**Project Report**

Title:

**Tag-to-Tag Routing Algorithm Implementation using Python and Machine Learning**

Prepared By:

Sourabh Tantuway

MSCS at The University of Texas at Dallas

Spring 2023 semester

Instructed By:

Dr. Zygmunt J. Haas

Professor at The University of Texas at Dallas

**Overview:**

This project is based on the research paper by Dr. Zygmunt J. Haas, which deals with the idea of passive tags communicating with each other in a large-scale network. While the research paper provides an extensive study about how T2T communication results in significantly larger capacity than centralized RFID reader-based system, this project is just an effort to practically implement the basic routing algorithm mentioned as “*Algorithm 1:**Routing Protocol for Scenario A*” in the paper. Kindly go through the paper to better understand the project.

*Research paper link: (*[*https://ieeexplore.ieee.org/document/9146888*](https://ieeexplore.ieee.org/document/9146888)*)*

**Tools and Technologies Used:**

In this project, I used *Python* language and some *Machine Learning libraries* for the implementation of the algorithm. Entire project can be run using the python notebook file (i.e. *Project.ipynb*). In my case I used *Jupyter Notebook* server locally and ran the .*ipynb* file on that.

**Assumptions:**

* For making things simpler, in this project we assumed that each tag cannot have more than one message to transmit.
* Weights of all messages will be same.

**Implementation:**

The project consists of *Project.ipynb* file, which has the main code and some additional *csv* files having some data.

Python code generates the initial arrangement of RF tags in a 2D plane. Specifically, **20** random points are selected inside a **15x15** grid. Number of points and grid dimensions can be set in the code. Arrangement used for this project is shown below.

A picture containing text, diagram, line, plot

Description automatically generated

After positions of tags are decided, distance between each pair of tags is calculated. These distances will be stored in a matrix. This *distance matrix* can be stored locally so that this arrangement of tags can be reused later to perform analysis. In our case, I stored one specific arrangement in a *csv* file (*dist\_table.csv*) and entire project analysis is done over that specific arrangement only.

Now the *Radius* of transmission is selected, which is same for all tags. *Radius* selection is subject to *distance matrix*, such that each tag is having at least one other tag inside it’s radius. More specifically, If the distance between a pair of tags is less than or equal to the *radius*, there is an edge between them. So, select a radius such that connected graph is formed where tags are the nodes. In my case, I selected the radius value *8.00*. Graph of tags is shown below.

A picture containing diagram, line, plot, text

Description automatically generated

Now the idea is that if some number of tags want to transmit their messages to their respective destinations. The *Algorithm1*, tells which tags will transmit finally, such that there will be minimum interference between tags.

*Algorithm1* uses constraints to decide which tags can’t transmit while some other tag is transmitting. That is avoiding the interference between tags.

Python code generate an “*Instance*” of tags. By instance we mean number of candidate tags which want to transmit. And corresponding “*Solution*” of the instance, which is final tags selected for transmission. *Instance* and *Solution* are vectors of size equal to number of nodes, i.e. 20.

In *Instance* vector value *a* at index *i,* means *node i* wants to transmit to destination *node a*.

In *Solution* vector value *0/1* at index *i*, means node *i* is finally selected for transmission.

Examples of *Instance*(left) and *Solution*(right) shown below.

A picture containing diagram, line, text, plot

Description automatically generatedA picture containing diagram, line, plot, text

Description automatically generated

*[16,19,7,15,7,17,5,13,11,14,0,15,0,0,16,0,6,4,0,11] [0,1,0,0,1,1,0,0,0,1,0,1,0,0,0,0,0,1,0,1]*

With python code we can generate as many rows as we want, of the *instances* and corresponding *solutions* given by the *Algorithm1*. We use *generate\_data ()*function for generating the dataset. It is the main function which uses all other functions used in the *Algorithm1*. This function takes single parameter which is the total number of rows we want to generate.

In my specific arrangement of tags, I generated *90k* rows using this function and saved the dataset locally. And used 80% of this dataset to train a machine learning model.

This model can predict the solution for any given instance with a certain amount of accuracy. In this project I was able to achieve nearly *70%* accuracy for the ML model.

Steps to generate the dataset and train the model is given in ‘Running Instruction’ section.

**Running Instruction:**

Python notebook file should be executed in sequence, from starting to end.

Some dependencies might be required to run the code successfully. They can be installed as you run the code.

Before executing the python file make sure if the csv files are in place:

* *dist\_table.csv* should be in the same folder as *Project.ipynb*
* folder named ‘*dataset’* should also be in the same folder as *Project.ipynb* file. This folder contains *dataset.csv* used for training and testing the ML model.

*dist\_table.csv* contains specific arrangement of tags for which ML model will be trained.

*dataset.csv* contains the 90k rows of instances and solutions. These rows correspond to the tag arrangement stored in *dist\_table.csv*.

Now change the path of *dataset.csv* in the code and execute all the cells. The model will be trained, and the accuracy of model will come nearly *70%.*

New arrangement of tags can be generated and stored in *dist\_table.csv* and new dataset can be generated using *generate\_data(num\_of\_rows)* function and saved in *dataset.csv.* If done so, ML model will be generated for this new data and accuracy will be according to the amount of data generated.

**Challenges:**

I faced some main challenges in preparing the data for the classifier. We are predicting a solution for an instance which is a vector representing the number of nodes. This type of problem comes under multilabel classification problem. So, preparation of data took some time, which involved making 20 separate features for our 20 nodes arrangement. Also, there are 20 labels corresponding to our 20 nodes. If *label\_1* has value 1, it means node 1 is selected for transmission, otherwise value will be 0. After preparation of data applying the model was relatively easier.

**Results:**

I used *DecisionTreeClassifier* from *sklearn* library. *MultiOutputClassifier* classin *sklearn* helped dealing with multi label classification and supported vector type prediction in our case. I divided all *90k* data rows in *training* and *testing* set *80%* and *20%* respectively. Trained the classifier over *training* set and tested over *testing* set to achieve ***70%* accuracy**.

**Future Work and Improvement:**

There is a scope of improvement in the current project as mentioned below:

* Other ML classifiers can be explored which will support our type of dataset.
* Hyperparameter tuning can be done over current model to see if higher accuracy can be achieved.
* Optimization can be done in current code.
* Accommodating code to allow more than one message for a single tag.
* Allowing weights/priorities of messages.
* Implementing *Algorithm2* and *Algorithm3* mentioned in the paper.

**Iteration 2:**

**Understanding the accuracy:**

If any solution has more than one incorrectly selected node, the whole solution is discarded.

Only the ones with all nodes selected correctly are contributing to the accuracy.

**Overfitting and hyperparameter tuning:**

It turned out that our model was overfitting because training accuracy was 100%. So, we performed hyperparameter tuning over our model.

First we analyzed the decision tree being formed to check it’s max-depth, min-samples-leaf, min-samples-split. These are the hyperparameters of the decision tree.

Now using sklearn’s GridSearchCV, we figured out best hyperparameters as shown below.

A screenshot of a computer

Description automatically generated with low confidence

Using these parameters in our model on the same dataset, we got higher test accuracy of 75%. Training accuracy also got reduced to 83%. It means model is not overfitting.

A screenshot of a computer code

Description automatically generated with low confidence

**Relation of accuracy with amount of training data:**

As figured, increasing the amount of training data also increases the accuracy. We calculated train and test accuracy of model by providing different amount of training data and plotted a graph as shown below:

A picture containing text, line, screenshot, diagram

Description automatically generated A picture containing text, line, diagram, plot

Description automatically generated

**Distribution of incorrect nodes in all incorrect predictions:**

We want to see in all the wrong predictions made by model, how many incorrect nodes are there. Some predictions can have 1 incorrect nodes , 2 incorrect nodes and so on. This distribution can be represented by the graph shown below:

A picture containing text, diagram, screenshot, plot

Description automatically generated

Test data rows : 21118

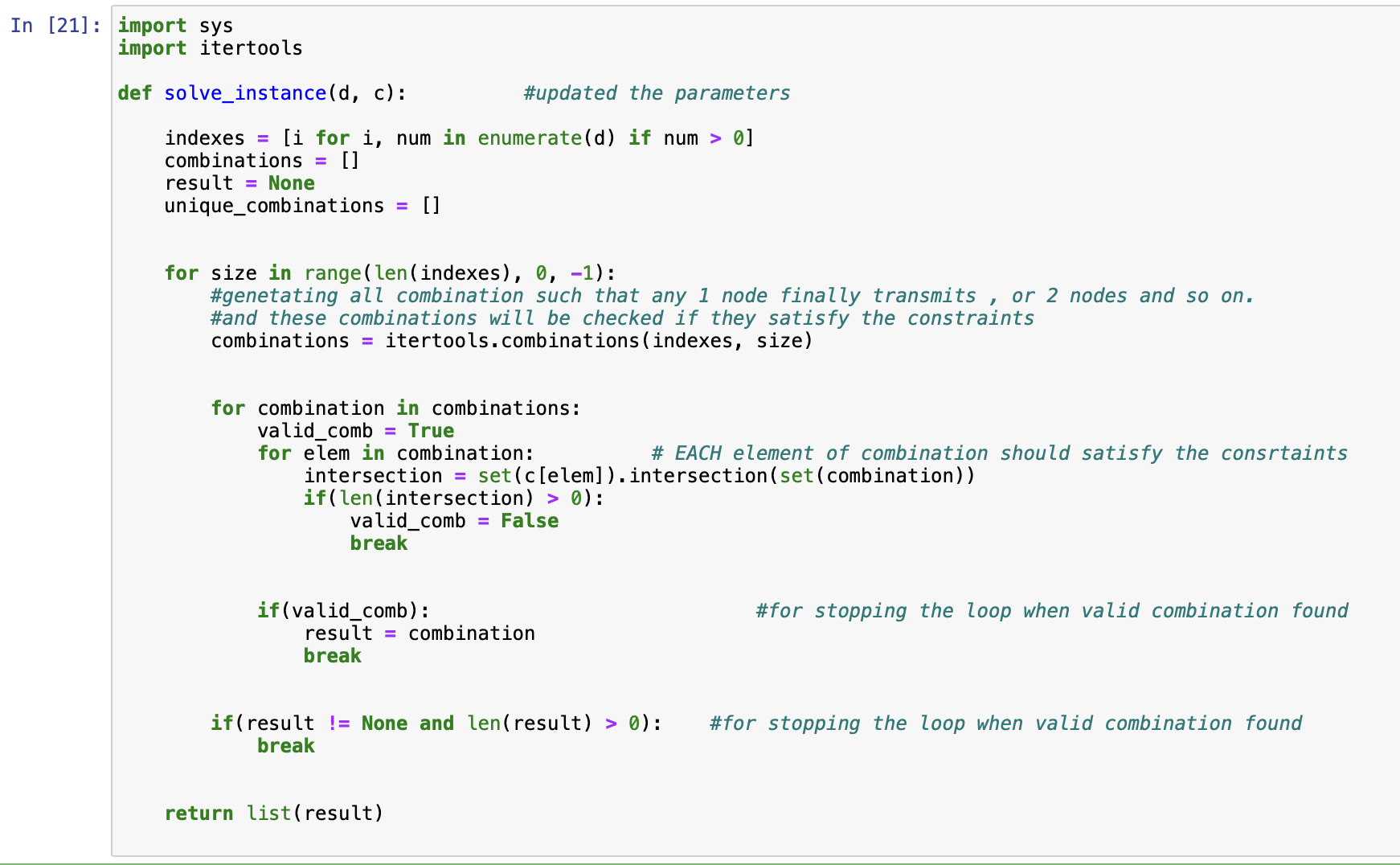
Incorrect predictions : 5032

Around 2800 predictions has only 1 incorrectly selected node

**Optimization in generating Datasets:**

Previously it was taking way too longer to generate the dataset for some specific arrangement of tags. For example, for generating 20k rows it was taking whole day. Now the code has been changed. There were some unnecessary conditions which are removed now. Also, loop was running unnecessarily which is corrected and break statements are added.

Following is the optimized code for solve\_instance() function which mainly decided the running time:

[](https://user-images.githubusercontent.com/26655938/261076579-92a2c992-bdc7-46fb-858e-62c66c09c4a4.png)

**Hoeffding Tree for Incremental Learning:**

As we want to achieve dynamic incremental learning for our case, we are updating our classifier to incremental classifier which preserves old information as well as keeps on learning from new incoming data from stream or batch. One such classifier is HoeffdingAdaptiveDecisionTree from river library.

* classifier works well with dictionary datatypes.

**Handling case when arrangement of tags changes:**

If tags are moved. In this case new data generated will be different from the previous arrangement. So our classifier should detect the change in data and train itself according to the new data. So drift detection mechanism is to be used.

A screenshot of a computer code

Description automatically generated

Aug 21, 2023 & Aug 22, 2023:

Task it to check the effect on the accuracy of the model when it is trained by one type of data and later incrementally trained with slightly different type of data. In our case training the classifier for first arrangement of tags and later for second arrangement of tags and check the accuracies.

For this, I generated 5 *training datasets* for arrangement\_1, each having 20k rows:

1. X\_set1\_1, y\_set1\_1
2. X\_set2\_1, y\_set2\_1
3. X\_set3\_1, y\_set3\_1
4. X\_set4\_1, y\_set4\_1
5. X\_set5\_1, y\_set5\_1

A screenshot of a computer

Description automatically generated

And 5 datasets for arrangement\_2 :

1. X\_set1\_2, y\_set1\_2
2. X\_set2\_2, y\_set2\_2
3. X\_set3\_2, y\_set3\_2
4. X\_set4\_2, y\_set4\_2
5. X\_set5\_2, y\_set5\_2

A screenshot of a computer

Description automatically generated

Along with above , 1 *test dataset* for each arrangement, having 50k rows:

A screenshot of a cell phone

Description automatically generated

Now trained the classifier with arrangement\_1(incrementally with all 5 training sets) and then arrangement\_2. And plotted the graph of accuracy:

A screenshot of a computer program

Description automatically generated

1.

A graph of a number of numbers and a line

Description automatically generated

Now, resetting the classifier and reversed the order of training.

Now trained the classifier with arrangement\_2 and then arrangement\_1. And plotted the graph of accuracy:

2.

A graph with numbers and lines

Description automatically generated

Repeated above two steps again:

3.

A graph of a line

Description automatically generated with medium confidence

4.

A graph of a line

Description automatically generated with medium confidence

**Observation**: In plot 1 and 3, it appears that the classifier is not able to persist the previous learning, as accuracy is declining

In plot lot 2 and 4 accuracy later, accuracy is higher. Thus, ambiguity is there.

Aug 23, 2023:

More research about how to persist previous learning of classifier in case the dataset changes.

<https://machinelearningmastery.com/transfer-learning-for-deep-learning/>

[https://builtin.com/data-science/transfer-learning](https://builtin.com/data-science/transfer-learning%20)

[https://neptune.ai/blog/concept-drift-best-practices](https://neptune.ai/blog/concept-drift-best-practices%20)

https://en.wikipedia.org/wiki/Catastrophic\_interference#:~:text=When%20such%20deep%20generative%20models,than%20at%20the%20input%20level.

Aug 24, 2023:

Analyzed how accuracy varies:

Results: if dataset changes, classifier forgets the older learning and gradually learns the new dataset.

If I train the classifier with first dataset and reaches maximum accuracy of 53 %, and then train with a part of second dataset accuracy goes down 19% for older examples but for new examples it goes up from 0% to 34% and it gradually increases till max accuracy of 47% if we keep on training with new dataset(second one)

X\_set\_1 => [ .**53**, .**34**(.**19**), **.39, .41, .43, .47**]

Similarly, if we reverse the dataset and train it with second dataset first.

X\_set\_2 => [ .**48**, .**39**(.**25**), **.44, .49, .51, .54**]

Aug 25, 2023:

Research about Evolving classifiers

Reference : https://arxiv.org/pdf/0709.3965.pdf

Aug 28, 2023 & Aug 29, 2023:

Ran demo tutorial and research about Continual Learning

References :

* <https://github.com/ContinualAI/colab>
* <https://www.continualai.org>
* <https://wiki.continualai.org/the-continualai-wiki/introduction-to-continual-learning>

Scope of Neural network in previous memory utilization:

Reference:

* <https://www.coursera.org/learn/advanced-learning-algorithms/>
* <https://medium.com/@satnalikamayank12/on-learning-embeddings-for-categorical-data-using-keras-165ff2773fc9>
* <https://www.simplilearn.com/tutorials/deep-learning-tutorial/neural-network>

Algorithm for managing memory outside classifiers:

1. Train the classifier with, say, 100,000 samples of scenario 1.
2. Obtain the solution as the vector ***x1***
3. Reset the classifier
4. Train the classifier with, say, 100,000 samples of scenario 2.
5. Obtain the solution as the vector ***x2***
6. Calculate: ***x****3=α∙****x1+(1-α)∙ x2*,**starting with ***α=****0.1*
7. Obtain the accuracy of the ***x****3* solution for scenario 2
8. Plot on the graph the accuracy as a function of ***α***
9. Repeat steps 1) through 8) for  ***α***changing from 0 to 1, in steps of 0.1 . You will end up with 11 points on the graph.

Aug 30, 2023:

We have 2 training datasets for tag **scenario\_1**[X\_set\_1(instances), y\_set\_1(solutions)]and **scenario\_2** (X\_set\_2, y\_set\_2). Which we will use for the training of classifier sequentially.

Training dataset looks like following:

A screenshot of a calendar

Description automatically generatedA white background with black numbers

Description automatically generated

(nodes want to transfer to the destination) (solution)

We are considering a **test** dataset(**X\_test\_2**, **y\_test\_2**) which is generated for scenario 2. Over this we will be testing our combined prediction of both classifiers. This dataset has 50k rows.

Training the classifier for scenario 1:

A screenshot of a computer code

Description automatically generated

Finding accuracy for this classifier for some unseen test dataset of scenario 1:

A screenshot of a computer code

Description automatically generated

Storing solution for scenario\_1 in **y\_pred\_1** (**x1**). Having 50k rows. **y\_pred\_1** stores predicted solution by classifier 1 for our **test** dataset(**X\_test\_2**, **y\_test\_2**).

A screenshot of a computer

Description automatically generated

Resetting the classifier:

A white box with green and purple text

Description automatically generated

training again for scenario 2.

A screenshot of a computer code

Description automatically generated

Accuracy for unseen data of scenario 2:

A screenshot of a test

Description automatically generated

Note: Accuracy is lesser for scenario 2 but it is different issue which I am looking concurrently. But for now considering 56% is the highest accuracy for next arrangement.

Storing solution for scenario\_2 in **y\_pred\_2** (**x2**). Having **50k** rows:

A screenshot of a computer screen

Description automatically generated

Now, calculating **x3** for **alpha** values [0.1 – 0.9] , storing nine x3 sets for each alpha namely [**y\_pred\_3\_1** to **y\_pred\_3\_9**]

Displaying **y\_pred\_3\_1 , y\_pred\_3\_5 and y\_pred\_3\_9:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Storing accuracy calculation for each x3 in a vector **accuracy\_vec** comparing with actual solution y\_test\_2.

A screenshot of a computer program

Description automatically generated

Plotting graph of accuracy against **alpha** values:

A screen shot of a graph

Description automatically generated

Aug 31, 2023 & Sep 01, 2023

Revisiting the accuracy calculation:

Unlike previous accuracy calculations, we are checking fraction of nodes classifier is predicting correctly for each solution. Then taking average of all calculated fractions for the prediction set.

Definition of the function **accuracy\_function()** responsible of calculating accuracy is fillowing.

A screenshot of a computer code

Description automatically generated

Example:

Classifier training over some training data X\_set\_3 , y\_set\_3

A computer code with black text

Description automatically generated

Older accuracy calculation:

Accuracy over unseen data **X\_test\_3**(instances) **, y\_test\_3**(solutions)

A screenshot of a cell phone

Description automatically generated

New accuracy calculation:

First taking out predictions of the classifier over unseen instance set **X\_test\_3.**

A screenshot of a number

Description automatically generated

Calculating accuracy using our function, with prediction and actual solution set.

A screenshot of a computer

Description automatically generated

Here, accuracy value is much higher than before, because before even 1 bit in the solution was wrong whole solution was considered wrong. Now, instead we are considering number of bits correctly predicted.

Analyzing accuracy behavior when tags move:

I generated other scenarios from scratch and moved some nodes and for both scenarios’ accuracies are nearly same.

Maybe the inconsistency in accuracies were for one particular combination of arrangement, which I am looking further into.