Final Project Report

Alexander Oliver **Hussain Quadri**

1 Introduction/Modified Proposal

`https://www.kaggle.com/c/pubg-finish-placement-prediction`

Our team has chosen the Kaggle challenge of predicting player placements in the Player Unknown Battle Grounds (PUBG) video game. The project link is given above. By using the algorithms and techniques learned in this class, we can use various methods to determine the relevance of different aspects of the player statistics, and decide on how different features affect the scoring. A possible challenge would be that the players performance is not strictly linked to either pure skill or their in-game statistics, and instead has an immeasurable and more random quality. We hope to show that this is not the case and that the players' observable statistics have a large bearing on performance.

1.1 Why Use Machine Learning

Machine learning techniques are appropriate for this data because we must predict the final placement of players using their statistics gathered in the game. We must see how different features such as the number of eliminations, the amount of boosts used, etc impact the possibility of a player winning. Noting patterns manually given a small enough set of data is personally intensive, and the accuracy becomes limited to the size of training set a person is able to analyze. However, a machine learning algorithm begins in the training effort how we do, by knowing nothing about these patterns, then gaining knowledge in a measurable way, and can eventually come to classify the data to a notable accuracy.

1.2 Dataset Acquisition

The dataset is available on the main Kaggle project page, under the "Data" tab. We are provided with two sets of data; the training set, and the test set, which contain a large number of anonymous PUBG game stats from around 65000 games, formatted so that each row contains one player's post-game stats. The data comes from matches of all types: solos, duos, squads, and custom; there is no guarantee of there being 100 players per match, nor at most 4 players per group. So while a player may choose different strategies to play in solo games compared to games with 4-person teams, we must account for all data equally in order to determine common elements of success for all of these game modes. The data itself is provided by the PUBG Developer API.

1.3 Initial thoughts on Possible Approaches

Our goal is to implement a Multi-layered Neural Network for this dataset. This type of neural net requires minimal pre-processing, making it optimal for the large amount of data and is able to accommodate for non-linear trends in the data. We will start off with a simple, single layered Neural Network baseline and train our data using that. Once we are able to do that, we intend to build our MNN from the baseline.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

2 The Benefits/Drawbacks of this Project

This project intrigued us mainly because we are involved in this gaming community. It seemed an interesting idea to apply the machine learning techniques we have discovered in this class, and to explore statistics for predictions that we otherwise would not have thought about. The idea to predict how a player will place is a very good idea, and can have a great impact on competitive matches where professional gamers can train using the data provided by this exploration. They can, for example, see where their skills/play-style is lacking and improve on that, since there will be evidence to suggest that improving on a certain feature heavily increases the likelihood of winning. This can be also used internally by the PUBG Development team if they were ever to introduce an actual ranking system where the player base is grouped into tiers based on their performance and their predicted placements. In addition, if factors such as boosters and other health regenerating items were the soul determiner of success in a match, it may be worthwhile for the developers to reduce the impact of such items so as to create a more even playing field among the players. We also realised from the start that this operation is very time expensive, and there are a lot of steps needed to be done before we can even begin implementing our algorithms.

3 Methods/Implementation

We started off with a theoretical implementation of decision trees using this data. We did NOT implement any of this using code, but we used this as a baseline to our baseline, i.e. we created models of what this model would look like, what features we should use in our prediction calculations, etc for the purpose of getting ourselves started on the project. All implementation were worked on equally by Alex and Hussain. Different functionality was shared between us, and we would merge once we get our seperate parts working.

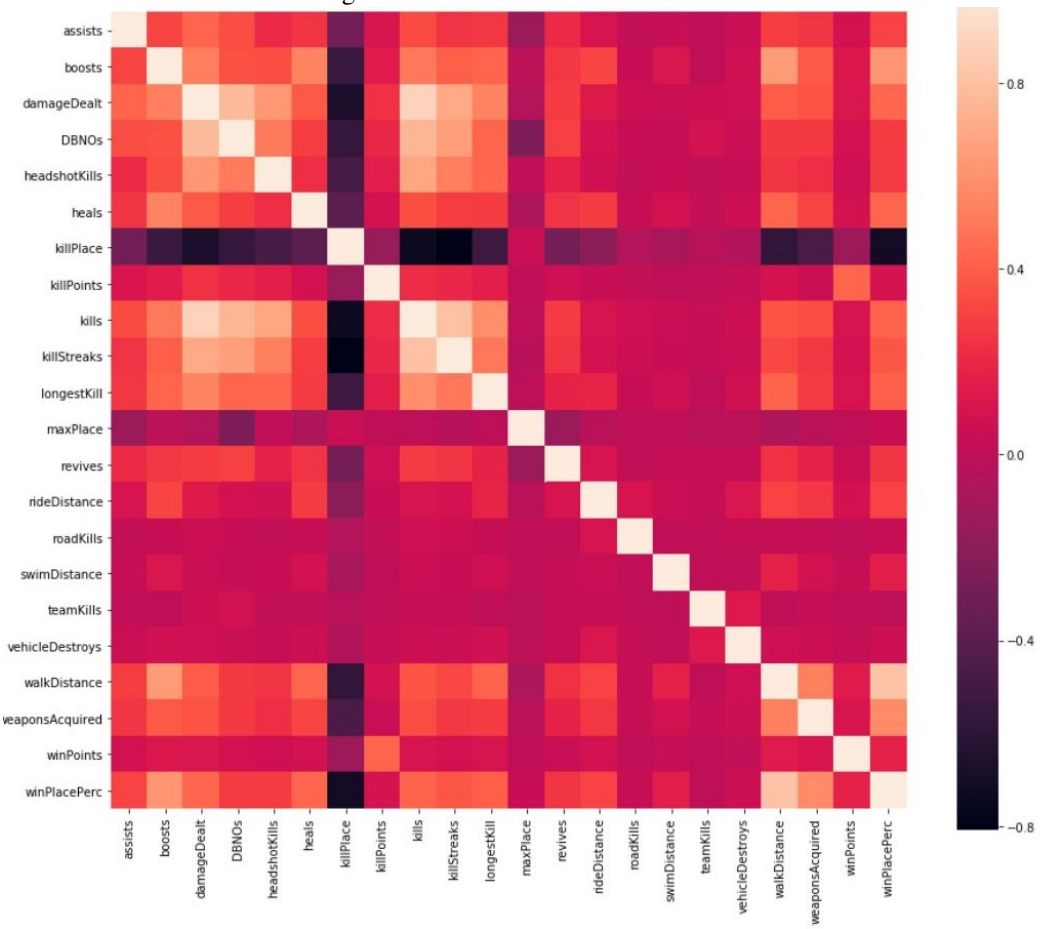
Before we started our implementation, we realised (after using a base kernel on the kaggle competitions web-page) that our run-times are well over 8 hours. This is when we decided that we must first clean up our data before progressing further. We cycled through the training/test data-frames and modified the data types in order to reduce memory usage. The results of that can be seen below. This decreased our run-times to where we could reliably test our implementations.

Figure 1: Memory Usage Reduction

We decided that the most important variable that we want to predict is winPlacePerc. We first checked the variables that are correlated to winPlacePerc. winPoints is the external ranking of the player based on past performance (higher value means more wins in the past). It was interesting to see that past successes are not a huge contributing factor to winning an ongoing match. walkDistance is definitely correlated with winning since a player must travel long distances in order to survive for a long time. damageDealt and kills correlate well, higher values mean that the player is playing well. killPlace (ranking of kills vs other players) is not correlated since someone not getting any kills is most likely to not survive long.

We plotted a heatmap to visualise how correlated each variable is to one another. We can see the maxPlace and numGroups are highly correlated. As expected, the winPoints correlates very highly to killPoints (more kills means more wins). maxPlace and numGroups is something that we should not focus on, as we found that the maxPlace is always greater than or equal to numGroups, which means that some of the groups' data is missing from matches. This helped us decide which features to discard (we get rid of these in our implementation of the neural net). From this, we know which features we are selecting.

Figure 2: Correlations between features



We then built our MNN based off our theories. Generally, we built our Neural Network with Keras, with 3 hidden layers using RELU Activation. The model is sequential. We started off by grouping our data on matchID and gameID and getting the features (means, maxes, mins) for each group, since this is a team game and scored within a group would be the same. We then added the feature of each match, in order to compare individual stats for each player in each group with individuals in other groups. We merged these new dataframes on the means/maxes/mins of the statistics, and targeted/trained on the winPlacePercentages. We discarded the columns that we believe have no contribution to the predictions (ID, matchID, groupID, IDmean/max/min/match-mean) Team skill

level is more important that individual skill level (if one teammate is bad, it's more likely to lose the match), so we remove the features of each player and select the features for just the groups and match. The 3 hidden layers used RELU activation. We normalised after each layer, as well as had a dropout(1) after each layer. The first layer has a density of 512 nodes, the second is of 256 nodes, the third is of 128 nodes. Refer to Fig. 3.

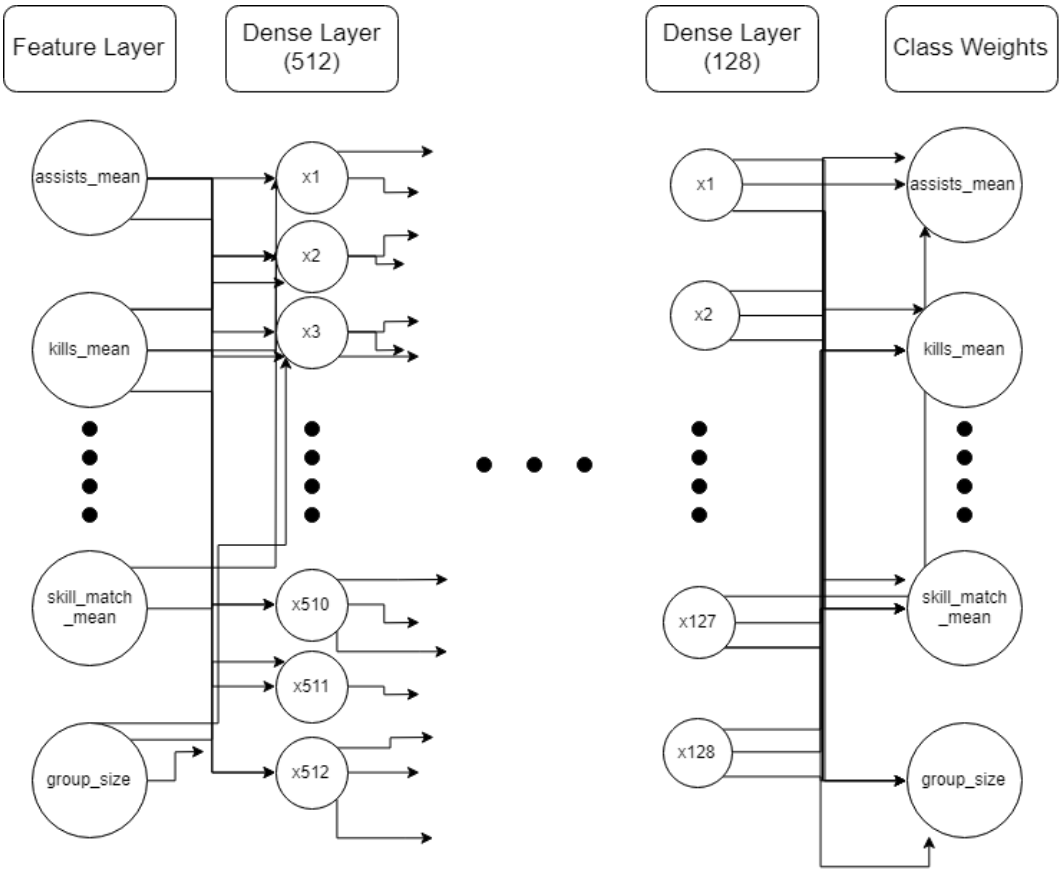


Figure 3: Neural Network Structure

The results from this were that we are indeed on the correct track. Our results show that the training results absolutely matched our training data to a high degree of accuracy (0.80). The graphs below show our model being fit to the data, showing the loss/absolute errors between the test and training data for 30 epochs. We tried running the program for more epochs (30000) in order to get a more accurate fit of the model, however we encountered crashes and realised that it would take too much time to to get such graphs, hence we stuck to a small number of epochs. Below is also a sample output from the training.

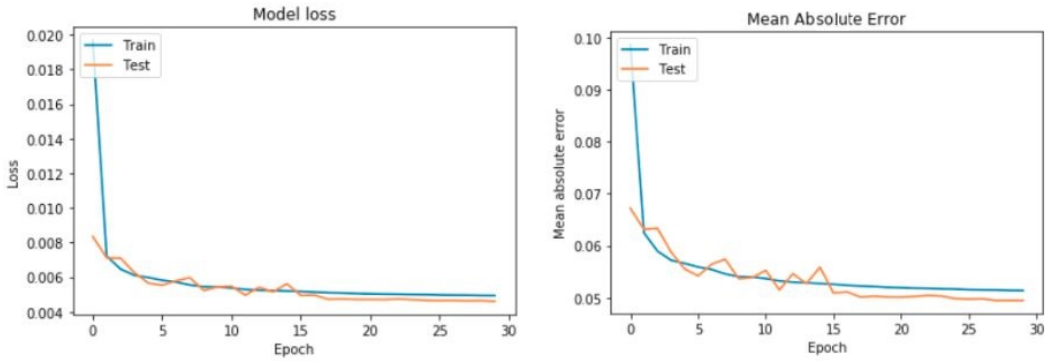


Figure 4: Training data vs Validation Loss/Mean Absolute Error

	Id	winPlacePerc
0	9329eb41e215eb	0.268149
1	639bd0dcd7bda8	0.905389
2	63d5c8ef8dfe91	0.620866
3	cf5b81422591d1	0.600212
4	ee6a295187ba21	0.957888

Figure 5: Training data vs Validation Loss/Mean Absolute Error

4 Error Analysis

We initially started our data exploration on a full dataset, underestimating the time commitment required. Hence we followed the memory efficiency guide/kernel on kaggle to optimise our datasets. After a few attempts at implementing our neural net, we decided to look at which features are unnecessary and are taking valuable resources. To do this, we plotted the heatmap and figured out which features correlate best with each other, and which do not/which have discrepancies. Once getting a visual look at this, we decided what features we must discard since they play no role in the prediction of our placements.

References

[1] Deltanullnull https://www.kaggle.com/deltanullnull/neural-chicken-dinner-network/notebook

[2] Anycode Baseline NeuralNet Kernel https://www.kaggle.com/anycode/simple-nn-baseline-4/code.

5 Appendix

```
import os
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn import preprocessing

from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from keras.callbacks import LearningRateScheduler, EarlyStopping,
    ModelCheckpoint, ReduceLROnPlateau
from keras import optimizers
from keras.models import load_model

df_train = pd.read_csv('../input/train_V2.csv')
df_test = pd.read_csv('../input/test_V2.csv')

""" Memory saving function credit to
https://www.kaggle.com/gemartin/load-data-reduce-memory-usage"""
def reduce_mem_usage(df):
    """ iterate through all the columns of a dataframe and modify the data type
        to reduce memory usage.
    """
    #start_mem = df.memory_usage().sum() / 1024**2
    #print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

    for col in df.columns:
        col_type = df[col].dtype

        if col_type != object:
            c_min = df[col].min()
            c_max = df[col].max()
            if str(col_type)[:3] == 'int':
                if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                    df[col] = df[col].astype(np.int8)
                elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                    df[col] = df[col].astype(np.int16)
                elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                    df[col] = df[col].astype(np.int32)
                elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                    df[col] = df[col].astype(np.int64)
            else:
                if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
                    df[col] = df[col].astype(np.float16)
                elif c_min > np.finfo(np.float32).min and c_max
                < np.finfo(np.float32).max:
                    df[col] = df[col].astype(np.float32)
                else:
                    df[col] = df[col].astype(np.float64)

    #end_mem = df.memory_usage().sum() / 1024**2
```

```
    #print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
    #print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_mem))

    return df

df_train = reduce_mem_usage(df_train)
df_test = reduce_mem_usage(df_test)


TrainingSetNumerics = df_train.select_dtypes(include=[np.number])
NumericCorrelations = TrainingSetNumerics.corr()
#print(NumericCorrelations['winPlacePerc'].sort_values(ascending = False),'\n')

#Produce Heatmap of feature correlations
f , ax = plt.subplots(figsize = (15,13))
sns.heatmap(NumericCorrelations, square = True,  vmax=.82)

# Unifying Features
df_train["distance"] = df_train["rideDistance"]+df_train["walkDistance"]
    +df_train["swimDistance"]
# df_train["healthpack"] = df_train["boosts"] + df_train["heals"]
df_train["skill"] = df_train["headshotKills"]+df_train["roadKills"]
df_test["distance"] = df_test["rideDistance"]+df_test["walkDistance"]
    +df_test["swimDistance"]
# df_test["healthpack"] = df_test["boosts"] + df_test["heals"]
df_test["skill"] = df_test["headshotKills"]+df_test["roadKills"]

"""
Grabbing Feature of each group
"""
df_train_size =
    df_train.groupby(['matchId','groupId']).size().reset_index(name='group_size')
df_test_size =
    df_test.groupby(['matchId','groupId']).size().reset_index(name='group_size')

df_train_mean = df_train.groupby(['matchId','groupId']).mean().reset_index()
df_test_mean = df_test.groupby(['matchId','groupId']).mean().reset_index()

df_train_max = df_train.groupby(['matchId','groupId']).max().reset_index()
df_test_max = df_test.groupby(['matchId','groupId']).max().reset_index()

df_train_min = df_train.groupby(['matchId','groupId']).min().reset_index()
df_test_min = df_test.groupby(['matchId','groupId']).min().reset_index()

"""
Grabbing Features of Each Match
"""
df_train_match_mean = df_train.groupby(['matchId']).mean().reset_index()
df_test_match_mean = df_test.groupby(['matchId']).mean().reset_index()

df_train = pd.merge(df_train, df_train_mean, suffixes=["", "_mean"],
    how='left', on=['matchId', 'groupId'])
df_test = pd.merge(df_test, df_test_mean, suffixes=["", "_mean"],
    how='left', on=['matchId', 'groupId'])
del df_train_mean
del df_test_mean

df_train = pd.merge(df_train, df_train_max, suffixes=["", "_max"],
    how='left', on=['matchId', 'groupId'])
df_test = pd.merge(df_test, df_test_max, suffixes=["", "_max"],
    how='left', on=['matchId', 'groupId'])
del df_train_max
del df_test_max

df_train = pd.merge(df_train, df_train_min, suffixes=["", "_min"],
```

```
    how='left', on=['matchId', 'groupId'])
df_test = pd.merge(df_test, df_test_min, suffixes=["", "_min"],
    how='left', on=['matchId', 'groupId'])
del df_train_min
del df_test_min

df_train = pd.merge(df_train, df_train_match_mean, suffixes=["", "_match_mean"],
    how='left', on=['matchId'])
df_test = pd.merge(df_test, df_test_match_mean, suffixes=["", "_match_mean"],
    how='left', on=['matchId'])
del df_train_match_mean
del df_test_match_mean

df_train = pd.merge(df_train, df_train_size, how='left', on=['matchId', 'groupId'])
df_test = pd.merge(df_test, df_test_size, how='left', on=['matchId', 'groupId'])
del df_train_size
del df_test_size

target = 'winPlacePerc'
train_columns = list(df_test.columns)

""" Discard redundant columns """
train_columns.remove("Id")
train_columns.remove("matchId")
train_columns.remove("groupId")
train_columns.remove("Id_mean")
train_columns.remove("Id_max")
train_columns.remove("Id_min")
train_columns.remove("Id_match_mean")

"""
Remove Features of each player, focus on group/match
Team Skill level has higher precedence over personal skill
"""
train_columns_new = []
for name in train_columns:
    if '_' in name:
        train_columns_new.append(name)
train_columns = train_columns_new
print(train_columns)

X = df_train[train_columns]
Y = df_test[train_columns]
T = df_train[target]

del df_train

x_train, x_test, t_train, t_test = train_test_split(X, T, test_size = 0.2,
    random_state = 1234)

# scaler = preprocessing.MinMaxScaler(feature_range=(-1, 1)).fit(x_train)
scaler = preprocessing.QuantileTransformer().fit(x_train)

x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)
Y = scaler.transform(Y)

print("x_train", x_train.shape, x_train.min(), x_train.max())
print("x_test", x_test.shape, x_test.min(), x_test.max())
print("Y", Y.shape, Y.min(), Y.max())

model = Sequential()
model.add(Dense(512, kernel_initializer='he_normal', input_dim=x_train.shape[1],
    activation='relu'))
model.add(BatchNormalization())
```

```
model.add(Dropout(0.1))
model.add(Dense(256, kernel_initializer='he_normal', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.1))
model.add(Dense(128, kernel_initializer='he_normal', activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.1))

optimizer = optimizers.Adam(lr=0.01, epsilon=1e-8, decay=1e-4, amsgrad=False)
model.compile(optimizer=optimizer, loss='mse', metrics=['mae'])

def step_decay_schedule(initial_lr=1e-3, decay_factor=0.75, step_size=10, verbose=0):
    '''
    Wrapper function to create a LearningRateScheduler with step decay schedule.
    '''
    def schedule(epoch):
        return initial_lr * (decay_factor ** np.floor(epoch/step_size))

    return LearningRateScheduler(schedule, verbose)

lr_sched = step_decay_schedule(initial_lr=0.1, decay_factor=0.9, step_size=1, verbose=1)
early_stopping = EarlyStopping(monitor='val_mean_absolute_error', mode = 'min',
    patience=4, verbose=1)

history = model.fit(x_train, t_train,
                    validation_data=(x_test, t_test),
                    epochs=3000,
                    batch_size=32768,
                    callbacks=[lr_sched,early_stopping],
                    verbose=1)


plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

#Plotting Mean Absolute Error of Test Data
plt.plot(history.history['mean_absolute_error'])
plt.plot(history.history['val_mean_absolute_error'])
plt.title('Mean Absolute Error')
plt.ylabel('Mean Absolute Error')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

pred = model.predict(Y)
pred = pred.ravel()

# pred = (pred + 1) / 2
df_test['winPlacePercPred'] = np.clip(pred, a_min=0, a_max=1)


aux = df_test.groupby(['matchId','groupId'])
    ['winPlacePercPred'].agg('mean').groupby('matchId').rank(pct=True).reset_index()
aux.columns = ['matchId','groupId','winPlacePerc']
df_test = df_test.merge(aux, how='left', on=['matchId','groupId'])

submission = df_test[['Id', 'winPlacePerc']]
#submission.head()
submission.to_csv('submission.csv', index=False)
```