**Multiple Sequences Learning**

The existing multisequence learning application is working with learning/understanding of sequences and the matching sequences, but the application completely works with the hardcoded input sequences.

**Team\_MSL** will analyse and work on taking the input sequences from any config file can be in any format (e.g. .txt, .csv, .xml, .json, .excel).

Team\_MSL will also work on increasing the efficiency of the code by comparing the different approaches for file operations, along with team will also work on sub sequences matching and prediction.

**Names of Team\_MSL members:**

• Poonam Dashrath Paraskar

• Pratik Desai

• Ankita Talande

**Strategy Team\_MSL is following for code changes and merge is as following:**

1. All the team members will commit their changes in their own feature branch and raise PR for merging those changes into Team\_MSL branch.
2. After reviewing the PR changes team members will merge those changes into Team\_MSL branch.
3. All the team members must take pull of latest Team\_MSL branch before committing any changes and before raising PR.
4. Team members will have to resolve merge conflicts (if any) before raising PR by their own.
5. Once we are done with the development and the Team\_MSL branch is up to date with all the code changes then we will raise PR from Team\_MSL branch to master branch.

**Highlighted Points -**

1. The Hierarchical Temporal Memory (HTM)
2. Spatial Pooler
3. To optimize the sequence learning behaviour of spatial pooler and temporal memory layer in dependence on HTM Sparsity
4. The sequence learning behaviour of spatial pooler and temporal memory layer in dependence on learning parameter-Cells per Column
5. Improve HTM Spatial Pooler with Homeostatic Plasticity Control

**List of tasks done by Team\_MSL per sprint:**

**Sprint 1:**

1. Team\_MSL - forked the Neo-cortex repository.
2. Created new repository out of existing neo-cortex and named as neocortexapi\_Team\_MSL
3. All team members have created their own branches and one common branch for final code changes merge.

**Sprint 2:**

1. All team members analysed the existing multisequence learning project and tried to run it on their local visual studio code.
2. Resolved the build failures and able to run the program and view the output locally.
3. Created a Readme.md file and added basic description about what are all the tasks each team member will perform.

**Sprint 3:**

1. Team started actual development in this sprint and worked on taking the input sequences from local text file.
2. Created new method in Program.cs for eliminating the hard coded inputs and getting from local text file.
3. Able to debug the flow with new input sequences.
4. Team also worked on logic for splitting different sequences.

**Sprint 4:**

1. Created new method – GetInputFromTextFile
2. Implemented new logic for input sequence splitting and testing

**Sprint 5:**

1. Split input sequences using regular expression.
2. Written the logic for subsequence inputs.
3. Created new method – GetSubSequenceInputFromTextFile

**Sprint 6:** This can be considered as last sprint.

1. Team has implemented 2 new methods GetInputSequenceFromCSVFile().
2. Compared the performance and implementation strategy of newly implemented method with the existing one.
3. Team has implemented another method for taking input sequences GetInputSequenceFromExcelFile(), to minimize the issues of above 2 methods which has been elaborated briefly below in descriptive manner.

**Testing Phase:**

1. Team has done performance testing by comparing the output for matching percentage of the test sequence and for predicting next element of the sub sequence.
2. Team has also compared the required execution time both legacy code and newly implemented changes.
3. Test execution report can is attached in following link: [https://docs.google.com/spreadsheets/d/1DefOwD5Xcg0SZ9lGAKfDWmm9pbE3M-xHERitR8tOLTg/edit#gid=0](https://docs.google.com/spreadsheets/d/1DefOwD5Xcg0SZ9lGAKfDWmm9pbE3M-xHERitR8tOLTg/edit%23gid=0)

**Hierarchical Temporal Memory**

**Encoder:**

Encoder is chosen according to the type of the inputs. There are some encoders available for popular input type:

* 1. Scalar Encoder
  2. Datetime Encoder
  3. Boolean Encoder
  4. Category Encoder
  5. Geo-Spatial Encoder

In this project we are using - **Scalar Encoder**

**Scalar Encoder:** It is one of the encoding techniques and is a part of Hierarchical Temporal Memory (HTM). HTM is a machine intelligence technology which is trying to imitate the process and architecture of neocortex. The main purpose for scalar encoder is to encode numeric or floating-point value into an array of bits, where the output has 0’s with an adjacent block of 1’s. The location of the block of 1’s varies continuously depending on the input value.

The scalar representation of value (e.g., for categories, this is the internal index used by the encoder). This number is consistent with what is returned by **[getScalars()](https://nupic.docs.numenta.org/1.0.3/api/algorithms/encoders.html" \l "nupic.encoders.base.Encoder.getScalars" \o "nupic.encoders.base.Encoder.getScalars)**. This value is always an int or float and can be used for numeric comparisons.

HTM consists of 2 different components: Spatial Pooler and Temporal Memory.

1. Spatial Pooler -

Encoder produces output to be fed into Spatial Pooler algorithm. Type of Spatial Pooler (SP) that is used in this example is the multithreaded version that utilize multicore of the machine to run the spatial pooler algorithm.

SpatialPoolerMT spatialPooler = new SpatialPoolerMT(hpa);

patialPooler.Init(memory, UnitTestHelpers.GetMemory());

1. Temporal Memory -

The output of Spatial Pooler (SDR) is used as the input of Temporal Memory.Temporal memory algorithm will then learn the temporal pattern from spatial pattern.

TemporalMemory temporalMemory = new TemporalMemory();

temporalMemory.Init(mem).

Team Implemented three methods to take inputs

1. GetInputFromTextFile( )
2. GetInputFromCsvFile( )
3. GetInputFromExcelFile( )

We implemented three methods to check which file type is convenient and fulfil our requirements in terms of speed, functionality and use case. We took real time approach into consideration so system performs operation smoothly in real world scenario.

1. **GetInputFromTextFile( ):**

Team has implemented *GetInputFromTextFile( )* method to take the inputs from the Text file.

We have tried 2 approaches to split the multiple input sequences:

1. By using comma ‘,’ to separate each digit of the input sequence and using special character at the end of each sequence for splitting it from other input sequences. In this case we used semi-colon ‘;’ to split.

The significant issue we faced by using this approach is we had to add both comma ‘,’ and semi-colon ‘;’ at the end of each input sequence, which is not a feasible solution and by which text file also looks inappropriate.

Following image shows the inputs in Text file format. Where highlighted part shows addition of special character.

Text

Description automatically generated with medium confidence

1. To resolve issue, we faced in the first approach we used regular expression to split multiple sequences based on detecting the enter ‘/r/n’. Using this approach wherever we added enter for next input, is getting detected by our regular expression logic.

For this we had to read all the rows together using ‘reader.ReadToEnd()’ method and then split it by detecting the enter keyword.

This can cause an issue in real time working environment.

Following image shows the input in Text file format.

Graphical user interface, application

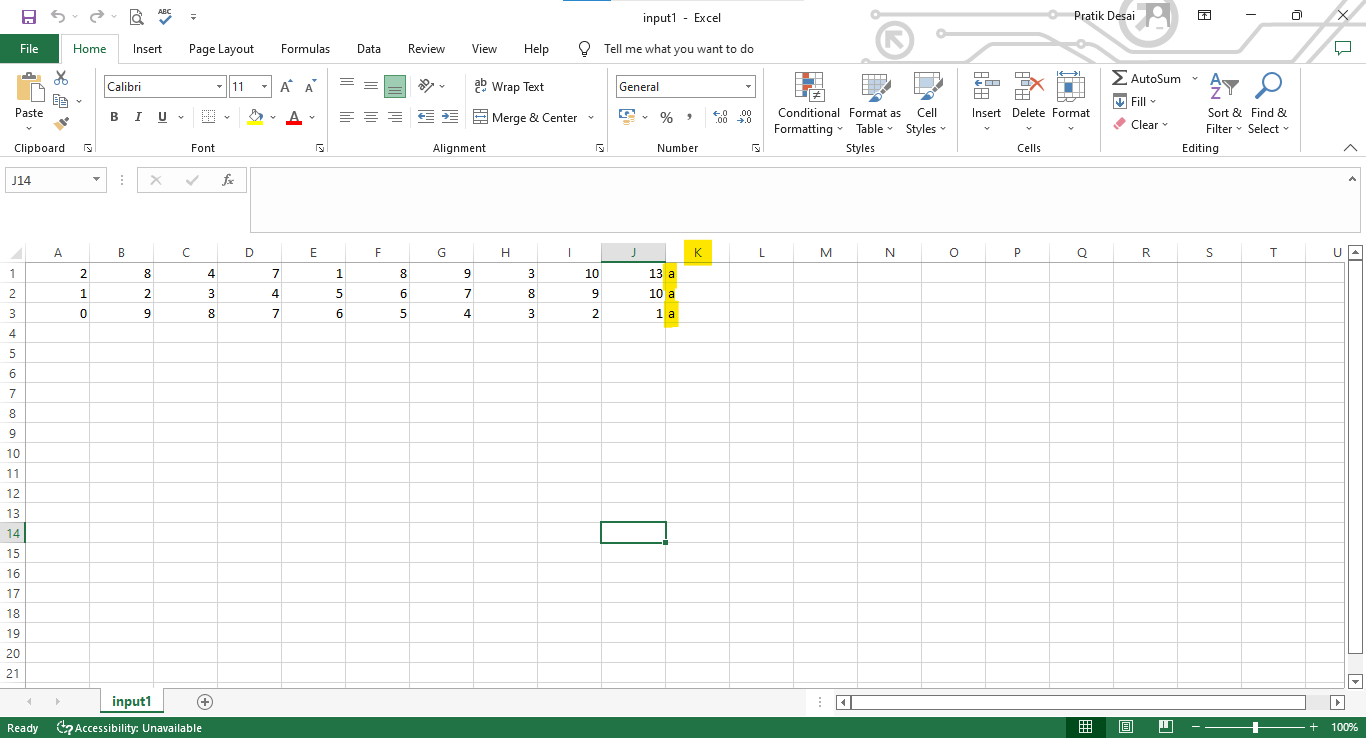
Description automatically generated

1. **GetInputFromCsvFile( ):**

Team has implemented *GetInputFromCsvFile( )* method to take the inputs from the CSV file. CSV stands for "Comma-Separated Values". It is a file format used for storing and exchanging tabular data, such as spreadsheets or databases. In a CSV file, each line represents a row of data and each field within a row is separated by a comma. CSV files are simple and widely supported, making them a popular choice for data exchange between different systems and applications.

The problem with CSV file is we need to add one non double character at end of each row to terminate the row/sequence and take the next sequence. This can cause an issue in real time working environment.

Following image shows the inputs in CSV file format. Where highlighted part in column ‘K’ works as a terminator for the sequence or row.

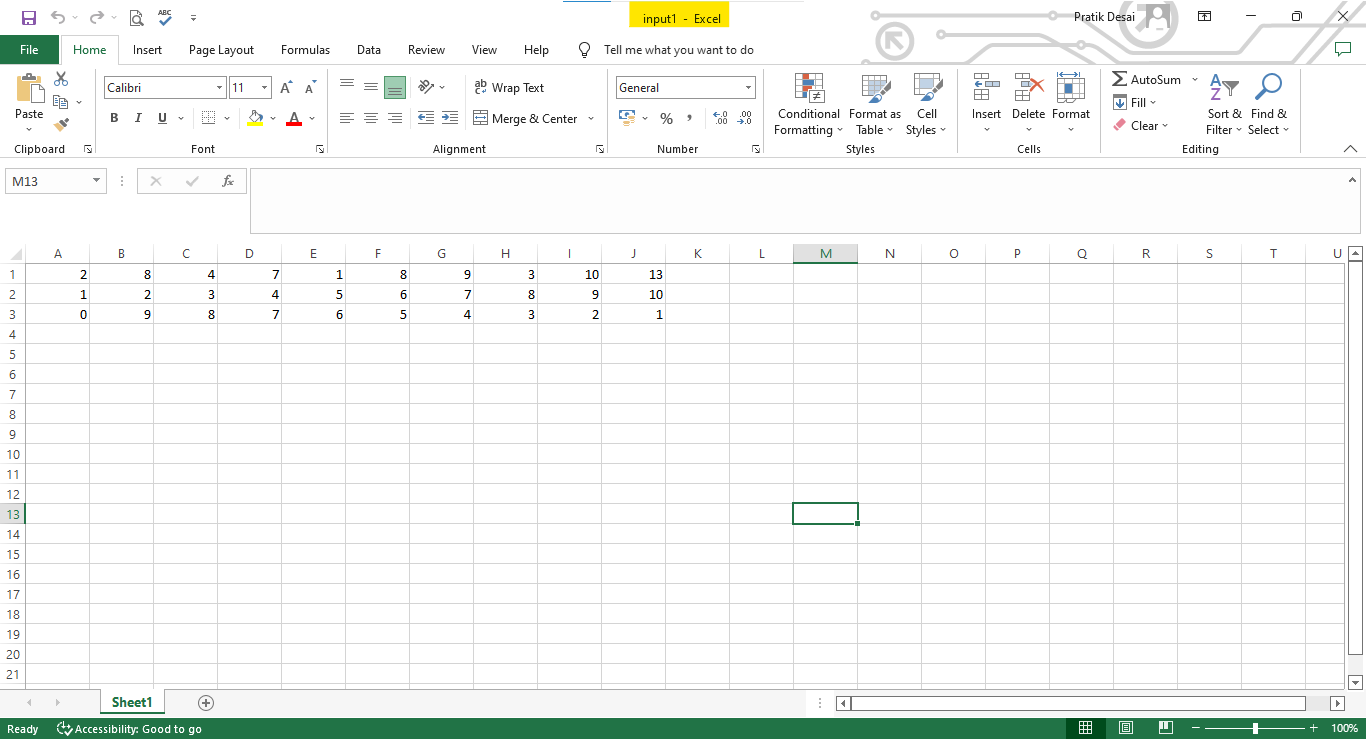


1. **GetInputFromExcelFile( ):**

In the *GetInputFromExcelFile( )* method we are using .xlsx file type to take the input sequences.

Here we overcame the issues of the previous methods *GetInputFromCsvFile()* and *GetInputFromTextFile()* where we need to add any non-double value to terminate the row/sequence and to jump to the next row/sequence and any special in case of text file to jump over the next input sequence. To implement this feature we used the *string.IsNullOrWhiteSpace( )* property.

Following image shows the input taken from .xlsx file.



This is the most efficient method and can be used in real time scenarios. Here we don’t need to add any special character as terminator. It automatically detects the empty cell and considers it as an end of row/sequence.

**Additional Nugget Packages:**

We used few third-party nugget packages to implement this feature. Following table shows the details of nugget packages we used in the code.

| **Nugget Package** | **Version** |
| --- | --- |
| EPPlus | 6.1.3 |
| ExcelDataReader | 3.7.0-develop00310 |

**File Path:**

Getting file path automatically for the input.xlsx and subsequence.xlsx files is necessary because whenever team member or any user tries to run the code into their local environment they need to add the paths of the file explicitly to take the files as a input. To overcome this issue, we implemented property *Environment.CurrentDirectory* to take the current directory where the code debug and takes the input files. This solved our issue of adding path manually in everyone’s local environment.