CAP5404 Deep Learning for Computer Graphics
Dr. Corey Toler-Franklin
Course Project Part I Simple Learning Models: Classifiers and Regressors

Team Members:
Aadithya Kandeth - 69802791
Sai Nikhil Dondapati - 22286439
Shaanya Singh - 34762752

TIC TAC TOE

IMPLEMENTATION

Classifiers

Multilayer Perceptron

Multilayer Perceptron is a type of feed forward neural network. It consists of 3 main layers - input, hidden and output layer. There may be multiple hidden layers in a MLP. They can solve problems that are not linearly separable. MLP uses backpropagation to be able to adjust weights and minimize cost functions.

Implementation for Tic Tac Toe:

In the tic tac toe problem, we use the MLP integrated with sklearn. We have applied MLPClassifier to the three given datasets (tictac_final.txt, tictac_single.txt and tictac_multi.txt).

An example of MLPClassifier initialization:
MLPClassifier(solver='adam', alpha=1e-3, max_iter=5, hidden_layer_sizes=(256,256,128,), random_state=20, activation = 'tanh')

Here we have used the default solver ‘adam’. A solver helps with weight optimization.
Alpha is the strength of the L2 regularization term
Max_iter gives the number of iterations for which the solver tries to converge.
Hidden_layer_sizes gives the number of neurons in the hidden layers
Activation attributes to the activation function being used for the model.

Train-Test Split for Single Dataset: 80-20 respectively
Train-Test Split for Final Dataset: 80-20 respectively
Train-Test Split for Multi Dataset: 80-20 respectively

K-Nearest Neighbors

K-Nearest Neighbors is a non-parametric supervised learning that can be used for both classification and regression problems. The arguments of the KNeighborsClassifier that are commonly used to improve the model performance are n_neighbors (number of neighbors) and metric (distance metric used to calculate the proximity between two data points)

Implementation for Tic Tac Toe:

In the tic tac toe problem, we use the KNeighborsClassifier integrated with sklearn. We applied KNeighborsClassifier to the three given datasets (tictac_final.txt, tictac_single.txt and tictac_multi.txt).

We performed hyperparameter tuning for finding out the ideal number of neighbors that gives the best fit. For this, we split the data into 60% train, 20% validation and 20% test. We performed hyperparameter tuning on the validation data and we obtained the best fit when the number of neighbors is 1 for all the tic tac toe datasets. Using K-Fold cross validation, we found out that KNeighborsClassifier overfits on the training data when the number of neighbors is 1

Linear SVM

Support Vector Machine, sometimes known as SVM, is a linear model used to solve classification and regression issues. It works well for many real-world issues and can solve both linear and non-linear problems. The SVM concept is straightforward: A line or a hyperplane that divides the data into classes is produced by the algorithm.

Implementation for Tic Tac Toe:

To solve the tic tac toe problem, we use svm from the sklearn library. The model is evaluated for both the tictac_final.txt and tictac_single.txt datasets. The train test split is set to 80:20. After training the model on the training set, we obtain statistics like accuracy and print the confusion matrix.

The next step is to test the classifier on unseen data using K-fold cross validation and generate the accuracy. We use stratified K-Fold from the sklearn library. Stratified k-fold cross-validation does stratified sampling instead of random sampling. The model is then trained and then the accuracy is collected again.

Regressors

Multilayer Perceptron

Implementation for Tic Tac Toe:

We have applied MLPRegressor to the three given datasets (tictac_final.txt, tictac_single.txt and tictac_multi.txt).

An example of MLPRegressor initialization:
MLPRegressor(solver='adam', alpha=1e-6, max_iter=300, hidden_layer_sizes=(256,256,128,9), random_state=777, activation = 'relu')

Here we have used the default solver ‘adam’. A solver helps with weight optimization.
Alpha is the strength of the L2 regularization term
Max_iter gives the number of iterations for which the solver tries to converge.
Hidden_layer_sizes gives the number of neurons in the hidden layers
Activation attributes to the activation function being used for the model.

K- Nearest Neighbors

Implementation for Tic Tac Toe:

In the tic tac toe problem, we use the KNeighborsRegressor integrated with sklearn. We applied KNeighborsRegressor to the three given datasets (tictac_final.txt, tictac_single.txt and tictac_multi.txt).

We performed hyperparameter tuning for finding out the ideal number of neighbors that gives the best fit. For this, we split the data into 60% train, 20% validation and 20% test. We performed hyperparameter tuning on the validation data and we obtained the best fit when the number of neighbors is 1 for all the tic tac toe datasets. Using K-Fold cross validation, we found out that KNeighborsRegressor overfits on the training data when the number of neighbors is 1

Linear Regression using Normal Equations

Implementation for Tic Tac Toe:

Normal Equation used:

$$\theta = (X^T X)^{-1} X y$$

To implement linear regression using normal equations, we perform the below steps to manipulate the data:

1. Convert the data into numpy nd arrays.
2. Add a bias vector in the first column of the input data.
3. Take the transpose of this matrix X.
4. Multiply the X^T by X.
5. Take the inverse of X^T*X.
6. Multiple X^T by the output matrix Y.
7. Take $\theta = (X^T*X)^{-1} * (X^T * Y)$
8. $\theta[0]$ and $\theta[1]$ will give us the intercept and slope respectively
9. Repeat the steps for each of the 9 output vectors (Y1 …. Y9)

The accuracy is then calculated manually using the predicted results from X_test.

To use K-Fold cross validation, we use KFold from the sklearn library.

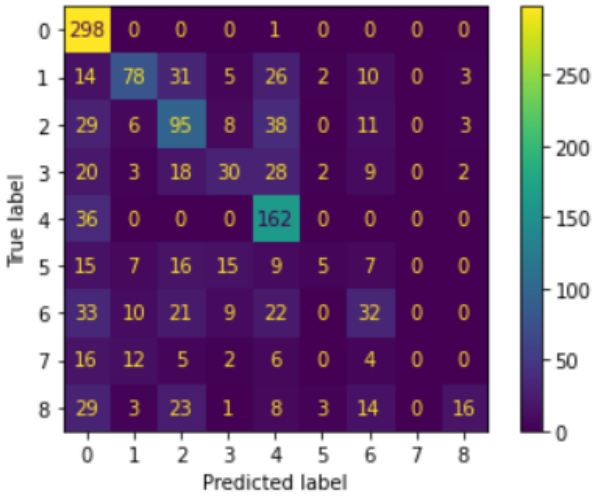
EVALUATION ON TIC TAC TOE BOARDS

Classifiers

Multilayer Perceptron

Performance on Single dataset (tictac_single.txt):

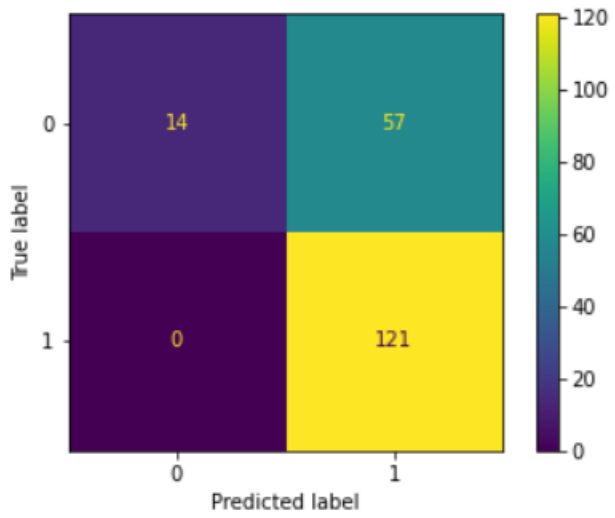
```
Classification Accuracy : 0.5461479786422578
Confusion Matrix:
```



```
Normalized confusion matrix: [[0.99665552 0.         0.         0.         0.00505051 0.
  0.         0.         0.        ]
 [0.04682274 0.46153846 0.16315789 0.04464286 0.13131313 0.02702703
  0.07874016 0.         0.03092784]
 [0.09698997 0.03550296 0.5        0.07142857 0.19191919 0.
  0.08661417 0.         0.03092784]
 [0.06688963 0.01775148 0.09473684 0.26785714 0.14141414 0.02702703
  0.07086614 0.         0.02061856]
 [0.12040134 0.         0.         0.         0.81818182 0.
  0.         0.         0.        ]
 [0.05016722 0.04142012 0.08421053 0.13392857 0.04545455 0.06756757
  0.05511811 0.         0.        ]
 [0.11036789 0.0591716  0.11052632 0.08035714 0.11111111 0.
  0.2519685  0.         0.        ]
 [0.05351171 0.07100592 0.02631579 0.01785714 0.03030303 0.
  0.03149606 0.         0.        ]
 [0.09698997 0.01775148 0.12105263 0.00892857 0.04040404 0.04054054
  0.11023622 0.         0.16494845]]
```

Performance on Final dataset (tictac_final.txt):

```
Classification Accuracy : 0.703125
Confusion Matrix:
```




```
Normalized confusion matrix: [[0.1971831  0.47107438]
 [0.         1.        ]]
```

Test results after K-Fold Cross Validation:

Single dataset:
Maximum test accuracy achieved with K-Fold Cross validation is 0.8947368421052632

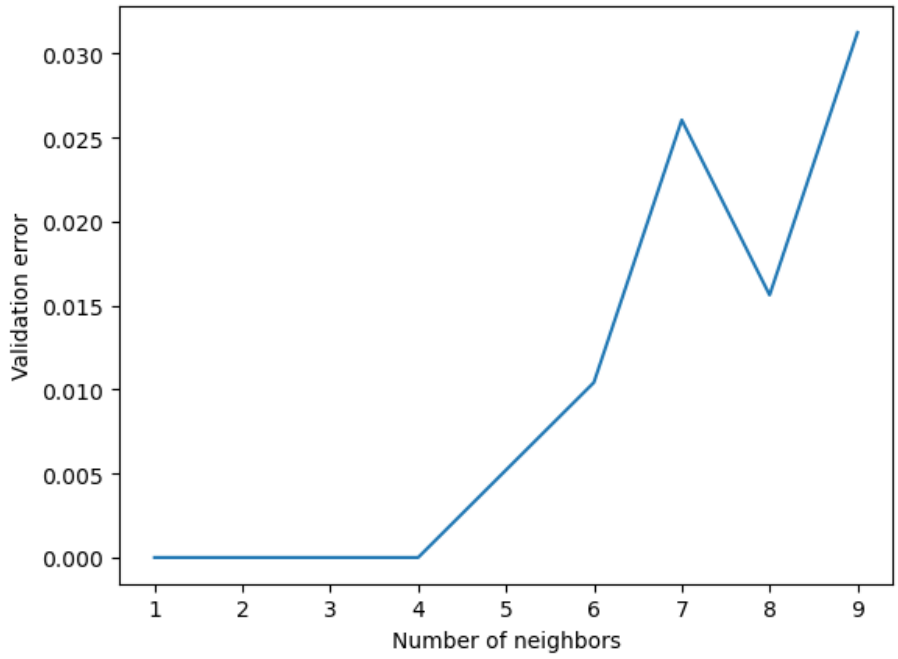
Final dataset:
Maximum test accuracy achieved with K-Fold Cross validation is 0.9947916666666666

Multi dataset:
Maximum test accuracy achieved with K-Fold Cross validation is 0.927027714213069

K-Nearest Neighbors

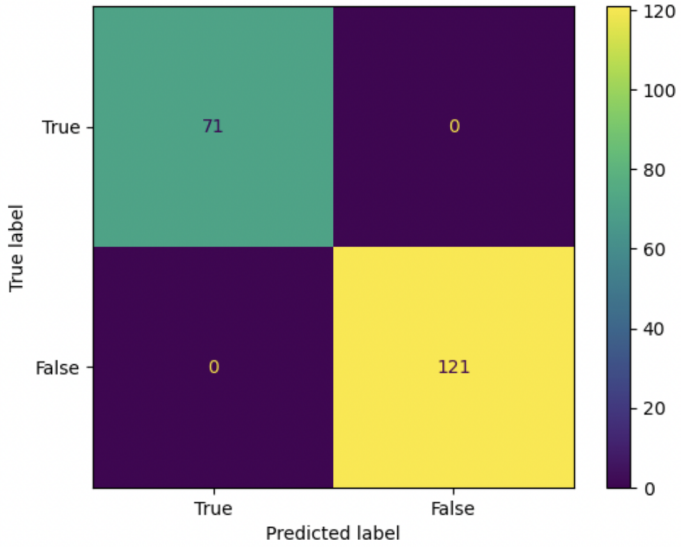
Performance on Final dataset (tictac_final.txt):

Hyperparameter tuning:



Accuracy Statistics and Confusion Matrix:

```
Accuracy of KNN Classifier on Final dataset is 1.0
******
Confusion matrix
```

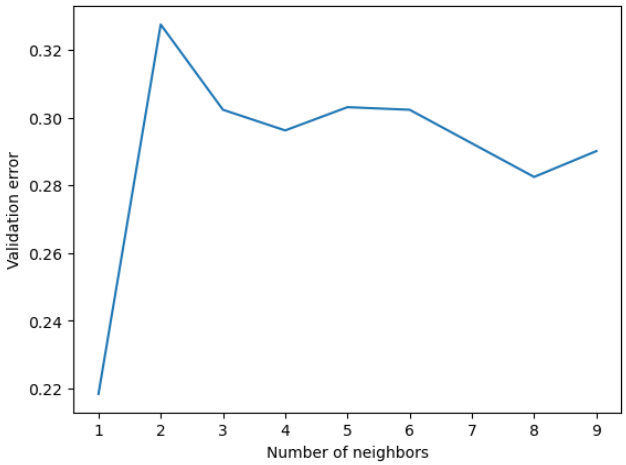



```
******
Normalized confusion matrix
[[1 0]
 [0 1]]
```

Maximum test accuracy acheived with K-Fold Cross validation is 1.0

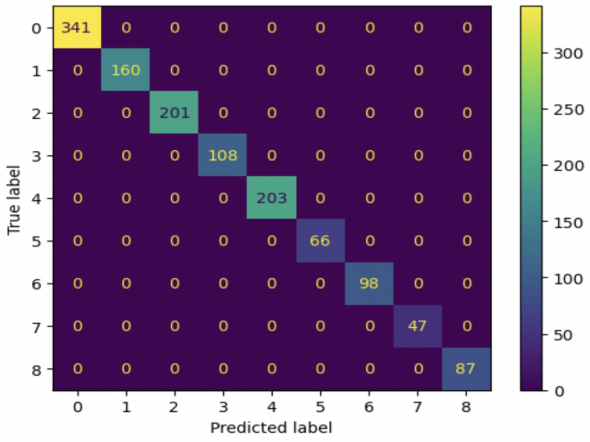
Performance on Single dataset (tictac_single.txt):

Hyperparameter tuning:



Accuracy Statistics and Confusion Matrix:

```
Accuracy of KNN Classifier on Final dataset is 1.0
******
Confusion matrix
```




```
******
Normalized confusion matrix
[[1 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 1]]
```

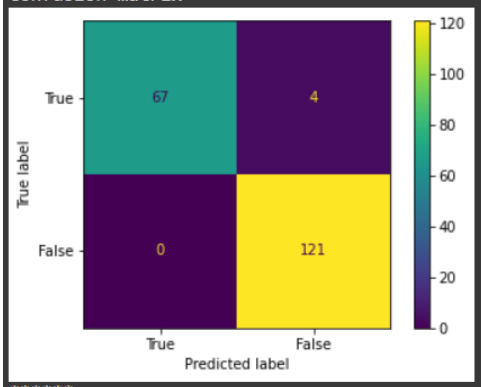
Maximum test accuracy acheived with K-Fold Cross validation is 0.899236641221374

Linear SVM

Performance on Single dataset (tictac_single.txt):

Accuracy Statistics and Confusion Matrix:

```
Accuracy of SVM on Final dataset is 0.9791666666666666
******
Confusion matrix
```

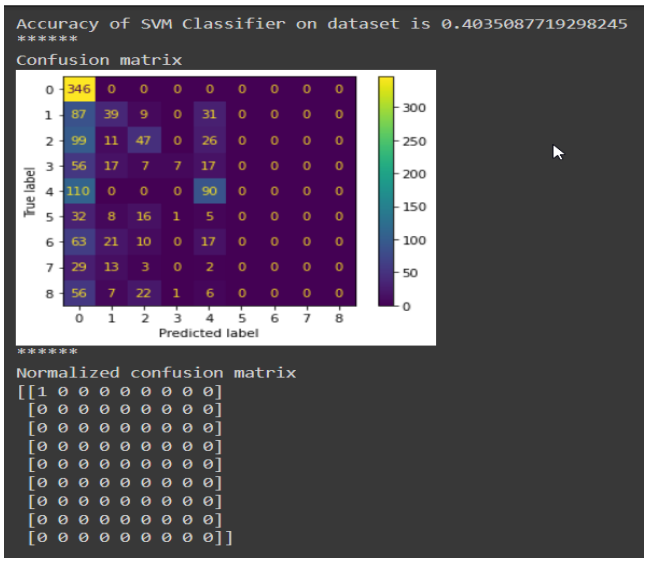



```
******
Normalized confusion matrix
[[0 0]
 [0 1]]
```

After performing K-Fold cross validation, Maximum test accuracy acheived was 0.9947916666666666.

Performance on Final dataset (tictac_final.txt):

Accuracy Statistics and Confusion Matrix:



After performing K-Fold cross validation, Maximum test accuracy acheived was 0.5469107551487414.

Regressors

Multilayer Perceptron

Single dataset:
Maximum test accuracy achieved with K-Fold Cross validation is 0.3251908396946565

Final dataset:
Accuracy of MLP Regressor on Final dataset is 0.9444444444444444

Multi dataset:
Maximum test accuracy achieved with K-Fold Cross validation is 0.9195928753180661

K- Nearest Neighbors

Single dataset:
Maximum test accuracy acheived with K-Fold Cross validation is 0.9114503816793893

Final dataset:
Maximum test accuracy acheived with K-Fold Cross validation is 1.0

Multi dataset:
Maximum test accuracy acheived with K-Fold Cross validation is 0.963019508057676

Linear Regression using Normal Equations.

For normal equations, accuracy had to be calculated manually since the normal equation had to be created manually. The calculated accuracy was 0.546910

For K-Fold cross validation using the SK-learn library:
Accuracy acheived with K-Fold Cross validation was 0.7903307888040713

Questions & Answers

Explain which method worked best for classification and why?

KNearestNeighborClassifier worked very well for classification with an average accuracy of more than 80% with K-fold cross validation. MLP Classifier also gave very good accuracy on the datasets. In tac tac toe game, moves that happen immediately after each other have a high correlation and we get to see a lot of similarity between the states. KNN Classifier works by finding similarity between the input data points. Therefore, we expect the KNN Classifier to work well on tic tac toe data and we have seen that happen.

Explain which method worked best for regression and why?

KNearestNeighborRegressor worked very well for classification with an average accuracy of more than 80% with K-fold cross validation. The performance of KNN Regressor is however less than that of the Classifier because we are predicting continuous outputs and then rounding them to zeros and ones to match with the original data. Because of the rounding process, we are losing some accuracy.

In tac tac toe game, moves that happen immediately after each other have a high correlation and we get to see a lot of similarity between the states. KNN Regressor works by finding similarity between the input data points. Therefore, we expect the KNN Regressor to work well on tic tac toe data and we have seen that happen.

MLP Regressor and Linear Regresson on the other gave only decent accuracy on the datasets because of the rounding process (converting continuous output to zeroes and ones).

Investigate (and report) what happens to the accuracy of the classifiers if they are trained on 1/10 as much data.

The accuracy of KNN Classifier with 1/10 data and K-fold cross validation is 0.865. There is not much difference with KNN Classifier. However, with MLP, the accuracy decreased by more than

25% owing to the fact that Neural Network architectures are data hungry. To conclude, MLP Classifier performs better with more data.

Explain why certain methods scale better to larger datasets than the others. Hint: the multilayer perceptron should work better than k-nearest neighbors regressors, but they both should have above 80% accuracy, and should play a decent game of Tic Tac Toe in the next step.

MLP Regressor gave more than 80% accuracy because the neural network architecture which has a lot of parameters is able to capture the relation between different states of the Tic Tac Toe game. KNN Regressor also gave more than 80% accuracy because KNN Regressor is good in capturing the similarities between data points which is very essential for games

INSTRUCTIONS

1. Open a Jupyter Notebook.
2. Open a new notebook and upload Final_Tic_Tac_Toe.ipynb from the Tic_Tac_Toe folder.
3. Upload all the datasets (tictac_final.txt, tictac_multi.txt and tictac_single.txt).
4. Run all cells to run all the models (or) import the libraries from the first cell and run the desired model.

BUGS/DIFFICULTIES

- Implementing K fold cross validation manually for the linear regression using normal equations was difficult so we had to use the sklearn library.
- The linear regression using normal equations produced low accuracy. This could possibly be because of:
    - Incorrect round off values for the predicted results
    - Loss of accuracy when converting continuous output values to 0,1
- Linear SVM did not produce accurate results on the tictac_single.txt dataset.
- MLP Classifier overfits if we do not have a validation dataset
- Hyperparameter tuning for finding ideal set of parameters for MLP Classifier is time consuming

CONNECT FOUR

IMPLEMENTATION

We trained Multi layer Perceptron Classifier on the Connect 4 dataset. We did hyperparameter tuning for finding the right set of parameters that give the best fit on the data. We selected Multi layer Perceptron because we wanted to train a complex model that has a lot of parameters and the ability to learn the complex relationship between different states and output. Neural Networks also has the ability to learn the correlation between different input data points and learning the correlation is important because the model needs to identify a lot of optimal moves. The more optimal moves the model can capture, the better chance the bot will have to win the game.

We did hyperparamter tuning with the following parameters:

activations=['relu','tanh','logistic']
solvers=['adam','sgd']
learning_rates=['constant','adaptive']
learning_rate_inits=[0.001,0.01]

The best architecture of the MLP Classifier we found has the following parameters:

{**'hidden_layer_sizes'**: (150, 100, 50), **'activation'**: 'relu', **'solver'**: 'sgd', **'max_iter'**: 1000, **'learning_rate'**: 'constant', **'learning_rate_init'**: 0.001}

```
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'tanh', 'solver': 'sgd', 'max_iter': 1000, 'learning_rate': 'con
stant', 'learning_rate_init': 0.001}
The validation error is: 0.20257567907630825
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'tanh', 'solver': 'sgd', 'max_iter': 1000, 'learning_rate': 'con
stant', 'learning_rate_init': 0.01}
The validation error is: 0.20146547257789948
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'tanh', 'solver': 'sgd', 'max_iter': 1000, 'learning_rate': 'ada
ptive', 'learning_rate_init': 0.001}
The validation error is: 0.20005921101324842
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'tanh', 'solver': 'sgd', 'max_iter': 1000, 'learning_rate': 'ada
ptive', 'learning_rate_init': 0.01}
The validation error is: 0.2104211383317297
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'logistic', 'solver': 'adam', 'max_iter': 1000, 'learning_rate':
'constant', 'learning_rate_init': 0.001}
The validation error is: 0.2078306565021094
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'logistic', 'solver': 'adam', 'max_iter': 1000, 'learning_rate':
'constant', 'learning_rate_init': 0.01}
The validation error is: 0.20353785804159574
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'logistic', 'solver': 'adam', 'max_iter': 1000, 'learning_rate':
'adaptive', 'learning_rate_init': 0.001}
The validation error is: 0.20790467026867
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'logistic', 'solver': 'adam', 'max_iter': 1000, 'learning_rate':
'adaptive', 'learning_rate_init': 0.01}
The validation error is: 0.20975501443268452
**************
The parameters are:
{'hidden_layer_sizes': (150, 100, 50), 'activation': 'logistic', 'solver': 'sgd', 'max_iter': 1000, 'learning_rate':
```

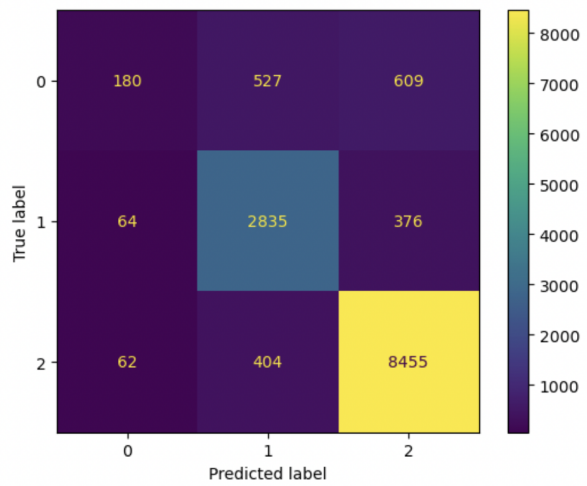
Hyperparameter tuning

We saved the model weights and generated a pickle file. We then programmed a bot which uses the saved model to predict the next step and return it to the human user. The programmed bot takes a state of the game from human user, plays its step (update the state of the Connect 4 board) and return the updated state to the user. It will then be users turn to play the next move. We implemented the gameplay using Command line interface.

EVALUATION

The best MLP classifier with the parameters mentioned above gave the accuracy of 0.8488

```
Accuracy of MLP Classifier on Connect dataset is 0.8488750740082889
******
Confusion matrix
```




```
******
```

The bot that made use of the above model can win against humans. However in some cases, the bot is losing against human by making silly moves. This is happening because the model doesn't have 100% accuracy. The bot is not able to recognize some optimal moves because of this reason and therefore is losing against human. But the bot we have developed is winning against human in most situations.

```
In [244]: def bot_playing_game(state,model):
              state=np.array(state,dtype=object)
              state=function_to_num(state)
              state=np.array(state)
              possible_new_state=[]
              all_indices=[]
              for i in range(7):
                  for step in range(6):
                      index=(41-i)-(7*step)
                      if state[index]==-1:
                          all_indices.append(index)
                          break
              for i in all_indices:
                  if state[i]==-1:
                      possible_new_state=copy.deepcopy(state)
                      possible_new_state[i]=0
                      pred_state=np.expand_dims(possible_new_state, axis=0)
                      if model.predict(pred_state)==1:
                          break
              possible_new_state=np.array(possible_new_state,dtype=object)
              return_state=np.array(function_to_alphabet(possible_new_state))
              return_state=return_state.reshape(6,7)
              return return_state

In [255]: human_move=[['b', 'b', 'b', 'b', 'b', 'b', 'b'],
                      ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
                      ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
                      ['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                      ['b', 'b', 'b', 'b', 'x', 'x', 'o'],
                      ['b', 'b', 'b', 'b', 'x', 'x', 'o']]

In [256]: bot_playing_game(human_move,loaded_model)

Out[256]: array([['b', 'b', 'b', 'b', 'b', 'b', 'b'],
                 ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
                 ['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                 ['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                 ['b', 'b', 'b', 'b', 'x', 'x', 'o'],
                 ['b', 'b', 'b', 'b', 'x', 'x', 'o']], dtype=object)

In [ ]:
```

P1.ipynb

Show

An instance of bot (Player O) winning the game

```
In [244]: def bot_playing_game(state,model):
              state=np.array(state,dtype=object)
              state=function_to_num(state)
              state=np.array(state)
              possible_new_state=[]
              all_indices=[]
              for i in range(7):
                  for step in range(6):
                      index=(41-i)-(7*step)
                      if state[index]==-1:
                          all_indices.append(index)
                          break
              for i in all_indices:
                  if state[i]==-1:
                      possible_new_state=copy.deepcopy(state)
                      possible_new_state[i]=0
                      pred_state=np.expand_dims(possible_new_state, axis=0)
                      if model.predict(pred_state)==1:
                          break
              possible_new_state=np.array(possible_new_state,dtype=object)
              return_state=np.array(function_to_alphabet(possible_new_state))
              return_state=return_state.reshape(6,7)
              return return_state

In [246]: human_move=[['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                      ['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                      ['b', 'b', 'b', 'b', 'b', 'b', 'x'],
                      ['b', 'b', 'b', 'b', 'b', 'x', 'o'],
                      ['b', 'b', 'b', 'b', 'x', 'x', 'o'],
                      ['b', 'b', 'b', 'b', 'x', 'x', 'o']]

In [247]: bot_playing_game(human_move,loaded_model)

Out[247]: array([['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                 ['b', 'b', 'b', 'b', 'b', 'b', 'o'],
                 ['b', 'b', 'b', 'b', 'b', 'o', 'x'],
                 ['b', 'b', 'b', 'b', 'b', 'x', 'o'],
                 ['b', 'b', 'b', 'b', 'x', 'x', 'o'],
                 ['b', 'b', 'b', 'b', 'x', 'x', 'o']], dtype=object)

In [ ]:
```

P1.ipynb

Show

An instance of bot blocking human from winning

INSTRUCTIONS

To Train:

1. Run Connect4_Training.ipynb

To Run:

1. Download Connect4.py
2. Have mlp_classifier_connect_4 in the same folder
3. Open a new python notebook/ Command prompt in the same directory
4. A sample notebook has been provided called Run_connect_four.ipynb.
5. Run Connect4.py

```
In [8]: 1 run Connect4.py
```

6. Call main and pass the input

```
In [9]: 1 from Connect4 import *
        2
        3 main([['b', 'b', 'b', 'b', 'b', 'b', 'b'],
        4       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
        5       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
        6       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
        7       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
        8       ['b', 'b', 'b', 'b', 'x', 'b', 'b']])

[['b' 'b' 'b' 'b' 'b' 'b' 'b']
 ['b' 'b' 'b' 'b' 'b' 'b' 'b']
 ['b' 'b' 'b' 'b' 'b' 'b' 'b']
 ['b' 'b' 'b' 'b' 'b' 'b' 'b']
 ['b' 'b' 'b' 'b' 'b' 'b' 'b']
 ['b' 'b' 'b' 'b' 'x' 'b' 'o']]

In [10]: 1 main([['b', 'b', 'b', 'b', 'b', 'b', 'b'],
         2       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
         3       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
         4       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
         5       ['b', 'b', 'b', 'b', 'b', 'b', 'b'],
         6       ['b', 'b', 'b', 'b', 'x', 'x', 'o']])
```

7. Keep running until either the human or the computer wins.

BUGS/DIFFICULTIES

- Converting input data that has ‘x’, ‘o’ and ‘b’ characters into numbers. This step is crucial because neural networks need numbers as input
- Using the developed model to play a move against a move made by human is challenging
- Hyperparameter tuning for finding ideal set of parameters for MLP Classifier is time consuming

APIs USED IN THIS PROJECT

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import LabelEncoder
from sklearn import svm
import pickle
import copy
```