Lab 5 (Team Mean)

- Authors:
  - Ahmet Ata Ersoy
  - Yasin Cagatay Duygu

We used Soccer Player Dataset (https://www.kaggle.com/datasets/thedevastator/fifa-world-cup-anomaly-detection-in-player-ratin) and we tried to predict players market values in 4 different ranges [0, 3e+6, 10e+6, 25e+6, 120e+06] Euros.

In [22]:
```
import pandas as pd
import numpy as np
```

In [23]:
```
pd.options.display.max_columns = None
pd.options.display.max_rows = 3
```

Reading data

In [24]:
```
df = pd.read_csv(r'players_20.csv')
df
```

Out[24]:

	sofifa_id	player_url	short_name	long_name	age	dob	height_cm	weight_kg	nationality	club	overall	potential	value_eur	wage_eur	player_positions	preferred_foot	internation
0	158023	https://sofifa.com/player/158023/lionel-messi/...	L. Messi	Lionel Andrés Messi Cuccittini	32	6/24/1987	170	72	Argentina	FC Barcelona	94	94	95500000	565000	RW, CF, ST	Left	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
18277	233449	https://sofifa.com/player/233449/ximing-pan/20...	Pan Ximing	潘喜明	26	1/11/1993	182	78	China PR	Hebei China Fortune FC	48	51	40000	2000	CM	Right	

18278 rows × 104 columns

Column name	Description
player_url	The URL of the player's FIFA profile. (String)
short_name	The player's short name. (String)
long_name	The player's long name. (String)
age	The player's age. (Integer)
dob	The player's date of birth. (String)
height_cm	The player's height in centimeters.
weight_kg	The player's weight in kilograms.
nationality	The player's nationality. (String)
club	The player's club. (String)
overall	The player's overall rating. (Integer)
potential	The player's potential rating. (Integer)
value_eur	The player's value in Euros. (Integer)
wage_eur	The player's wage in Euros. (Integer)
player_positions	The player's positions. (String)
preferred_foot	The player's preferred foot. (String)
international_reputation	The player's international reputation. (Integer)
weak_foot	The player's weak foot rating. (Integer)
skill_moves	The player's skill moves rating. (Integer)
work_rate	The player's work rate. (String)
body_type	The player's body type. (String)
gk_positioning	The player's goalkeeper positioning. (Integer)
player_traits	The player's traits. (String)
attacking_crossing	The player's crossing. (Integer)
attacking_finishing	The player's finishing. (Integer)
attackingheadingaccuracy	The player's heading accuracy. (Integer)
attackingshortpassing	The player's short passing.
attacking_volleys	The player's volleys. (Integer)
skill_dribbling	The player's dribbling.
skill_curve	The player's curve. (Integer)
skillfkaccuracy	The player's free kick accuracy. (Integer)
skilllongpassing	The player's long passing. (Integer)
skillballcontrol	The player's ball control. (Integer)
movement_acceleration	The player's acceleration. (Integer)
movementsprintspeed	The player's sprint speed. (Integer)
movement_agility	The player's agility. (Integer)
movement_reactions	The player's reactions. (Integer)
movement_balance	The player's balance. (Integer)
powershotpower	The player's shot power. (Integer)
power_jumping	The player's jumping. (Integer)
power_stamina	The player's stamina. (Integer)

- Taken from https://www.kaggle.com/datasets/thedevastator/fifa-world-cup-anomaly-detection-in-player-ratin

- Removing unnecesary attributes from the dataset.
- Replaced string variable to binary. The attribute was preferred foot of the player (Left or Right).

In [25]:
```
df = df.drop(['player_url', 'short_name' ,"long_name",
         'sofifa_id', 'real_face', 'loaned_from',
         'nation_position', 'nation_jersey_number',
         'gk_handling', 'gk_kicking', 'gk_reflexes',
         'gk_speed','gk_positioning', 'team_jersey_number',
         'gk_diving', 'goalkeeping_handling', 'goalkeeping_kicking',
         'goalkeeping_positioning', 'goalkeeping_reflexes', 'goalkeeping_diving',
         'dob'], axis=1)
df = df.replace({'preferred_foot': {'Right': 1, 'Left': 0}})
df
```

Out[25]:

	age	height_cm	weight_kg	nationality	club	overall	potential	value_eur	wage_eur	player_positions	preferred_foot	international_reputation	weak_foot	skill_moves	work_rate	body_type	release_clause_eu
0	32	170	72	Argentina	FC Barcelona	94	94	95500000	565000	RW, CF, ST	0	5	4	4	Medium/Low	Messi	195800000.(
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	..
18277	26	182	78	China PR	Hebei China Fortune FC	48	51	40000	2000	CM	1	1	3	2	Medium/Medium	Normal	Na

18278 rows × 83 columns

Changing join dates to years in current club.

- This data was from 2019, we decided to create a new variable by as a function of current date [2019] and the year they joined their current club (kind of Cross-Variable). Therefore this new attribute [joined] was created by:
  - joined = 2019 - join_year
- We had to modify date data from dd/mm/yy to year only.

In [26]:
```
print("Original DataFrame:")
print(df.joined)
df['joined'] = df["joined"].str.split("/", expand = True)[2]
df = df.dropna(
    axis=0,
    how='any',
    thresh=None,
    subset='joined',
)
df.joined = 2019 - pd.to_numeric(df.joined)
print("New DataFrame:")
print(df.joined)
```

```
Original DataFrame:
0        7/1/2004
           ...
18277         NaN
Name: joined, Length: 18278, dtype: object
New DataFrame:
0        15
         ..
18276     0
Name: joined, Length: 16990, dtype: int64
C:\Users\ataer\AppData\Local\Temp\ipykernel_26468\3535153878.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df.joined = 2019 - pd.to_numeric(df.joined)
```

Changing Position scores to numeric data

- These attributes below was indicator of how well the player performed in that position. We couldn't understand the +X part of the variable therefore, we reduced it to just an integer.
  - Before -> lw : 89+2 ::: After -> lw : 89

In [27]:
```
list1=['ls', 'st', 'rs', 'lw','lf','cf','rf','rw','lw','lam',
      'cam', 'ram', 'lm','lcm', 'cm', 'rcm', 'rm', 'lwb', 'ldm',
       'cdm', 'rdm', 'rwb', 'lb', 'lcb', 'cb', 'rcb', 'rb']

for i in list1:
    df[str(i)]= df[str(i)].str.split("+", expand = True)[0]
```

```
C:\Users\ataer\AppData\Local\Temp\ipykernel_26468\3804009049.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df[str(i)]= df[str(i)].str.split("+", expand = True)[0]
```

Removes Goal Keepers

- Goal keepers have completely different performance metric and they don't have measured performance variables for most of the attributes. Therefore, we elimiated all goal keepers by dropping all player without "PACE" attribute. This ensured we only had non-goal keeper players in dataset.

In [28]:
```
# Drop rows with any empty cells
df =df.dropna(
    axis=0,
    how='any',
    thresh=None,
    subset='pace',
)
df.release_clause_eur.fillna(0, inplace = True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15087 entries, 0 to 18276
Data columns (total 83 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   age                        15087 non-null  int64
```

```
 1   height_cm                 15087 non-null  int64
 2   weight_kg                 15087 non-null  int64
 3   nationality               15087 non-null  object
 4   club                      15087 non-null  object
 5   overall                   15087 non-null  int64
 6   potential                 15087 non-null  int64
 7   value_eur                 15087 non-null  int64
 8   wage_eur                  15087 non-null  int64
 9   player_positions          15087 non-null  object
 10  preferred_foot            15087 non-null  int64
 11  international_reputation  15087 non-null  int64
 12  weak_foot                 15087 non-null  int64
 13  skill_moves               15087 non-null  int64
 14  work_rate                 15087 non-null  object
 15  body_type                 15087 non-null  object
 16  release_clause_eur        15087 non-null  float64
 17  player_tags               1405 non-null   object
 18  team_position             15087 non-null  object
 19  joined                    15087 non-null  int64
 20  contract_valid_until      15087 non-null  float64
 21  pace                      15087 non-null  float64
 22  shooting                  15087 non-null  float64
 23  passing                   15087 non-null  float64
 24  dribbling                 15087 non-null  float64
 25  defending                 15087 non-null  float64
 26  physic                    15087 non-null  float64
 27  player_traits             6412 non-null   object
 28  attacking_crossing        15087 non-null  int64
 29  attacking_finishing       15087 non-null  int64
 30  attacking_heading_accuracy  15087 non-null  int64
 31  attacking_short_passing   15087 non-null  int64
 32  attacking_volleys         15087 non-null  int64
 33  skill_dribbling           15087 non-null  int64
 34  skill_curve               15087 non-null  int64
 35  skill_fk_accuracy         15087 non-null  int64
 36  skill_long_passing        15087 non-null  int64
 37  skill_ball_control        15087 non-null  int64
 38  movement_acceleration     15087 non-null  int64
 39  movement_sprint_speed     15087 non-null  int64
 40  movement_agility          15087 non-null  int64
 41  movement_reactions        15087 non-null  int64
 42  movement_balance          15087 non-null  int64
 43  power_shot_power          15087 non-null  int64
 44  power_jumping             15087 non-null  int64
 45  power_stamina             15087 non-null  int64
 46  power_strength            15087 non-null  int64
 47  power_long_shots          15087 non-null  int64
 48  mentality_aggression      15087 non-null  int64
 49  mentality_interceptions   15087 non-null  int64
 50  mentality_positioning     15087 non-null  int64
 51  mentality_vision          15087 non-null  int64
 52  mentality_penalties       15087 non-null  int64
 53  mentality_composure       15087 non-null  int64
 54  defending_marking         15087 non-null  int64
 55  defending_standing_tackle  15087 non-null  int64
 56  defending_sliding_tackle  15087 non-null  int64
 57  ls                        15087 non-null  object
 58  st                        15087 non-null  object
 59  rs                        15087 non-null  object
 60  lw                        15087 non-null  object
 61  lf                        15087 non-null  object
 62  cf                        15087 non-null  object
 63  rf                        15087 non-null  object
 64  rw                        15087 non-null  object
 65  lam                       15087 non-null  object
 66  cam                       15087 non-null  object
 67  ram                       15087 non-null  object
 68  lm                        15087 non-null  object
 69  lcm                       15087 non-null  object
 70  cm                        15087 non-null  object
 71  rcm                       15087 non-null  object
 72  rm                        15087 non-null  object
 73  lwb                       15087 non-null  object
 74  ldm                       15087 non-null  object
 75  cdm                       15087 non-null  object
 76  rdm                       15087 non-null  object
 77  rwb                       15087 non-null  object
 78  lb                        15087 non-null  object
 79  lcb                       15087 non-null  object
 80  cb                        15087 non-null  object
 81  rcb                       15087 non-null  object
 82  rb                        15087 non-null  object
dtypes: float64(8), int64(41), object(34)
memory usage: 9.7+ MB
```

- Investigating the dataset, we found there are some meaningless values given to some player's stats. For instance, there are 9 body types recorded into dataset but only 3 of them are meaningful and 6 of them are meaningless.

- Joined datasi olmayanlar silindi.

- Some of the player had unique assigned body types. There were only 5 of them, therefore we manually changed those to Normal, Stocky or Lean body types.

In [29]:

```
print(f'Original unique body types: \n\t{df.body_type.unique()}')
df = df.replace(['Messi', 'C. Ronaldo', 'Neymar', 'PLAYER_BODY_TYPE_25'], 'Normal')
df = df.replace(['Shaqiri', 'Akinfenwa'], 'Stocky')
print(f'Changed unique body types: \n\t{df.body_type.unique()}')
```

```
Original unique body types: 
	['Messi' 'C. Ronaldo' 'Neymar' 'Normal' 'Lean' 'PLAYER_BODY_TYPE_25'
 'Stocky' 'Shaqiri' 'Akinfenwa']
Changed unique body types: 
	['Normal' 'Lean' 'Stocky']
```

ONE-HOT Encoding

- Most of our data had attributes that held multiple tags (attributes) assigned to them and those were player_tags (Dribbler, Distance Shooter etc.), player_traits (Beat Offside Trap, Argues with Officials etc.), player_positions (LS (Left Striker), RW (Right Wing) etc.).
- We had to seperate those traits, tags etc. by ',' before one-hot encoding them.
- Some players didn't have any recorded traits or tags, one-hot encoding made it possible to assign players with binary integers for given attributes, therefore there was no NaN variable at the end.

In [30]:

```
df = (
    df.drop(columns='player_tags')
      .join(df['player_tags'].str.get_dummies(sep=','))
)
df = (
    df.drop(columns='player_traits')
      .join(df['player_traits'].str.get_dummies(sep=','))
)
df = (
    df.drop(columns='player_positions')
```

```
      .join(df['player_positions'].str.get_dummies(sep=','))
)
```

- One hot encoding remaining categorical attributes.

In [31]:
```
df = df.join(pd.get_dummies(data = df[['club', 'nationality', 'team_position', 'body_type']],
                    drop_first = True)
       )

df = df.drop(['club', 'nationality', 'team_position', 'body_type'], axis = 1)
```

Split Work Rate into Defensive / Offensive and convert into numerical value

- work_rate attibute had defensive and offensive stats. We seperated those into two variables.

In [32]:
```
#print("Original DataFrame:")
#print(df.joined)
df['defensive_work_rate'] = df["work_rate"].str.split("/", expand = True)[1]
df['offensive_work_rate'] = df["work_rate"].str.split("/", expand = True)[0]
df = df.dropna(
    axis=0,
    how='any',
    thresh=None,
    subset='joined',
)

df = df.drop(['work_rate'], axis=1)
```

- Change the ordinal attributes to ordinal integers.

In [33]:
```
df = df.replace({'offensive_work_rate': {'Low': 0, 'Medium': 1, 'High':2}})
df = df.replace({'defensive_work_rate': {'Low': 0, 'Medium': 1, 'High':2}})
```

Change year that contract is valid into how many years left in the contract

Changed valid contract year attribute from date to how many years left until it ends

In [34]:
```
pd.options.display.max_rows = 10
df.contract_valid_until=df.contract_valid_until-2019
```

In [35]:
```
df
```

Out[35]:

	age	height_cm	weight_kg	overall	potential	value_eur	wage_eur	preferred_foot	international_reputation	weak_foot	skill_moves	release_clause_eur	joined	contract_valid_until	pace	shooting	passing	dribbling	d
0	32	170	72	94	94	95500000	565000	0	5	4	4	195800000.0	15	2.0	87.0	92.0	92.0	96.0	
1	34	187	83	93	93	58500000	405000	1	5	4	5	96500000.0	1	3.0	90.0	93.0	82.0	89.0	
2	27	175	68	92	92	105500000	290000	1	5	5	5	195200000.0	2	3.0	91.0	85.0	87.0	95.0	
4	28	175	74	91	91	90000000	470000	1	4	4	4	184500000.0	0	5.0	91.0	83.0	86.0	94.0	
5	28	181	70	91	91	90000000	370000	1	4	5	4	166500000.0	4	4.0	76.0	86.0	92.0	86.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
18271	20	180	72	48	59	50000	1000	1	1	3	2	88000.0	1	0.0	52.0	37.0	47.0	46.0	
18273	22	186	79	48	56	40000	2000	1	1	3	2	70000.0	1	0.0	57.0	23.0	28.0	33.0	
18274	22	177	66	48	56	40000	2000	1	1	2	2	72000.0	0	3.0	58.0	24.0	33.0	35.0	
18275	19	186	75	48	56	40000	1000	1	1	2	2	70000.0	0	0.0	54.0	35.0	44.0	45.0	
18276	18	185	74	48	54	40000	1000	1	1	2	2	70000.0	0	3.0	59.0	35.0	47.0	47.0	

15087 rows × 1025 columns

Value Distrubition Visualization

We will select player values as the target. Before turning them into classes, we checked their distrubition.

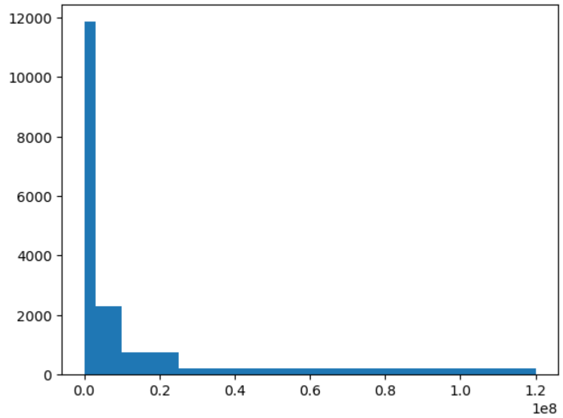
In [36]:
```
df['value_eur'].describe()
```

Out[36]:
```
count    1.508700e+04
mean     2.630759e+06
std      5.780851e+06
min      0.000000e+00
25%      3.500000e+05
50%      7.500000e+05
75%      2.300000e+06
max      1.055000e+08
Name: value_eur, dtype: float64
```

In [37]:
```
import matplotlib.pyplot as plt

plt.hist(df.value_eur, range=[0, 1.055000e+08], bins=[0, 3e+6, 10e+6, 25e+6, 120e+06 ])
```

Out[37]:
```
(array([11860.,  2300.,   729.,   198.]),
 array([0.0e+00, 3.0e+06, 1.0e+07, 2.5e+07, 1.2e+08]),
 <BarContainer object of 4 artists>)
```



Dividing players into 4 tier groups in terms of their market value.

In [38]:
```
pd_cut = pd.cut(df.value_eur, bins = [-10, 3e+6, 10e+6, 25e+6, 120e+08 ], labels = [3,2,1,0])

df = df.drop('value_eur',axis=1)
df2 = df.copy()
```

Normalize Numeric data

- First, we changed all variables to numeric data.

In [39]:
```
df.dtypes[df.dtypes == object]
for i in list1:
    df[i]=pd.to_numeric(df[i])
print(f'We current have only numeric data:\n\t{df.dtypes.unique()}')
```

```
We current have only numeric data:
        [dtype('int64') dtype('float64') dtype('uint8')]
```

In [40]:
```
normalized_df=(df-df.min())/(df.max()-df.min())

y = np.array(pd_cut)
# y = (y-y.min())/(y.max()-y.min())
X = np.asarray(normalized_df)

print(f'Normalized dataframes max and min values:\n\t Xmax:{X.max()}\t Xmin: {X.min()}')
print(f'Normalized dataframes max and min values:\n\t ymax:{y.max()}\t ymin: {y.min()}')
```

```
Normalized dataframes max and min values:
         Xmax:1.0        Xmin: 0.0
Normalized dataframes max and min values:
         ymax:3  ymin: 0
```

Cross-Product Features

- We didn't use cross-products of attributes that much. That's because, none of attributes are a direct of function each other.

KERAS

In [41]:
```
from sklearn import metrics as mt
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
import tensorflow as tf
from tensorflow import keras
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'

print(tf.__version__)
print(keras.__version__)

from keras.wrappers.scikit_learn import KerasClassifier

from tensorflow.keras.layers import Dense, Activation, Input
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.utils import plot_model
```

```
2.10.0
2.10.0
```

Our Model Configuration

- Model class takes the hyper parameters like:
  - loss_function, optimizer, testSize, number of layers layer and accuracy type.
- For shuffling, we use cross-validation Stratified Shuffle Split because:
  - We have unbalanced data. Stratified Shuffle Split conserves the ratio of the classes for all folds, test and train data.

In [56]:
```
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import KFold
import numpy as np

class ModelConfig(object):

    def __init__(
                self, batch_size = 50,
                loss_function = sparse_categorical_crossentropy,
                no_classes = 4,
                no_epochs = 10,
                optimizer = Adam(),
                verbosity = 0,
```

```
                 num_folds = 2,
                 randomState = 42,
                 testSize = 0.2 ,
                 trainSize = 0.8,
                 no_layer = 3,
                 first_layer_size = 1024,
                 accuracy_type = 'sparse_categorical_accuracy',
                 inputs = [],
                 targets = []):

        self.batch_size = batch_size
        self.loss_function = loss_function
        self.no_classes = no_classes
        self.no_epochs = no_epochs
        self.optimizer = optimizer
        self.verbosity = verbosity
        self.num_folds = num_folds
        self.randomState = randomState
        self.testSize = testSize
        self.trainSize = trainSize
        self.no_layer = no_layer
        self.inputs = inputs
        self.targets = targets
        self.accuracy_type = accuracy_type
        self.first_layer_size = first_layer_size

    def run(self):
        # Define per-fold score containers
        self.acc_per_fold = []
        loss_per_fold = []

        # Define StratifiedShuffleSplit Cross Validator
        shuffle = StratifiedShuffleSplit(n_splits=self.num_folds,
                                         test_size=self.testSize,
                                         train_size=self.trainSize,
                                         random_state=self.randomState)
        input_shape=self.inputs.shape

        # StratifiedShuffleSplit Cross Validation model evaluation
        fold_no = 1

        for train, test in shuffle.split(self.inputs, self.targets):

            # Define the model architecture
            model = Sequential()
            i = 0
            while i < self.no_layer - 1:
                i += 1
                model.add(Dense(np.round(self.first_layer_size/i), activation='relu'))
            model.add(Dense(self.no_classes, activation='softmax'))

            # Compile the model
            model.compile(loss=self.loss_function,
                          optimizer=self.optimizer,
                          metrics=[self.accuracy_type])

            print(f'------------------------------------------------------------------------\n',
                  f'Training for fold {fold_no} ...')


            # Fit data to model
            self.Model = model.fit(self.inputs[train], self.targets[train],
                        batch_size=self.batch_size,
                        epochs=self.no_epochs,
                        verbose=self.verbosity,
                        validation_data=(self.inputs[test], self.targets[test]))

            # Generate generalization metrics
            self.X_test = self.inputs[test]
            self.y_test = self.targets[test]
            scores = model.evaluate(self.inputs[test], self.targets[test], verbose=0)
            print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}%')
            self.acc_per_fold.append(scores[1])
            loss_per_fold.append(scores[0])

            # Increase fold number
            fold_no = fold_no + 1
            self.ypred = model.predict(self.X_test).argmax(axis=-1)
```

Accuracy Metric:

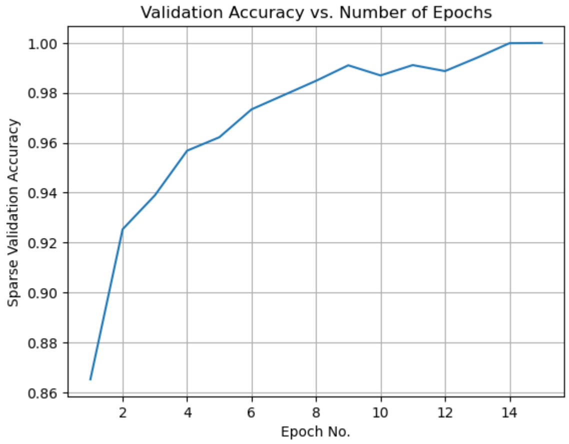
- Both accuracy and sparse_categorical_accuracy gives similar results however, as we have multiclass target which has unbalance between different classes we decided to stick with sparse_categorical_accuracy because:
  - Calculates how often predictions matches integer labels.
  - Our classes are equally important eventhough they are unbalanced.
  - We have multiclass targets.

In [ ]:

```
testModel = ModelConfig(num_folds = 1, no_layer = 3, no_epochs = 15, verbosity = 1,
                        accuracy_type = 'accuracy',  inputs = X, targets = y )
%time testModel.run()
```

In [191…

```
plt.plot(range(1,16),testModel.Model.history['accuracy'])
plt.title('Validation Accuracy vs. Number of Epochs')
plt.xlabel('Epoch No.')
plt.ylabel('Sparse Validation Accuracy')
plt.grid('both')
```

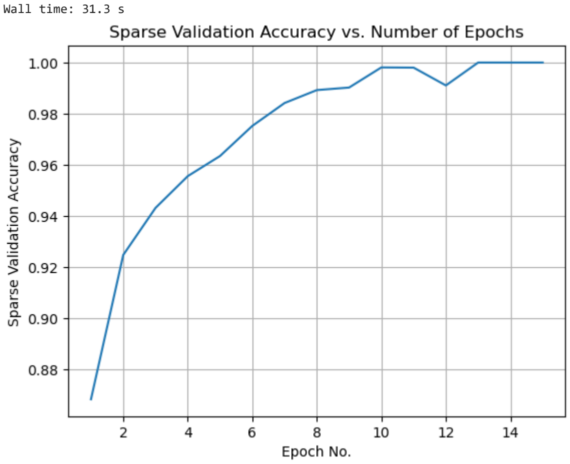


In [192…

```
testModel = ModelConfig(num_folds = 1, no_layer = 3, no_epochs = 15, verbosity = 1,
                        accuracy_type = 'sparse_categorical_accuracy', inputs = X, targets = y )
%time testModel.run()

plt.plot(range(1,16),testModel.Model.history['sparse_categorical_accuracy'])
plt.title('Sparse Validation Accuracy vs. Number of Epochs')
plt.xlabel('Epoch No.')
plt.ylabel('Sparse Validation Accuracy')
plt.grid('both')
```

```
Epoch 1/15
242/242 [==============================] - 2s 8ms/step - loss: 0.3270 - sparse_categorical_accuracy: 0.8683 - val_loss: 0.2058 - val_sparse_categorical_accuracy: 0.9102
Epoch 2/15
242/242 [==============================] - 2s 8ms/step - loss: 0.1732 - sparse_categorical_accuracy: 0.9248 - val_loss: 0.1827 - val_sparse_categorical_accuracy: 0.9231
Epoch 3/15
242/242 [==============================] - 2s 8ms/step - loss: 0.1344 - sparse_categorical_accuracy: 0.9432 - val_loss: 0.1672 - val_sparse_categorical_accuracy: 0.9251
Epoch 4/15
242/242 [==============================] - 2s 8ms/step - loss: 0.1071 - sparse_categorical_accuracy: 0.9556 - val_loss: 0.1744 - val_sparse_categorical_accuracy: 0.9294
Epoch 5/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0882 - sparse_categorical_accuracy: 0.9635 - val_loss: 0.2172 - val_sparse_categorical_accuracy: 0.9238
Epoch 6/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0643 - sparse_categorical_accuracy: 0.9752 - val_loss: 0.1885 - val_sparse_categorical_accuracy: 0.9271
Epoch 7/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0432 - sparse_categorical_accuracy: 0.9842 - val_loss: 0.2081 - val_sparse_categorical_accuracy: 0.9304
Epoch 8/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0336 - sparse_categorical_accuracy: 0.9892 - val_loss: 0.3168 - val_sparse_categorical_accuracy: 0.9162
Epoch 9/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0272 - sparse_categorical_accuracy: 0.9902 - val_loss: 0.2226 - val_sparse_categorical_accuracy: 0.9321
Epoch 10/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0085 - sparse_categorical_accuracy: 0.9981 - val_loss: 0.2884 - val_sparse_categorical_accuracy: 0.9324
Epoch 11/15
242/242 [==============================] - 2s 10ms/step - loss: 0.0086 - sparse_categorical_accuracy: 0.9980 - val_loss: 0.2825 - val_sparse_categorical_accuracy: 0.9321
Epoch 12/15
242/242 [==============================] - 3s 11ms/step - loss: 0.0251 - sparse_categorical_accuracy: 0.9911 - val_loss: 0.2591 - val_sparse_categorical_accuracy: 0.9334
Epoch 13/15
242/242 [==============================] - 2s 8ms/step - loss: 0.0023 - sparse_categorical_accuracy: 1.0000 - val_loss: 0.2825 - val_sparse_categorical_accuracy: 0.9331
Epoch 14/15
242/242 [==============================] - 2s 8ms/step - loss: 9.3221e-04 - sparse_categorical_accuracy: 1.0000 - val_loss: 0.2975 - val_sparse_categorical_accuracy: 0.9331
Epoch 15/15
242/242 [==============================] - 2s 8ms/step - loss: 5.4642e-04 - sparse_categorical_accuracy: 1.0000 - val_loss: 0.2999 - val_sparse_categorical_accuracy: 0.9357
CPU times: total: 3min 24s
Wall time: 31.3 s
```



Epoch size study

- We have seen platoing after Epoch No. 20. Therefore, we will use number of epochs as 20 from now on.

Layer Number study

In [195…

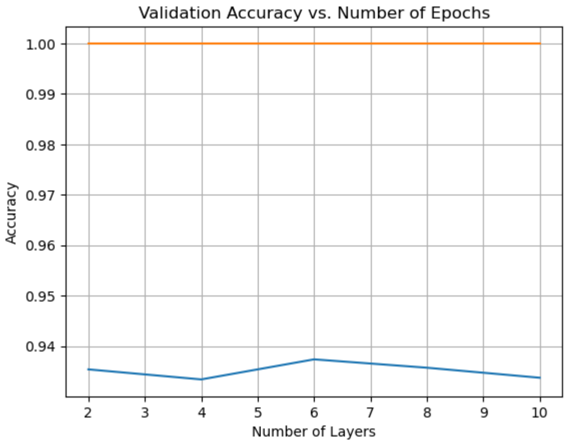
```
valAccuracy_List = []
Accuracy_List = []
i = 0
while i < 10:
    i += 2
    testModel = ModelConfig(num_folds = 1, no_layer = i, no_epochs = 20,
                            accuracy_type = 'sparse_categorical_accuracy',
                            inputs = X, targets = y )
    testModel.run()
    valAccuracy_List.append(testModel.Model.history['val_sparse_categorical_accuracy'][-1])
    Accuracy_List.append(testModel.Model.history['sparse_categorical_accuracy'][-1])
    print(f"no layer: {i}\tAccuracy: {Accuracy_List[-1]} \tValidation Accuracy: {valAccuracy_List[-1]}")
```

```
no layer: 2     Accuracy: 1.0   Validation Accuracy: 0.9353876709938049
no layer: 4     Accuracy: 1.0   Validation Accuracy: 0.9333996176719666
```

```
no layer: 6     Accuracy: 1.0   Validation Accuracy: 0.9373757243156433
no layer: 8     Accuracy: 1.0   Validation Accuracy: 0.9357190132141113
no layer: 10    Accuracy: 1.0   Validation Accuracy: 0.933730959892273
```

In [289…

```
plt.plot(range(2,11,2),valAccuracy_List)
plt.plot(range(2,11,2),Accuracy_List)
plt.title('Validation Accuracy vs. Number of Epochs')
plt.xlabel('Number of Layers')
plt.ylabel('Accuracy')
plt.grid('both')
```



Cross-Validation

- Even with 3 layers and 5 epoch, the sparse_categorical_accuracy doesn't change between the folds that much.

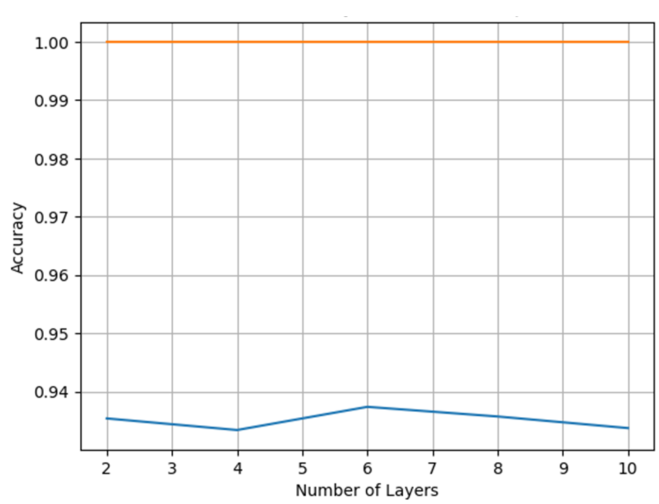
In [157…

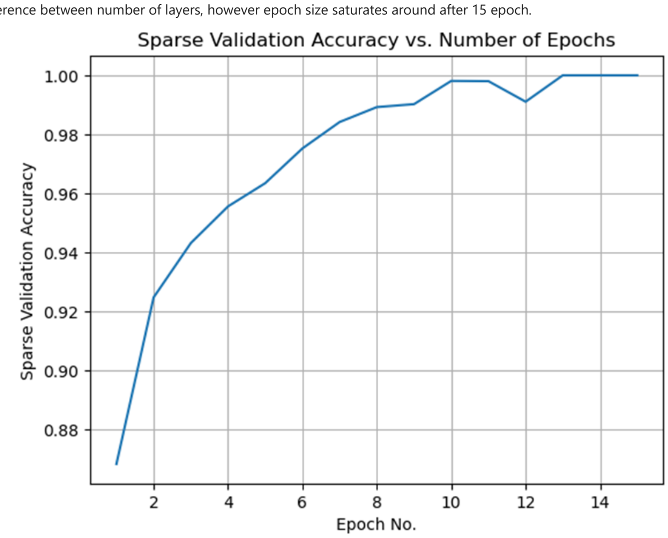
```
valAccuracy_List = []
Accuracy_List = []
testModel = ModelConfig(num_folds = 5, no_layer = 3, no_epochs = 5,
                        accuracy_type = 'sparse_categorical_accuracy',
                        verbosity=0,
                        inputs = X, targets = y )
testModel.run()
```

```
------------------------------------------------------------------------
 Training for fold 1 ...
Score for fold 1: loss of 0.18399615585803986; sparse_categorical_accuracy of 92.74353981018066%
95/95 [==============================] - 0s 2ms/step
------------------------------------------------------------------------
 Training for fold 2 ...
Score for fold 2: loss of 0.15735597908496857; sparse_categorical_accuracy of 94.10205483436584%
95/95 [==============================] - 0s 2ms/step
------------------------------------------------------------------------
 Training for fold 3 ...
Score for fold 3: loss of 0.173183873295784; sparse_categorical_accuracy of 93.57190132141113%
95/95 [==============================] - 0s 2ms/step
------------------------------------------------------------------------
 Training for fold 4 ...
Score for fold 4: loss of 0.14824780821800232; sparse_categorical_accuracy of 93.903249502182%
95/95 [==============================] - 0s 2ms/step
------------------------------------------------------------------------
 Training for fold 5 ...
Score for fold 5: loss of 0.1549304872751236; sparse_categorical_accuracy of 93.70443820953369%
95/95 [==============================] - 0s 2ms/step
```

Comparison with Standard MLP

- Earlier we have shown how accuracy changes with different epoch and layer numbers. There is a slight difference between number of layers, however epoch size saturates around after 15 epoch.





!

In [ ]:

```
plt.plot(range(2,11,2),valAccuracy_List)
plt.plot(range(2,11,2),Accuracy_List)
plt.title('Validation Accuracy vs. Number of Epochs')
plt.xlabel('Number of Layers')
plt.ylabel('Accuracy')
plt.grid('both')
```

ROC Curve

In [58]:
```
testModel = ModelConfig(num_folds = 1, no_layer = 6, no_epochs = 20, verbosity = 0,
                        accuracy_type = 'sparse_categorical_accuracy',
                        inputs = X, targets = y)
testModel.run()
```

```
------------------------------------------------------------------------
 Training for fold 1 ...
Score for fold 1: loss of 0.44457846879959106; sparse_categorical_accuracy of 91.9151782989502%
95/95 [==============================] - 0s 3ms/step
```

In [59]:
```
from sklearn.preprocessing import LabelBinarizer
lb = LabelBinarizer()
lb.fit(testModel.y_test)
y_test_ROC=lb.transform(testModel.y_test)
lb = LabelBinarizer()
lb.fit(testModel.ypred)
y_pred_ROC=lb.transform(testModel.ypred)
```

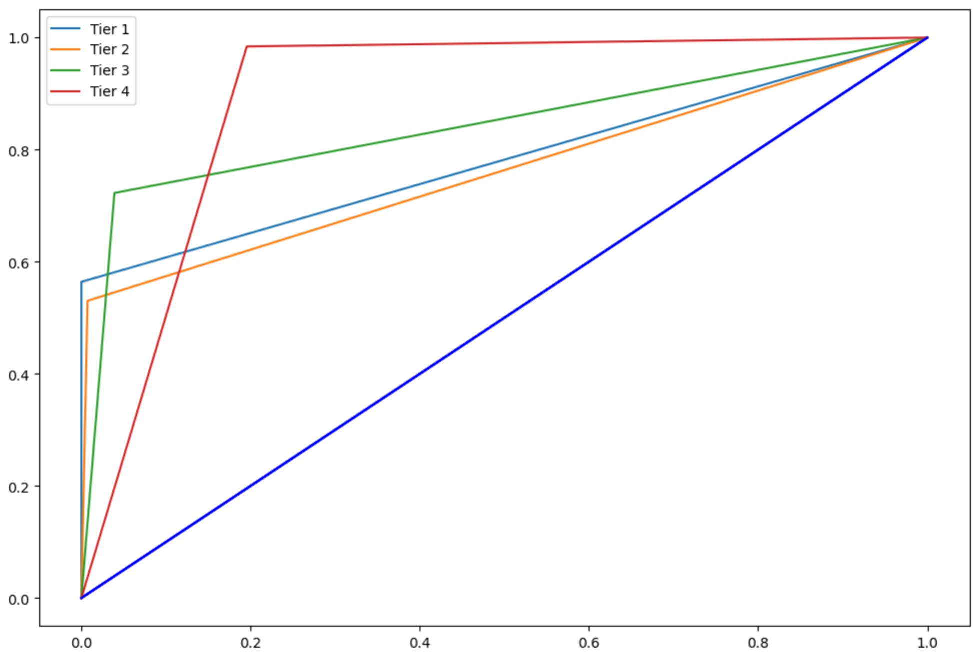
In [152…
```
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score

# function for scoring roc auc score for multi-class
def multiclass_roc_auc_score(y_test, y_pred, average="micro"):
    fig, c_ax = plt.subplots(1,1, figsize = (12, 8))
    lb = LabelBinarizer()
    lb.fit(y_test)
    y_test_ROC=lb.transform(y_test)
    lb = LabelBinarizer()
    lb.fit(y_pred)
    y_pred_ROC=lb.transform(y_pred)

    for (idx, c_label) in enumerate(['Tier 1','Tier 2','Tier 3','Tier 4']):
        fpr, tpr, thresholds = roc_curve(y_test_ROC[:,idx], y_pred_ROC[:,idx])
        c_ax.plot(fpr, tpr, label = f'{c_label}')
        c_ax.plot(fpr, fpr, 'b-')
        c_ax.legend()
    #c_ax.label(['0','1','2','3'])
    return roc_auc_score(y_test_ROC, y_pred_ROC, average=average)
```

In [153…
```
print('ROC AUC score:', multiclass_roc_auc_score(testModel.y_test, testModel.ypred))
```

```
ROC AUC score: 0.9461011707532583
```



- We can see that model predicts Tier 4 and Tier 3 better. Our data was unbalanced and they dominate the dataset.
- Tier 1 players were predicted slightly better than Tier 2 players even though they are the least presented class. Because there is high skill gap between Tier 1 and other Tiers.