




















ELE7038B
AI AND DISTRIBUTED
COMPUTING

Course Work
Abdulhameed Yunusa

Check List

Tasks	Completion Status
Dataset Pre-processing <ul style="list-style-type: none"> - Loading the dataset in different vectors - Perform data cleansing in each data frame - Combine / Merge all data frames - Dimensionality / Features reduction - Data Segmentation 	      
Machine Learning Model Creation and Training <ul style="list-style-type: none"> - Python code to create and train a selected shallow learning model and train it using the combined NSL_KDD dataset with accuracies logged - Python code to create and train a selected deep learning model and train it using the combined NSL_KDD dataset with accuracies logged - Plot the statistics from training on both models SL and DL showing accuracies - Perform parallel training for the SL and DL model for 20 Clients and 20 Batches - Plot graphs to show the comparison of time taken by SL and DL in sequential and Parallel 	    
Machine Learning model evaluation <ul style="list-style-type: none"> - Extend the code from the ML model creation in sequential and parallel - SL sequential vs SL parallel Training (Time) - SL sequential vs DL sequential (Performance) - SL vs parallel vs DL Parallel Training (Performance) - DL sequential vs DL Parallel training (Time) - Plot Graphs for all comparison 	      



Extending Parallel learning code from previous task to FL concept <ul style="list-style-type: none"> - Improve accuracy of SL using FL - Improve accuracy of DL using FL 	 
---	--

Table of contents

Check List	2
Table of contents	4
List of Figures	6
List of Abbreviations	8
1: Dataset loading and pre-processing	9
1.1 Loading the data in different vectors	9
A. Display the dataset information	9
B. Display number of records with non-numeric values	9
1.2 Perform data cleansing in each data frame /vector	10
A. Search for all NULL/empty cells and remove the rows	10
B. Search for alpha-numeric or alphabetical values in the dataset's vectors	10
C. Displaying the Dataset Information after cleansing	10
1.3 Merging all the data frames/vectors from NSL-KDD	11
1.4 Dimensionality/Features reduction	11
A. Display number of features/columns and number of records from the combined NSL_KDD data and the CICIDS Data	11
B. Perform the dimensionality reduction also called features reduction operations on both vectors	11
C. Displaying Number of features/columns and number of records in both vectors after feature reduction	12
1.5 Data segmentation	12
A. Data segmentation on both NSL-KDD and CICIDS datasets	12
B. Display the number of columns and rows in each vector	12
1.6 Summary of Dataset Loading and Prepossessing	12
2: Machine learning model creation and training	14

2.1 Create and train a selected Shallow learning model using the Combined NSL_KDD Dataset	14
2.2 create and train a selected Deep learning model using the Combined NSL_KDD Dataset	14
2.3 Plotting the statistics from training iterations of both SL and DL models showing learning accuracies	15
2.4 Performing Parallel training on both KNN and DNN	15
2.5 Plotting the graph to show comparison of time taken by SL and DL for sequential and for parallel training	16
2.6 Summary of Machine Learning model creation and Training	17
3: Machine learning model evaluation	19
3.1 Analysis, Graphs and Visualizations	19
A. SL sequential Vs SL parallel training (Compare Time)	19
B. SL sequential Vs DL Sequential (Compare Performance metrics)	20
C. SL parallel Vs DL parallel training (Compare Performance metrics)	20
D. DL sequential Vs DL parallel training (Compare Time)	21
3.2 Dataframes	22
3.3 Summary of Machine Learning Model Evaluation	24
4: Extending Parallel learning code to Federated Learning Implementation.	27

List of Figures

Figure 1: Loaded KDD_DDoS data showing dataset information.....	9
Figure 2: Loaded KDD_Probe data showing dataset information	9
Figure 3: Loaded KDD_R2L data showing dataset information	9
Figure 4: Loaded KDD_U2R data showing dataset information.....	9
Figure 5: Loaded CICIDS data showing dataset information	10
Figure 6: KDD_DDoS dataset information after cleansing	10
Figure 7: KDD_Probe dataset information after cleansing	10
Figure 8: KDD_R2L dataset information after cleansing	10
Figure 9: KDD_U2R dataset information after cleansing.....	10
Figure 10: CICIDS dataset information after cleansing	10
Figure 11: Image showing how the datasets were merged	11
Figure 12: Image showing the features from the combined NSL_KDD data set and CICIDS Dataset.....	11
Figure 13: Feature reduction on Combined NSL_KDD dataset	11
Figure 14: Feature reduction on CICIDS dataset.....	11
Figure 15: Number of features in both vectors after reduction	12
Figure 16: Segmentation of combined NSL_KDD and display of Dataset details	12
Figure 17: Segmentation of combined CICIDS and display of Dataset details ...	12
Figure 18: Creation and Training of SL (KNN) with accuracies, time, and Overall accuracy logged.....	14
Figure 19: Creation and Training of DL (DNN) with accuracies, time, and Overall accuracy logged.....	14
Figure 20: Graph Showing the Accuracies of SL and DL at each epoch.....	15
Figure 21: Parallel Training on KNN.....	15
Figure 22: Parallel Training on DNN.....	16
Figure 23: Graph showing comparison of time taken by SL and DL for sequential and for parallel training.....	16
Figure 24: Data frame showing the total time taken to execute KNN and DNN in Sequential and Parallel.....	17
Figure 25: Data frame showing the Accuracies of KNN and DNN executed in Sequential and Parallel at different epochs.....	17

Figure 26: Graph of SL sequential Vs SL parallel training (Compare Time) at every epoch.....	19
Figure 27: Graph of SL sequential Vs DL Sequential showing Accuracy, TPR and Error Rate for both Models at every Epoch	20
Figure 28: Graph of SL Parallel Vs DL Parallel showing Accuracy, TPR and Error Rate for both Models at every Epoch.....	21
Figure 29: Graph for DL sequential Vs DL parallel training Time.....	22
Figure 30: Data frame showing recorded metrics from KNN's execution in Sequential	22
Figure 31: Data frame showing recorded metrics from DNN's execution in Sequential	23
Figure 32: Data frame showing recorded metrics from KNN's execution in Parallel	23
Figure 33: Data frame showing recorded metrics from DNN's execution in Parallel	24
Figure 34: DataFrame showing the total time taken to train SL (KNN) and DL (DNN) model in sequential and parallel	24
Figure 35: Result from implementing FL on KNN in parallel	27
Figure 36: Result from implementing FL on DNN in parallel.....	28

List of Abbreviations

CM: Confusion Matrix
DNN: Deep Neural Nets
DL: Deep Learning
DT: Decision Tree
FL: Federated Learning
FPR: False Positive Rate
FNR: False Negative Rate
FPR: False Positive Rate
IPYNB: IPython Notebook
KNN: K-Nearest Neighbors Algorithm
SL: Shallow Learning
TNR: True Negative Rate
TPR: True Positive Rate

1: Dataset loading and pre-processing

The Solution to this task is in an IPYNB file named **2.1_Course_work.ipynb**

1.1 Loading the data in different vectors

A. Display the dataset information

B. Display number of records with non-numeric values

```
# Read data KDD_DDoS
kdd_ddos_data = pd.read_excel("KDD_DDoS.xlsx")

#Display number of rows, columns and non-numeric data
kdd_ddos_data_det = display_dataset_info(kdd_ddos_data)

Number of rows: 29132
Number of columns: 42
Non-numeric data: 1470 rows
```

Figure 1: Loaded KDD_DDoS data showing dataset information

```
# Read data KDD_Probe
kdd_probe_data = pd.read_excel("KDD_Probe.xlsx")

#Display number of rows, columns and non-numeric data
kdd_probe_data_det = display_dataset_info(kdd_probe_data)

Number of rows: 20292
Number of columns: 42
Non-numeric data: 1373 rows
```

Figure 2: Loaded KDD_Probe data showing dataset information

```
# Read data KDD_R2L
kdd_r2l_data = pd.read_excel("KDD_R2L.xlsx")

#Display number of rows, columns and non-numeric data
kdd_r2l_data_det = display_dataset_info(kdd_r2l_data)

Number of rows: 18564
Number of columns: 42
Non-numeric data: 535 rows
```

Figure 3: Loaded KDD_R2L data showing dataset information

```
# Read data KDD_U2R
kdd_u2r_data = pd.read_excel("KDD_U2R.xlsx")

#Display number of rows, columns and non-numeric data
kdd_u2r_data_det = display_dataset_info(kdd_u2r_data)

Number of rows: 15734
Number of columns: 42
Non-numeric data: 1022 rows
```

Figure 4: Loaded KDD_U2R data showing dataset information

```
# Read data CICIDS_DoS
CICIDS_DoS_data = pd.read_excel("CICIDS_DoS.xlsx")

#Display number of rows, columns and non-numeric data
CICIDS_DoS_data_det = display_dataset_info(CICIDS_DoS_data)

Number of rows: 225745
Number of columns: 79
Non-numeric data: 2112 rows
```

Figure 5: Loaded CICIDS data showing dataset information

1.2 Perform data cleansing in each data frame /vector

- Search for all NULL/empty cells and remove the rows
- Search for alpha-numeric or alphabetical values in the dataset's vectors
- Displaying the Dataset Information after cleansing

```
#Cleansing for KDD DDoS
kdd_ddos_data = data_cleansing(kdd_ddos_data)

Number of rows: 27662
Number of columns: 42
```

Figure 6: KDD_DDoS dataset information after cleansing

```
#Cleansing for KDD_Probe
kdd_probe_data = data_cleansing(kdd_probe_data)

Number of rows: 18919
Number of columns: 42
```

Figure 7: KDD_Probe dataset information after cleansing

```
#Cleansing for KDD_R2L Data
kdd_r2l_data = data_cleansing(kdd_r2l_data)

Number of rows: 18029
Number of columns: 42
```

Figure 8: KDD_R2L dataset information after cleansing

```
#Cleansing for KDD_u2r_data
kdd_u2r_data = data_cleansing(kdd_u2r_data)

Number of rows: 14712
Number of columns: 42
```

Figure 9: KDD_U2R dataset information after cleansing

```
#Cleansing for CICIDS_DoS
CICIDS_DoS_data = data_cleansing(CICIDS_DoS_data)

Number of rows: 223633
Number of columns: 79
```

Figure 10: CICIDS dataset information after cleansing

1.3 Merging all the data frames/vectors from NSL-KDD

```
DS_NSL_Final = merge_datasets(kdd_ddos_data, kdd_probe_data, kdd_r2l_data, kdd_u2r_data)
```

Figure 11: Image showing how the datasets were merged

1.4 Dimensionality/Features reduction

A. Display number of features/columns and number of records from the combined NSL_KDD data and the CICIDS Data

```
# Display the features before feature reduction from the combined NSL-KDD dataset
DS_NSL_Final_info = display_dataset_info(DS_NSL_Final)
```

```
Number of rows: 79322
Number of columns: 42
Non-numeric data: 0 rows
```

```
# Display the features before feature reduction from the combined CICIDS dataset
CICIDS_DoS_data_info = display_dataset_info(CICIDS_DoS_data)
```

```
Number of rows: 223633
Number of columns: 79
Non-numeric data: 0 rows
```

Figure 12: Image showing the features from the combined NSL_KDD data set and CICIDS Dataset

B. Perform the dimensionality reduction also called features reduction operations on both vectors

```
# Feature reduction on combined NSL-KDD data set
DS_NSL_Final = perform_dimensionality_reduction(DS_NSL_Final)
```

```
Number of correlated features at 0.99 threshold: 1
Number of features/columns before reduction: 42
Number of features/columns after reduction: 41
Number of records after reduction: 79322
```

Figure 13: Feature reduction on Combined NSL_KDD dataset

```
# Feature reduction on combined CICIDS data set
CICIDS_DoS_data = perform_dimensionality_reduction(CICIDS_DoS_data)
```

```
Number of correlated features at 0.99 threshold: 15
Number of features/columns before reduction: 79
Number of features/columns after reduction: 64
Number of records after reduction: 223633
```

Figure 14: Feature reduction on CICIDS dataset

C. Displaying Number of features/columns and number of records in both vectors after feature reduction

```
# Display the features after feature reduction from the combined NSL-KDD dataset
DS_NSL_Final_info_final = display_dataset_info(DS_NSL_Final)
```

```
Number of rows: 79322
Number of columns: 41
Non-numeric data: 0 rows
```

```
# Display the features after feature reduction from the combined CICIDS dataset
CICIDS_DoS_data_info_final = display_dataset_info(CICIDS_DoS_data)
```

```
Number of rows: 223633
Number of columns: 64
Non-numeric data: 0 rows
```

Figure 15: Number of features in both vectors after reduction

1.5 Data segmentation

A. Data segmentation on both NSL-KDD and CICIDS datasets

B. Display the number of columns and rows in each vector

```
X_train_KDD, X_test_KDD, y_train_KDD, y_test_KDD = split_dataset(DS_NSL_Final, 0.3)
```

```
Shape of X_train: (55525, 40)
Shape of X_test: (23797, 40)
Shape of y_train: (55525, 1)
Shape of y_test: (23797, 1)
```

Figure 16: Segmentation of combined NSL_KDD and display of Dataset details

```
X_train_KDD, X_test_KDD, y_train_KDD, y_test_KDD = split_dataset(CICIDS_DoS_data, 0.3)
```

```
Shape of X_train: (156543, 40)
Shape of X_test: (67090, 40)
Shape of y_train: (156543, 1)
Shape of y_test: (67090, 1)
```

Figure 17: Segmentation of combined CICIDS and display of Dataset details

1.6 Summary of Dataset Loading and Preprocessing

The provided code includes several functions and operations for data preprocessing and analysis. It starts by importing the necessary libraries, such as Pandas, NumPy, and Sklearn. The code provides functions like `display_dataset_info()` to show information about the dataset, including the

number of rows, columns, and the sum of rows with non-numeric data. Another function called `merge_datasets()` allow merging multiple datasets into one.

The `data_cleansing()` function performs cleansing operations by removing rows with NULL or empty cells and non-numeric values from the dataset.

To reduce the dimensionality of the dataset, the code offers the `perform_dimensionality_reduction()` function, which uses correlation analysis to identify and remove highly correlated features. The `split_dataset()` function splits the dataset into training and testing sets.

The code demonstrates the usage of these functions by applying them to several datasets, including KDD_DDoS, KDD_Probe, KDD_R2L, KDD_U2R, and CICIDS_DoS. It loads each dataset, displays its information, and performs data cleansing operations.

The cleaned datasets are then merged into a single dataset named `DS_NSL_Final` using the `merge_datasets()` function. Dimensionality reduction is applied to `DS_NSL_Final` and `CICIDS_DoS_data` using the `perform_dimensionality_reduction()` function, which removes highly correlated features.

The merged and reduced datasets are further split into training and testing sets using the `split_dataset()` function. Finally, the code saves the final merged dataset (`DS_NSL_Final`) to an Excel file named "`DS_NSL_Final.xlsx`".

In summary, the provided code offers a comprehensive set of functions and operations for effective data preprocessing, merging, dimensionality reduction, and dataset splitting, providing a solid foundation for further data analysis or machine learning tasks.

2: Machine learning model creation and training

The Solution to this task is in an IPYNB file named **2.2_Course_work.ipynb**

2.1 Create and train a selected Shallow learning model using the Combined NSL_KDD Dataset

Shallow Learning Model KNN

```
knn_acc, SL_seq_final_time, KNN = create_and_train_shallow_model(X_train, y_train, X_test, y_test)
Epoch 3 : Accuracy 0.9903286807744259
Epoch 4 : Accuracy 0.9903286807744259
Epoch 5 : Accuracy 0.9903286807744259
Epoch 6 : Accuracy 0.9903286807744259
Epoch 7 : Accuracy 0.9903286807744259
Epoch 8 : Accuracy 0.9903286807744259
Epoch 9 : Accuracy 0.9903286807744259
Epoch 10 : Accuracy 0.9903286807744259
Epoch 11 : Accuracy 0.9903286807744259
Epoch 12 : Accuracy 0.9903286807744259
Epoch 13 : Accuracy 0.9903286807744259
Epoch 14 : Accuracy 0.9903286807744259
Epoch 15 : Accuracy 0.9903286807744259
Epoch 16 : Accuracy 0.9903286807744259
Epoch 17 : Accuracy 0.9903286807744259
Epoch 18 : Accuracy 0.9903286807744259
Epoch 19 : Accuracy 0.9903286807744259
Epoch 20 : Accuracy 0.9903286807744259
Shallow Learning Model Accuracy (KNN) : 0.9903286807744259
Shallow Learning Model Sequential Trainig time (KNN) : 114.86100196838379 seconds
```

Figure 18: Creation and Training of SL (KNN) with accuracies, time, and Overall accuracy logged

2.2 create and train a selected Deep learning model using the Combined NSL_KDD Dataset

```
# Train the Deep Learning Model (DNN)
DL_acc_list, DL_seq_final_time, DNN = train_deep_model(X_train, y_train, X_test, y_test)
Epoch 3 : Accuracy 0.9552464485168457
Epoch 4 : Accuracy 0.9565491676330566
Epoch 5 : Accuracy 0.9553725123405457
Epoch 6 : Accuracy 0.9578098058700562
Epoch 7 : Accuracy 0.9567172527313232
Epoch 8 : Accuracy 0.959028422832489
Epoch 9 : Accuracy 0.9577257633209229
Epoch 10 : Accuracy 0.943900465965271
Epoch 11 : Accuracy 0.9578938484191895
Epoch 12 : Accuracy 0.9592806100845337
Epoch 13 : Accuracy 0.9562970399856567
Epoch 14 : Accuracy 0.9567592740058899
Epoch 15 : Accuracy 0.958061933517456
Epoch 16 : Accuracy 0.9587762951850891
Epoch 17 : Accuracy 0.9563390612602234
Epoch 18 : Accuracy 0.9573895931243896
Epoch 19 : Accuracy 0.95650714635849
Epoch 20 : Accuracy 0.9562970399856567
Deep Learning Model Accuracy (DNN): 0.9562970399856567
Deep Learning Model Sequential Trainig time (DNN) : 1941.634501695633 seconds
```

Figure 19: Creation and Training of DL (DNN) with accuracies, time, and Overall accuracy logged

2.3 Plotting the statistics from training iterations of both SL and DL models showing learning accuracies

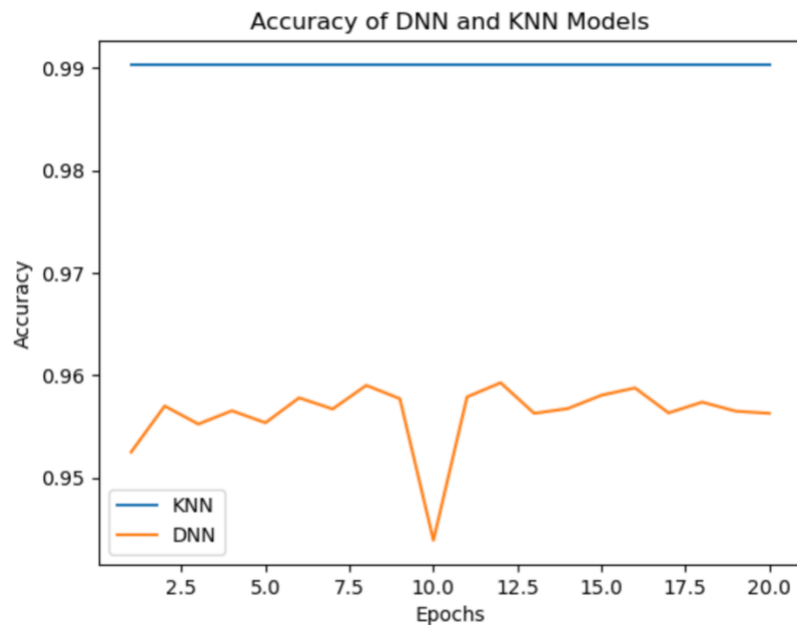


Figure 20: Graph Showing the Accuracies of SL and DL at each epoch

2.4 Performing Parallel training on both KNN and DNN

```
# Train the Deep Learning Model (KNN) in parallel

knn_tasks=[]
knn_results=[]

#measure training time
knn_start_time = time.time()

#initialize ray
ray.shutdown()
ray.init()

#train model x-times
for i in range(20):
    knn_tasks.append(train_shallow_model.remote(X_train_batches[i], y_train_batches[i], X_test, y_test))
knn_total_time = time.time() - knn_start_time

#calculate metrics
for task in knn_tasks:
    accuracy, model = ray.get(task)
    knn_results.append(accuracy)

# Print the time
print("Total KNN Model Training Time in parallel :", knn_total_time, "seconds")
```

```
2023-05-26 20:01:29,019 INFO worker.py:1625 -- Started a local Ray instance.
Total KNN Model Training Time in parallel : 4.866597890853882 seconds
```

Figure 21: Parallel Training on KNN

```

# Train the Deep Learning Model (DNN) in parallel
dnn_tasks = []
dnn_results = []

#measure training time
dnn_start_time = time.time()

#initialize ray
ray.shutdown()
ray.init()

#train model x-times
for i in range(20):
    dnn_tasks.append(train_deep_model.remote(X_train_batches[i], y_train_batches[i], X_test, y_test))

#calculate metrics
for task in dnn_tasks:
    accuracy, model = ray.get(task)
    dnn_results.append(accuracy)
dnn_total_time = time.time() - dnn_start_time

# Print the time
print("Total DNN Model Training Time in parallel :", dnn_total_time, "seconds")

```

2023-05-26 20:03:45,737 INFO worker.py:1625 -- Started a local Ray instance.

Figure 22: Parallel Training on DNN

2.5 Plotting the graph to show comparison of time taken by SL and DL for sequential and for parallel training

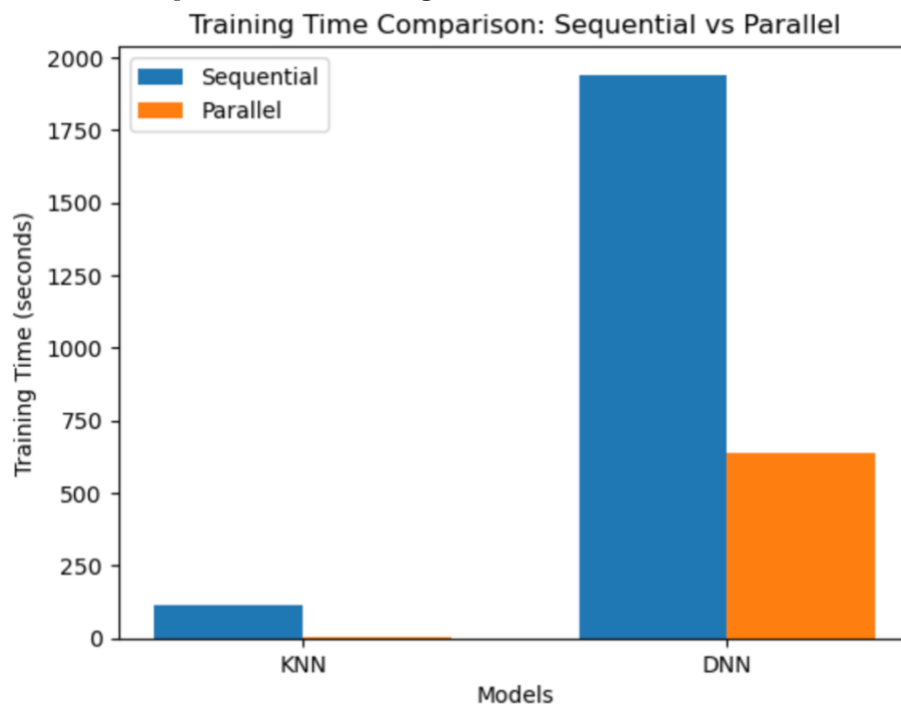


Figure 23: Graph showing comparison of time taken by SL and DL for sequential and for parallel training

	KNN	DNN	KNN_Parallel	DNN_Parallel
SN				
1	114.86	1941.63	4.87	636.11

Figure 24: Data frame showing the total time taken to execute KNN and DNN in Sequential and Parallel

	KNN	DNN	KNN_Parallel	DNN_Parallel
Epoch				
1	0.9903	0.9525	0.9492	0.9418
2	0.9903	0.9570	0.9429	0.9344
3	0.9903	0.9552	0.9480	0.9457
4	0.9903	0.9565	0.9468	0.9479
5	0.9903	0.9554	0.9456	0.9311
6	0.9903	0.9578	0.9474	0.9412
7	0.9903	0.9567	0.9498	0.9471
8	0.9903	0.9590	0.9448	0.9468
9	0.9903	0.9577	0.9487	0.9421
10	0.9903	0.9439	0.9453	0.9424
11	0.9903	0.9579	0.9427	0.9385
12	0.9903	0.9593	0.9508	0.9259
13	0.9903	0.9563	0.9451	0.9494
14	0.9903	0.9568	0.9399	0.9437
15	0.9903	0.9581	0.9536	0.9276
16	0.9903	0.9588	0.9492	0.9392
17	0.9903	0.9563	0.9497	0.9434
18	0.9903	0.9574	0.9500	0.9521
19	0.9903	0.9565	0.9491	0.9463
20	0.9903	0.9563	0.9439	0.9550

Figure 25: Data frame showing the Accuracies of KNN and DNN executed in Sequential and Parallel at different epochs

2.6 Summary of Machine Learning model creation and Training

The code begins by importing necessary libraries such as time, ray, pandas, NumPy, TensorFlow, sklearn, and matplotlib.pyplot. These libraries are essential for data manipulation, model creation, and result visualisation.

Next, it defines functions for splitting the dataset into training and testing sets (`split_dataset()`), creating a shallow learning model using KNN (`create_shallow_model()`), and training the shallow model (`create_and_train_shallow_model()`). Similarly, functions for creating and training a deep learning model using DNN are defined (`create_deep_model()` and `train_deep_model()`).

The preprocessed dataset is loaded from an Excel file using `pd.read_excel()`. The dataset is then split into training and testing sets using the `split_dataset()` function.

The shallow learning model (KNN) is trained sequentially using a loop, and the accuracy for each epoch is printed. The final accuracy and training time for sequential training are displayed.

Similarly, the deep learning model (DNN) is trained sequentially using a loop, and the accuracy for each epoch is printed. The final accuracy and training time for sequential training are displayed.

Next, the KNN and DNN models are trained in parallel using the Ray library. The training time for both models in parallel is calculated and displayed. Visualisations are generated using `matplotlib.pyplot` to plot the accuracy of the KNN and DNN models for each epoch.

Dataframes are created to summarise the training results, including the sequential and parallel training times for both models.

Finally, the training times for KNN and DNN models (both sequential and parallel) are printed for comparison.

Overall, the code demonstrates the implementation of sequential and parallel training for KNN and DNN models and provides insights into their performance and training times.

3: Machine learning model evaluation

The Solution to this task is in an IPYNB file named **2.3_Course_work.ipynb**

3.1 Analysis, Graphs and Visualizations

A. SL sequential Vs SL parallel training (Compare Time)

The parallel execution of the KNN model significantly reduces training time compared to sequential execution. The average epoch duration in sequential execution is 2.5 seconds, while in parallel execution, it is approximately 0.03 seconds, resulting in a speedup of about 83 times. This highlights the advantage of parallel execution in accelerating the training process and maximizing computational resources. By leveraging multiple processors or computing units to perform computations simultaneously, parallel execution effectively saves time.

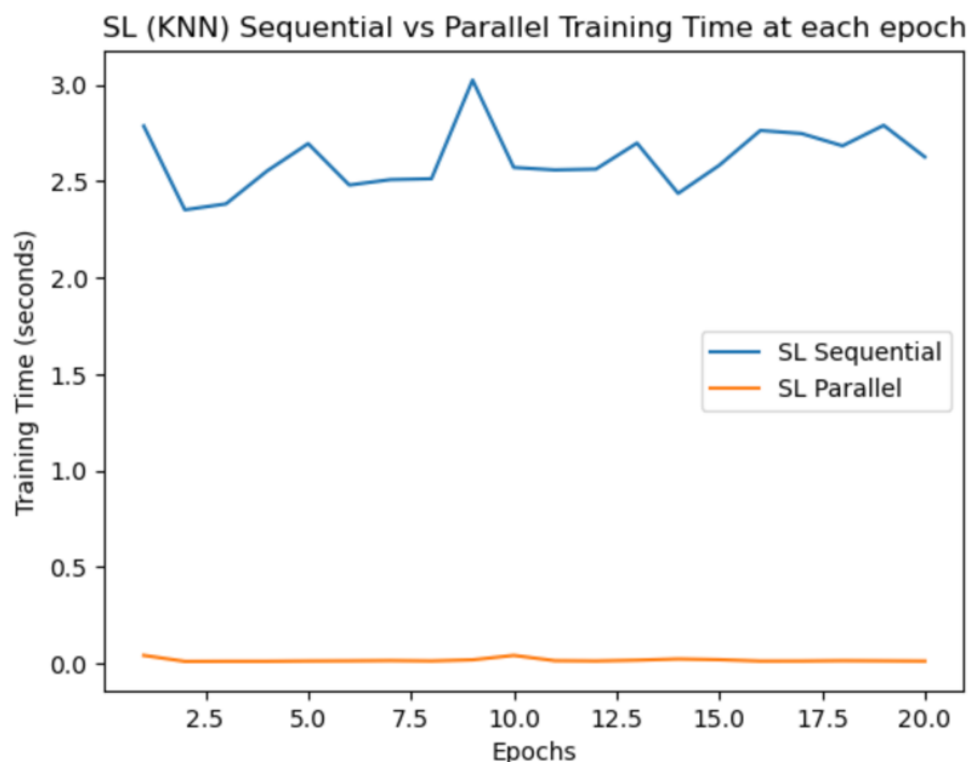


Figure 26: Graph of SL sequential Vs SL parallel training (Compare Time) at every epoch

B. SL sequential Vs DL Sequential (Compare Performance metrics)

In sequential execution, the KNN model outperformed the DNN model with higher accuracy (DNN: 0.76, KNN: 0.98). The DNN model had a lower TPR, and lower error rate, indicating its lack of ability to identify positive examples accurately. Conversely, the KNN model showed lower FPR and higher TNR, indicating its proficiency in correctly classifying negative instances. Overall, the KNN model displayed better performance across accuracy, TPR, FNR, and error rate, making it the preferred choice for predicting the target variable.

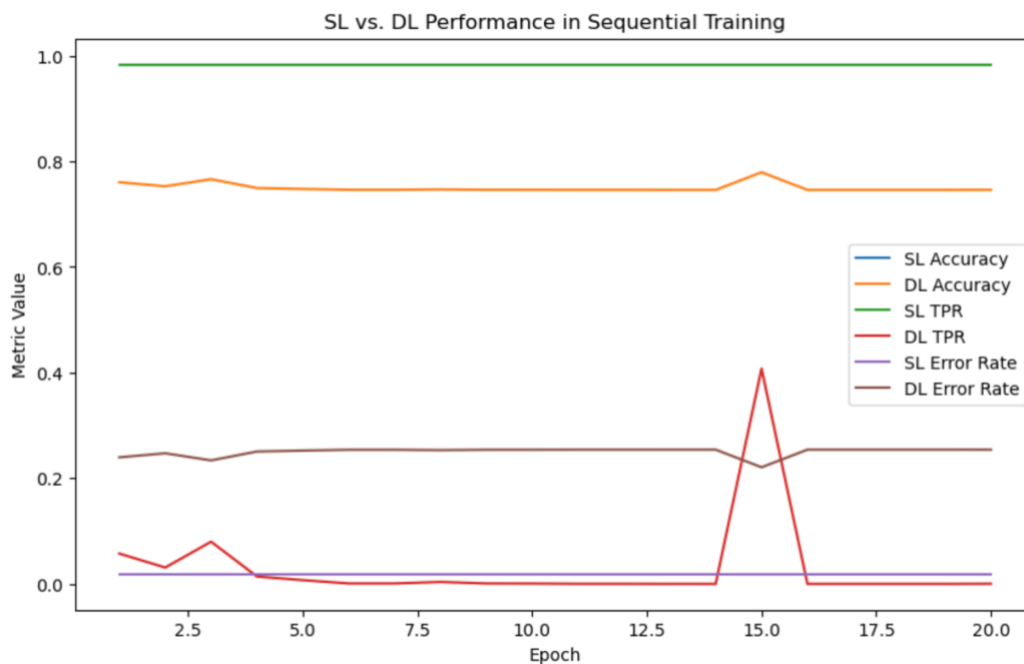


Figure 27: Graph of SL sequential Vs DL Sequential showing Accuracy, TPR and Error Rate for both Models at every Epoch

C. SL parallel Vs DL parallel training (Compare Performance metrics)

The DNN model beat the KNN model in terms of TPR, reaching an average of 0.88 compared to 0.82, according to parallel execution. Both models' training times were drastically cut, although the DNN model benefited more. While the KNN model showed lower FPR and greater TNR, excelling in categorising negative cases, the DNN model demonstrated higher TPR, reduced error rate, and better overall performance. Either model can be considered for parallel

execution depending on individual requirements, with the DNN model providing better overall performance.

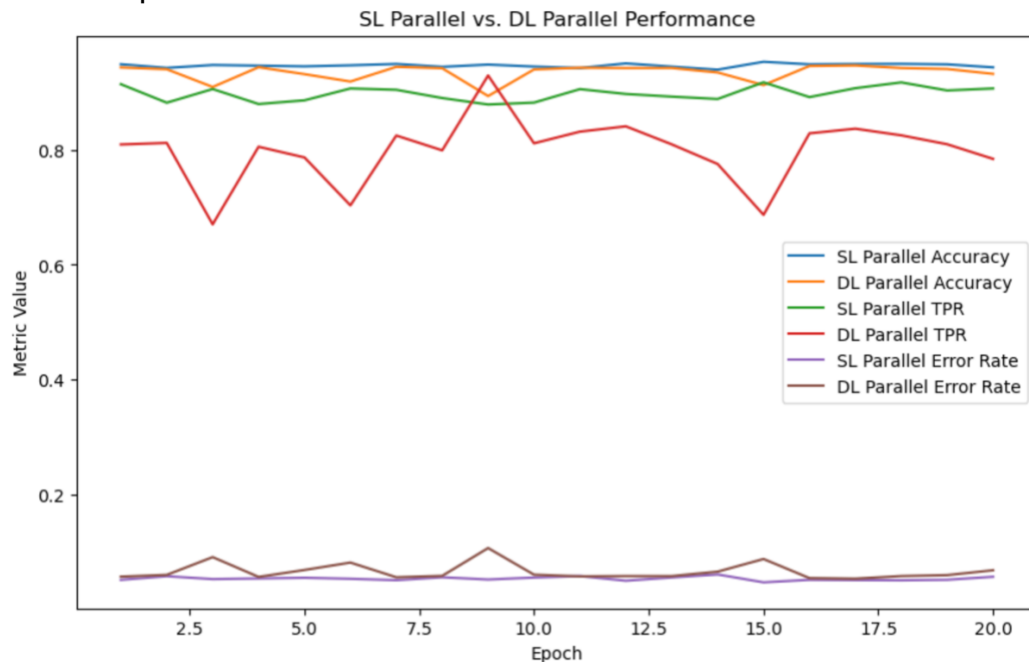


Figure 28: Graph of SL Parallel Vs DL Parallel showing Accuracy, TPR and Error Rate for both Models at every Epoch

D. DL sequential Vs DL parallel training (Compare Time)

The training time per epoch is greatly decreased by parallel execution when comparing the time required for DNN in sequential and parallel execution. Each epoch requires much more time during sequential execution, lasting anything from 86 to more than 120 seconds. The time per epoch is, however, greatly decreased by parallel processing and ranges from about 0.008 seconds to 0.047 seconds.

These results demonstrate the benefit of parallel execution for DNN, which enables substantially quicker training by utilising several processors or processing units to carry out computations concurrently. As a result, training time can be reduced overall and computational resources can be used more effectively.

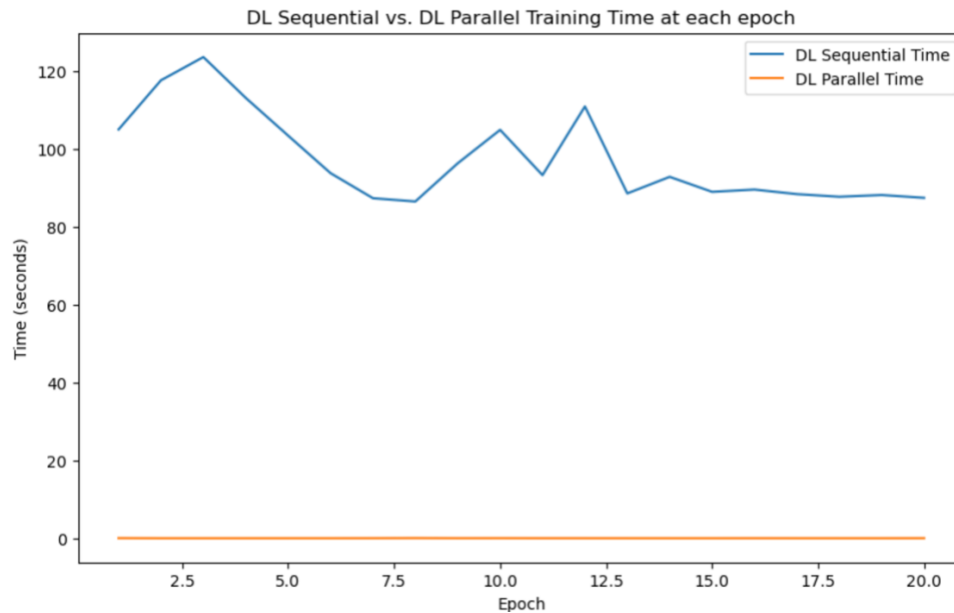


Figure 29: Graph for DL sequential Vs DL parallel training Time

3.2 Dataframes

The Dataframes below shows the recorded metrics at each epoch for SL and DL both in sequential and Parallel, the time taken, accuracy, TPR, FPR, FNR, and Error Rate were logged. The total time taken was also recorded.

Epoch	Time	Accuracy	TPR	FPR	TNR	FNR	Error Rate
1	2.7855	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
2	2.3510	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
3	2.3818	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
4	2.5525	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
5	2.6942	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
6	2.4795	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
7	2.5075	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
8	2.5121	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
9	3.0236	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
10	2.5706	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
11	2.5577	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
12	2.5621	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
13	2.6969	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
14	2.4362	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
15	2.5816	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
16	2.7623	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
17	2.7464	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
18	2.6828	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
19	2.7889	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169
20	2.6252	0.9831	0.9833	0.0176	0.9824	0.0167	0.0169

Figure 30: Data frame showing recorded metrics from KNN's execution in Sequential

DNN Metrics from Sequential Execution							
	Time	Accuracy	TPR	FPR	TNR	FNR	Error Rate
Epoch							
1	105.0426	0.7604	0.0574	0.0001	0.9999	0.9426	0.2396
2	117.6745	0.7527	0.0309	0.0013	0.9987	0.9691	0.2473
3	123.6362	0.7661	0.0799	0.0000	1.0000	0.9201	0.2339
4	113.1412	0.7494	0.0139	0.0000	1.0000	0.9861	0.2506
5	103.4466	0.7477	0.0071	0.0000	1.0000	0.9929	0.2523
6	93.8094	0.7461	0.0008	0.0000	1.0000	0.9992	0.2539
7	87.3770	0.7461	0.0008	0.0000	1.0000	0.9992	0.2539
8	86.5365	0.7468	0.0036	0.0000	1.0000	0.9964	0.2532
9	96.3523	0.7461	0.0008	0.0000	1.0000	0.9992	0.2539
10	104.9302	0.7460	0.0007	0.0000	1.0000	0.9993	0.2540
11	93.3023	0.7459	0.0002	0.0000	1.0000	0.9998	0.2541
12	110.9472	0.7459	0.0002	0.0000	1.0000	0.9998	0.2541
13	88.6097	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
14	92.8632	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
15	89.0167	0.7793	0.4076	0.0941	0.9059	0.5924	0.2207
16	89.5974	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
17	88.4284	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
18	87.7505	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
19	88.1951	0.7459	0.0000	0.0000	1.0000	1.0000	0.2541
20	87.4780	0.7459	0.0003	0.0000	1.0000	0.9997	0.2541

Figure 31: Data frame showing recorded metrics from DNN's execution in Sequential

KNN Metrics from Parallel Execution							
	Time	Accuracy	TPR	FPR	TNR	FNR	Error Rate
Epoch							
1	0.0412	0.9492	0.9145	0.0389	0.9611	0.0855	0.0508
2	0.0104	0.9429	0.8823	0.0364	0.9636	0.1177	0.0571
3	0.0109	0.9480	0.9061	0.0377	0.9623	0.0939	0.0520
4	0.0110	0.9468	0.8800	0.0304	0.9696	0.1200	0.0532
5	0.0124	0.9456	0.8864	0.0342	0.9658	0.1136	0.0544
6	0.0131	0.9474	0.9071	0.0388	0.9612	0.0929	0.0526
7	0.0145	0.9498	0.9048	0.0349	0.9651	0.0952	0.0502
8	0.0126	0.9448	0.8904	0.0367	0.9633	0.1096	0.0552
9	0.0183	0.9487	0.8791	0.0276	0.9724	0.1209	0.0513
10	0.0411	0.9453	0.8824	0.0333	0.9667	0.1176	0.0547
11	0.0140	0.9427	0.9059	0.0448	0.9552	0.0941	0.0573
12	0.0125	0.9508	0.8975	0.0311	0.9689	0.1025	0.0492
13	0.0164	0.9451	0.8929	0.0371	0.9629	0.1071	0.0549
14	0.0228	0.9399	0.8887	0.0427	0.9573	0.1113	0.0601
15	0.0190	0.9536	0.9178	0.0342	0.9658	0.0822	0.0464
16	0.0121	0.9492	0.8920	0.0314	0.9686	0.1080	0.0508
17	0.0123	0.9497	0.9076	0.0360	0.9640	0.0924	0.0503
18	0.0139	0.9500	0.9177	0.0390	0.9610	0.0823	0.0500
19	0.0130	0.9491	0.9036	0.0354	0.9646	0.0964	0.0509
20	0.0120	0.9439	0.9071	0.0436	0.9564	0.0929	0.0561

Figure 32: Data frame showing recorded metrics from KNN's execution in Parallel

DNN Metrics from Parallel Execution							
	Time	Accuracy	TPR	FPR	TNR	FNR	Error Rate
Epoch							
1	0.0370	0.9437	0.8095	0.0106	0.9894	0.1905	0.0563
2	0.0084	0.9406	0.8122	0.0156	0.9844	0.1878	0.0594
3	0.0090	0.9097	0.6703	0.0088	0.9912	0.3297	0.0903
4	0.0097	0.9442	0.8056	0.0086	0.9914	0.1944	0.0558
5	0.0106	0.9321	0.7869	0.0185	0.9815	0.2131	0.0679
6	0.0113	0.9191	0.7034	0.0073	0.9927	0.2966	0.0809
7	0.0228	0.9448	0.8251	0.0144	0.9856	0.1749	0.0552
8	0.0472	0.9423	0.7993	0.0089	0.9911	0.2007	0.0577
9	0.0211	0.8939	0.9297	0.1184	0.8816	0.0703	0.1061
10	0.0276	0.9403	0.8115	0.0158	0.9842	0.1885	0.0597
11	0.0179	0.9432	0.8318	0.0188	0.9812	0.1682	0.0568
12	0.0171	0.9427	0.8411	0.0227	0.9773	0.1589	0.0573
13	0.0142	0.9428	0.8097	0.0119	0.9881	0.1903	0.0572
14	0.0120	0.9350	0.7755	0.0106	0.9894	0.2245	0.0650
15	0.0111	0.9129	0.6868	0.0100	0.9900	0.3132	0.0871
16	0.0189	0.9463	0.8289	0.0137	0.9863	0.1711	0.0537
17	0.0131	0.9472	0.8370	0.0152	0.9848	0.1630	0.0528
18	0.0126	0.9427	0.8254	0.0174	0.9826	0.1746	0.0573
19	0.0122	0.9410	0.8099	0.0143	0.9857	0.1901	0.0590
20	0.0234	0.9326	0.7844	0.0170	0.9830	0.2156	0.0674

Figure 33: Data frame showing recorded metrics from DNN's execution in Parallel

	KNN	DNN	KNN_Parallel	DNN_Parallel
SN				
1	52.37	1948.23	5.97	7.93

Figure 34: DataFrame showing the total time taken to train SL (KNN) and DL (DNN) model in sequential and parallel

3.3 Summary of Machine Learning Model Evaluation

The code begins by cloning **2.2_Course_work.ipynb** and then extending its functionality. Then importing necessary libraries, such as time, ray, pandas, numpy, tensorflow, sklearn, and matplotlib. These libraries provide various functionalities for data manipulation, machine learning, and visualisation.

The dataset is split into training and testing sets using the train_test_split function from scikit-learn. The features (X) and target variable (y) are extracted from the dataset.

A shallow learning model (KNN) is created using the `KNeighborsClassifier` class from `scikit-learn`. The model is trained for multiple epochs, and the accuracy, true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), false negative rate (FNR), error rate, and time per epoch are recorded.

Similarly, a deep learning model (DNN) is created using the `sequential` class from `Keras` and trained for multiple epochs. The accuracy, TPR, FPR, TNR, FNR, error rate, and time per epoch are calculated and stored.

The metrics from the KNN model in sequential execution are displayed in a formatted `DataFrame`, including the time, accuracy, TPR, FPR, TNR, FNR, and error rate for each epoch.

The deep learning model (DNN) is trained using the training and testing sets. The metrics are calculated, including accuracy, TPR, FPR, TNR, FNR, error rate, and time per epoch. These metrics are displayed in a formatted data frame.

To measure the effect of parallelism using Ray, the training of both the shallow learning model (KNN) and deep learning model (DNN) is parallelized. The training is performed in parallel for multiple epochs using the Ray framework. The total training time and time per epoch for the parallel execution of the KNN model are recorded.

The training of the KNN model in parallel is executed using the `train_shallow_model` function, which creates and trains the KNN model. The accuracy, model, and confusion matrix (CM) for each epoch are obtained. The total training time and time per epoch for the parallel execution of the KNN model are recorded and stored.

The results from the parallel execution of the KNN model are evaluated, and the accuracy, TPR, FPR, TNR, FNR, and error rate for the final epoch are extracted. Similarly, the training of the DNN model in parallel is performed using the `train_deep_model` function. The accuracy, model, and CM for each epoch are obtained.

The total training time and time per epoch for the parallel execution of the DNN model are recorded and stored.

Finally, the results from the parallel execution of the DNN model are evaluated, and the accuracy, TPR, FPR, TNR, FNR, and error rate for the final epoch are extracted.

These parallel execution results provide insights into the training efficiency and speedup achieved through parallelism.

4: Extending Parallel learning code to Federated Learning Implementation.

The Solution to this task is in an IPYNB file named **2.4_Course_work.ipynb**.

Whilst trying to implement Federated learning on the parallel training for KNN and DNN, the models chosen for SL and DL respectively, I encountered an error, that prevented the DNN model from running beyond 1 epoch.

The **2.4_Course_work.ipynb** file contain the functional code for KNN parallel implemented with DL and the output.

It also contains the result of the first epoch of DNN model.

The FL model was run in 4 rounds amongst 10 clients to see if aggregation can improve the accuracy.

The results obtained were then compared to Figure 32 which shows the results from executing KNN in parallel

Execute FL on SL (KNN)

```
fl_knn()
2023-05-27 17:13:33,748 INFO worker.py:1625 -- Started a local Ray instance.
Round: 1 Average Accuracy: 0.9158549396982811 Total time: 84.37151384353638 seconds
Round: 2 Average Accuracy: 0.9158549396982811 Total time: 80.81651782989502 seconds
Round: 3 Average Accuracy: 0.9158549396982811 Total time: 81.24969291687012 seconds
Round: 4 Average Accuracy: 0.9158549396982811 Total time: 81.79906272888184 seconds
Test Accuracy: 0.9548262386015044
***** The value of Accuracy = 0.9548262386015044 *****
```

Figure 35: Result from implementing FL on KNN in parallel

From Figure 35, we can see that when FL increases the time at each round (epoch) neutralizing the inherent parallelism features of Ray, and there's a decrease in accuracy, applying Federated Learning on a SL, KNN in this case does not do it any benefit.

```

Model: "sequential_55"
Layer (type)                Output Shape                Param #
=====
dense_220 (Dense)           (None, 64)                  2624
dense_221 (Dense)           (None, 32)                  2080
dense_222 (Dense)           (None, 16)                  528
dense_223 (Dense)           (None, 2)                   34
=====
Total params: 5,266
Trainable params: 5,266
Non-trainable params: 0

```

```

(pid=1698) 2023-05-29 15:00:37.351100: I tensorflow/core/platform/cpu_feature_guard.cc:111
with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in
4.1 SSE4.2 AVX AVX2 FMA
(pid=1698) To enable them in other operations, rebuild TensorFlow with the appropriate co
(train_deep_model pid=1698) 2023-05-29 15:00:46.702213: I tensorflow/core/platform/cpu_fe
ary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CI
operations: SSE4.1 SSE4.2 AVX AVX2 FMA [repeated 8x across cluster]
(train_deep_model pid=1698) To enable them in other operations, rebuild TensorFlow with
d 8x across cluster]
Round: 1 Average Accuracy: 0.9467243790626526 Total time: 493.97032713890076 seconds

```

Figure 36: Result from implementing FL on DNN in parallel

From Figure 35, when compared to Figure 33 shows that FL marginally increases the accuracy on DL in parallel (0.9467) vs (0.9437), but it increases the time. From this we can infer that implementing FL on a parallel DL trade off time for accuracy.