Information Technology Course
Module Project
by Damir Dobric / Andreas Pech

Song recommendation using Hierarchical Temporal Memory

Mandar Anil Patkar
mandar.patkar@stud.fra-uas.de

***Abstract*—This paper presents an innovative approach to song recommendation using Hierarchical Temporal Memory (HTM) to build a song recommending system which would recommend the next song depending on the playlist the person is listing. In the context of an increasingly digital and social media-centric world, we aim to enhance user engagement with music playing applications by personalizing song selections to individual preferences. Our methodology leverages Neocortex API to develop a dynamic playlist recommendation system that not only aligns with the user's current interests and tastes but also adapts to changing preferences over time. This system will help to maximize the time users spend interacting with the application, ensuring a seamless and enjoyable user experience. Through a series of experiments and user trials, we demonstrate the effectiveness of our HTM-based recommendation system in providing highly relevant and engaging musical content. Also, the parameters chosen while creating the system are simple attributes of songs which influence the recommendation are name of artist, the genre of the song and the mood of the song. The experiment was run on different configuration and then a specific HTM configuration which matches with highest accuracy was chosen. The results indicate a significant matching result in song recommendation compared to the next song in the playlist. So far during various runs the system has given 93.1% accuracy predicting the next song to be same as what is already present next in the playlist.**

Keywords—AI, HTM, SDRs, spatial pooler, API, multisequence learning, neocortex.

I. INTRODUCTION

There are many events which occur rhythmically, and us as human try to follow it naturally, here we are trying to identify such music sequence which is associated entity, and the end goal is to learn song with key and predict the next song in given playlist. In brief for this experiment, we have around 5000 songs which forms around 100 playlists. Yes, each playlist has 50 songs. This experiment with help of Neocortex API learns the playlists as sequences in a model. During prediction we give an input as one of the songs out 5000 songs randomly chosen, and the algorithm gives us next song from a playlist which it has learns from previous playlists. The 100 playlists which I choose are from the top 50 songs played internationally and is given to us by popular song streaming application.

The neocortex is the seat of intelligent thought in the mammalian brain. High level vision, hearing, touch, movement, language, and planning are all performed by the neocortex. Given such a diverse suite of cognitive functions, you might expect the neocortex to implement an equally diverse suite of specialized neural algorithms. This is not the case. The neocortex displays a remarkably uniform pattern of neural circuitry. The biological evidence suggests that the neocortex implements a common set of algorithms to perform many different intelligence functions [1].

The neocortex continually processes an endless stream of rich sensory information. It does this remarkably well, better than any existing AI computer. A wealth of empirical evidence demonstrates that cortical regions represent all information using sparse patterns of activity. To function effectively throughout a lifetime these representations must have tremendous capacity and must be extremely tolerant to noise. However, a detailed theoretical understanding of the capacity and robustness of cortical sparse representations has been missing [2] .

HTM provides a theoretical framework for understanding the neocortex and its many capabilities. To date we have implemented a small subset of this theoretical framework. Over time, more and more of the theory will be implemented. Today we believe we have implemented a sufficient subset of what the neocortex does to be of commercial and scientific value [1].

II. LITERATURE SURVEY

A. *Sequence*

A particular order in which related things follow each other. For our experiment the sequence is nothing but the playlist of songs.

B. Key

The sequence which is associated with an entity is called key. Here we create the key using the name of playlist with song number as they come in the playlist. Song numbers are unique to the songs.

C. HTM

Hierarchical Temporal Memory (HTM) provides a flexible and biologically accurate framework for solving prediction, classification, and anomaly detection problems for a broad range of data types. HTM systems require data input in the form of Sparse Distributed Representations (SDRs). SDRs are quite different from standard computer representations, such as ASCII for text, in that meaning is encoded directly into the representation. An SDR consists of a large array of bits of which most are zeros, and a few are ones. Each bit carries some semantic meaning so if two SDRs have more than a few overlapping one-bits, then those two SDRs have similar meanings [3].

HTMs can be viewed as a type of neural network. By definition, any system that tries to model the architectural details of the neocortex is a neural network. However, on its own, the term "neural network" is not very useful because it has been applied to a large variety of systems. HTMs model neurons (called cells when referring to HTM), which are arranged in columns, in layers, in regions, and in a hierarchy. The details matter, and in this regard HTMs are a new form of neural network [1].

D. Sparse Distributed Representations (SDRs)

Empirical evidence shows that every region of the neocortex represents information using sparse activity patterns made up of a small percentage of active neurons, with the remaining neurons being inactive. An SDR is a set of binary vectors where a small percentage of 1s represent active neurons, and the 0s represent inactive neurons. The small percentage of 1s, denoted the *sparsity*, varies from less than one percent to several percent. SDRs are the primary data structure used in the neocortex and used everywhere in HTM systems. There is not a single type of SDRs in HTM but distinct types for various purposes [4].

While a bit position in a dense representation like ASCII has no semantic meaning, the bit positions in an SDR represent a particular property. The semantic meaning depends on what the input data represents. Some bits may represent edges or big patches of color; others might correspond to different musical notes. Figure 1 shows a somewhat contrived but illustrative example of an SDR representing parts of a zebra. If we flip a single bit in a vector from a dense representation, the vector may take an entirely different value. In an SDR, nearby bit positions represent similar properties. If we invert a bit, then the description changes but not radically [4].

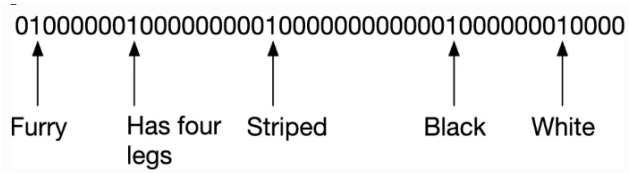

Figure 1 : Encoded data for an animal

III. METHODOLOGY

In the experiment, the dataset which was picked up is for 100 days of top 50 international playlist which consists of most played, liked and listened 50 songs from Spotify [5], so overall we have 100 sequences to for the experiment and totals 5000 songs. The dataset for playlists was refined for any error, further broken down and dropped unnecessary column or attributes. The class of Song is the most minute and important model for the experiment which holds the data. The collection of these songs together forms a Playlist. The attribute of model Song further needs to be converted in a form that HTM can be process which is mostly numeric. These numeric values are saved in so called class of Database which hold all the attributes of a song in unique fashion. The playlist is then encoded using Scalar Encoder. The encoded data was then trained using an algorithm called Multi-sequence Learning algorithm. Detailed explanation and steps are broken down further in detail and is given below.

The following methods were used to preprocess, learn and postprocess as part of the experiment.

A. Read from CSV file and fill Dataset class

The original Dataset [5] which is formatted in CSV format had lot attributes which a song has, but not all the fields were necessary. Most of these attributes were dropped manually and a shorter and Dataset of 5000 entries was selected.

The important fields which were kept were *the position* of the song in given playlist, *name* of the song, *artists* of the song, *genre* of the song and *mood* of the song. The *artists* had one or more artists anything more than 2 artists were needed to drop so we do not have more attributes of same type of significant data.

position	song	artist	genre	mood
1	Ella Baila S	Eslabon Ar	Pop	Energetic
2	un x100to	Grupo Fro	Pop	Energetic
3	La Bebe - I	Yng Lvcas	Reggaeton	Upbeat
4	Cupid - Tw	FIFTY FIFT	Pop	Romantic
5	Flowers	Miley Cyru	Pop	Romantic
6	Daylight	David Kush	Pop	Upbeat
7	Kill Bill	SZA	Soundtrack	Intense
8	Tattoo	Loreen	Pop	Melancholic
9	As It Was	Harry Style	Pop	Melancholic
10	TQG	KAROL G	Pop	Energetic

Figure 2 : Dataset

This data was read from the CSV file and put into class of list of *Dataset* and if the position attribute restarts the counter value, then a new *Dataset* list was created.

B. Fill Song and create Playlist

The class of *Song* is the smallest entity of the sequence which we use in our experiment to hold all the required and necessary attributes songs. The model consists of *Name* of song, *Singer 1* and *Singer 2* of song, *Genre 1* and *Genre 2* of the song, and finally *Mood* of the song. A list of *Song* when given a name form a *Playlist*. The model of Song and Playlist can be seen in Listing 1.

The dataset which was read previously from the CSV is now particularly formatted and processed to fill model of *Song*. When a list of *Dataset* is filled with *Song*, a name is given to the list and added to the model of *Playlist*.

```
public class Song
{
      public String Name { get; set; }
      public String Singer1 { get; set; }
      public String Singer2 { get; set; }
      public String Genre1 { get; set; }
      public String Genre2 { get; set; }
      public String Mood { get; set; }
}

public class Playlist
{
      public String Name { get; set; }
      public List<Song> Songs { get; set;}
}
```

Listing 1: Model of *Song* and *Playlist*

The data has irregularities such as multiple comma values in single attribute and each attribute needs to be mapped to a unique value.

C. Fill Database from attributes of Song

The *Database* model is class of dictionary which hold all the attributes of *Song* as key with a unique numeric value assigned to it.

```
public class Database
{
    public Dictionary<string, int> SongNames = new
Dictionary<string, int>();

    public Dictionary<string, int> Singers = new
Dictionary<string, int>();

    public Dictionary<string, int> Genres = new
Dictionary<string, int>();

    public Dictionary<string, int> Moods = new
Dictionary<string, int>();
}
```

Listing 2: Model of *Database*

Listing 3 is the model of *Database* for attributes of song. Figure 3 show the transformation from raw data to attributes of say genre having mapped to a unique numeric value.

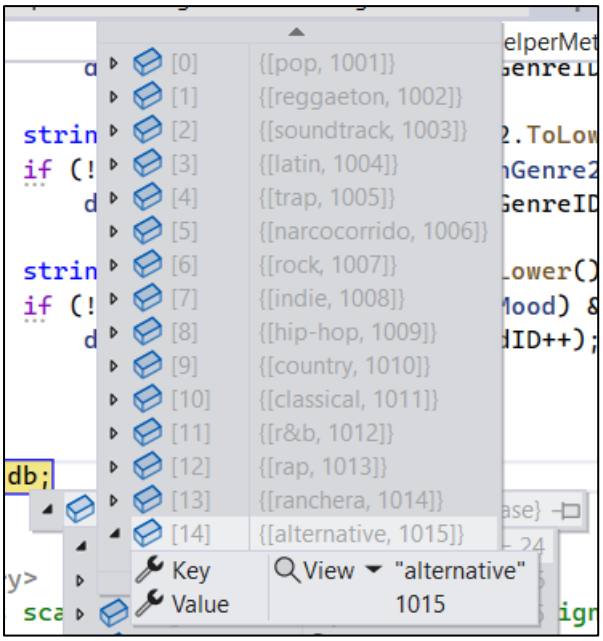

Figure 3 : Unique values assigned to genres in playlist

The initial values for all the attributes of songs are predefined in static variables. So, if it's a three-digit value we could accommodate 999 unique entries in database for the selected attribute. Thus, we fill all the attributes of the songs which can be seen in Listing 2.

```
public static Database FillDatabase(List<Playlist>
playlists)
{
   Database db = new Database();
   foreach (Playlist playlist in playlists)
   {
      foreach (Song song in playlist.Songs)
      {
         string cleanSongName =
song.Name.ToLower().Replace(" ", "_");
         if (!db.SongNames.ContainsKey
(cleanSongName) && !string.IsNullOrEmpty
(cleanSongName))
            db.SongNames.Add(cleanSongName,
SongID++);
:        :        :        :        :        :        :
         string cleanMood =
song.Mood.ToLower().Replace(" ", "_");
         if (!db.Moods.ContainsKey(cleanMood) &&
!string.IsNullOrEmpty(cleanMood))
            db.Moods.Add(cleanMood, MoodID++);
      }
   }
   return db;
}
```

Listing 3: Function to fill *Database*

In short, a database is a class holding dictionary of all the attributes of a song. Database is a very important class which is used later in scalarizing, configuring encoder, determining size of SDR, and decoding the predictions.

D. Scalarize the Playlist

For the experiment, the class of Database holds all the actual value which is filled in dataset and mapped scalar values. But these values need to be mapped to the song with their own attributes in given sequence. The model is *ScalarSong* which is similar to *Song* only holds the scalar values fetched from *Database*. And the model *ScalarModel* helps to map an actual song with scalar values and saves 1-to-1 mapping of *Song* with *ScalarSong*. As we have *ScalarSong* which is equivalent to *Song*, we have model *PlaylistScalarModel* which is equivalent to *Playlist*.

```
//equivalent to Song class
public class ScalarSong
{
    public int Name { get; set; }
    public int Singer1 { get; set; }
    public int Singer2 { get; set; }
    public int Genre1 { get; set; }
    public int Genre2 { get; set; }
    public int Mood { get; set; }
}

//one to one mapping of Song and ScalarSong
public class ScalarModel
{
    public int ID { get; set; }
    public Song Song { get; set; }
    public ScalarSong ScalarSong { get; set; }
}

//equivalent to Playlist class
public class PlaylistScalarModel
{
    public String Name { get; set; }
    public List<ScalarModel> ScalarModelList { get; set; }
}
```

Listing 4: Models of ScalarSong, ScalarModel and PlaylistScalarModel

The ScalarModel which saves one-to-one mapping saves an *ID* which is the unique value given to each song. This *ID* is later used to create a part of *Key* for learning by the algorithm in later parts of the experiment.

As referred further Figure 4 and Figure 5 is the to show the mapping of *Song* and *ScalarSong* with *ID* assigned in *ScalarModel*.

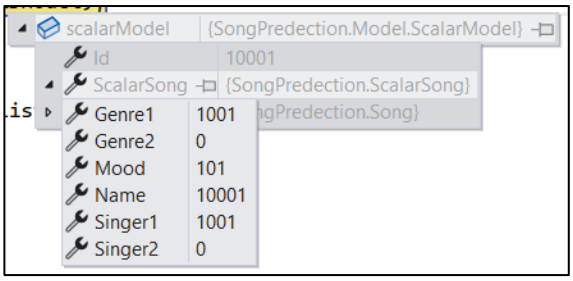

Figure 4 : Filled model of *ScalarSong* with attributes

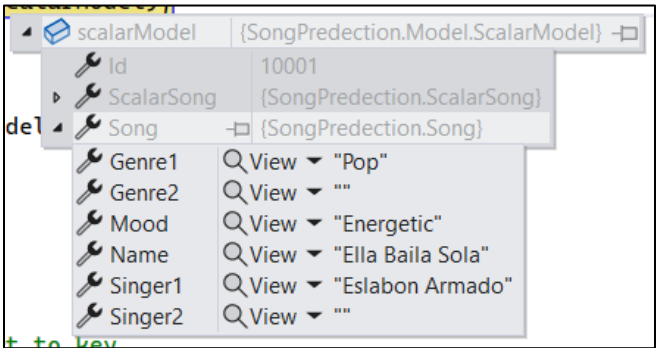

Figure 5 : Filled model an actual *Song* with attributes

As shown in

Figure 5, the *Song* is the smallest level of data structure which can used and manipulated. All the values from class *Song* are scalarized and stored in *ScalarSong* and if found any null values they are replace by 0 representing that it does not exists.

Referring in Figure 3 where all the genres are assigned unique value and Figure 5 the *Genre1* is "*Pop*" so the ScalarSong has scalar value of *Genre1* as "*101*" if compared back in Figure 3 and so on the for all the playlists and all the attributes of song is populated the same way. Thus, we obtain *ScalarSong* with one-to-one mapping saved for easy retrieval later.

E. Get the Encoder

An encoder is a mechanism to convert the data in a format which can be learned and understood by the experiment. Here we are using *ScalarEncoder* from Neocortex API, which converts numeric or scalar values to an SDR.

To encode a *Song*, we need to understand the features of song necessary for creating SDR and what features needs to be selected. There are some common features like singers, genre and mood but name is always unique which helps to achieve unique SDR for each song.

The encoders are created dynamically using counts of attributes, so the size of the *Dataset* does not matter here.

F. Encode the Playlist

As mentioned earlier, encoding is done using Scalar Encoder of NeoCortexApi and the output is SDR. But before we start encoding, we need to configure the Scalar

Encoder and understand how the multiple attributes of the songs come together to form and complete SDR.

The main features required by the Scalar Encoder are the size which will represent the attribute, minimum scalar value of the attribute and the maximum scalar value. All these values are dynamically set as per our dataset and usually calculated while creating class of Database. There are more features in the encoder but are of the least significant during our experiment.

Following is the encoded output for a random song during running the experiment in Figure 6.

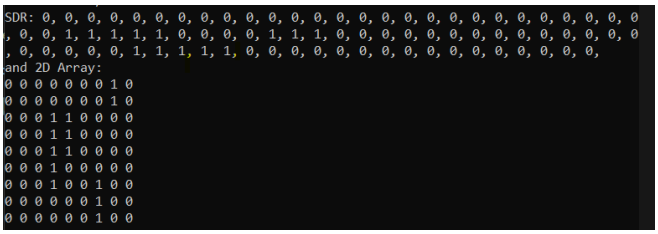

Figure 6 : SDR of encoded attribute of song

Now that we have configured the encoder, the SDRs are concatenated as Name, Singer1, Genre1 and Mood of the song in the respective sequence.

G. Run experiment with Multisequence Algorithm

The Multisequence Learning is based on HTM Classifier taken from the NeoCortexApi. In this algorithm, the connections of taken as per HTM Configuration. These configurations are responsible for the mapping to actual cortex of human brain. Spatial Pooler is used to create a sparse representation of encoded data and Temporal Memory is used to remember the sparse representation. There is also a Cortex Layer and HTM Classifier which gets trained model and predict the label.

Following is the algorithm for Multisequence Algorithm.

01. Get HTM Config and initialize memory of Connections
02. Initialize HTM Classifier and Cortex Layer
03. Initialize HomeostaticPlasticityController
04. Initialize memory for Spatial Pooler and Temporal Memory
05. Add Spatial Pooler memory to Cortex Layer
05.01 Compute the SDR of all encoded segment for multi-sequences using Spatial Pooler
05.02 Continue for maximum number of cycles
06. Add Temporal Memory to Cortex Layer
06.01 Compute the SDR as Compute Cycle and get Active Cells
06.02 Learn the Key with Active Cells
06.03 Get the input predicted values and update the last predicted value depending upon the similarity
06.04 Reset the Temporal Memory
06.05 Continue all above steps for sequences of multi-sequences for maximum cycles
07. Get the trained Cortex Layer and HTM Classifier

The Key which is learn with active cells is most important piece which is remember while learning happens. The has name of the playlist and ID of the songs which is generated during the scalarizing of playlist is done. Thus, it looks like name of playlist plus a chain of IDs of song in order which they appear in a playlist. Figure 7 is for better visualization of the Key.

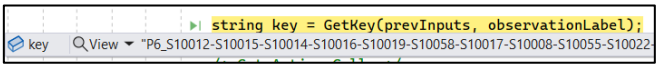

Figure 7 : Key during one of the cycles while running experiments

H. Prediction using model and decoding prediction

The object of *CortexLayer* has the learned combination of key and the SDR (Sparse Distributed Representation). The object of *HtmClassifier* is learned prediction model which is responsible for prediction of possible value.

The predicted output is *Key* and needs to be interpreted as per the inputs which were given so a decoding is necessary before publishing results. The predicted key holds the playlist number and the from the chain of songs the last appearing song should be picked up as that's the song which appears at the end of playlist, and it also matches the key when the algorithm learned the relationship between key and active cells.

To predict something, we first need an input *Song* from our collection of songs. At random the experiment chooses a *Song* for input and encodes them using the same encoder used to encode the *Playlist*. This encoded input is then passed to the learned model as the result of algorithm and gives us the encoded result as *Key* which it learned. The predicted result is broken down as *Playlist* N*ame* and *Song ID*, which is mapped to ID from the *PlaylistScalarModel* and *ScalarModel* to get the actual *Playlist* and *Song* which has the one-to-one mapping saved.

IV. RESULTS

The experiment was conducted several times with different *HTMConfig* which affects the learning process and tries not to over fit the model. Each sequence was learned in 50, 100, 200 cycles which gave different predicting results. To measure the results and accuracy of predicting correct next song in the playlist, a set of random *Song* were selected with the subsequent *Song* of the *Playlist*. If the predicted *Song* matches the subsequent *Song*, then it's a correct match and accuracy per prediction is added up.

In this experiment, *Key* created from the song in a playlist holds significance important for learning the sequence. So, each playlist has been given a unique name and each song holds a unique entry in *Database*. The key is created as *'{Playlist.Name}-{Song*W*}-{Song*X*}-{Song*Y*}-{Song*Z*} and so on'* adding the *Song ID* at end of *Key* and learn with a SDR of a last song in the key.

Once the learning part is done, a set *Song* is the prediction is executed on it. On the result of the prediction,

the *Key* is mapped back after looking in *ScalarModel* and *Song* is displayed. Multisequence Learning algorithm's model has theoretically 100% accuracy [6] when ran with 200 cycles. It was successfully able to learn 5000 songs has given 93.1% of accuracy when a set of 30 songs were randomly chosen and the results were compared and matched with the immediate next song in sequence/playlist.

Figure 8 shows the final output from one of runs of experiment.

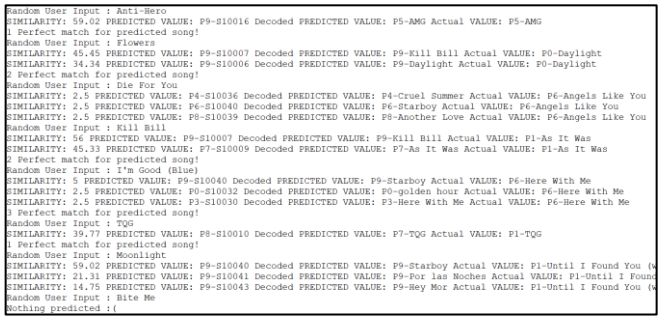

Figure 8 : Random user input and predicted output

V. DISCUSSIONS

Learning and predicting sequences have been experimented with for decades and various methods have been used. Adapting an existing method for large and noisy data remains a challenge. In the experiment, *HtmClassifier* is used which is a recently developed neural network based on cortex of human brain and not just a single neuron model.

Following are couple of improvements which can be done to further refine the experiments:

A. Consider null values while learning

When we consider null values for each attribute, we add one more tuple and this can be filled when we do not know the value, or if the value is null.

For example: Usually a song has a primary singer and secondary singer but not all songs have a known singer, or the artist might be unknown. So, in such scenarios we can assign a value which adds one more tuple and increases some meaning while learning for unknown attributes.

B. Consider all significant possible parameters for inputs

This improvement comes from the previous improvement where null values should be filled with data; this helps to use new attributes while learning.

For example: Usually a song has a primary singer and secondary singer but not all songs have them, so in such cases we can have secondary singer a null and consider singer 2 as attribute while learning.

C. Create larger dataset to improve input sequences

The size of the dataset has always been a significant challenge to deal with. There are a couple of ways to tackle it.

1) Synthetic data

One can create a playlist by randomly choosing songs from a large data set of songs. The downside to this is that the learning algorithm ends up learning the randomness of data instead of real-world relationship between two songs.

2) Scrapping web data

This one of the easy methods where web scrappers can be used to scrape the playlist from music related web application hosted online. To make it even quicker, one can request playlists from generative AI.

3) Data from user

This method is most genuine method and most time consuming too. Here we need to gather data from real-time users where the user creates their own playlist in application, and we add that data while continuous learning.

VI. REFERENCES

[1] J. Hawkins, "numenta.com," Numenta, 12 September 2011. [Online]. Available: https://numenta.com/neuroscience-research/research-publications/papers/hierarchical-temporal-memory-white-paper/. [Accessed 22 March 2023].

[2] S. A. &. J. Hawkins, "arxiv.org," Arxiv, 25 March 2015. [Online]. Available: https://arxiv.org/abs/1503.07469. [Accessed 22 June 2023].

[3] S. purdy, "arxiv.org," Arxiv, 18 February 2016. [Online]. Available: https://arxiv.org/abs/1602.05925. [Accessed 22 June 2023].

[4] K. j. H. &. S. Ahmad, "link.springer.com," SpringerLink, 20 July 2021. [Online]. Available: https://link.springer.com/article/10.1007/s42452-021-04715-0#Sec14. [Accessed 22 March 2022].

[5] "Dataset:https://github.com/anxods/spotify-top-50-songs/blob/main/data/spotify-streaming-top-50-world.csv".

[6] "NeoCortexApi : https://github.com/ddobric/neocortexapi".