MLP for Regression and Classification

CS550 Assignment 3

Name: Sudhir Sharma

ID: 12041500

```
# !pip install -q kaggle
```

```
# from google.colab import files
# files.upload()
```

```
# !mkdir ~/.kaggle
# !cp kaggle.json ~/.kaggle/
# !chmod 600 ~/.kaggle/kaggle.json
```

```
# !kaggle competitions download -c new-york-city-taxi-fare-prediction
```

```
# !unzip new-york-city-taxi-fare-prediction
```

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-pythor
# For example, here's several helpful packages to load in
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns


import math
from math import sqrt

from numpy import absolute
from numpy import mean
from numpy import std

from sklearn import metrics
from sklearn.feature_selection import f_regression
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import Ridge
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import RepeatedKFold
from sklearn import neighbors

!pip install tensorflow
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras import Sequential
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.losses import MeanSquaredError
from keras.layers import BatchNormalization
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from tensorflow.keras.optimizers import Adam
from keras.callbacks import EarlyStopping
!pip install haversine
from tensorflow.random import set_seed
!pip install xgboost
import xgboost as xgb
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/
Requirement already satisfied: tensorflow in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: gast>=0.2.1 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: tensorboard<2.9,>=2.8 in /usr/local/lib/python3.7/dis
Requirement already satisfied: protobuf<3.20,>=3.9.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: absl-py>=0.4.0 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/li
Requirement already satisfied: flatbuffers>=1.12 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: keras<2.9,>=2.8.0rc0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: tensorflow-estimator<2.9,>=2.8 in /usr/local/lib/pyth
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: keras-preprocessing>=1.1.1 in /usr/local/lib/python3.
Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.7/
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/pytho
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dist-pa
```

```
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/lib/py
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.7/dis
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/local/l
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.7/
Requirement already satisfied: importlib-metadata>=4.4 in /usr/local/lib/python3.7/d
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-pack
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/
Requirement already satisfied: haversine in /usr/local/lib/python3.7/dist-packages (
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/
Requirement already satisfied: xgboost in /usr/local/lib/python3.7/dist-packages (0.
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from
```

Part 1: Predicting Taxi Fares

Implement MLP model using Keras

- The implementation of MLP model in Keras comprises of three steps:-
  - Compiling the model with the compile() method.
  - Training the model with fit() method.
  - Evaluating the model performance with evaluate() method.

Reference https://www.kaggle.com/code/afrinp/nyc-taxi-fare-prediction-5-lakh-rows#Making-ML-Model

```
import warnings
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import seaborn as sns


from keras.models import Sequential
```

```
from keras.layers import Dense,LSTM, TimeDistributed, Flatten, MaxPooling1D,Conv1D,Dropout
from sklearn.metrics import precision_recall_fscore_support, precision_score, recall_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression, Ridge, Lasso,ElasticNet,HuberRegressor,
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeRegressor,ExtraTreeRegressor
from sklearn.svm import SVR
from sklearn.ensemble import AdaBoostRegressor,BaggingRegressor,RandomForestRegressor,Extr
from sklearn.model_selection import train_test_split,GridSearchCV,RandomizedSearchCV
from sklearn.preprocessing import StandardScaler
from statsmodels.tsa.arima.model import ARIMA
from math import radians, cos, sin, asin, sqrt
pd.set_option('display.float_format', lambda x: '%.3f' % x)
warnings.filterwarnings("ignore")
%matplotlib inline
```

There is a lot of data, which will require a lot of computing resources, so I took only a part of itloading data

```
# loading train data
my_dataframe = pd.read_csv('train.csv',nrows = 1000000)
# loading test data
testdf = pd.read_csv('test.csv')
```

will only be including 1,000,000 rows in this notebook due to size constraints

```
print(f'Number of records: {my_dataframe.shape[0]}')
print(f'Number of columns: {my_dataframe.shape[1]}')
```

```
Number of records: 1000000
Number of columns: 8
```

```
my_dataframe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 8 columns):
 #   Column             Non-Null Count    Dtype  
---  ------             --------------    -----  
 0   key                1000000 non-null  object 
 1   fare_amount        1000000 non-null  float64
 2   pickup_datetime    1000000 non-null  object 
 3   pickup_longitude   1000000 non-null  float64
 4   pickup_latitude    1000000 non-null  float64
 5   dropoff_longitude  999990 non-null   float64
 6   dropoff_latitude   999990 non-null   float64
 7   passenger_count    1000000 non-null  int64  
dtypes: float64(5), int64(1), object(2)
memory usage: 61.0+ MB
```

Info about our data

```
my_dataframe.describe()
```

	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_la
count	1000000.000	1000000.000	1000000.000	999990.000	999
mean	11.348	-72.527	39.929	-72.528	
std	9.822	12.058	7.626	11.324	
min	-44.900	-3377.681	-3116.285	-3383.297	-3
25%	6.000	-73.992	40.735	-73.991	
50%	8.500	-73.982	40.753	-73.980	
75%	12.500	-73.967	40.767	-73.964	
max	500.000	2522.271	2621.628	45.582	1

```
my_dataframe.head()
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitud
0	2009-06-15 17:26:21.0000001	4.500	2009-06-15 17:26:21 UTC	-73.844	40.72
1	2010-01-05 16:52:16.0000002	16.900	2010-01-05 16:52:16 UTC	-74.016	40.71
2	2011-08-18 00:35:00.00000049	5.700	2011-08-18 00:35:00 UTC	-73.983	40.76

Statistical summary of data

```
my_dataframe[my_dataframe.isnull().any(1)]
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_lat
120227	2012-12-11 12:57:00.00000013	12.500	2012-12-11 12:57:00 UTC	-73.993	

From this we learn that

- The minimum fare is negative, which is impossible
- Some travel points are missing the city
- The maximum number of passengers is equal to 208, which is impossible
- The maximum fare also unreal

471472		7.800		0.000

A Data Cleaning & Feature Engineering

Rows with missing values

```
my_dataframe.columns[my_dataframe.isnull().any()]
```

```
Index(['dropoff_longitude', 'dropoff_latitude'], dtype='object')
```

columns with null values

```
my_dataframe1 = my_dataframe[~my_dataframe.isnull().any(1)]
```

Dropping null values rows

longitude and latitude

found NaN values in columns dropoff_longitude and dropoff_latitude
We found NaN values in columns

i) dropoff_longitude and

ii) dropoff_latitude

which is not much as comapared to our trainset. So we will Drop it.

```
# swap these values
wrong_location = my_dataframe1[((my_dataframe1['dropoff_latitude'] < 0) | (my_dataframe1['
```

```
# swap columns
wrong_location.columns = ['key','fare_amount',"pickup_datetime","pickup_latitude","pickup_
                          "dropoff_latitude","dropoff_longitude","passenger_count"]
```

```
# merge these values back into original my_dataframe
my_dataframe1.loc[my_dataframe1.index.isin(wrong_location.index),["pickup_latitude","picku

# remaining odd coordinates, drop them
my_dataframe1[((my_dataframe1['dropoff_latitude'] < 0) | (my_dataframe1['pickup_latitude']
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_la
66433	2010-07-28 09:22:00.000000254	7.300	2010-07-28 09:22:00 UTC	-0.004	
98000	2011-10-16 19:39:00.00000069	5.700	2011-10-16 19:39:00 UTC	0.003	
174356	2011-11-21 21:36:00.00000081	9.700	2011-11-21 21:36:00 UTC	1703.093	2
444028	2010-11-28 12:16:00.000000129	5.700	2010-11-28 12:16:00 UTC	-0.002	
549740	2011-04-21 15:58:00.00000020	5.700	2011-04-21 15:58:00 UTC	0.000	

```
# drop the remaining 77 rows
drop = my_dataframe1[((my_dataframe1['dropoff_latitude'] < 0) | (my_dataframe1['pickup_lat
my_dataframe1 = my_dataframe1[~my_dataframe1.index.isin(drop.index)]
```

Longitudes should be negative and latitudes should be positive

```
my_dataframe1 = my_dataframe1.drop(my_dataframe1[(my_dataframe1['dropoff_latitude'] == 0)
```

Since range of longitudes and latitudes for cities in USA is between -125 & -67 and 24 & 50 respectively, remove all the other records that fall outside of these ranges

dropping these records that sit in the ATLANTIC OCEAN:

```
my_dataframe1.head()
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitud
0	2009-06-15 17:26:21.0000001	4.500	2009-06-15 17:26:21 UTC	-73.844	40.72
1	2010-01-05 16:52:16.0000002	16.900	2010-01-05 16:52:16 UTC	-74.016	40.71
2	2011-08-18 00:35:00.00000049	5.700	2011-08-18 00:35:00 UTC	-73.983	40.76

```
my_dataframe1[((my_dataframe1["pickup_latitude"] < 24) & (my_dataframe1["pickup_latitude"]
```

key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_long

```
my_dataframe1.loc[((my_dataframe1["pickup_longitude"] < -125) & (my_dataframe1["pickup_lor
```

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_la
1260	2011-03-10 20:25:00.00000049	5.700	2011-03-10 20:25:00 UTC	-73.974	
2280	2011-08-29 08:24:00.000000107	8.900	2011-08-29 08:24:00 UTC	-73.937	
4278	2015-04-07 23:33:02.0000005	7.000	2015-04-07 23:33:02 UTC	-73.973	
8647	2014-03-27 18:01:00.00000071	21.500	2014-03-27 18:01:00 UTC	-74.002	
10458	2013-02-23 20:58:00.000000150	2.500	2013-02-23 20:58:00 UTC	-73.980	
...	...	...	...	...	
993372	2014-03-26 08:08:00.000000105	9.500	2014-03-26 08:08:00 UTC	-73.952	
993672	2014-04-22 12:47:34.0000003	5.500	2014-04-22 12:47:34 UTC	-73.992	

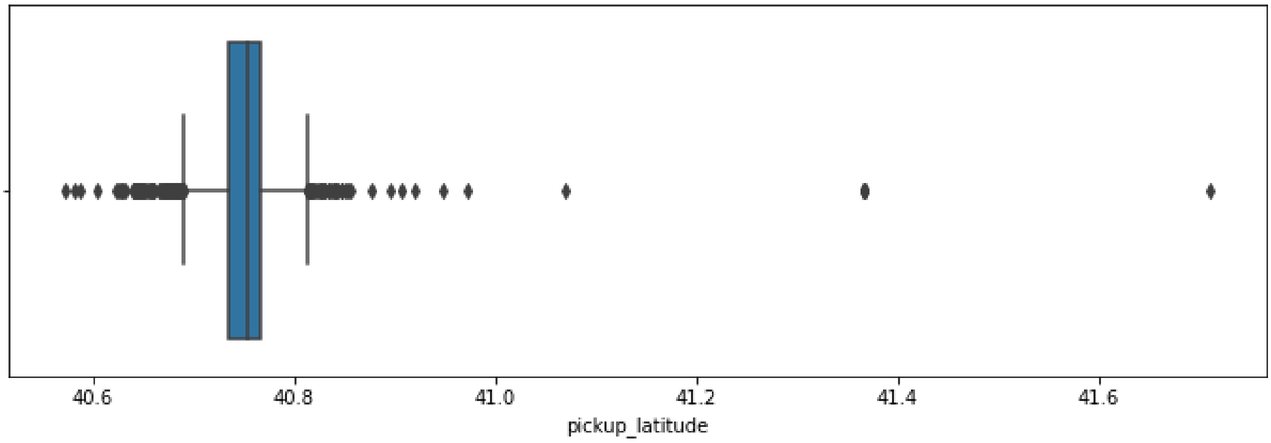
dropping odd latitudes and odd longitudes

```
drop = my_dataframe1.loc[((my_dataframe1["pickup_longitude"] < -125) & (my_dataframe1["pic
my_dataframe1 = my_dataframe1[~my_dataframe1.index.isin(drop.index)]
```

Drop records that fall outside of testdf's coordinates

```
fig,ax= plt.subplots(2,figsize = (12,8))
sns.boxplot(testdf['pickup_latitude'],ax = ax[0])
sns.boxplot(testdf['pickup_longitude'],ax = ax[1])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb836697810>
```



looking at range of pickup latitude and longitude in test set

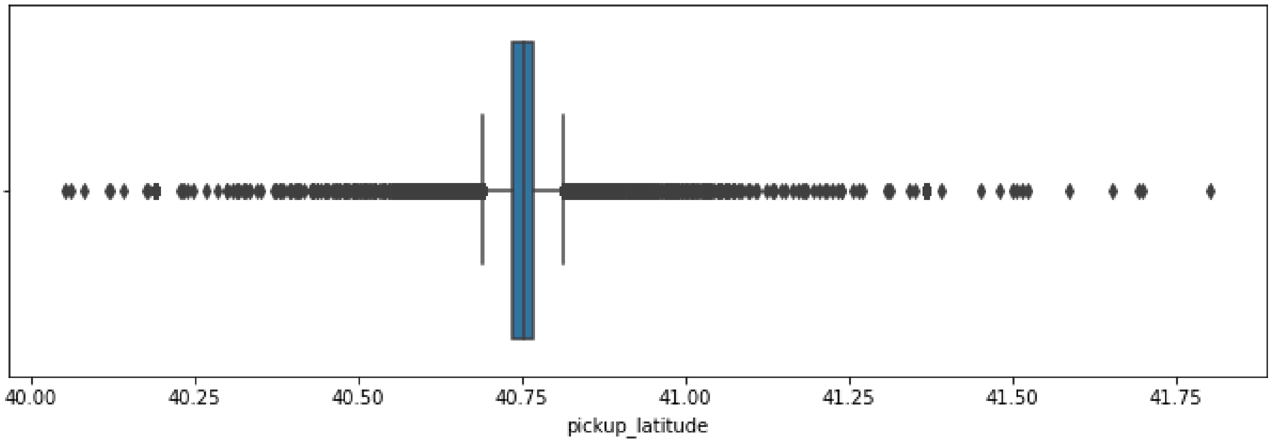
```
testdf.describe()
```

	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passe
count	9914.000	9914.000	9914.000	9914.000	
mean	-73.975	40.751	-73.974	40.752	
std	0.043	0.034	0.039	0.035	
min	-74.252	40.573	-74.263	40.569	
25%	-73.993	40.736	-73.991	40.735	
50%	-73.982	40.753	-73.980	40.754	
75%	-73.968	40.767	-73.964	40.769	
max	-72.987	41.710	-72.991	41.697	

```
my_dataframe1 = my_dataframe1[((my_dataframe1['pickup_longitude'] > -75) & (my_dataframe1[
```

```
fig,ax= plt.subplots(2,figsize = (12,8))
sns.boxplot(my_dataframe1['pickup_latitude'],ax = ax[0])
sns.boxplot(my_dataframe1['pickup_longitude'],ax = ax[1])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb8360f3bd0>
```



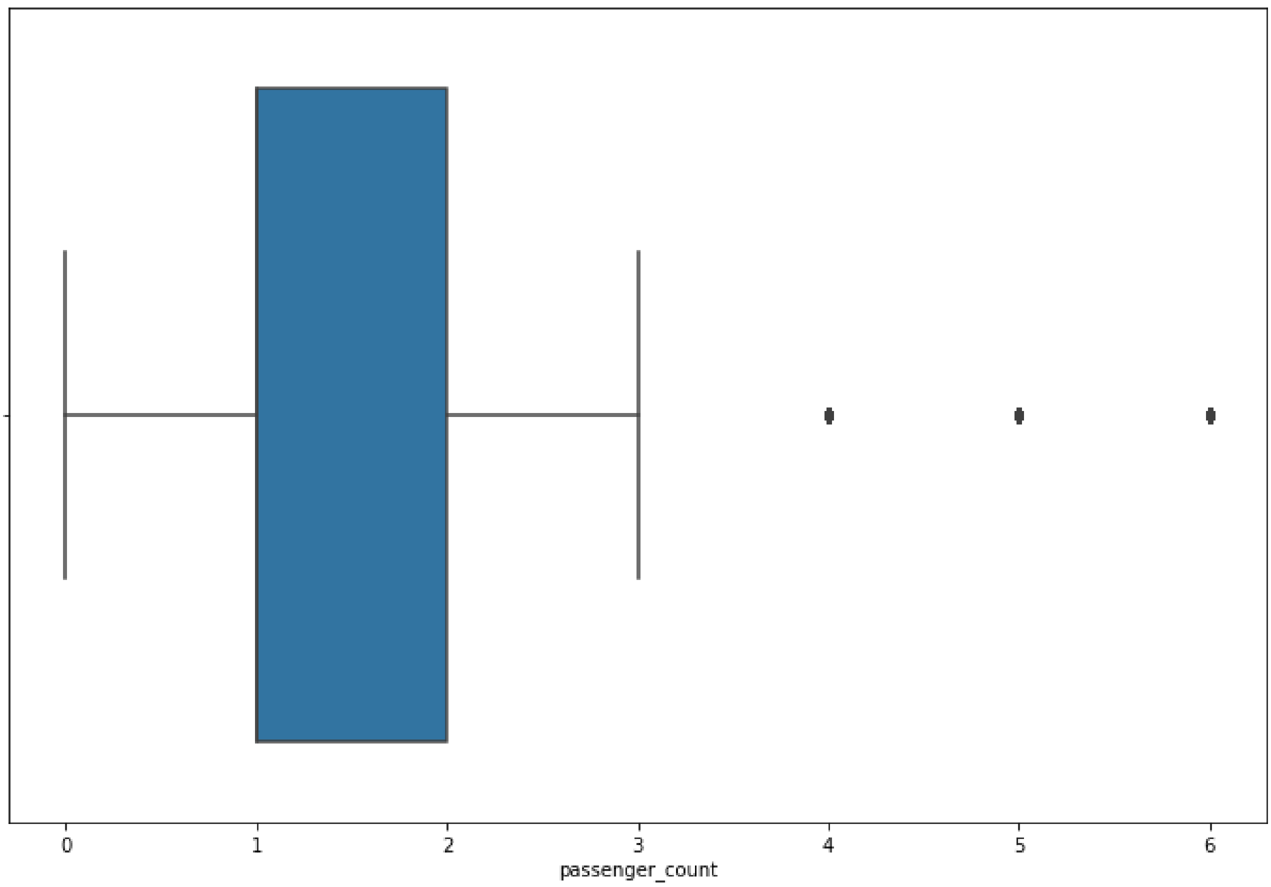
looking at range of pickup latitude and longitude in test se

```
my_dataframe1 = my_dataframe1.drop(my_dataframe1[my_dataframe1['fare_amount'] <= 0].index)
```

Dropping unrealistic and negative cab fares

```
fig,ax= plt.subplots(figsize = (12,8))
sns.boxplot(my_dataframe1['passenger_count'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb8360959d0>
```



```
my_dataframe1[my_dataframe1['passenger_count'] >50]
```

key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_long

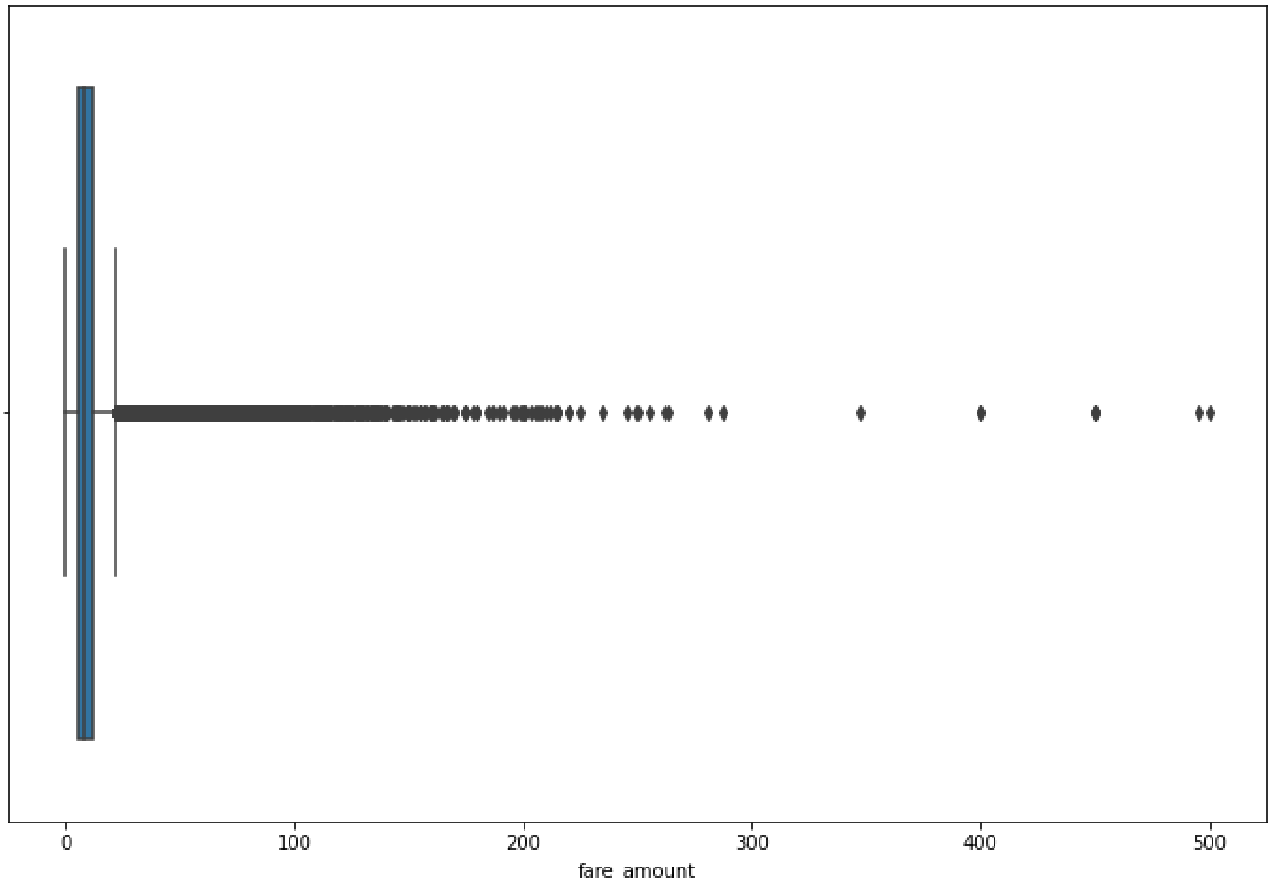
Dropping unrealistic passenger count

```
my_dataframe1 = my_dataframe1.drop(my_dataframe1[my_dataframe1['passenger_count'] > 50].ir
```

dropping the 2 extreme values

```
fig,ax= plt.subplots(figsize = (12,8))
sns.boxplot(my_dataframe1['fare_amount'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb83671f5d0>
```

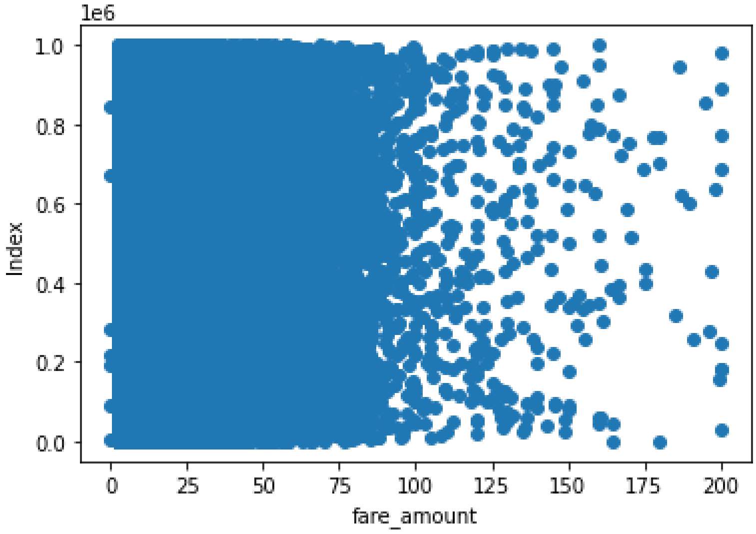


There will be no negative tax and you may not be able to pay more than a certain limit depending on the circumstances, let's say this limit is 200\$

Also, I came to know from google that the minimum fare for a New York taxi is 2,50\$

```
# Assumption: cab fares are all below $200
my_dataframe1 = my_dataframe1.drop(my_dataframe1[my_dataframe1['fare_amount'] > 200].inde>
```

```
plt.scatter(x=my_dataframe1.fare_amount,y=my_dataframe1.index)
plt.ylabel('Index')
plt.xlabel('fare_amount')
plt.show()
```




```
print("Number of Fare_amount <=0 is")
my_dataframe1['fare_amount'][(my_dataframe1.fare_amount<=0)].count()
```

```
Number of Fare_amount <=0 is
0
```

Final checking if still some outliers are left

```
print('Number of observations out of valid range in coordinate columns:', end="\n")

print('pickup_longitude', end=': ')
print((my_dataframe1.pickup_longitude <-180).sum()+(my_dataframe1.pickup_longitude > 180).

print('pickup_latitude', end=': ')
print((my_dataframe1.pickup_latitude <-90).sum()+(my_dataframe1.pickup_latitude > 90).sum(

print('dropoff_longitude', end=': ')
print((my_dataframe1.dropoff_longitude <-180).sum()+(my_dataframe1.dropoff_longitude > 18(

print('dropoff_latitude', end=': ')
print((my_dataframe1.dropoff_latitude <-90).sum()+(my_dataframe1.dropoff_latitude > 90).su

if(((my_dataframe1.pickup_longitude <-180).sum()+(my_dataframe1.pickup_longitude > 180).su
    print("No OutLiers Left")
else:
    print("Outliers Still Left")
```

```
Number of observations out of valid range in coordinate columns:
pickup_longitude: 0
pickup_latitude: 0
dropoff_longitude: 0
```

```
dropoff_latitude: 0
No OutLiers Left
```

Dropping unrealitstic fare amount

Feature Engineering

Changing datatypes and creating new fields

```
pd.to_datetime(pd.to_datetime(my_dataframe1.head()['pickup_datetime']).dt.strftime("%Y-%m-
```

```
0   2009-06-15 17:26:00
1   2010-01-05 16:52:00
2   2011-08-18 00:35:00
3   2012-04-21 04:30:00
4   2010-03-09 07:51:00
Name: pickup_datetime, dtype: datetime64[ns]
```

```
my_dataframe1['pickup_datetime'] = pd.to_datetime(pd.to_datetime(my_dataframe1['pickup_dat
testdf['pickup_datetime'] = pd.to_datetime(pd.to_datetime(testdf['pickup_datetime']).dt.st
```

```
my_dataframe1['year'] = my_dataframe1['pickup_datetime'].dt.year
my_dataframe1['month'] = my_dataframe1['pickup_datetime'].dt.month
my_dataframe1['day'] = my_dataframe1['pickup_datetime'].dt.day
my_dataframe1['weekday'] = my_dataframe1['pickup_datetime'].dt.weekday
my_dataframe1['hour'] = my_dataframe1['pickup_datetime'].dt.hour
my_dataframe1['min'] = my_dataframe1['pickup_datetime'].dt.minute

testdf['year'] = testdf['pickup_datetime'].dt.year
testdf['month'] = testdf['pickup_datetime'].dt.month
testdf['day'] = testdf['pickup_datetime'].dt.day
testdf['weekday'] = testdf['pickup_datetime'].dt.weekday
testdf['hour'] = testdf['pickup_datetime'].dt.hour
testdf['min'] = testdf['pickup_datetime'].dt.minute
```

We can understand displacement through start and end points.

We will use the Haversine formula to calculate the distance between two geolocations

*Distance *

Calculate the distance based on longitude and latitude

Haversine formula:

dlon = lon2 - lon1 dlat = lat2 - lat1 a = (sin(dlat/2))^2 + cos(lat1) * cos(lat2) * (sin(dlon/2))^2 c = 2 * atan2(sqrt(a), sqrt(1-a)) d = R * c (where R is the radius of the Earth)

$a = \sin^2(\Delta\varphi/2) + \cos\varphi1 \cdot \cos\varphi2 \cdot \sin^2(\Delta\lambda/2)$

$c = 2 \cdot \mathrm{atan2}(\sqrt{a}, \sqrt{(1-a)})$

$d = R \cdot c$

```
# define haversine formula to convert points to distance in km
def haversine(my_dataframe2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1 = my_dataframe2['pickup_longitude']
    lon2 = my_dataframe2['dropoff_longitude']
    lat1 = my_dataframe2['pickup_latitude']
    lat2 = my_dataframe2['dropoff_latitude']

    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat2 = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers.
    return c * r
```

Let's also calculate the distance using the Chebyshev method

The Chebyshev iteration is an iterative method for determining the solutions of a system of linear equations.

Reference https://brilliant.org/wiki/chebyshevs-formula/#:~:text=x%20%3D%20a%20%2B%20b%202%20%2B,%3D1%2C%20t%3D1%2C&text=x%3Db.,-Hence%2C

```
def chebyshev(pickup_long, dropoff_long, pickup_lat, dropoff_lat):
    return np.maximum(np.absolute(pickup_long - dropoff_long), np.absolute(pickup_lat - dr
```

```
my_dataframe1['distance'] = my_dataframe1.apply(haversine,axis = 1)
```

```
my_dataframe2 = my_dataframe1.copy()
my_dataframe2 = my_dataframe2.drop(columns = ['pickup_latitude','pickup_longitude','dropo
```

Dropping correlated and redundant columns

```
testdf['distance'] = testdf.apply(haversine,axis = 1)
testdf = testdf.drop(columns = ['pickup_latitude','pickup_longitude','dropoff_latitude','c
```

Data modeling

```
X = my_dataframe2.drop(columns = ['fare_amount','key','pickup_datetime'])
y = my_dataframe2['fare_amount']
```

```
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state = 42)
```

```
# looking at rows, columns for train and validation set
print(f'train: {X_train.shape}')
print(f'test: {y_train.shape}')
print(f'val train: {X_test.shape}')
print(f'val test: {y_test.shape}')
```

```
train: (734713, 9)
test: (734713,)
val train: (244905, 9)
val test: (244905,)
```

Neural Nets

```
X_train.head()
```

	passenger_count	year	month	day	weekday	hour	min	Chebyshev	distance
623653	1	2009	9	26	5	13	29	0.011	1.392
275569	1	2010	9	5	6	14	16	0.027	2.771
87623	1	2011	9	15	3	16	32	0.018	1.542
913744	1	2012	5	29	1	2	54	0.007	0.803
390215	1	2010	4	3	5	9	38	0.008	0.885

```
ss = StandardScaler()
ss.fit(X_train)
X_train_ss = ss.transform(X_train)
X_test_ss = ss.transform(X_test)
```

Scaling the data

```
def get_models(my_models=dict()):
    my_models['lr'] = LinearRegression()
    my_models['lasso'] = Lasso()
    my_models['ridge'] = Ridge()
    my_models['en'] = ElasticNet()
    my_models['huber'] = HuberRegressor()
    my_models['pa'] = PassiveAggressiveRegressor(max_iter=1000, tol=1e-3)

    return my_models
```

Linear Model

```
def get_models_nl(my_models=dict()):
    my_models['svr'] = SVR()
    n_trees = 100
    my_models['ada'] = AdaBoostRegressor(n_estimators=n_trees)
    my_models['bag'] = BaggingRegressor(n_estimators=n_trees)
    my_models['rf'] = RandomForestRegressor(n_estimators=n_trees)
    my_models['et'] = ExtraTreesRegressor(n_estimators=n_trees)
    my_models['gbm'] = GradientBoostingRegressor(n_estimators=n_trees)
    return my_models
```

Non linear model

```
def evaluate_models(my_models, X_train_ss,y_train,X_test_ss,y_test):
    for my_name, model in my_models.items():
        model_fit = model.fit(X_train_ss,y_train)
        # making the predictions
        train_prediction = model_fit.predict(X_train_ss)
        test_prediction = model_fit.predict(X_test_ss)
        # evaluating the forecast
        train_mse = mean_squared_error(y_train,train_prediction)
        test_mse = mean_squared_error(y_test,test_prediction)
        print(f'{my_name}:')
        print(f'----')
        print(f'Train MAE: {round(train_mse,2)}')
        print(f'Test MAE: {round(test_mse,2)}')
        print(f'\n')
```

Fit Model

```
def pipeline(model):
    pipe = Pipeline([(model, model_dict[model])])
    return pipe
```

Defining the pipeline

```
def params(mt_model):


    if mt_model == 'lasso':
        return {"alpha":[0.01,0.1,1,2,5,10],
               }


    elif mt_model == 'ridge':
        return {
            "alpha":[0.01,0.1,1,2,5,10],
            }

    elif mt_model == 'en':
        return {
            'alpha':[0.01,0.1,1,10],
            'l1_ratio':[0.2,0.3,0.4,0.5,0.6]
            }
    elif mt_model == 'knn':
        return {
            'n_neighbors':[4,5,6,7]}

    elif mt_model == 'dt':
        return {
            'max_depth':[3,4,5],
            'min_samples_split':[2,3,4],
            'min_samples_leaf':[2,3,4]
        }
    elif mt_model == 'bag':
        return {
            'max_features':[100, 150]
        }

    elif mt_model == 'rf':
        return {
            'n_estimators':[100,150],
            'max_depth':[4],
            'min_samples_leaf':[2,3,4]
        }
    elif mt_model == 'et':
        return {
            'n_estimators':[50,100,150,200],
            'max_depth':[1000,2000,3000],
            'min_samples_leaf':[10000,20000,30000],
        }
    elif mt_model == 'abc':
        return {
            'n_estimators':[50,100,150,200],
            'learning_rate':[0.3,0.6,1]
        }
```

```
    elif mt_model == 'gbc':
        return {
            'learning_rate':[0.2],
            'max_depth':[1000,2000,3000],
            'min_samples_split':[10000,20000,30000]

        }
    elif mt_model == 'xgb':
        return {
            'eval_metric' : ['auc'],
            'subsample' : [0.8],
            'colsample_bytree' : [0.5],
            'learning_rate' : [0.1],
            'max_depth' : [5],
            'scale_pos_weight': [5],
            'n_estimators' : [100,200],
            'reg_alpha' : [0, 0.05],
            'reg_lambda' : [2,3],
            'gamma' : [0.01]

        }
    elif mt_model == 'svr':
        return {
            'kernel': ['rbf', 'linear','poly'],
            'C': [1,20,50,100],
            'gamma':['scale','auto'],
            'epsilon':[0.1,1,10]
        }
    elif mt_model == 'ada':
        return {
            'n_estimators':[50,100,150],
            'learning_rate':[0.01,0.1,1],

        }
    elif mt_model == 'bag':
        return {
            'n_estimators':[20,50,100,150],
            'max_features':[2,4,6],
            'max_samples':[0.1,0.2,0.3,0.5,0.7],
            'bootstrap':[True]

        }
    elif mt_model == 'rf':
        return {
             'bootstrap': [True],
             'max_depth': [5,10,15],
             'max_features': ["auto", "sqrt", "log2"],
             'min_samples_leaf': [10000,20000,30000],
             'min_samples_split': [10000,20000,30000],
             'n_estimators': [50,200,300,400],
             'random_state': 42,
             }
    elif mt_model == 'et':
        return {
             'bootstrap': [True],
```

```
            'max_depth': [5,10,15],
            'max_features': ["auto", "sqrt", "log2"],
            'min_samples_leaf': [10000,20000,30000],
            'min_samples_split': [10000,20000,30000],
            'n_estimators': [50,200,300,400],
            'random_state': 42,
        }

    elif mt_model == 'gbm':
        return {
            'learning_rate' : [0.1,0.3,0.6,1],
            'min_samples_split':[10000,20000,30000],
            'min_samples_leaf': [10000,20000,30000],
            'max_depth' : [8,10,20]
        }


def grid_search_rs(model,my_models,X_train = X_train_ss,y_train = y_train,X_test = X_test_
    pipe_params = params(model)
    model = my_models[model]
    grid_search = RandomizedSearchCV(model,param_distributions = pipe_params,cv = 5,scorir
    grid_search.fit(X_train_ss,y_train)
    train_score = grid_search.score(X_train_ss,y_train)
    test_score = grid_search.score(X_test_ss,y_test)

    print(f'Results from: {model}')
    print(f'----------------------------------')
    print(f'Best Hyperparameters: {grid_search.best_params_}')
    print(f'Mean MSE: {-round(grid_search.best_score_,4)}')
    print(f'Train Score: {-round(train_score,4)}')
    print(f'Test Score: {-round(test_score,4)}')
    print(' ')
```

grid search with randomizedsearchcv

A. (20 marks) Create a baseline Neural network with the following specifications

2 hidden layers: Each with 16 and 8 neurons respectively.

Sigmoid activation,

Batch Size=128 for Gradient Descent.

```
model1 = Sequential()
model1.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tra
model1.add(Dense(8,activation = 'sigmoid'))
model1.add(Dense(1))
model1.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model1 = model1.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 11s - loss: 77.4430 - mae: 4.3292 - mse: 77.4430 - mape: 34.0639 - val_l
Epoch 2/7
5740/5740 - 11s - loss: 25.5659 - mae: 2.3662 - mse: 25.5659 - mape: 23.0930 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 21.0575 - mae: 2.1423 - mse: 21.0575 - mape: 21.8667 - val_l
Epoch 4/7
5740/5740 - 11s - loss: 20.6969 - mae: 2.1108 - mse: 20.6969 - mape: 21.5974 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 20.5854 - mae: 2.1032 - mse: 20.5854 - mape: 21.6036 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 20.5326 - mae: 2.0975 - mse: 20.5326 - mape: 21.5332 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 20.4995 - mae: 2.0960 - mse: 20.4995 - mape: 21.5477 - val_l
```

```python
print(f'Train Score:{mean_squared_error(y_train,model1.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model1.predict(X_test_ss))}')
```

```
Train Score:20.468251750572843
Test Score:19.692928487927567
```

Minimizing the mean absolute error loss

```python
model2 = Sequential()
model2.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tra
model2.add(Dense(8,activation = 'sigmoid'))
model2.add(Dense(1))
model2.compile(loss = 'mae',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model2 = model2.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 10s - loss: 4.3433 - mae: 4.3433 - mse: 84.1605 - mape: 29.4184 - val_lo
Epoch 2/7
5740/5740 - 10s - loss: 2.3924 - mae: 2.3924 - mse: 32.1281 - mape: 18.5859 - val_lo
Epoch 3/7
5740/5740 - 11s - loss: 2.1019 - mae: 2.1019 - mse: 22.9514 - mape: 18.0639 - val_lo
Epoch 4/7
5740/5740 - 10s - loss: 2.0470 - mae: 2.0470 - mse: 21.8377 - mape: 17.9502 - val_lo
Epoch 5/7
5740/5740 - 10s - loss: 2.0433 - mae: 2.0433 - mse: 21.7460 - mape: 17.9257 - val_lo
Epoch 6/7
5740/5740 - 10s - loss: 2.0400 - mae: 2.0400 - mse: 21.6847 - mape: 17.9122 - val_lo
Epoch 7/7
5740/5740 - 10s - loss: 2.0358 - mae: 2.0358 - mse: 21.6177 - mape: 17.9181 - val_lo
```

```python
print(f'Train Score:{mean_squared_error(y_train,model2.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model2.predict(X_test_ss))}')
```

```
Train Score:21.642035444013487
Test Score:20.857208259413124
```

Minimizing the mean absolute percentage error loss

```
model3 = Sequential()
model3.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tra
model3.add(Dense(8,activation = 'sigmoid'))
model3.add(Dense(1))
model3.compile(loss = 'mape',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model3 = model3.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 11s - loss: 34.5016 - mae: 5.2974 - mse: 107.8736 - mape: 34.5016 - val_
Epoch 2/7
5740/5740 - 10s - loss: 20.4428 - mae: 3.3127 - mse: 63.4592 - mape: 20.4428 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 19.0038 - mae: 2.9003 - mse: 50.1242 - mape: 19.0038 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 18.3973 - mae: 2.6688 - mse: 42.1009 - mape: 18.3973 - val_l
Epoch 5/7
5740/5740 - 14s - loss: 18.0154 - mae: 2.5112 - mse: 36.7181 - mape: 18.0154 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 17.7860 - mae: 2.4091 - mse: 33.2900 - mape: 17.7860 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 17.6394 - mae: 2.3383 - mse: 30.8636 - mape: 17.6394 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model3.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model3.predict(X_test_ss))}')
```

```
Train Score:29.343349619657438
Test Score:28.649116110864686
```

B. (20 marks) Experiment with number of layers and neurons per layer to increase the performance metrics.

Taking the number of layers=7

neurons per layer=64,56,48,32,24,16,8

minimizing the mse loss

activation function: sigmoid

```
model4 = Sequential()
model4.add(Dense(64,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tra
model4.add(Dense(56,activation = 'sigmoid'))
model4.add(Dense(48,activation = 'sigmoid'))
model4.add(Dense(32,activation = 'sigmoid'))
model4.add(Dense(24,activation = 'sigmoid'))
model4.add(Dense(16,activation = 'sigmoid'))
```

```
model4.add(Dense(8,activation = 'sigmoid'))
model4.add(Dense(1))
model4.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model4 = model4.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 16s - loss: 101.3079 - mae: 6.0310 - mse: 101.3079 - mape: 59.1738 - val
Epoch 2/7
5740/5740 - 14s - loss: 75.8133 - mae: 4.8224 - mse: 75.8133 - mape: 48.9664 - val_l
Epoch 3/7
5740/5740 - 14s - loss: 28.1697 - mae: 2.4332 - mse: 28.1697 - mape: 22.8428 - val_l
Epoch 4/7
5740/5740 - 14s - loss: 21.6325 - mae: 2.2016 - mse: 21.6325 - mape: 22.2809 - val_l
Epoch 5/7
5740/5740 - 14s - loss: 20.8097 - mae: 2.1440 - mse: 20.8097 - mape: 21.9209 - val_l
Epoch 6/7
5740/5740 - 14s - loss: 20.3870 - mae: 2.0866 - mse: 20.3870 - mape: 21.4449 - val_l
Epoch 7/7
5740/5740 - 15s - loss: 20.2301 - mae: 2.0677 - mse: 20.2301 - mape: 21.3336 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model4.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model4.predict(X_test_ss))}')
```

```
Train Score:20.37257995993675
Test Score:19.5632091556042
```

Taking the number of layers=1

Neurons per layer=1024

minimizing the mse loss

```
model5 = Sequential()
model5.add(Dense(1024,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_t
model5.add(Dense(1))
model5.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model5 = model5.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 19s - loss: 24.3014 - mae: 2.3625 - mse: 24.3014 - mape: 23.1077 - val_l
Epoch 2/7
5740/5740 - 16s - loss: 21.9372 - mae: 2.2071 - mse: 21.9372 - mape: 21.7796 - val_l
Epoch 3/7
5740/5740 - 18s - loss: 21.4212 - mae: 2.1736 - mse: 21.4212 - mape: 21.8157 - val_l
Epoch 4/7
5740/5740 - 19s - loss: 21.1651 - mae: 2.1470 - mse: 21.1651 - mape: 21.7190 - val_l
Epoch 5/7
5740/5740 - 16s - loss: 21.0074 - mae: 2.1293 - mse: 21.0074 - mape: 21.5962 - val_l
Epoch 6/7
5740/5740 - 16s - loss: 20.8385 - mae: 2.1103 - mse: 20.8385 - mape: 21.3965 - val_l
Epoch 7/7
5740/5740 - 16s - loss: 20.7370 - mae: 2.0985 - mse: 20.7370 - mape: 21.2822 - val_l
```

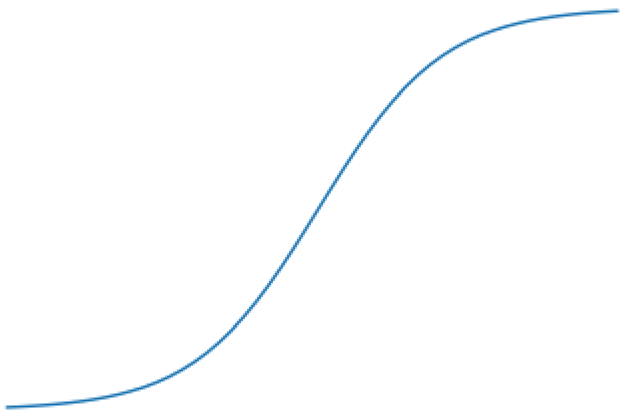
```
print(f'Train Score:{mean_squared_error(y_train,model5.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model5.predict(X_test_ss))}')
```

```
Train Score:20.588655554736523
Test Score:19.851468381369873
```

```
def logistic_func(x): return np.e**x/(np.e**x + 1)
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(np.arange(-5, 5, 0.2), [logistic_func(x) for x in np.arange(-5, 5, 0.2)])
plt.axis('off')
```

```
(-5.49, 5.290000000000008, -0.04256437797184291, 1.0410946577429678)
```



C. (10 marks) Experiment with activation functions

Calculating the MSE loss

Neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: tanh

```
model6 = Sequential()
model6.add(Dense(16,activation = 'tanh',kernel_initializer = 'normal',input_dim = X_train_
model6.add(Dense(8,activation = 'tanh'))
model6.add(Dense(1))
model6.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model = model6.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_da
```

```
Epoch 1/7
5740/5740 - 10s - loss: 58.8463 - mae: 3.5303 - mse: 58.8463 - mape: 27.8539 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 23.3596 - mae: 2.2446 - mse: 23.3596 - mape: 22.1031 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 20.8943 - mae: 2.1306 - mse: 20.8943 - mape: 21.9187 - val_l
Epoch 4/7
5740/5740 - 9s - loss: 20.5669 - mae: 2.1160 - mse: 20.5669 - mape: 21.9544 - val_lo
Epoch 5/7
5740/5740 - 10s - loss: 20.1555 - mae: 2.0883 - mse: 20.1555 - mape: 21.7756 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 19.7969 - mae: 2.0697 - mse: 19.7969 - mape: 21.6790 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 19.5541 - mae: 2.0581 - mse: 19.5541 - mape: 21.6504 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model6.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model6.predict(X_test_ss))}')
```

```
Train Score:19.4056099187136
Test Score:18.66270479161919
```

Calculating the MSE loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: relu

```
model7 = Sequential()
model7.add(Dense(16,activation = 'relu',kernel_initializer = 'normal',input_dim = X_train_
model7.add(Dense(8,activation = 'relu'))
model7.add(Dense(1))
model7.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model7 = model7.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_c
```

```
Epoch 1/7
5740/5740 - 11s - loss: 30.2392 - mae: 2.5367 - mse: 30.2392 - mape: 25.4569 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 21.6781 - mae: 2.1691 - mse: 21.6781 - mape: 22.1951 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 20.6062 - mae: 2.1383 - mse: 20.6062 - mape: 22.0189 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 20.3170 - mae: 2.1201 - mse: 20.3170 - mape: 21.8268 - val_l
Epoch 5/7
5740/5740 - 9s - loss: 20.1709 - mae: 2.1095 - mse: 20.1709 - mape: 21.6711 - val_lo
Epoch 6/7
5740/5740 - 10s - loss: 20.0816 - mae: 2.1068 - mse: 20.0816 - mape: 21.6301 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 20.0276 - mae: 2.1064 - mse: 20.0276 - mape: 21.6451 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model7.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model7.predict(X_test_ss))}')
```

```
Train Score:19.919556952833773
Test Score:19.342295217049163
```

calculating the mse loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: softsign

```
model8 = Sequential()
model8.add(Dense(16,activation = 'softsign',kernel_initializer = 'normal',input_dim = X_tr
model8.add(Dense(8,activation = 'softsign'))
model8.add(Dense(1))
model8.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model8 = model8.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_d
```

```
Epoch 1/7
5740/5740 - 11s - loss: 63.3270 - mae: 3.8109 - mse: 63.3270 - mape: 30.6851 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 24.5739 - mae: 2.3328 - mse: 24.5739 - mape: 22.5783 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 21.6243 - mae: 2.2191 - mse: 21.6243 - mape: 22.7047 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 21.2670 - mae: 2.1971 - mse: 21.2670 - mape: 22.5977 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 21.1438 - mae: 2.1827 - mse: 21.1438 - mape: 22.4101 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 21.0473 - mae: 2.1725 - mse: 21.0473 - mape: 22.3376 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 20.9665 - mae: 2.1643 - mse: 20.9665 - mape: 22.2774 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model8.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model8.predict(X_test_ss))}')
```

```
Train Score:20.922296613264468
Test Score:20.107495076046636
```

calculating the mse loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: elu

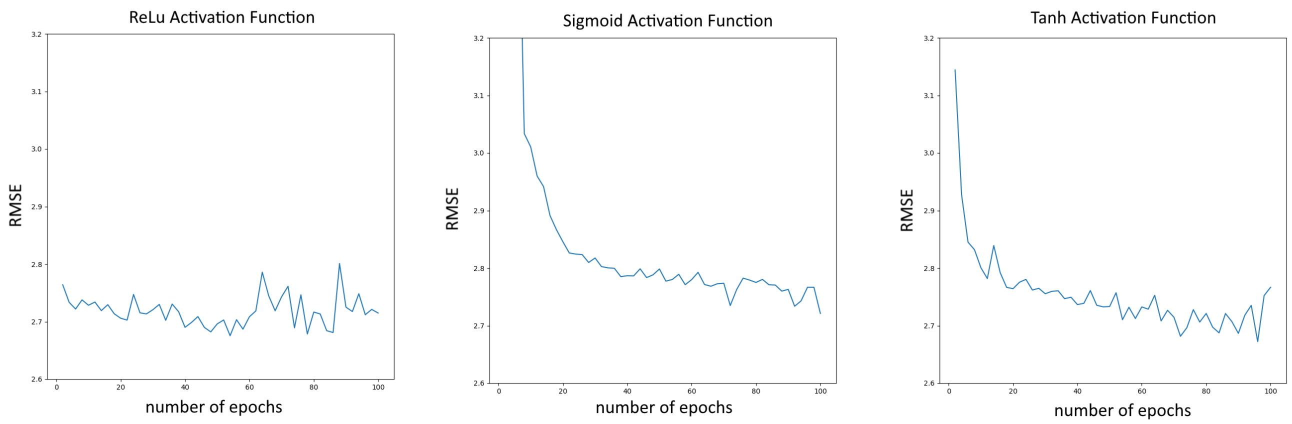
```
model9 = Sequential()
model9.add(Dense(16,activation = 'elu',kernel_initializer = 'normal',input_dim = X_train_s
model9.add(Dense(8,activation = 'elu'))
model9.add(Dense(1))
model9.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model9 = model9.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validation_d
```

```
Epoch 1/7
5740/5740 - 11s - loss: 28.2877 - mae: 2.4723 - mse: 28.2877 - mape: 23.5142 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 21.9024 - mae: 2.1413 - mse: 21.9024 - mape: 20.7639 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 20.7109 - mae: 2.1105 - mse: 20.7109 - mape: 21.1949 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 20.3731 - mae: 2.1040 - mse: 20.3731 - mape: 21.3858 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 20.1830 - mae: 2.0939 - mse: 20.1830 - mape: 21.3852 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 20.0351 - mae: 2.0877 - mse: 20.0351 - mape: 21.3898 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 19.9440 - mae: 2.0828 - mse: 19.9440 - mape: 21.3784 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model9.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model9.predict(X_test_ss))}')
```

```
Train Score:19.864358082783337
Test Score:19.26340119529084
```

Comparison of Activation Functions: The figure above gives a comparison of how the RMSE value developed with the number of epochs for the Relu, tanh and sigmoid activation functions, when training our Manhattan NN Model. The activation functions all behaved similarly, and we had no reason to prefer any one in particular and decided to use a tanh activation function.



D. (15 marks) Experiment with regularization techniques: Early stopping, Dropout rate

adding dropout=0.3 to the model

activation function as sigmoid

```
model10 = Sequential()
model10.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tr
model10.add(Dropout(0.3))
model10.add(Dense(8,activation = 'sigmoid'))
model10.add(Dense(1))
model10.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model10 = model10.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validatior
```

```
Epoch 1/7
5740/5740 - 11s - loss: 92.3021 - mae: 5.0233 - mse: 92.3021 - mape: 39.7041 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 31.8897 - mae: 2.6885 - mse: 31.8897 - mape: 25.5653 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 23.4229 - mae: 2.4008 - mse: 23.4229 - mape: 24.5455 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 22.3791 - mae: 2.3341 - mse: 22.3791 - mape: 24.1249 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 22.2051 - mae: 2.3169 - mse: 22.2051 - mape: 23.9062 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 22.0306 - mae: 2.3033 - mse: 22.0306 - mape: 23.7606 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 21.9522 - mae: 2.2905 - mse: 21.9522 - mape: 23.5712 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model10.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model10.predict(X_test_ss))}')
```

```
Train Score:21.056697701183502
Test Score:20.25563715127119
```

Tabulate the 95% confidence intervals of each of the 3 metrics from each of the parts above neatly based on at least 5 experiments on validation.

Applying early stopping with patience=3 and dropout=0.3

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
model11 = Sequential()
model11.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tr
model11.add(Dropout(0.3))
model11.add(Dense(8,activation = 'sigmoid'))
model11.add(Dense(1))
model11.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model11 = model11.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validatior
```

```
Epoch 1/7
5740/5740 - 10s - loss: 109.0595 - mae: 5.6949 - mse: 109.0595 - mape: 43.9009 - val
Epoch 2/7
5740/5740 - 10s - loss: 42.6676 - mae: 2.9943 - mse: 42.6676 - mape: 26.2622 - val_l
Epoch 3/7
```

```
5740/5740 - 10s - loss: 26.7108 - mae: 2.5136 - mse: 26.7108 - mape: 24.6561 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 23.1000 - mae: 2.3685 - mse: 23.1000 - mape: 24.1732 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 22.3368 - mae: 2.3185 - mse: 22.3368 - mape: 23.8682 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 22.1155 - mae: 2.3010 - mse: 22.1155 - mape: 23.6912 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 22.0135 - mae: 2.2939 - mse: 22.0135 - mape: 23.5931 - val_l
```

```
print(f'Train Score:{mean_squared_error(y_train,model11.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model11.predict(X_test_ss))}')
```

```
Train Score:21.126765773748005
Test Score:20.315663997780284
```

D. (15 marks) Experiment with regularization techniques: Early stopping, Dropout rate

Optimization (optimizer adam)

- With optimization, the objective is to minimize the loss function. The idea is that if the loss is reduced to an acceptable level, the model has indirectly learned the function mapping input to output.
- In Keras, there are several choices for optimizers. The most commonly used optimizers are; **Stochastic Gradient Descent (SGD)**, **Adaptive Moments (Adam)** and **Root Mean Squared Propagation (RMSprop)**.
- Each optimizer features tunable parameters like learning rate, momentum, and decay.
- Adam and RMSprop are variations of SGD with adaptive learning rates. In the proposed classifier network, Adam is used since it has the highest test accuracy.

Double-click (or enter) to edit

E. (10 marks) Experiment with at least 2 more Optimizers

calculating the mse loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: sigmoid

Optimizer: SGD

```
model12 = Sequential()
model12.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tr
model12.add(Dense(8,activation = 'sigmoid'))
model12.add(Dense(1))
model12.compile(loss = 'mse',optimizer = 'SGD',metrics = ['mae','mse','mape'])
history_model12 = model12.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validatior
```

```
Epoch 1/7
5740/5740 - 11s - loss: 23.6343 - mae: 2.3285 - mse: 23.6343 - mape: 23.8687 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 21.0868 - mae: 2.1767 - mse: 21.0868 - mape: 22.2845 - val_l
Epoch 3/7
5740/5740 - 10s - loss: 20.8949 - mae: 2.1497 - mse: 20.8949 - mape: 22.0779 - val_l
Epoch 4/7
5740/5740 - 10s - loss: 20.5059 - mae: 2.1010 - mse: 20.5059 - mape: 21.6545 - val_l
Epoch 5/7
5740/5740 - 9s - loss: 20.2914 - mae: 2.0771 - mse: 20.2914 - mape: 21.4570 - val_lo
Epoch 6/7
5740/5740 - 10s - loss: 20.1380 - mae: 2.0616 - mse: 20.1380 - mape: 21.3473 - val_l
Epoch 7/7
5740/5740 - 9s - loss: 20.1196 - mae: 2.0555 - mse: 20.1196 - mape: 21.2923 - val_lo
```

```
print(f'Train Score:{mean_squared_error(y_train,model12.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model12.predict(X_test_ss))}')
```

```
Train Score:19.951352787451576
Test Score:19.192778015734373
```

Metrics (accuracy)

- Performance metrics are used to determine if a model has learned the underlying data distribution. The default metric in Keras is loss.
- During training, validation, and testing, other metrics such as **accuracy** can also be included.
- **Accuracy** is the percent, or fraction, of correct predictions based on ground truth.

calculating the mse loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: sigmoid

Optimizer: Ftrl

```
model13 = Sequential()
```

```
model13.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tr
model13.add(Dense(8,activation = 'sigmoid'))
model13.add(Dense(1))
model13.compile(loss = 'mse',optimizer = 'Ftrl',metrics = ['mae','mse','mape'])
history_model13 = model13.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validatior
```

```
Epoch 1/7
5740/5740 - 11s - loss: 187.9994 - mae: 9.8120 - mse: 187.9994 - mape: 81.0599 - val
Epoch 2/7
5740/5740 - 10s - loss: 171.7602 - mae: 8.9749 - mse: 171.7602 - mape: 70.5827 - val
Epoch 3/7
5740/5740 - 10s - loss: 163.9362 - mae: 8.5294 - mse: 163.9362 - mape: 64.9923 - val
Epoch 4/7
5740/5740 - 9s - loss: 158.2960 - mae: 8.1923 - mse: 158.2960 - mape: 60.8062 - val_
Epoch 5/7
5740/5740 - 10s - loss: 153.6381 - mae: 7.9054 - mse: 153.6381 - mape: 57.2926 - val
Epoch 6/7
5740/5740 - 9s - loss: 149.7254 - mae: 7.6598 - mse: 149.7254 - mape: 54.3509 - val_
Epoch 7/7
5740/5740 - 9s - loss: 146.4107 - mae: 7.4489 - mse: 146.4107 - mape: 51.8978 - val_
```

```
print(f'Train Score:{mean_squared_error(y_train,model13.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model13.predict(X_test_ss))}')
```

```
Train Score:144.91412562950296
Test Score:144.26936916616563
```

calculating the mse loss

neural network has 2 hidden layers with 16 and 8 neurons respectively.

Activation function: sigmoid

Optimizer: Adagrad

```
model14 = Sequential()
model14.add(Dense(16,activation = 'sigmoid',kernel_initializer = 'normal',input_dim = X_tr
model14.add(Dense(8,activation = 'sigmoid'))
model14.add(Dense(1))
model14.compile(loss = 'mse',optimizer = 'Adagrad',metrics = ['mae','mse','mape'])
history_model14 = model14.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validatior
```

```
Epoch 1/7
5740/5740 - 10s - loss: 195.8697 - mae: 10.1944 - mse: 195.8697 - mape: 85.8223 - va
Epoch 2/7
5740/5740 - 10s - loss: 178.8448 - mae: 9.3488 - mse: 178.8448 - mape: 75.2495 - val
Epoch 3/7
5740/5740 - 10s - loss: 169.9536 - mae: 8.8642 - mse: 169.9536 - mape: 69.1400 - val
Epoch 4/7
5740/5740 - 9s - loss: 163.3903 - mae: 8.4864 - mse: 163.3903 - mape: 64.4069 - val_
Epoch 5/7
5740/5740 - 9s - loss: 157.7910 - mae: 8.1533 - mse: 157.7910 - mape: 60.2881 - val_
Epoch 6/7
```

```
5740/5740 - 9s - loss: 153.0510 - mae: 7.8629 - mse: 153.0510 - mape: 56.7506 - val_
Epoch 7/7
5740/5740 - 10s - loss: 149.0858 - mae: 7.6143 - mse: 149.0858 - mape: 53.7994 - val
```

```
print(f'Train Score:{mean_squared_error(y_train,model14.predict(X_train_ss))}')
print(f'Test Score:{mean_squared_error(y_test,model14.predict(X_test_ss))}')
```

```
Train Score:147.313845748648
Test Score:146.66527294882468
```

The best mse loss is 20.07, where the model's hyperparameters are:

2 hidden layers with 16 and 8 neurons respectively

activation function: tanh

```
final_model = Sequential()
final_model.add(Dense(16,activation = 'tanh',kernel_initializer = 'normal',input_dim = X_t
final_model.add(Dense(8,activation = 'tanh'))
final_model.add(Dense(1))
final_model.compile(loss = 'mse',optimizer = 'adam',metrics = ['mae','mse','mape'])
history_model = final_model.fit(X_train_ss,y_train, epochs = 7, batch_size = 128, validati
```

```
Epoch 1/7
5740/5740 - 10s - loss: 58.0160 - mae: 3.4902 - mse: 58.0160 - mape: 27.3835 - val_l
Epoch 2/7
5740/5740 - 10s - loss: 23.2750 - mae: 2.2424 - mse: 23.2750 - mape: 22.0564 - val_l
Epoch 3/7
5740/5740 - 9s - loss: 20.9031 - mae: 2.1310 - mse: 20.9031 - mape: 21.8082 - val_lo
Epoch 4/7
5740/5740 - 10s - loss: 20.4585 - mae: 2.1047 - mse: 20.4585 - mape: 21.6543 - val_l
Epoch 5/7
5740/5740 - 10s - loss: 20.2514 - mae: 2.0826 - mse: 20.2514 - mape: 21.4426 - val_l
Epoch 6/7
5740/5740 - 10s - loss: 20.1467 - mae: 2.0729 - mse: 20.1467 - mape: 21.3624 - val_l
Epoch 7/7
5740/5740 - 10s - loss: 20.0802 - mae: 2.0702 - mse: 20.0802 - mape: 21.3827 - val_l
```

Test set final prediction:

```
ss = StandardScaler()
ss.fit(X)
X_ss = ss.transform(X)
test_my_dataframe_ss = ss.transform(testdf.iloc[:,2:])
```

```
print(f'Shape of X: {X_ss.shape}')
print(f'Shape of y: {y.shape}')
```

```
Shape of X: (979618, 8)
Shape of y: (979618,)
```

```
ann_prediction = pd.DataFrame({"key": testdf['key'], "fare_amount":final_model.predict(tes
ann_prediction.to_csv("my_final_submission", index=False)
```

```
310/310 [==============================] - 0s 782us/step
```

```
final_model = Sequential()
final_model.add(Dense(64,activation = 'relu',kernel_initializer = 'normal',input_dim = X_s
final_model.add(Dropout(0.3))
final_model.add(Dense(3,activation = 'relu'))
final_model.add(Dense(1))
final_model.compile(loss = 'mse',optimizer = 'adam',metrics = 'mae')
history_final_model = final_model.fit(X_ss,y, epochs = 100, batch_size = 50000,verbose = 2
```

```
Epoch 1/100
20/20 - 1s - loss: 220.0970 - mae: 11.3110 - 1s/epoch - 52ms/step
Epoch 2/100
20/20 - 1s - loss: 216.2257 - mae: 11.1671 - 706ms/epoch - 35ms/step
Epoch 3/100
20/20 - 1s - loss: 211.4902 - mae: 10.9892 - 703ms/epoch - 35ms/step
Epoch 4/100
20/20 - 1s - loss: 205.1654 - mae: 10.7523 - 705ms/epoch - 35ms/step
Epoch 5/100
20/20 - 1s - loss: 196.6974 - mae: 10.4389 - 710ms/epoch - 35ms/step
Epoch 6/100
20/20 - 1s - loss: 185.3697 - mae: 10.0176 - 705ms/epoch - 35ms/step
Epoch 7/100
20/20 - 1s - loss: 170.9553 - mae: 9.4629 - 711ms/epoch - 36ms/step
Epoch 8/100
20/20 - 1s - loss: 153.4986 - mae: 8.7606 - 711ms/epoch - 36ms/step
Epoch 9/100
20/20 - 1s - loss: 133.7306 - mae: 7.9112 - 712ms/epoch - 36ms/step
Epoch 10/100
20/20 - 1s - loss: 111.8265 - mae: 6.9142 - 699ms/epoch - 35ms/step
Epoch 11/100
20/20 - 1s - loss: 88.7544 - mae: 5.8185 - 712ms/epoch - 36ms/step
Epoch 12/100
20/20 - 1s - loss: 67.9653 - mae: 4.8162 - 710ms/epoch - 36ms/step
Epoch 13/100
20/20 - 1s - loss: 52.5391 - mae: 4.0963 - 700ms/epoch - 35ms/step
Epoch 14/100
20/20 - 1s - loss: 43.7501 - mae: 3.7093 - 709ms/epoch - 35ms/step
Epoch 15/100
20/20 - 1s - loss: 39.6227 - mae: 3.5305 - 712ms/epoch - 36ms/step
Epoch 16/100
20/20 - 1s - loss: 37.7778 - mae: 3.4175 - 705ms/epoch - 35ms/step
Epoch 17/100
20/20 - 1s - loss: 36.6897 - mae: 3.3061 - 704ms/epoch - 35ms/step
Epoch 18/100
20/20 - 1s - loss: 35.8920 - mae: 3.2040 - 717ms/epoch - 36ms/step
Epoch 19/100
```

```
20/20 - 1s - loss: 35.2001 - mae: 3.1247 - 720ms/epoch - 36ms/step
Epoch 20/100
20/20 - 1s - loss: 34.9262 - mae: 3.0573 - 708ms/epoch - 35ms/step
Epoch 21/100
20/20 - 1s - loss: 34.7468 - mae: 3.0044 - 711ms/epoch - 36ms/step
Epoch 22/100
20/20 - 1s - loss: 34.4754 - mae: 2.9660 - 713ms/epoch - 36ms/step
Epoch 23/100
20/20 - 1s - loss: 34.2844 - mae: 2.9315 - 708ms/epoch - 35ms/step
Epoch 24/100
20/20 - 1s - loss: 34.0972 - mae: 2.9044 - 704ms/epoch - 35ms/step
Epoch 25/100
20/20 - 1s - loss: 33.9077 - mae: 2.8816 - 707ms/epoch - 35ms/step
Epoch 26/100
20/20 - 1s - loss: 33.7771 - mae: 2.8627 - 711ms/epoch - 36ms/step
Epoch 27/100
20/20 - 1s - loss: 33.6204 - mae: 2.8460 - 706ms/epoch - 35ms/step
Epoch 28/100
20/20 - 1s - loss: 33.4455 - mae: 2.8285 - 708ms/epoch - 35ms/step
Epoch 29/100
```

```
ann_prediction = pd.DataFrame({"key": testdf['key'], "fare_amount":final_model.predict(tes
ann_prediction.to_csv("taxi_fare_prediction.csv", index=False)
```

```
310/310 [==============================] - 1s 2ms/step
```

Saving the prediction in the taxi_fare_prediction file

Part 2: Breaking hcaptcha

```
import os
classes = os.listdir("hcaptcha_dataset/train/")
print(classes)
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

There are 7 classes to the dataset.\ Taking 20% data as Testing and taking the remaining 80% for Training.

For splitting out dataset into 2 categories we will use the split-folders package in python. It can be downloaded using pip install split-folders

```
%%capture
!pip install split-folders
```

```
%%capture
import splitfolders
splitfolders.ratio("hcaptcha_dataset/train/", output="./", seed=1337, ratio=(0.8, 0.2))
```

```
os.listdir("./train")
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

```
os.listdir("./val")
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

(20 marks) Create a baseline Neural network with the following specifications.

```
labels = os.listdir("./train")
print(labels)
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

labels in dataset

```
import pandas as pd

names=[]
my_train=[]
my_test=[]
my_total = []

for name in labels:
    im_num_train = len(os.listdir(f"./train/{name}"))
    im_num_test = len(os.listdir(f"./val/{name}"))

    names.append(name)
    my_train.append(im_num_train)
    my_test.append(im_num_test)
    my_total.append(im_num_train+im_num_test)
dic = {'Label': names, 'Number(training)': my_train, 'Number(testing)': my_test, 'Total':

my_dataframe = pd.DataFrame(dic)
my_dataframe
```

	Label	Number(training)	Number(testing)	Total
0	boat	422	106	528
1	motorcycle	473	119	592
2	airplane	321	81	402
3	truck	524	132	656

No of images belonging to each class

```
print(f"Total training images: {my_dataframe['Number(training)'].sum()}")
print(f"Total testing images: {my_dataframe['Number(testing)'].sum()}")
```

```
Total training images: 2411
Total testing images: 607
```

Write a function to display a random image and its shape. Find out whether the shape of each image is the same or not. If not then make all images of the same shape.

```
import random
def random_image(path):
    labels = os.listdir(path)
    random_label = random.choice(labels)

    image_with_choson_label = os.listdir(f"./train/{random_label}")
    random_image = random.choice(image_with_choson_label)
    img_path = f"./train/{random_label}/{random_image}"
    print(random_label)
    return img_path
```

```
import matplotlib.pyplot as plt
labels = os.listdir("./train")

size = {}
for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        img = plt.imread(f"./train/{label}/{image}")
        if img.shape not in size:
            size[img.shape]=1
        else:
            size[img.shape]+=1

print(size)
```

```
{(128, 128): 2411}
```

Running the above code shows us there are 2 different file dimensions.

```
from PIL import Image
import cv2

for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        img = cv2.imread(f"./train/{label}/{image}")
        try:
            if img.shape == list(size)[1]:
                img = cv2.resize(img, dsize=(128, 128), interpolation=cv2.INTER_AREA)
                cv2.imwrite(f"./train/{label}/{image}", img)
        except:
            continue
```

```
labels = os.listdir("./train")

size = {}
for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        img = plt.imread(f"./train/{label}/{image}")
        if img.shape not in size:
            size[img.shape]=1
        else:
            size[img.shape]+=1

print(size)
my_train = list(size.values())[0]
print(my_train)
```

```
{(128, 128): 2411}
2411
```

We maked all the images dimension = 128x128x3

```
from PIL import Image
import cv2
import matplotlib.pyplot as plt


for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        try:
            img = cv2.imread(f"./train/{label}/{image}")
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            cv2.imwrite(f"./train/{label}/{image}", gray)
        except:
            continue
```

```
import PIL
```

```
import matplotlib.pyplot as plt

img_check = random_image("./train")
img = PIL.Image.open(img_check)
gray_img = img.convert("L")
plt.imshow(gray_img, cmap='gray')
```

```
motorbus
<matplotlib.image.AxesImage at 0x7f90ab174be0>
```

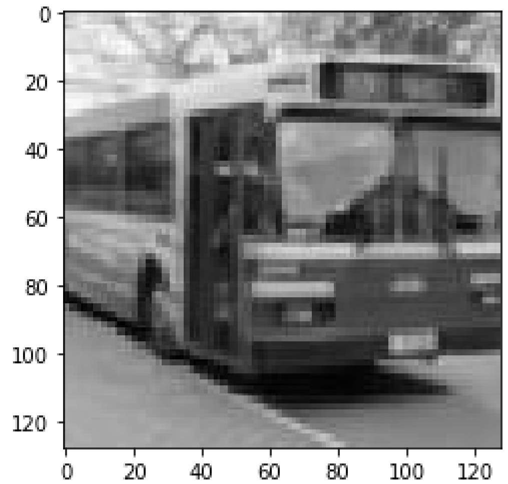


Image normalization is a typical process in image processing that changes the range of pixel intensity values. Its normal purpose is to convert an input image into a range of pixel values that are more familiar or normal to the senses, hence the term normalization.

Image data should be normalised when we want the model to be brightness invariant. There are chances some images are clicked in dim lighting conditions while some were clicked in bright illumination. Normalisation will help all image to weight equally irrespective of illumination. We should definately normalise our color channels.

Let us find the mean and standard deviation of pixel intensity.

```
labels = os.listdir("./train")

maximum_pixel_intensity = float("-inf")
minimum_pixel_intensity = float("inf")
mean_pixel_intensity = 0
std_pixel_intensity = 0

for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        img = plt.imread(f"./train/{label}/{image}",cv2.IMREAD_UNCHANGED)
        if minimum_pixel_intensity>img.min():
            minimum_pixel_intensity = img.min()
        if maximum_pixel_intensity < img.max():
            maximum_pixel_intensity = img.max()
        mean_pixel_intensity+=img.sum()
mean_pixel_intensity = mean_pixel_intensity/(128*128*my_train)
```

```
std_pixel_intensity = 0
sum_diff = 0

for label in labels:
    images = os.listdir(f"./train/{label}")
    for image in images:
        img = plt.imread(f"./train/{label}/{image}",cv2.IMREAD_UNCHANGED)
        temp = ((img.sum()/(128*128))-mean_pixel_intensity)**2
        sum_diff+=temp
std_pixel_intensity = (sum_diff/(my_train))**(1/2)

print("minimum_pixel_intensity: ", minimum_pixel_intensity)
print("maximum_pixel_intensity: ", maximum_pixel_intensity)
print("mean_pixel_intensity:", mean_pixel_intensity)
print("std_pixel_intensity: ",std_pixel_intensity)
```

```
minimum_pixel_intensity:  0
maximum_pixel_intensity:  255
mean_pixel_intensity: 137.64696161878499
std_pixel_intensity:  33.24571848596172
```

We can pre-process our images in three ways. One of them is pixel scaling. The three main types of pixel scaling techniques as follows:

Pixel Normalization: scale pixel values to the range 0-1.

Pixel Centering: scale pixel values to have a zero mean.

Pixel Standardization: scale pixel values to have a zero mean and unit variance.

Dividing all pixel values by 255 to bring them betweeen 0 and 1. Reducing the image dimensions to 28x28

```
from PIL import Image
import cv2
import matplotlib.pyplot as plt

def resize_and_normalise(img_path):
    img = cv2.imread(img_path,cv2.IMREAD_UNCHANGED)
    img = cv2.resize(img, dsize=(40, 40), interpolation = cv2.INTER_AREA)
    img_resized = img / 255
    return img_resized
```

```
import os
classes = os.listdir("./train")
print(classes)
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

```
my_dataframe
```

	Label	Number(training)	Number(testing)	Total
0	boat	422	106	528
1	motorcycle	473	119	592
2	airplane	321	81	402
3	truck	524	132	656
4	seaplane	224	56	280
5	bicycle	243	61	304
6	motorbus	204	52	256

As we can see after splitting into test and train the number of images in each class are different.The problem with such a datset is that models trained on it wll be biased towards the more occuring classes. There will no propoer representation of minority classes in both validation and training.

Balanced training sample become very important.

There are several ways to tackle this unbalanced data problem. 2 basic ones are:

Undersampling the majority classes Oversampling the minority classes Apart from this we can also take balanced subsets of the datset during training.

To ensure that every fold contains images from each class and no duplicates, We can use stratified k-fold cross-validation which is the same as just k-fold cross-validation, But Stratified k-fold cross-validation, it does stratified sampling instead of random sampling. Here we are working on the original dataset only without augmenting the dataset.

```
class_number = 0
num_classes = len(os.listdir("./train"))
count = 0

x_train = []
y_train = []


for classes in os.listdir("./train"):
    y_array = class_number
    all_images = os.listdir(f"./train/{classes}")
    tot = len(all_images)
    count = 0
    for images in all_images:
        try:
            img = resize_and_normalise(f"./train/{classes}/{images}")
            x_train.append(img)
            y_train.append(y_array)
            count+=1
        except:
```

```
            continue
    class_number+=1
    print(f"Class {classes} processessing done")

print("All classes have been processed")
```

```
Class boat processessing done
Class motorcycle processessing done
Class airplane processessing done
Class truck processessing done
Class seaplane processessing done
Class bicycle processessing done
Class motorbus processessing done
All classes have been processed
```

```
import numpy as np
x_train =  (np.array(x_train).reshape(np.array(x_train).shape[0],-1))
y_train = np.array(y_train)
print(x_train.shape,y_train.shape)
```

```
(2411, 1600) (2411,)
```

```
from sklearn.utils import shuffle
x_train, y_train = shuffle(x_train, y_train)
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(
      x_train, y_train, test_size = 0.30)
```

```
X_test.shape
```

```
(724, 1600)
```

Double-click (or enter) to edit

Double-click (or enter) to edit

Training a neural network with 4 layers

number of neurons = 1024,256,64,7 respectively

batch size=128 for gradient descent

activation function is sigmoid.

```
model1 = Sequential()
model1.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model1.add(Dense(256,activation = 'relu'))
model1.add(Dense(64,activation = 'relu'))
```

```
model1.add(Dense(7,activation='softmax'))
model1.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics = 'accu
history_model1 = model1.fit(X_train,Y_train, epochs = 80, batch_size = 128, validation_dat
best_score = max(history_model1.history['accuracy'])
print("The best accuracy is",best_score)
```

```
Epoch 1/80
14/14 - 5s - loss: 1.8784 - accuracy: 0.2833 - val_loss: 1.4411 - val_accuracy: 0
Epoch 2/80
14/14 - 0s - loss: 1.3137 - accuracy: 0.5157 - val_loss: 1.2624 - val_accuracy: 0
Epoch 3/80
14/14 - 0s - loss: 1.1572 - accuracy: 0.5726 - val_loss: 1.1012 - val_accuracy: 0
Epoch 4/80
14/14 - 0s - loss: 1.0287 - accuracy: 0.6574 - val_loss: 0.9593 - val_accuracy: 0
Epoch 5/80
14/14 - 0s - loss: 0.9386 - accuracy: 0.6752 - val_loss: 1.0864 - val_accuracy: 0
Epoch 6/80
14/14 - 0s - loss: 0.9374 - accuracy: 0.6740 - val_loss: 0.8359 - val_accuracy: 0
Epoch 7/80
14/14 - 0s - loss: 0.8105 - accuracy: 0.7196 - val_loss: 0.8082 - val_accuracy: 0
Epoch 8/80
14/14 - 0s - loss: 0.7447 - accuracy: 0.7475 - val_loss: 0.7693 - val_accuracy: 0
Epoch 9/80
14/14 - 0s - loss: 0.7214 - accuracy: 0.7504 - val_loss: 0.7635 - val_accuracy: 0
Epoch 10/80
14/14 - 0s - loss: 0.7534 - accuracy: 0.7398 - val_loss: 0.8389 - val_accuracy: 0
Epoch 11/80
14/14 - 0s - loss: 0.7104 - accuracy: 0.7570 - val_loss: 0.7336 - val_accuracy: 0
Epoch 12/80
14/14 - 0s - loss: 0.6246 - accuracy: 0.7878 - val_loss: 0.8996 - val_accuracy: 0
Epoch 13/80
14/14 - 0s - loss: 0.6400 - accuracy: 0.7825 - val_loss: 0.7095 - val_accuracy: 0
Epoch 14/80
14/14 - 0s - loss: 0.5999 - accuracy: 0.7943 - val_loss: 0.6999 - val_accuracy: 0
Epoch 15/80
14/14 - 0s - loss: 0.5926 - accuracy: 0.8068 - val_loss: 0.8798 - val_accuracy: 0
Epoch 16/80
14/14 - 0s - loss: 0.6021 - accuracy: 0.7943 - val_loss: 0.7158 - val_accuracy: 0
Epoch 17/80
14/14 - 0s - loss: 0.5435 - accuracy: 0.8168 - val_loss: 0.6873 - val_accuracy: 0
Epoch 18/80
14/14 - 0s - loss: 0.4923 - accuracy: 0.8322 - val_loss: 0.6951 - val_accuracy: 0
Epoch 19/80
14/14 - 0s - loss: 0.4820 - accuracy: 0.8405 - val_loss: 0.6840 - val_accuracy: 0
Epoch 20/80
14/14 - 0s - loss: 0.4683 - accuracy: 0.8394 - val_loss: 0.6357 - val_accuracy: 0
Epoch 21/80
14/14 - 0s - loss: 0.4281 - accuracy: 0.8637 - val_loss: 0.6311 - val_accuracy: 0
Epoch 22/80
14/14 - 0s - loss: 0.4430 - accuracy: 0.8483 - val_loss: 0.7554 - val_accuracy: 0
Epoch 23/80
14/14 - 0s - loss: 0.4475 - accuracy: 0.8471 - val_loss: 0.7278 - val_accuracy: 0
Epoch 24/80
14/14 - 0s - loss: 0.4325 - accuracy: 0.8595 - val_loss: 0.8694 - val_accuracy: 0
Epoch 25/80
14/14 - 0s - loss: 0.4773 - accuracy: 0.8328 - val_loss: 0.7197 - val_accuracy: 0
Epoch 26/80
14/14 - 0s - loss: 0.4012 - accuracy: 0.8631 - val_loss: 0.6072 - val_accuracy: 0
Epoch 27/80
```

```
14/14 - 0s - loss: 0.4074 - accuracy: 0.8583 - val_loss: 0.5794 - val_accuracy: 0
Epoch 28/80
14/14 - 0s - loss: 0.3437 - accuracy: 0.8886 - val_loss: 0.5976 - val_accuracy: 0
```

```
pred = tf.argmax(model1.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 7ms/step
Macro score: 0.8022271103574308
Recall Score 0.757670675247035
```

Macro score: 0.8364602156139574

PLAYING WITH NUMBER OF LAYERS AND NUMBER OF NEURONS PER LAYER

number of layers=8

neurons per layer=512,256,128,64,32,24,16,7 respectively

```
model2 = Sequential()
model2.add(Dense(512,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trair
model2.add(Dense(256,activation = 'relu'))
model2.add(Dense(128,activation = 'relu'))
model2.add(Dense(64,activation = 'relu'))
model2.add(Dense(32,activation = 'relu'))
model2.add(Dense(24,activation = 'relu'))
model2.add(Dense(16,activation = 'relu'))
model2.add(Dense(7,activation="softmax"))
model2.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics = "accu
history_model2 = model2.fit(X_train,Y_train, epochs = 80, batch_size = 128, validation_dat
best_score = max(history_model2.history['accuracy'])
print("The best accuracy is",best_score)
```

```
Epoch 52/80
14/14 - 0s - loss: 0.2454 - accuracy: 0.9176 - val_loss: 0.8095 - val_accuracy: 0
Epoch 53/80
14/14 - 0s - loss: 0.2484 - accuracy: 0.9140 - val_loss: 0.7490 - val_accuracy: 0
Epoch 54/80
14/14 - 0s - loss: 0.2305 - accuracy: 0.9235 - val_loss: 0.8539 - val_accuracy: 0
Epoch 55/80
14/14 - 0s - loss: 0.2065 - accuracy: 0.9306 - val_loss: 0.7429 - val_accuracy: 0
Epoch 56/80
14/14 - 0s - loss: 0.2202 - accuracy: 0.9194 - val_loss: 0.8415 - val_accuracy: 0
Epoch 57/80
14/14 - 0s - loss: 0.2414 - accuracy: 0.9206 - val_loss: 1.0322 - val_accuracy: 0
Epoch 58/80
```

```
14/14 - 0s - loss: 0.3058 - accuracy: 0.8903 - val_loss: 1.2002 - val_accuracy: 0
Epoch 59/80
14/14 - 0s - loss: 0.5989 - accuracy: 0.7943 - val_loss: 1.1186 - val_accuracy: 0
Epoch 60/80
14/14 - 0s - loss: 0.4471 - accuracy: 0.8429 - val_loss: 0.6252 - val_accuracy: 0
Epoch 61/80
14/14 - 0s - loss: 0.2702 - accuracy: 0.9117 - val_loss: 0.6679 - val_accuracy: 0
Epoch 62/80
14/14 - 0s - loss: 0.2291 - accuracy: 0.9200 - val_loss: 0.7115 - val_accuracy: 0
Epoch 63/80
14/14 - 0s - loss: 0.1937 - accuracy: 0.9330 - val_loss: 0.6767 - val_accuracy: 0
Epoch 64/80
14/14 - 0s - loss: 0.1659 - accuracy: 0.9443 - val_loss: 0.7488 - val_accuracy: 0
Epoch 65/80
14/14 - 0s - loss: 0.2130 - accuracy: 0.9229 - val_loss: 0.7176 - val_accuracy: 0
Epoch 66/80
14/14 - 0s - loss: 0.1778 - accuracy: 0.9372 - val_loss: 0.7198 - val_accuracy: 0
Epoch 67/80
14/14 - 0s - loss: 0.1704 - accuracy: 0.9372 - val_loss: 0.7383 - val_accuracy: 0
Epoch 68/80
14/14 - 0s - loss: 0.1621 - accuracy: 0.9443 - val_loss: 0.7388 - val_accuracy: 0
Epoch 69/80
14/14 - 0s - loss: 0.1866 - accuracy: 0.9336 - val_loss: 1.0057 - val_accuracy: 0
Epoch 70/80
14/14 - 0s - loss: 0.1959 - accuracy: 0.9259 - val_loss: 0.7766 - val_accuracy: 0
Epoch 71/80
14/14 - 0s - loss: 0.1812 - accuracy: 0.9318 - val_loss: 0.7811 - val_accuracy: 0
Epoch 72/80
14/14 - 0s - loss: 0.1346 - accuracy: 0.9579 - val_loss: 0.7882 - val_accuracy: 0
Epoch 73/80
14/14 - 0s - loss: 0.1063 - accuracy: 0.9668 - val_loss: 0.8213 - val_accuracy: 0
Epoch 74/80
14/14 - 0s - loss: 0.1373 - accuracy: 0.9585 - val_loss: 0.8229 - val_accuracy: 0
Epoch 75/80
14/14 - 0s - loss: 0.1437 - accuracy: 0.9502 - val_loss: 1.0306 - val_accuracy: 0
Epoch 76/80
14/14 - 0s - loss: 0.1876 - accuracy: 0.9372 - val_loss: 0.7013 - val_accuracy: 0
Epoch 77/80
14/14 - 0s - loss: 0.1221 - accuracy: 0.9567 - val_loss: 0.8020 - val_accuracy: 0
Epoch 78/80
14/14 - 0s - loss: 0.1154 - accuracy: 0.9567 - val_loss: 0.8365 - val_accuracy: 0
Epoch 79/80
14/14 - 0s - loss: 0.0954 - accuracy: 0.9668 - val_loss: 0.9007 - val_accuracy: 0
Epoch 80/80
```

```
pred = tf.argmax(model2.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 4ms/step
Macro score: 0.7815235579726233
Recall Score 0.7711273025523229
```

Macro score: 0.8155478541691441

number of layers=3

neurons per layer=2048,1024,7

```
model3 = Sequential()
model3.add(Dense(2048,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model3.add(Dense(1024,activation = 'relu'))
model3.add(Dense(7,activation="softmax"))
model3.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics = "accu
history_model3 = model3.fit(X_train,Y_train, epochs = 70, batch_size = 128, validation_dat
best_score = max(history_model3.history['accuracy'])
print("The best accuracy is",best_score)
```

```
Epoch 1/70
14/14 - 1s - loss: 4.9443 - accuracy: 0.2733 - val_loss: 1.4590 - val_accuracy: 0
Epoch 2/70
14/14 - 1s - loss: 1.2435 - accuracy: 0.5471 - val_loss: 1.1247 - val_accuracy: 0
Epoch 3/70
14/14 - 1s - loss: 1.0178 - accuracy: 0.6313 - val_loss: 1.0325 - val_accuracy: 0
Epoch 4/70
14/14 - 1s - loss: 0.9194 - accuracy: 0.6663 - val_loss: 0.9200 - val_accuracy: 0
Epoch 5/70
14/14 - 1s - loss: 0.8277 - accuracy: 0.7143 - val_loss: 0.8911 - val_accuracy: 0
Epoch 6/70
14/14 - 1s - loss: 0.8133 - accuracy: 0.7155 - val_loss: 0.8394 - val_accuracy: 0
Epoch 7/70
14/14 - 1s - loss: 0.7544 - accuracy: 0.7273 - val_loss: 0.7456 - val_accuracy: 0
Epoch 8/70
14/14 - 1s - loss: 0.6762 - accuracy: 0.7742 - val_loss: 0.6908 - val_accuracy: 0
Epoch 9/70
14/14 - 1s - loss: 0.6217 - accuracy: 0.7848 - val_loss: 0.7218 - val_accuracy: 0
Epoch 10/70
14/14 - 1s - loss: 0.5844 - accuracy: 0.8032 - val_loss: 0.6654 - val_accuracy: 0
Epoch 11/70
14/14 - 1s - loss: 0.5843 - accuracy: 0.8056 - val_loss: 0.6984 - val_accuracy: 0
Epoch 12/70
14/14 - 1s - loss: 0.5670 - accuracy: 0.8062 - val_loss: 0.6546 - val_accuracy: 0
Epoch 13/70
14/14 - 1s - loss: 0.5293 - accuracy: 0.8186 - val_loss: 0.6411 - val_accuracy: 0
Epoch 14/70
14/14 - 1s - loss: 0.4921 - accuracy: 0.8358 - val_loss: 0.6154 - val_accuracy: 0
Epoch 15/70
14/14 - 1s - loss: 0.4339 - accuracy: 0.8595 - val_loss: 0.6314 - val_accuracy: 0
Epoch 16/70
14/14 - 1s - loss: 0.4272 - accuracy: 0.8660 - val_loss: 0.6388 - val_accuracy: 0
Epoch 17/70
14/14 - 1s - loss: 0.4267 - accuracy: 0.8601 - val_loss: 0.5967 - val_accuracy: 0
Epoch 18/70
14/14 - 1s - loss: 0.3530 - accuracy: 0.8903 - val_loss: 0.7661 - val_accuracy: 0
Epoch 19/70
14/14 - 1s - loss: 0.4410 - accuracy: 0.8388 - val_loss: 0.8245 - val_accuracy: 0
Epoch 20/70
14/14 - 1s - loss: 0.4154 - accuracy: 0.8518 - val_loss: 0.6597 - val_accuracy: 0
Epoch 21/70
14/14 - 1s - loss: 0.3497 - accuracy: 0.8797 - val_loss: 0.5749 - val_accuracy: 0
Epoch 22/70
```

```
14/14 - 1s - loss: 0.3474 - accuracy: 0.8814 - val_loss: 0.5944 - val_accuracy: 0
Epoch 23/70
14/14 - 1s - loss: 0.3090 - accuracy: 0.9022 - val_loss: 0.5502 - val_accuracy: 0
Epoch 24/70
14/14 - 1s - loss: 0.2568 - accuracy: 0.9200 - val_loss: 0.6391 - val_accuracy: 0
Epoch 25/70
14/14 - 1s - loss: 0.2459 - accuracy: 0.9229 - val_loss: 0.7213 - val_accuracy: 0
Epoch 26/70
14/14 - 1s - loss: 0.3007 - accuracy: 0.8892 - val_loss: 0.8086 - val_accuracy: 0
Epoch 27/70
14/14 - 1s - loss: 0.2814 - accuracy: 0.9087 - val_loss: 0.6831 - val_accuracy: 0
Epoch 28/70
14/14 - 1s - loss: 0.3137 - accuracy: 0.8986 - val_loss: 0.6554 - val_accuracy: 0
Epoch 29/70
```

```
pred = tf.argmax(model3.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 6ms/step
Macro score: 0.8385117760007484
Recall Score 0.8245976209524837
```

Macro score: 0.8389959148430156

Playing with Activation functions

Activation function: sigmoid

```
model4 = Sequential()
model4.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model4.add(Dense(256,activation = 'relu'))
model4.add(Dense(64,activation = 'relu'))
model4.add(Dense(7,activation="sigmoid"))
model4.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics = 'accu
history_model4 = model4.fit(X_train,Y_train, epochs = 80, batch_size = 128, validation_dat
best_score = max(history_model4.history['accuracy'])
print("The best accuracy is",best_score)
```

```
Epoch 1/80
14/14 - 1s - loss: 2.0629 - accuracy: 0.3065 - val_loss: 1.4050 - val_accuracy: 0
Epoch 2/80
14/14 - 0s - loss: 1.3521 - accuracy: 0.4956 - val_loss: 1.3407 - val_accuracy: 0
Epoch 3/80
14/14 - 0s - loss: 1.2340 - accuracy: 0.5376 - val_loss: 1.1006 - val_accuracy: 0
Epoch 4/80
14/14 - 0s - loss: 1.0638 - accuracy: 0.6129 - val_loss: 1.0017 - val_accuracy: 0
Epoch 5/80
14/14 - 0s - loss: 0.9726 - accuracy: 0.6520 - val_loss: 0.9510 - val_accuracy: 0
Epoch 6/80
```

```
14/14 - 0s - loss: 0.9071 - accuracy: 0.6912 - val_loss: 0.9377 - val_accuracy: 0
Epoch 7/80
14/14 - 0s - loss: 0.8721 - accuracy: 0.6971 - val_loss: 0.8465 - val_accuracy: 0
Epoch 8/80
14/14 - 0s - loss: 0.8203 - accuracy: 0.7178 - val_loss: 0.8851 - val_accuracy: 0
Epoch 9/80
14/14 - 0s - loss: 0.7981 - accuracy: 0.7315 - val_loss: 0.7933 - val_accuracy: 0
Epoch 10/80
14/14 - 0s - loss: 0.7600 - accuracy: 0.7362 - val_loss: 0.8233 - val_accuracy: 0
Epoch 11/80
14/14 - 0s - loss: 0.6998 - accuracy: 0.7593 - val_loss: 0.7735 - val_accuracy: 0
Epoch 12/80
14/14 - 0s - loss: 0.6910 - accuracy: 0.7647 - val_loss: 0.7727 - val_accuracy: 0
Epoch 13/80
14/14 - 0s - loss: 0.6919 - accuracy: 0.7635 - val_loss: 0.7813 - val_accuracy: 0
Epoch 14/80
14/14 - 0s - loss: 0.6507 - accuracy: 0.7724 - val_loss: 0.7468 - val_accuracy: 0
Epoch 15/80
14/14 - 0s - loss: 0.6208 - accuracy: 0.7943 - val_loss: 0.7294 - val_accuracy: 0
Epoch 16/80
14/14 - 0s - loss: 0.6053 - accuracy: 0.8026 - val_loss: 0.6713 - val_accuracy: 0
Epoch 17/80
14/14 - 0s - loss: 0.5362 - accuracy: 0.8376 - val_loss: 0.6555 - val_accuracy: 0
Epoch 18/80
14/14 - 0s - loss: 0.5159 - accuracy: 0.8299 - val_loss: 0.7002 - val_accuracy: 0
Epoch 19/80
14/14 - 0s - loss: 0.5200 - accuracy: 0.8133 - val_loss: 0.7420 - val_accuracy: 0
Epoch 20/80
14/14 - 0s - loss: 0.4889 - accuracy: 0.8364 - val_loss: 0.6434 - val_accuracy: 0
Epoch 21/80
14/14 - 0s - loss: 0.4629 - accuracy: 0.8483 - val_loss: 0.7315 - val_accuracy: 0
Epoch 22/80
14/14 - 0s - loss: 0.5001 - accuracy: 0.8317 - val_loss: 0.6570 - val_accuracy: 0
Epoch 23/80
14/14 - 0s - loss: 0.4405 - accuracy: 0.8506 - val_loss: 0.6024 - val_accuracy: 0
Epoch 24/80
14/14 - 0s - loss: 0.3997 - accuracy: 0.8643 - val_loss: 0.6331 - val_accuracy: 0
Epoch 25/80
14/14 - 0s - loss: 0.4265 - accuracy: 0.8488 - val_loss: 0.6113 - val_accuracy: 0
Epoch 26/80
14/14 - 0s - loss: 0.4357 - accuracy: 0.8453 - val_loss: 0.6319 - val_accuracy: 0
Epoch 27/80
14/14 - 0s - loss: 0.3846 - accuracy: 0.8702 - val_loss: 0.6300 - val_accuracy: 0
Epoch 28/80
14/14 - 0s - loss: 0.3897 - accuracy: 0.8702 - val_loss: 0.7817 - val_accuracy: 0
```

```
pred = tf.argmax(model4.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 4ms/step
Macro score: 0.7992918455075158
Recall Score 0.7922083195982204
```

Macro score: 0.7820051751980104

[ ] ↳ 3 cells hidden

Macro score: 0.018178225647105313

[ ] ↳ 3 cells hidden

Macro score: 0.023315550286504644

Applying early stopping with patience=3 and dropout=0.3

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
model7 = Sequential()
model7.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model7.add(Dropout(0.3))
model7.add(Dense(256,activation = 'relu'))
model7.add(Dropout(0.3))
model7.add(Dense(64,activation = 'relu'))
model7.add(Dense(7,activation="sigmoid"))
model7.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics ="accur
history_model7 = model7.fit(X_train,Y_train, epochs = 100, batch_size = 128, validation_da
```

```
Epoch 1/100
14/14 - 2s - loss: 1.9927 - accuracy: 0.2833 - val_loss: 1.4291 - val_accuracy: 0
Epoch 2/100
14/14 - 0s - loss: 1.4904 - accuracy: 0.4292 - val_loss: 1.2956 - val_accuracy: 0
Epoch 3/100
14/14 - 0s - loss: 1.3024 - accuracy: 0.5181 - val_loss: 1.1468 - val_accuracy: 0
Epoch 4/100
14/14 - 0s - loss: 1.2094 - accuracy: 0.5519 - val_loss: 1.1852 - val_accuracy: 0
Epoch 5/100
14/14 - 0s - loss: 1.1409 - accuracy: 0.5738 - val_loss: 0.9965 - val_accuracy: 0
Epoch 6/100
14/14 - 0s - loss: 1.0409 - accuracy: 0.6224 - val_loss: 0.9528 - val_accuracy: 0
Epoch 7/100
14/14 - 0s - loss: 0.9771 - accuracy: 0.6420 - val_loss: 0.9541 - val_accuracy: 0
Epoch 8/100
14/14 - 0s - loss: 0.9602 - accuracy: 0.6633 - val_loss: 0.9724 - val_accuracy: 0
Epoch 9/100
14/14 - 0s - loss: 0.9509 - accuracy: 0.6603 - val_loss: 0.8595 - val_accuracy: 0
Epoch 10/100
14/14 - 0s - loss: 0.8765 - accuracy: 0.6882 - val_loss: 0.8121 - val_accuracy: 0
Epoch 11/100
14/14 - 0s - loss: 0.8533 - accuracy: 0.7001 - val_loss: 0.7999 - val_accuracy: 0
Epoch 12/100
14/14 - 0s - loss: 0.8124 - accuracy: 0.7107 - val_loss: 0.8366 - val_accuracy: 0
Epoch 13/100
14/14 - 0s - loss: 0.8434 - accuracy: 0.6888 - val_loss: 0.8087 - val_accuracy: 0
Epoch 14/100
14/14 - 0s - loss: 0.7653 - accuracy: 0.7279 - val_loss: 0.7208 - val_accuracy: 0
Epoch 15/100
14/14 - 0s - loss: 0.7169 - accuracy: 0.7445 - val_loss: 0.7793 - val_accuracy: 0
Epoch 16/100
```

```
14/14 - 0s - loss: 0.7123 - accuracy: 0.7487 - val_loss: 0.7155 - val_accuracy: 0
Epoch 17/100
14/14 - 0s - loss: 0.7266 - accuracy: 0.7439 - val_loss: 0.7018 - val_accuracy: 0
Epoch 18/100
14/14 - 0s - loss: 0.6684 - accuracy: 0.7670 - val_loss: 0.8175 - val_accuracy: 0
Epoch 19/100
14/14 - 0s - loss: 0.7167 - accuracy: 0.7552 - val_loss: 0.7836 - val_accuracy: 0
Epoch 20/100
14/14 - 0s - loss: 0.6279 - accuracy: 0.7884 - val_loss: 0.6447 - val_accuracy: 0
Epoch 21/100
14/14 - 0s - loss: 0.6676 - accuracy: 0.7617 - val_loss: 0.6812 - val_accuracy: 0
Epoch 22/100
14/14 - 0s - loss: 0.6239 - accuracy: 0.7736 - val_loss: 0.6386 - val_accuracy: 0
Epoch 23/100
14/14 - 0s - loss: 0.6085 - accuracy: 0.7902 - val_loss: 0.6496 - val_accuracy: 0
Epoch 24/100
14/14 - 0s - loss: 0.6200 - accuracy: 0.7795 - val_loss: 0.6175 - val_accuracy: 0
Epoch 25/100
14/14 - 0s - loss: 0.5620 - accuracy: 0.8044 - val_loss: 0.6244 - val_accuracy: 0
Epoch 26/100
14/14 - 0s - loss: 0.5651 - accuracy: 0.7919 - val_loss: 0.6801 - val_accuracy: 0
Epoch 27/100
14/14 - 0s - loss: 0.5406 - accuracy: 0.8091 - val_loss: 0.5808 - val_accuracy: 0
Epoch 28/100
14/14 - 0s - loss: 0.5929 - accuracy: 0.7848 - val_loss: 0.6017 - val_accuracy: 0
```

```
pred = tf.argmax(model7.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 4ms/step
Macro score: 0.8244193829175657
Recall Score 0.7949519839567982
```

Macro score: 0.8256826227377789

Playing with optimizers

optimizer: Adagrad

```
model8 = Sequential()
model8.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model8.add(Dense(256,activation = 'relu'))
model8.add(Dense(64,activation = 'relu'))
model8.add(Dense(7,activation="sigmoid"))
model8.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'Adagrad',metrics = "a
history_model8 = model8.fit(X_train,Y_train, epochs = 100, batch_size = 128, validation_da
```

```
Epoch 1/100
14/14 - 1s - loss: 1.8499 - accuracy: 0.2780 - val_loss: 1.7447 - val_accuracy: 0
Epoch 2/100
```

```
14/14 - 0s - loss: 1.6655 - accuracy: 0.4078 - val_loss: 1.6316 - val_accuracy: 0
Epoch 3/100
14/14 - 0s - loss: 1.5577 - accuracy: 0.4849 - val_loss: 1.5438 - val_accuracy: 0
Epoch 4/100
14/14 - 0s - loss: 1.4718 - accuracy: 0.5193 - val_loss: 1.4703 - val_accuracy: 0
Epoch 5/100
14/14 - 0s - loss: 1.4088 - accuracy: 0.5519 - val_loss: 1.4280 - val_accuracy: 0
Epoch 6/100
14/14 - 0s - loss: 1.3556 - accuracy: 0.5738 - val_loss: 1.3611 - val_accuracy: 0
Epoch 7/100
14/14 - 0s - loss: 1.3057 - accuracy: 0.5993 - val_loss: 1.3291 - val_accuracy: 0
Epoch 8/100
14/14 - 0s - loss: 1.2667 - accuracy: 0.6218 - val_loss: 1.2731 - val_accuracy: 0
Epoch 9/100
14/14 - 0s - loss: 1.2288 - accuracy: 0.6230 - val_loss: 1.2705 - val_accuracy: 0
Epoch 10/100
14/14 - 0s - loss: 1.1977 - accuracy: 0.6520 - val_loss: 1.2371 - val_accuracy: 0
Epoch 11/100
14/14 - 0s - loss: 1.1693 - accuracy: 0.6520 - val_loss: 1.1937 - val_accuracy: 0
Epoch 12/100
14/14 - 0s - loss: 1.1362 - accuracy: 0.6615 - val_loss: 1.1575 - val_accuracy: 0
Epoch 13/100
14/14 - 0s - loss: 1.1118 - accuracy: 0.6781 - val_loss: 1.1528 - val_accuracy: 0
Epoch 14/100
14/14 - 0s - loss: 1.0877 - accuracy: 0.6805 - val_loss: 1.1250 - val_accuracy: 0
Epoch 15/100
14/14 - 0s - loss: 1.0683 - accuracy: 0.6888 - val_loss: 1.0923 - val_accuracy: 0
Epoch 16/100
14/14 - 0s - loss: 1.0461 - accuracy: 0.6894 - val_loss: 1.0767 - val_accuracy: 0
Epoch 17/100
14/14 - 0s - loss: 1.0293 - accuracy: 0.6995 - val_loss: 1.0640 - val_accuracy: 0
Epoch 18/100
14/14 - 0s - loss: 1.0139 - accuracy: 0.7048 - val_loss: 1.0717 - val_accuracy: 0
Epoch 19/100
14/14 - 0s - loss: 1.0020 - accuracy: 0.7101 - val_loss: 1.0348 - val_accuracy: 0
Epoch 20/100
14/14 - 0s - loss: 0.9841 - accuracy: 0.7119 - val_loss: 1.0228 - val_accuracy: 0
Epoch 21/100
14/14 - 0s - loss: 0.9739 - accuracy: 0.7119 - val_loss: 1.0068 - val_accuracy: 0
Epoch 22/100
14/14 - 0s - loss: 0.9585 - accuracy: 0.7214 - val_loss: 0.9939 - val_accuracy: 0
Epoch 23/100
14/14 - 0s - loss: 0.9471 - accuracy: 0.7143 - val_loss: 0.9875 - val_accuracy: 0
Epoch 24/100
14/14 - 0s - loss: 0.9354 - accuracy: 0.7220 - val_loss: 0.9711 - val_accuracy: 0
Epoch 25/100
14/14 - 0s - loss: 0.9248 - accuracy: 0.7279 - val_loss: 0.9601 - val_accuracy: 0
Epoch 26/100
14/14 - 0s - loss: 0.9130 - accuracy: 0.7309 - val_loss: 0.9510 - val_accuracy: 0
Epoch 27/100
14/14 - 0s - loss: 0.9060 - accuracy: 0.7261 - val_loss: 0.9584 - val_accuracy: 0
Epoch 28/100
14/14 - 0s - loss: 0.8966 - accuracy: 0.7255 - val_loss: 0.9591 - val_accuracy: 0
Epoch 29/100
```

```
pred = tf.argmax(model8.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 3ms/step
Macro score: 0.7850351916089732
Recall Score 0.7271484228867375
```

Macro score: 0.8037465090556944

Optimizer: Ftrl

```
model9 = Sequential()
model9.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_trai
model9.add(Dense(256,activation = 'relu'))
model9.add(Dense(64,activation = 'relu'))
model9.add(Dense(7,activation="sigmoid"))
model9.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'Ftrl',metrics = "accu
history_model9 = model9.fit(X_train,Y_train, epochs = 100, batch_size = 128, validation_da
```

```
Epoch 1/100
14/14 - 1s - loss: 1.9733 - accuracy: 0.2110 - val_loss: 1.9453 - val_accuracy: 0
Epoch 2/100
14/14 - 0s - loss: 1.9451 - accuracy: 0.2170 - val_loss: 1.9447 - val_accuracy: 0
Epoch 3/100
14/14 - 0s - loss: 1.9446 - accuracy: 0.2170 - val_loss: 1.9442 - val_accuracy: 0
Epoch 4/100
14/14 - 0s - loss: 1.9442 - accuracy: 0.2170 - val_loss: 1.9438 - val_accuracy: 0
Epoch 5/100
14/14 - 0s - loss: 1.9438 - accuracy: 0.2170 - val_loss: 1.9434 - val_accuracy: 0
Epoch 6/100
14/14 - 0s - loss: 1.9434 - accuracy: 0.2170 - val_loss: 1.9431 - val_accuracy: 0
Epoch 7/100
14/14 - 0s - loss: 1.9431 - accuracy: 0.2170 - val_loss: 1.9428 - val_accuracy: 0
Epoch 8/100
14/14 - 0s - loss: 1.9428 - accuracy: 0.2170 - val_loss: 1.9424 - val_accuracy: 0
Epoch 9/100
14/14 - 0s - loss: 1.9425 - accuracy: 0.2170 - val_loss: 1.9421 - val_accuracy: 0
Epoch 10/100
14/14 - 0s - loss: 1.9422 - accuracy: 0.2170 - val_loss: 1.9419 - val_accuracy: 0
Epoch 11/100
14/14 - 0s - loss: 1.9420 - accuracy: 0.2170 - val_loss: 1.9416 - val_accuracy: 0
Epoch 12/100
14/14 - 0s - loss: 1.9417 - accuracy: 0.2170 - val_loss: 1.9414 - val_accuracy: 0
Epoch 13/100
14/14 - 0s - loss: 1.9415 - accuracy: 0.2170 - val_loss: 1.9411 - val_accuracy: 0
Epoch 14/100
14/14 - 0s - loss: 1.9413 - accuracy: 0.2170 - val_loss: 1.9409 - val_accuracy: 0
Epoch 15/100
14/14 - 0s - loss: 1.9411 - accuracy: 0.2170 - val_loss: 1.9407 - val_accuracy: 0
Epoch 16/100
14/14 - 0s - loss: 1.9409 - accuracy: 0.2170 - val_loss: 1.9404 - val_accuracy: 0
Epoch 17/100
14/14 - 0s - loss: 1.9407 - accuracy: 0.2170 - val_loss: 1.9402 - val_accuracy: 0
Epoch 18/100
14/14 - 0s - loss: 1.9405 - accuracy: 0.2170 - val_loss: 1.9400 - val_accuracy: 0
Epoch 19/100
14/14 - 0s - loss: 1.9403 - accuracy: 0.2170 - val_loss: 1.9398 - val_accuracy: 0
```

```
Epoch 20/100
14/14 - 0s - loss: 1.9401 - accuracy: 0.2170 - val_loss: 1.9396 - val_accuracy: 0
Epoch 21/100
14/14 - 0s - loss: 1.9399 - accuracy: 0.2170 - val_loss: 1.9394 - val_accuracy: 0
Epoch 22/100
14/14 - 0s - loss: 1.9397 - accuracy: 0.2170 - val_loss: 1.9392 - val_accuracy: 0
Epoch 23/100
14/14 - 0s - loss: 1.9395 - accuracy: 0.2170 - val_loss: 1.9390 - val_accuracy: 0
Epoch 24/100
14/14 - 0s - loss: 1.9394 - accuracy: 0.2170 - val_loss: 1.9388 - val_accuracy: 0
Epoch 25/100
14/14 - 0s - loss: 1.9392 - accuracy: 0.2170 - val_loss: 1.9387 - val_accuracy: 0
Epoch 26/100
14/14 - 0s - loss: 1.9390 - accuracy: 0.2170 - val_loss: 1.9385 - val_accuracy: 0
Epoch 27/100
14/14 - 0s - loss: 1.9389 - accuracy: 0.2170 - val_loss: 1.9383 - val_accuracy: 0
Epoch 28/100
14/14 - 0s - loss: 1.9387 - accuracy: 0.2170 - val_loss: 1.9381 - val_accuracy: 0
```

```
pred = tf.argmax(model9.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 3ms/step
Macro score: 0.031176006314127862
Recall Score 0.14285714285714285
```

Macro score: 0.03260225251926497

Optimizer: SGD

```
model10 = Sequential()
model10.add(Dense(1024,activation = 'relu',kernel_initializer = 'normal',input_dim = X_tra
model10.add(Dense(256,activation = 'relu'))
model10.add(Dense(64,activation = 'relu'))
model10.add(Dense(7,activation="sigmoid"))
model10.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'SGD',metrics = "accu
history_model10 = model10.fit(X_train,Y_train, epochs = 100, batch_size = 128, validation_
```

```
Epoch 1/100
14/14 - 1s - loss: 1.7497 - accuracy: 0.3349 - val_loss: 1.5497 - val_accuracy: 0
Epoch 2/100
14/14 - 0s - loss: 1.4384 - accuracy: 0.5080 - val_loss: 1.4209 - val_accuracy: 0
Epoch 3/100
14/14 - 0s - loss: 1.3026 - accuracy: 0.5566 - val_loss: 1.5075 - val_accuracy: 0
Epoch 4/100
14/14 - 0s - loss: 1.2507 - accuracy: 0.5667 - val_loss: 1.1949 - val_accuracy: 0
Epoch 5/100
14/14 - 0s - loss: 1.1274 - accuracy: 0.6378 - val_loss: 1.1209 - val_accuracy: 0
Epoch 6/100
14/14 - 0s - loss: 1.0866 - accuracy: 0.6360 - val_loss: 1.0575 - val_accuracy: 0
Epoch 7/100
14/14 - 0s - loss: 1.0565 - accuracy: 0.6408 - val_loss: 1.3239 - val_accuracy: 0
```

```
Epoch 8/100
14/14 - 0s - loss: 1.0202 - accuracy: 0.6408 - val_loss: 1.0431 - val_accuracy: 0
Epoch 9/100
14/14 - 0s - loss: 0.9678 - accuracy: 0.6781 - val_loss: 0.9859 - val_accuracy: 0
Epoch 10/100
14/14 - 0s - loss: 0.9342 - accuracy: 0.6882 - val_loss: 1.0434 - val_accuracy: 0
Epoch 11/100
14/14 - 0s - loss: 0.9378 - accuracy: 0.6763 - val_loss: 1.0117 - val_accuracy: 0
Epoch 12/100
14/14 - 0s - loss: 0.8695 - accuracy: 0.7107 - val_loss: 0.9507 - val_accuracy: 0
Epoch 13/100
14/14 - 0s - loss: 0.8556 - accuracy: 0.7208 - val_loss: 1.0112 - val_accuracy: 0
Epoch 14/100
14/14 - 0s - loss: 0.8761 - accuracy: 0.7036 - val_loss: 1.2446 - val_accuracy: 0
Epoch 15/100
14/14 - 0s - loss: 0.8606 - accuracy: 0.7072 - val_loss: 0.8632 - val_accuracy: 0
Epoch 16/100
14/14 - 0s - loss: 0.8017 - accuracy: 0.7368 - val_loss: 0.8747 - val_accuracy: 0
Epoch 17/100
14/14 - 0s - loss: 0.7646 - accuracy: 0.7522 - val_loss: 0.8978 - val_accuracy: 0
Epoch 18/100
14/14 - 0s - loss: 0.8048 - accuracy: 0.7261 - val_loss: 0.8937 - val_accuracy: 0
Epoch 19/100
14/14 - 0s - loss: 0.7738 - accuracy: 0.7386 - val_loss: 0.8293 - val_accuracy: 0
Epoch 20/100
14/14 - 0s - loss: 0.7393 - accuracy: 0.7558 - val_loss: 0.9015 - val_accuracy: 0
Epoch 21/100
14/14 - 0s - loss: 0.7336 - accuracy: 0.7582 - val_loss: 0.8962 - val_accuracy: 0
Epoch 22/100
14/14 - 0s - loss: 0.7913 - accuracy: 0.7309 - val_loss: 0.7899 - val_accuracy: 0
Epoch 23/100
14/14 - 0s - loss: 0.7124 - accuracy: 0.7593 - val_loss: 0.8349 - val_accuracy: 0
Epoch 24/100
14/14 - 0s - loss: 0.6889 - accuracy: 0.7700 - val_loss: 0.9553 - val_accuracy: 0
Epoch 25/100
14/14 - 0s - loss: 0.7190 - accuracy: 0.7516 - val_loss: 0.7971 - val_accuracy: 0
Epoch 26/100
14/14 - 0s - loss: 0.6700 - accuracy: 0.7854 - val_loss: 0.9041 - val_accuracy: 0
Epoch 27/100
14/14 - 0s - loss: 0.6914 - accuracy: 0.7682 - val_loss: 0.8521 - val_accuracy: 0
Epoch 28/100
14/14 - 0s - loss: 0.6647 - accuracy: 0.7771 - val_loss: 0.9904 - val_accuracy: 0
```

```
pred = tf.argmax(model10.predict(X_test),axis=1)
print("Macro score:",precision_score(Y_test,pred,average='macro'))
print("Recall Score",recall_score(Y_test,pred,average='macro'))
```

```
23/23 [==============================] - 0s 4ms/step
Macro score: 0.7823659400490133
Recall Score 0.7302215831716004
```

Macro score: 0.8159472885292376

Conclusion:

The best Macro Score is : 0.8389959148430156, when

number of layers=3 and neurons per layer=2048,1024,7 respectively.

Checking the accuracy using our best model

```
best_model = Sequential()
best_model.add(Dense(2048,activation = 'relu',kernel_initializer = 'normal',input_dim = X_
best_model.add(Dense(1024,activation = 'relu'))
best_model.add(Dense(7,activation="softmax"))
best_model.compile(loss = 'sparse_categorical_crossentropy',optimizer = 'adam',metrics = '
history_best_model = best_model.fit(X_train,Y_train, epochs = 100, batch_size = 128, valic
best_score = max(history_best_model.history['accuracy'])
print("The best accuracy is",best_score)
```

```
Epoch 1/100
14/14 - 1s - loss: 5.0140 - accuracy: 0.2810 - val_loss: 1.7024 - val_accuracy: 0
Epoch 2/100
14/14 - 1s - loss: 1.4811 - accuracy: 0.4552 - val_loss: 1.2682 - val_accuracy: 0
Epoch 3/100
14/14 - 1s - loss: 1.1287 - accuracy: 0.5999 - val_loss: 1.1631 - val_accuracy: 0
Epoch 4/100
14/14 - 1s - loss: 1.0191 - accuracy: 0.6331 - val_loss: 0.9243 - val_accuracy: 0
Epoch 5/100
14/14 - 1s - loss: 0.8698 - accuracy: 0.7024 - val_loss: 0.8370 - val_accuracy: 0
Epoch 6/100
14/14 - 1s - loss: 0.7789 - accuracy: 0.7475 - val_loss: 0.8626 - val_accuracy: 0
Epoch 7/100
14/14 - 1s - loss: 0.7244 - accuracy: 0.7445 - val_loss: 0.7763 - val_accuracy: 0
Epoch 8/100
14/14 - 1s - loss: 0.6456 - accuracy: 0.7896 - val_loss: 0.7270 - val_accuracy: 0
Epoch 9/100
14/14 - 1s - loss: 0.6121 - accuracy: 0.7996 - val_loss: 0.8419 - val_accuracy: 0
Epoch 10/100
14/14 - 1s - loss: 0.5921 - accuracy: 0.8008 - val_loss: 0.7485 - val_accuracy: 0
Epoch 11/100
14/14 - 1s - loss: 0.5604 - accuracy: 0.8151 - val_loss: 0.6590 - val_accuracy: 0
Epoch 12/100
14/14 - 1s - loss: 0.5249 - accuracy: 0.8228 - val_loss: 0.8527 - val_accuracy: 0
Epoch 13/100
14/14 - 1s - loss: 0.6237 - accuracy: 0.7771 - val_loss: 0.6569 - val_accuracy: 0
Epoch 14/100
14/14 - 1s - loss: 0.5083 - accuracy: 0.8322 - val_loss: 0.6252 - val_accuracy: 0
Epoch 15/100
14/14 - 1s - loss: 0.4404 - accuracy: 0.8613 - val_loss: 0.6699 - val_accuracy: 0
Epoch 16/100
14/14 - 1s - loss: 0.4217 - accuracy: 0.8542 - val_loss: 0.6073 - val_accuracy: 0
```

```
Epoch 17/100
14/14 - 1s - loss: 0.4164 - accuracy: 0.8684 - val_loss: 0.6407 - val_accuracy: 0
Epoch 18/100
14/14 - 1s - loss: 0.4121 - accuracy: 0.8648 - val_loss: 0.6616 - val_accuracy: 0
Epoch 19/100
14/14 - 1s - loss: 0.4102 - accuracy: 0.8554 - val_loss: 0.5776 - val_accuracy: 0
Epoch 20/100
14/14 - 1s - loss: 0.3737 - accuracy: 0.8826 - val_loss: 0.7141 - val_accuracy: 0
Epoch 21/100
14/14 - 1s - loss: 0.3972 - accuracy: 0.8690 - val_loss: 0.5560 - val_accuracy: 0
Epoch 22/100
14/14 - 1s - loss: 0.3011 - accuracy: 0.9022 - val_loss: 0.5951 - val_accuracy: 0
Epoch 23/100
14/14 - 1s - loss: 0.3113 - accuracy: 0.8963 - val_loss: 0.7799 - val_accuracy: 0
Epoch 24/100
14/14 - 1s - loss: 0.3605 - accuracy: 0.8761 - val_loss: 0.6230 - val_accuracy: 0
Epoch 25/100
14/14 - 1s - loss: 0.3084 - accuracy: 0.8880 - val_loss: 0.6738 - val_accuracy: 0
Epoch 26/100
14/14 - 1s - loss: 0.2751 - accuracy: 0.9081 - val_loss: 0.6685 - val_accuracy: 0
Epoch 27/100
14/14 - 1s - loss: 0.2951 - accuracy: 0.9004 - val_loss: 0.6155 - val_accuracy: 0
Epoch 28/100
14/14 - 1s - loss: 0.2726 - accuracy: 0.9140 - val_loss: 0.5904 - val_accuracy: 0
```

Preparing the test data

```
X_test
```

```
array([[0.87843137, 0.80784314, 0.85882353, ..., 0.89803922, 0.90196078,
        0.90196078],
       [0.55294118, 0.50196078, 0.45098039, ..., 0.4627451 , 0.50980392,
        0.56078431],
       [0.99607843, 0.99607843, 0.99607843, ..., 1.        , 1.        ,
        1.        ],
       ...,
       [1.        , 1.        , 1.        , ..., 1.        , 1.        ,
        1.        ],
       [1.        , 0.99607843, 0.99607843, ..., 0.99215686, 1.        ,
        1.        ],
       [0.95294118, 0.96862745, 0.97254902, ..., 0.59215686, 0.59215686,
        0.59215686]])
```

```
import os
my_classes = os.listdir("hcaptcha_dataset/test/")
print(my_classes)
```

```
['boat', 'motorcycle', 'airplane', 'truck', 'seaplane', 'bicycle', 'motorbus']
```

```
import pandas as pd

names=[]
my_train=[]
```

```
my_test=[]
my_total = []

for name in labels:
    im_num_train = len(os.listdir(f"./hcaptcha_dataset/train/{name}"))
    im_num_test = len(os.listdir(f"./hcaptcha_dataset/test/{name}"))

    names.append(name)
    my_train.append(im_num_train)
    my_test.append(im_num_test)
    my_total.append(im_num_train+im_num_test)
dic = {'Label': names, 'Number(training)': my_train, 'Number(testing)': my_test, 'Total':

my_dataframe = pd.DataFrame(dic)
my_dataframe
```

	Label	Number(training)	Number(testing)	Total
0	boat	528	134	662
1	motorcycle	592	141	733
2	airplane	402	101	503
3	truck	656	163	819
4	seaplane	280	75	355
5	bicycle	304	71	375
6	motorbus	256	61	317

```
from PIL import Image
import cv2
import matplotlib.pyplot as plt


for my_label in os.listdir("./hcaptcha_dataset/test/"):
    my_images = os.listdir(f"./hcaptcha_dataset/test/{my_label}")
    for image in my_images:
        try:
            my_img = cv2.imread(f"./hcaptcha_dataset/test/{my_label}/{image}")
            gray = cv2.cvtColor(my_img, cv2.COLOR_BGR2GRAY)
            cv2.imwrite(f"./hcaptcha_dataset/test/{my_label}/{image}", gray)
        except:
            continue


class_number = 0
num_classes = len(os.listdir(f"./hcaptcha_dataset/test/"))
count = 0

x_test = []
y_test = []
```

```
for classes in os.listdir("./hcaptcha_dataset/test/"):
    y_array = class_number
    all_images = os.listdir(f"./hcaptcha_dataset/test/{classes}")
    tot = len(all_images)
    count = 0
    for images in all_images:
        try:
            img = resize_and_normalise(f"./hcaptcha_dataset/test/{classes}/{images}")
            x_test.append(img)
            y_test.append(y_array)
            count+=1
        except:
            continue
    class_number+=1
    print(f"Class {classes} processessing done!")

print("All classes have been processed")
```

```
Class boat processessing done!
Class motorcycle processessing done!
Class airplane processessing done!
Class truck processessing done!
Class seaplane processessing done!
Class bicycle processessing done!
Class motorbus processessing done!
All classes have been processed
```

```
x_test = np.array(x_test)
y_test = np.array(y_test)
```

```
x_test.shape
```

```
(746, 40, 40)
```

```
x_test =  (np.array(x_test).reshape(np.array(x_test).shape[0],-1))
```

```
x_test.shape
```

```
(746, 1600)
```

```
y_result = best_model.predict(x_test)
```

```
24/24 [==============================] - 0s 5ms/step
```

```
y_result.shape
```

```
(746, 7)
```

```
y_result=list(y_result)
```

```
my_data = {'id_of_image': [],'predicted_class_of_image': [],'actual_class_of_image': []}
```

```
my_data = {'id_of_image': [],'predicted_class_of_image': [],'actual_class_of_image': []}
count=0
for i in range(746):
    y_result[i] = list(y_result[i])
    my_index = y_result[i].index(max(y_result[i]))
    my_data['id_of_image'].append(i)
    my_data['predicted_class_of_image'].append(my_index)
    my_data['actual_class_of_image'].append(y_test[i])
    if my_index==y_test[i]:
        count+=1
count/746
```

```
0.8404825737265416
```

The accuracy of our model is 84.048%

Colab paid products - Cancel contracts here

0s completed at 9:41 PM