Concatenate to `df`. `df` is now:

```
SeqA SeqB SeqC
0  1  1  0
1  1  0  1
```

Processing 'G30A':

- Create `dicti` : `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}`
- Check SeqA: Does not have 'G30A'. `dicti` remains `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}` .
- Check SeqB: Has 'G30A'. `dicti` becomes `{'SeqA': 0, 'SeqB': 1, 'SeqC': 0}` .
- Check SeqC: Does not have 'G30A'. `dicti` remains `{'SeqA': 0, 'SeqB': 1, 'SeqC': 0}` .
- Concatenate to `df` . `df` is now:

```
SeqA SeqB SeqC
0  1  1  0
1  1  0  1
2  0  1  0
```

Processing 'G40C':

- Create `dicti` : `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}`
- Check SeqA: Does not have 'G40C'. `dicti` remains `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}` .
- Check SeqB: Does not have 'G40C'. `dicti` remains `{'SeqA': 0, 'SeqB': 0, 'SeqC': 0}` .
- Check SeqC: Has 'G40C'. `dicti` becomes `{'SeqA': 0, 'SeqB': 0, 'SeqC': 1}` .
- Concatenate to `df` . `df` is now:

```
SeqA SeqB SeqC
0  1  1  0
1  1  0  1
2  0  1  0
3  0  0  1
```

Finally, the code transposes the DataFrame (`df = df.T`).

The final `df` would look like this:

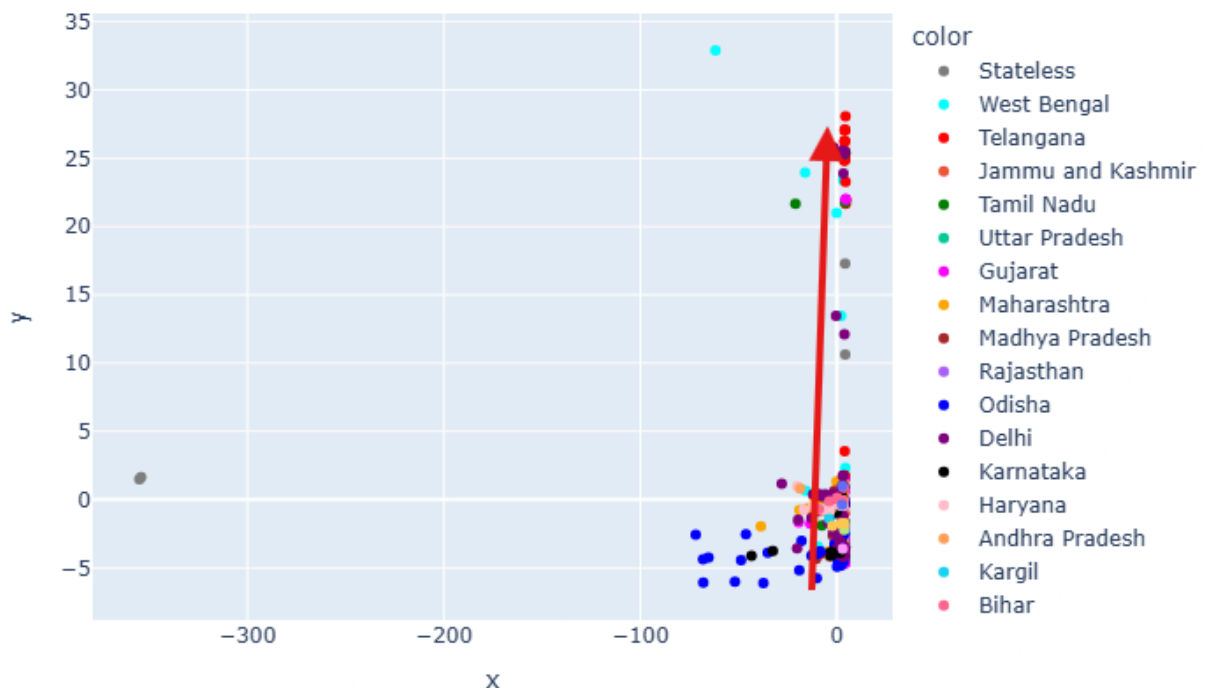
```
0 1 2 3
SeqA 1 1 0 0
SeqB 1 0 1 0
SeqC 0 1 0 1
```

In this final transposed DataFrame:

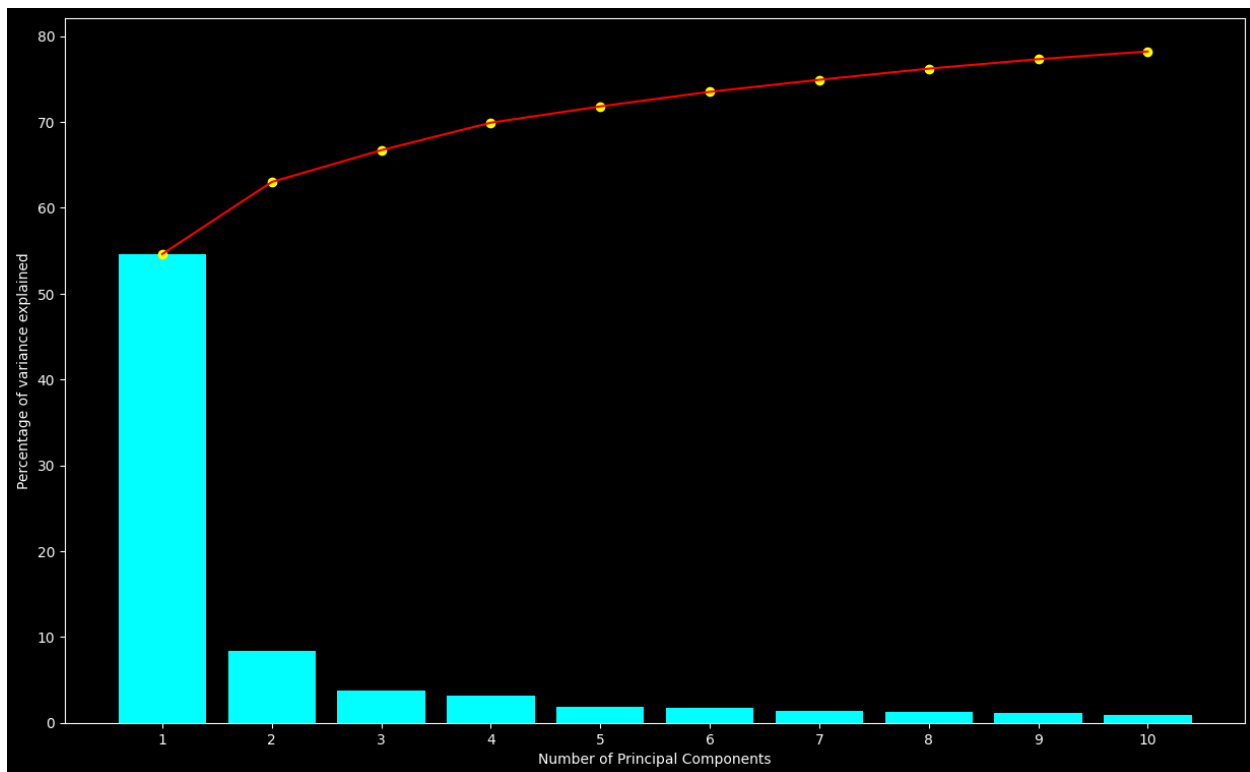
- Each row is a sequence ('SeqA', 'SeqB', 'SeqC').
- Each column is a unique mutation (represented by indices 0, 1, 2, 3, corresponding to 'A10G', 'C20T', 'G30A', 'G40C').
- The value is 1 if the sequence has that mutation, and 0 if it doesn't.

This binary matrix is then used for dimensionality reduction like PCA or t-SNE.

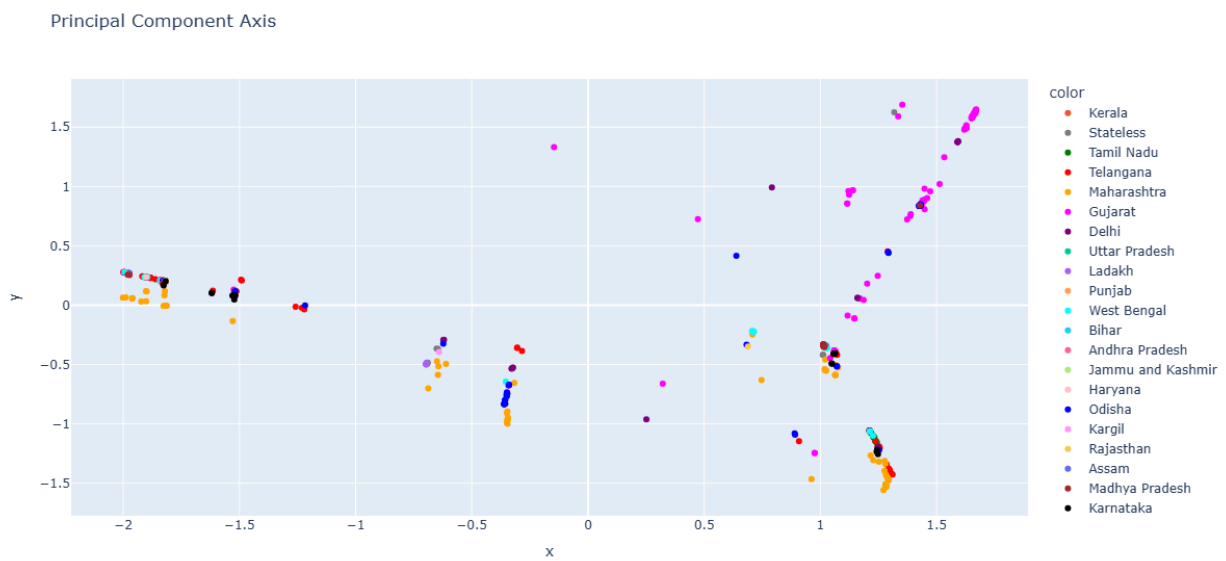
Principal Component Axis

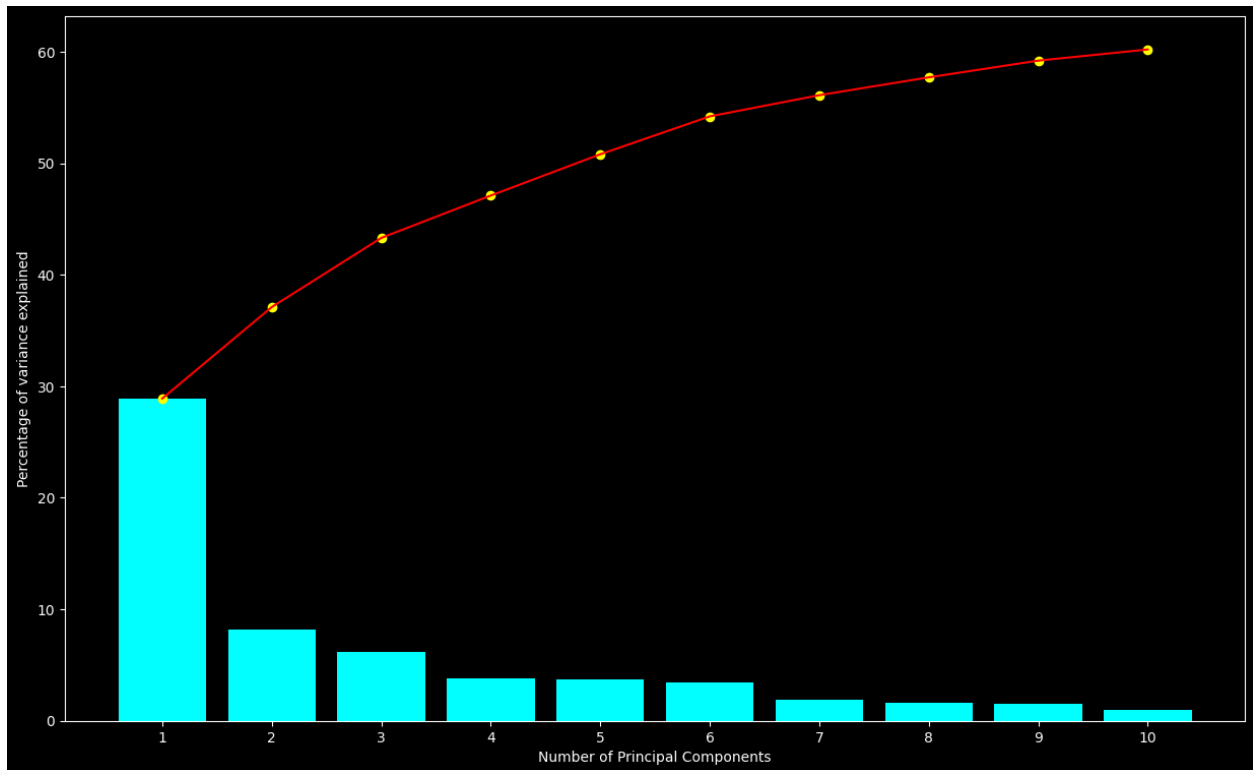


For PCA like this most of the data can be captured by PC1 example probable axis as shown as the data is spread of in one direction is less variant



But it isn't apparent for the PCA plot shown below





here the data is more spreadout , hence the PC1 axis cant capture most of the info

Can you show me an example data differences between the first and second analysis:

Okay, let's use a simplified example with a single short sequence and a few hypothetical mutations to show the difference in the data representation used for the two PCA analyses.

Imagine a short sequence: `ATGCGT`

And let's say for this sequence, the precomputed mutations compared to a reference are `['A1G', 'C4T']`.

Data Representation for the First PCA (k-mer based):

Let's use a small `k`, say `k=3`. The 3-mers in the sequence `ATGCGT` are: `ATG`, `TGC`, `GCG`, `CGT`.

The total number of possible 3-mers is $4^3 = 64$.

The k-mer count vector for this sequence would be a vector of size 64. Most values would be 0. The values would be 1 for the k-mers 'ATG', 'TGC', 'GCG', and 'CGT' (assuming they are unique within the sequence and only appear once).

For example, if the hash values for these 3-mers were: ATG → 8 TGC → 13 GCG → 5 CGT → 14

The k-mer count vector (simplified, showing only relevant positions) would look something like this:

Index: ... 5 ... 8 ... 13 ... 14 ... (out of 64 total indices) Value: ... 1 ... 1 ... 1 ... 1 ... (0 for all other indices)

This vector of counts (or frequencies) is what was used as input for the first PCA.

Data Representation for the Second PCA (Mutation based):

First, we need the list of all unique mutations observed in the *entire dataset*. Let's say for our small dataset, the unique mutations were ['A1G', 'C4T', 'G10A', 'T15C'] .

For the sequence ATGCGT , the precomputed mutations were ['A1G', 'C4T'] .

The mutation profile vector for this sequence would be a binary vector of size equal to the number of unique mutations (4 in this simplified example).

We check the presence of each unique mutation in the sequence's mutation list:

- 'A1G': Present (1)
- 'C4T': Present (1)
- 'G10A': Absent (0)
- 'T15C': Absent (0)

The binary mutation vector for this sequence would be:

Mutation: A1G C4T G10A T15C Value: 1 1 0 0

This binary vector (1 if the mutation is present, 0 if absent) is what was used as input for the second PCA.

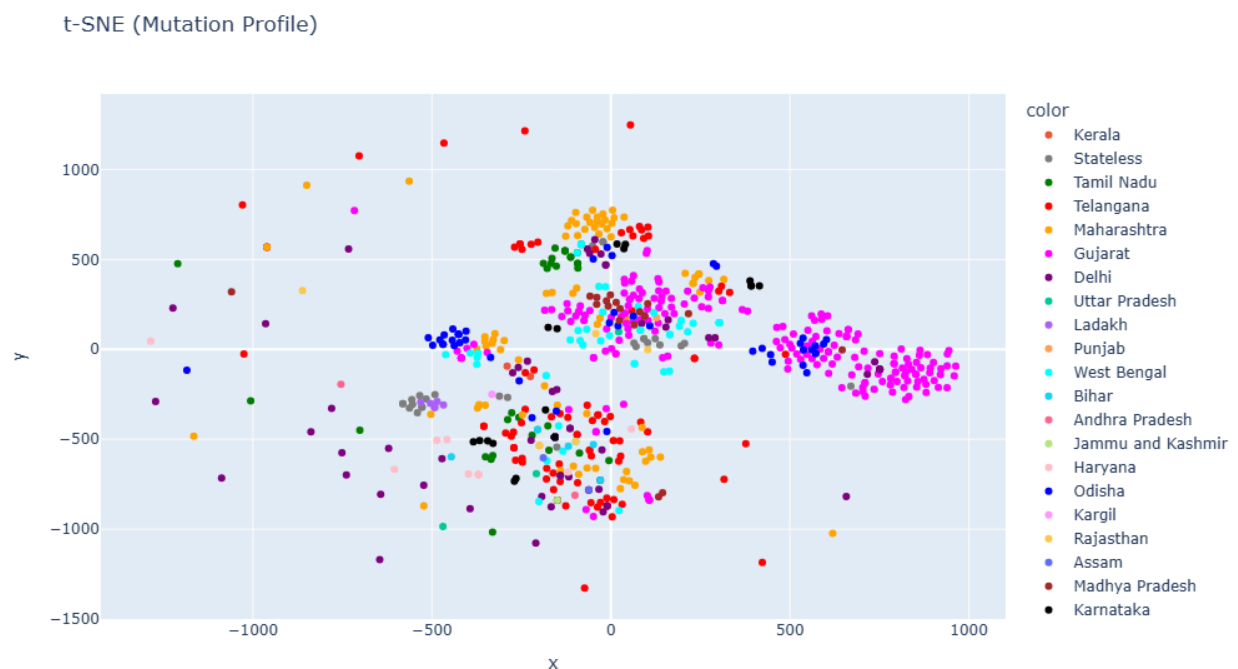
In summary:

- **First PCA (k-mer):** Uses vectors where values are *counts* of short sequence patterns (k-mers). Captures overall sequence composition and differences

based on these counts.

- **Second PCA (Mutation):** Uses vectors where values are *binary* (0 or 1) indicating the presence or absence of specific changes (mutations). Captures differences based on whether sequences share or lack particular mutations.

Both methods aim to represent the sequence numerically, but they capture different aspects of the sequence's characteristics, leading to potentially different clustering patterns in the resulting PCA plots.



the second t-SNE plot, which is based on the mutation profiles of the genome sequences, and the conclusions that can be drawn from it.

As explained in the markdown cell with ID [9Z0ZBB-6qCSY](#) after this plot, the t-SNE visualization based on mutation profiles is particularly good at highlighting the clustering of sequences that share similar sets of mutations. The conclusions drawn from this plot, supported by further genetic analysis (though the details are not strictly required for interpreting the visualization), are as follows:

1. **Odisha Outliers:** The plot clearly shows the blue points representing sequences from Odisha as being distinct and separated from the main body of data. The conclusion is that these sequences had unique genomic features (specifically, deletions) not found in other states, causing them to be outliers in both the k-mer and mutation-based analyses.
2. **Telangana, Delhi, and Tamil Nadu Cluster:** The secondary cluster formed by sequences primarily from Telangana (red), Delhi (purple), and Tamil Nadu (green) is very prominent in this t-SNE plot. This clustering is explained by the presence of a specific, co-occurring set of mutations (C6312A, C13730T, C23929T, and C28311T) in a significant proportion of sequences from these states. This suggests the rapid spread of a particular viral lineage in these regions, possibly due to early community transmission events.
3. **Gujarat Cluster:** The dense cluster of magenta points representing sequences from Gujarat is also clearly visible, often located within or adjacent to the main cluster but forming a distinct subgroup. This clustering is attributed to another shared set of characteristic mutations (C18877T, G25563T, and C26735T) that were less common in other parts of the country. The observation about the high fatality rate in Gujarat being comparable to Europe, where these mutations were also found, is presented as a possible correlation, highlighting how genetic differences might relate to clinical outcomes and transmission routes.

In essence, the t-SNE plot visually confirms that sequences from different states exhibit distinct patterns of mutations. These patterns likely reflect the introduction and spread of different viral lineages or variants in various parts of India during the early stages of the pandemic. The visualization provides compelling evidence for regional differences in circulating viral populations, which aligns with epidemiological investigations into the spread of the virus