

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>)

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.

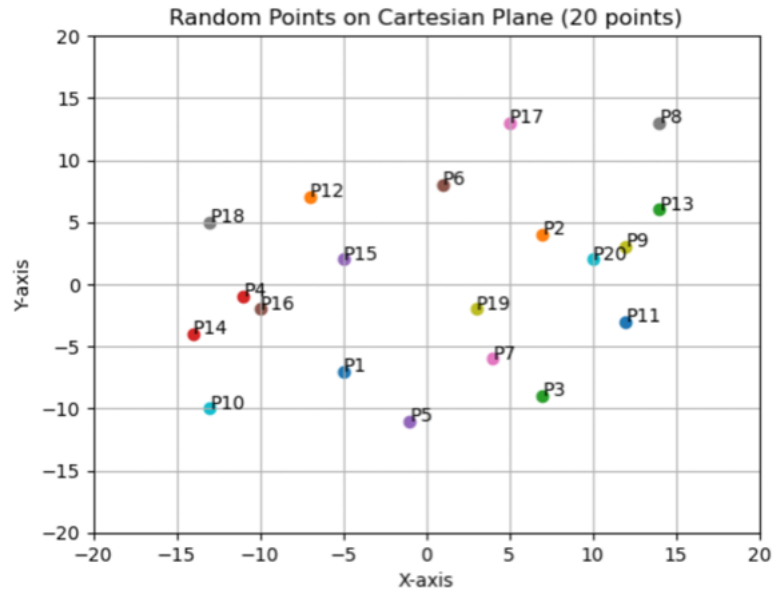
Assumptions:

- 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:

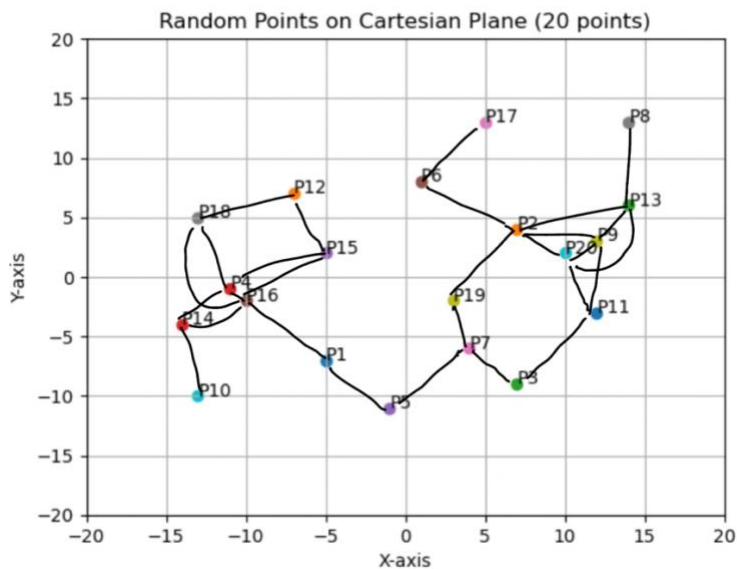
The project consists of *Project.ipynb* file, which has the main code and some additional *csv* files having some data.

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.



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.



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.

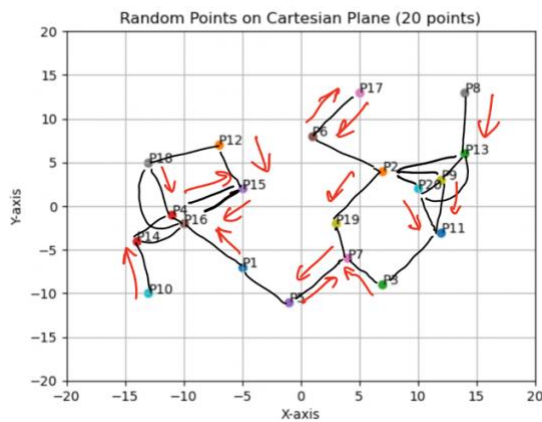
Algorithm1 uses constraints to decide which tags can't transmit while some other tag is transmitting. That is avoiding the interference between tags.

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.

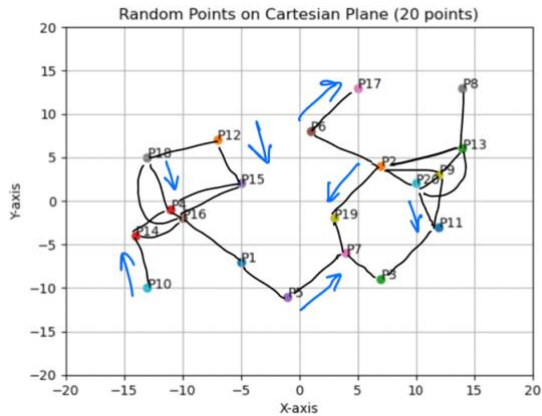
In *Instance* vector value a at index i , means *node i* wants to transmit to destination *node a*.

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.



[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* class in *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.

```
] best_params #printing best hyperparameters
]: {'estimator__criterion': 'gini',
    'estimator__max_depth': 16,
    'estimator__min_samples_leaf': 4,
    'estimator__min_samples_split': 16}
```

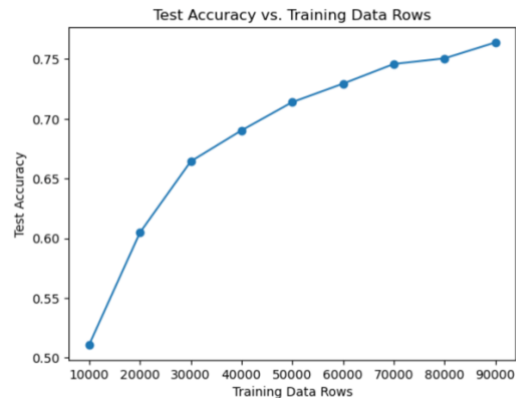
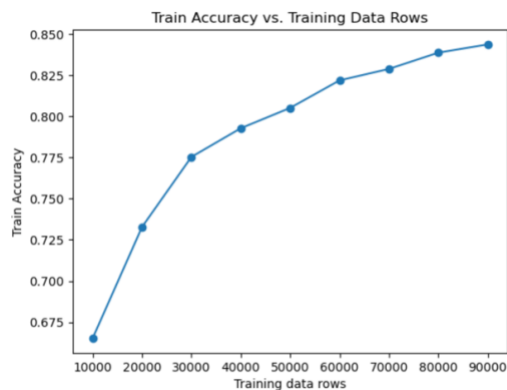
Using these parameters in our model on the same dataset, we got higher test accuracy of 75%. Training accuracy also got reduced to 83%. It means model is not overfitting.

```
score = clf.score(X_train, y_train)
print('Accuracy:', score)
score = clf.score(X_test, y_test)
print('Accuracy:', score)
```

Accuracy: 0.8306994993911514
Accuracy: 0.7499447548694637

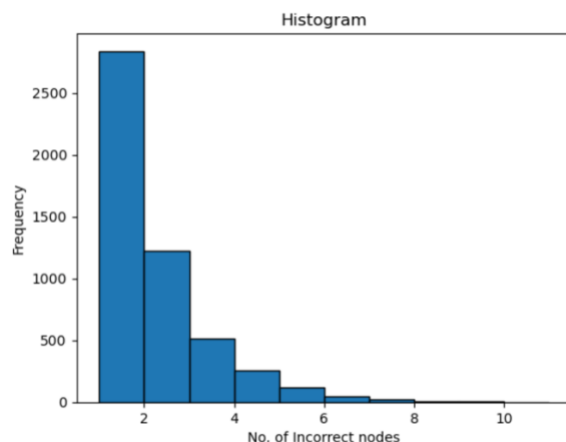
Relation of accuracy with amount of training data:

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:



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:



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:

```
In [21]: import sys
import itertools

def solve_instance(d, c):          #updated the parameters

    indexes = [i for i, num in enumerate(d) if num > 0]
    combinations = []
    result = None
    unique_combinations = []

    for size in range(len(indexes), 0, -1):
        #genetating all combination such that any 1 node finally transmits , or 2 nodes and so on.
        #and these combinations will be checked if they satisfy the constraints
        combinations = itertools.combinations(indexes, size)

        for combination in combinations:
            valid_comb = True
            for elem in combination:          # EACH element of combination should satisfy the consrtaints
                intersection = set(c[elem]).intersection(set(combination))
                if(len(intersection) > 0):
                    valid_comb = False
                    break

            if(valid_comb):                    #for stopping the loop when valid combination found
                result = combination
                break

        if(result != None and len(result) > 0):    #for stopping the loop when valid combination found
            break

    return list(result)
```


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.

```
ht_model = multioutput.ClassifierChain(  
    model=tree.HoeffdingAdaptiveTreeClassifier(  
        grace_period=100,  
        drift_detector=drift.binary.DDM(),  
        #max_depth = 16,  
        #split_criterion = 'gini',  
        #nominal_attributes=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],  
        nominal_attributes=['Att1', 'Att2', 'Att3', 'Att4', 'Att5', 'A  
    ))  
  
n_samples = 0  
correct_cnt = 0
```

Aug 21, 2022 & Aug 22, 2022:

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

```
In [155]: X_set1_1, y_set1_1 = generate_data(20000)
X_set2_1, y_set2_1 = generate_data(20000)
X_set3_1, y_set3_1 = generate_data(20000)
X_set4_1, y_set4_1 = generate_data(20000)
X_set5_1, y_set5_1 = generate_data(20000)

9293 19294 19295 19296 19297 19298 19299 19300 19301 19302 19303 19304 19305 19306 19307 19308 19309 19310 19
311 19312 19313 19314 19315 19316 19317 19318 19319 19320 19321 19322 19323 19324 19325 19326 19327 19328 193
29 19330 19331 19332 19333 19334 19335 19336 19337 19338 19339 19340 19341 19342 19343 19344 19345 19346 1934
7 19348 19349 19350 19351 19352 19353 19354 19355 19356 19357 19358 19359 19360 19361 19362 19363 19364 19365
19366 19367 19368 19369 19370 19371 19372 19373 19374 19375 19376 19377 19378 19379 19380 19381 19382 19383 1
```

9293	19294	19295	9296	19297	19298	19299	19300	19301	19302	19303	19304	19305	19306	19307	19308	19309	19310	19
311	19312	19313	19314	19315	19316	19317	19318	19319	19320	19321	19322	19323	19324	19325	19326	19327	19328	19
29	19330	19331	19332	19333	19334	19335	19336	19337	19338	19339	19340	19341	19342	19343	19344	19345	19346	1934
7	19348	19349	19350	19351	19352	19353	19354	19355	19356	19357	19358	19359	19360	19361	19362	19363	19364	19365
19366	19367	19368	19369	19370	19371	19372	19373	19374	19375	19376	19377	19378	19379	19380	19381	19382	19383	1

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

```
In [64]: X_set1_2, y_set1_2 = generate_data(20000)
X_set2_2, y_set2_2 = generate_data(20000)
X_set3_2, y_set3_2 = generate_data(20000)
X_set4_2, y_set4_2 = generate_data(20000)
X_set5_2, y_set5_2 = generate_data(20000)
```

2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079	2080	2081	2082	2083	2084	2085	2086	2087
2088	2089	2090	2091	2092	2093	2094	2095	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109
2110	2111	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127	2128	2129	2130	2131
2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153

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

```
In [29]: X_test_1, y_test_1 = generate_data(50000)
X_test_2, y_test_2 = generate_data(50000)
```

Now trained the classifier with arrangement_1 (incrementally with all 5 training sets) and then arrangement_2. And plotted the graph of accuracy:

```

# accuracy_arr1 = [0.2887, 0.38502, 0.46644, 0.51914, 0.54324]
# accuracy_arr2 = [0.21838, 0.4058, 0.43754, 0.47082, 0.49346]

# accuracy_arr2 = [0.33812, 0.38794, 0.42944, 0.44972, 0.469]
# accuracy_arr1 = [0.39884, 0.41236, 0.44722, 0.4761, 0.52682]
|

# accuracy_arr1 = [0.32092, 0.42144, 0.51282, 0.51024, 0.53542]
# accuracy_arr2 = [0.34202, 0.41472, 0.43708, 0.46606, 0.47888]

# accuracy_arr2 = [0.32962, 0.41138, 0.43526, 0.46308, 0.49114]
# accuracy_arr1 = [0.33266, 0.41292, 0.47982, 0.51108, 0.54498]

import matplotlib.pyplot as plt
input_rows = [20000, 40000, 60000, 80000, 100000]

# plt.subplot(3, 1, 1)
plt.plot(input_rows, accuracy_arr1, input_rows, accuracy_arr2, marker='o')
# plt.title('arrangement_1')

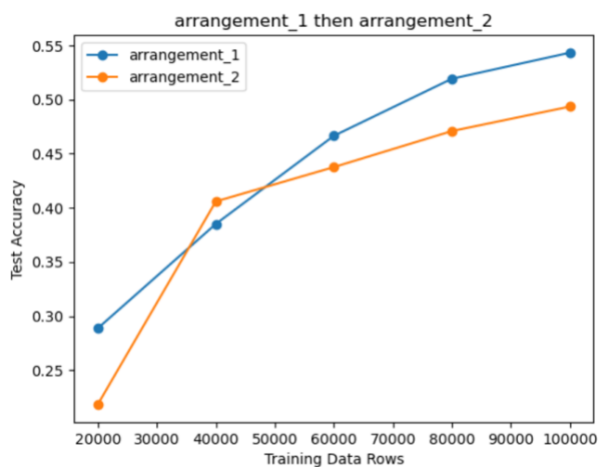
# plt.subplot(3, 1, 2)
# plt.plot(input_rows, , marker='x')
# plt.title('arrangement_2')

# Adding labels and title
plt.xlabel('Training Data Rows')
plt.ylabel('Test Accuracy')
plt.legend(['arrangement_1', 'arrangement_2'])
plt.title('arrangement_1 then arrangement_2')

# Display the plot
# plt.tight_layout()
plt.show()

```

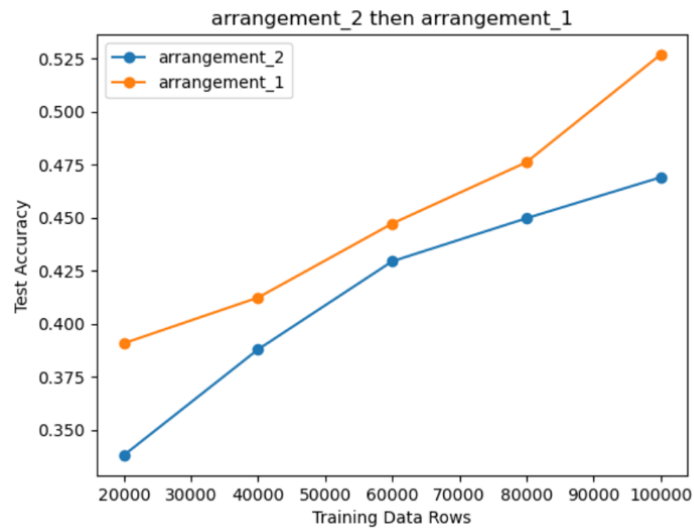
1.



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

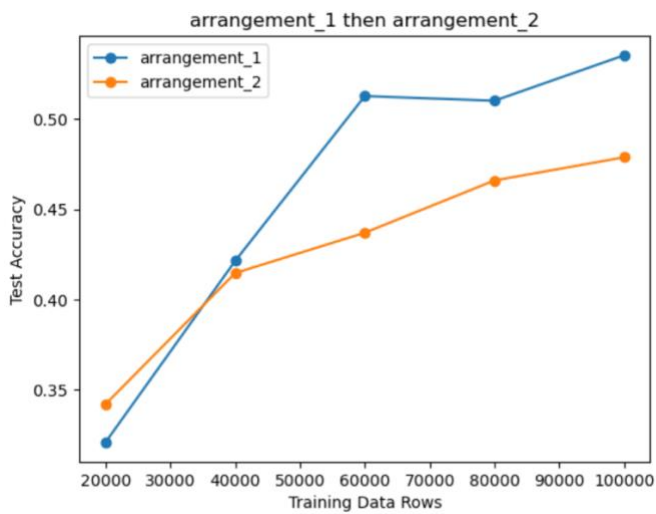
Now trained the classifier with arrangement_2 and then arrangement_1. And plotted the graph of accuracy:

2.

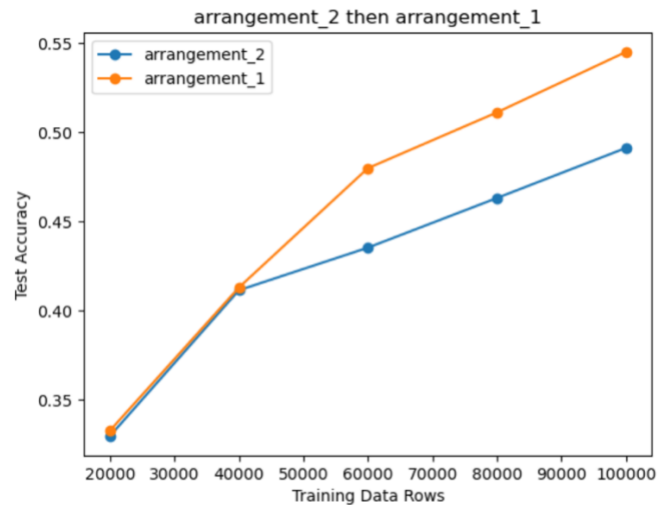


Repeated above two steps again:

3.



4.



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, 2022:

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://neptune.ai/blog/concept-drift-best-practices>

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



