HearthArena_Tensorflow

January 21, 2021

1 Deck Score

1.1 HearthArena Deck List Data

[13]:
```
deckLists = []
deckScores = []
addedLinks = []
link_dict_list = []
```

[10]:
```
import requests
from bs4 import BeautifulSoup
```

[9]:
```
profiles =␣
 ↪['zloyindy','krippers','boozor','adubs','dsorrow','woett','subsume','bighugger']
classes =␣
 ↪['demon-hunter','druid','hunter','mage','paladin','priest','rogue','shaman','warlock','warr
```

[11]:
```
# Generate links for individual decks

link_list = []

for profile in profiles:

    for clas in classes:

        URL = 'https://www.heartharena.com/profile/{}/{}'.format(profile,clas)
        page = requests.get(URL)

        soup = BeautifulSoup(page.content, 'html.parser')
        result = soup.find('section',class_='recent-arena-runs')

        try:

            trs = result.find_all('tr')

            for tr in trs:

                try:
                    link_list.append('https://www.heartarena.
 ↪com'+tr['data-href'])

                except:

                    pass

        except:

            pass
```

[ ]:

```
# Scrape individual decks for card data

for link in link_list:

    if link not in addedLinks:

        addedLinks.append(link)
        page = requests.get(link)
        page_html = BeautifulSoup(page.content,'html.parser')
        deckList = page_html.find('ul',class_='deckList')
        cardChoices = page_html.find(id='choices')

        cardCount = dict()

        link_dict = dict()
        pickedCardsList = []
        allChoicesNames = []
        allChoicesScores = []

        try:

            cards = deckList.find_all('li',class_='deckCard')

            for card in cards:

                name = card.find('span',class_='name').text
                quantity = card.find('span',class_='quantity').text
                cardCount.update([(name,quantity)])

            deckLists.append(cardCount)

            tierScore = page_html.find(id='deck-tier-score')
            score = tierScore.find('span')
            deckScores.append(score.text)
            pickedCards = cardChoices.find_all('li',class_='picked')

            for card in pickedCards:

                pick = card.find('span',class_='name')
                pickedCardsList.append(pick.text)

            allNames = cardChoices.find_all('span',class_='name')

            for name in allNames:

                allChoicesNames.append(name.text)

            allScores = cardChoices.find_all('span',class_='score')

            for score in allScores:

                allChoicesScores.append(score.text)

            link_dict['pickedCardsList'] = pickedCardsList
            link_dict['allChoicesNames'] = allChoicesNames
            link_dict['allChoicesScores'] = allChoicesScores

            link_dict_list.append(link_dict)

        except:

            pass
```

[18]:

```
# Only run when sure output should be saved

%store link_list
%store link_dict_list
%store deckLists
%store deckScores
%store addedLinks
```

```
Stored 'link_list' (list)
Stored 'link_dict_list' (list)
Stored 'deckLists' (list)
Stored 'deckScores' (list)
Stored 'addedLinks' (list)
```

[2]:

```
# Run to restore output

%store -r link_list
%store -r link_dict_list
%store -r deckLists
```

```
%store -r deckScores
%store -r addedLinks
```

1.2 Hearthstone API Card Data

[22]:
```
client_id = r'f850b1be704941eb94679d9ebe066f23'
client_secret = r'WU2QGIbCZYolIxJOvEDmp7GG1MnBxz9r'
redirect_uri = r'http://www.google.com'
```

[23]:
```
import sys
sys.path.append('C:
 ↪\\users\jarel\\appdata\local\programs\python\python38-32\lib\site-packages')
from requests_oauthlib import OAuth2Session
```

[24]:
```
oauth = OAuth2Session(client_id, redirect_uri=redirect_uri)
authorization_url, state = oauth.authorization_url(
        'https://us.battle.net/oauth/authorize')

print('Please go to %s and authorize access.' % authorization_url)
authorization_response = input('Enter the full callback url')
```

```
Please go to https://us.battle.net/oauth/authorize?response_type=code&client_id=
f850b1be704941eb94679d9ebe066f23&redirect_uri=http%3A%2F%2Fwww.google.com&state=
JUByHzYhHRfdf05fWIlky8JElZoDLd and authorize access.
Enter the full callback urlhttps://www.google.com/?code=USEXCBPZEP36AX5ZKQW21BNF
WEH8UOAXMH&state=JUByHzYhHRfdf05fWIlky8JElZoDLd
```

[25]:
```
token = oauth.fetch_token(
        'https://us.battle.net/oauth/token',
        authorization_response=authorization_response,
        client_secret=client_secret
)
```

[26]:
```
token
```

[26]:
```
{'access_token': 'USOsT8VUWHXvqTgHakWovttg9gtt4Ji3p7',
 'token_type': 'bearer',
 'expires_in': 86399,
 'expires_at': 1611289648.254414}
```

[27]:
```
import json
```

[31]:
```
a = oauth.get('https://us.api.blizzard.com/hearthstone/cards/?set=standard')
card_dict = json.loads(a.text)
```

[32]:
```
# Dictionary of card names with class mappings

cardNames = dict()
```

```
for page in range(card_dict['pageCount']):

    temp = oauth.get('https://us.api.blizzard.com/hearthstone/cards/?
 ↪set=standard&locale=en_US&page={}'.format(page+1))
    pageCards = json.loads(temp.text)

    for dictionary in pageCards['cards']:

        cardName = dictionary['name']
        if len(dictionary['multiClassIds']) == 0:

            classId = [dictionary['classId']]

        else:

            classId = dictionary['multiClassIds']

        cardNames.update([(cardName,classId)])
```

1.3 Data Preparation

[35]:

```
import numpy as np
```

[36]:

```
# Encoding deck lists as counts of cards

oneHotDeckList = []
oneHotDeckScores = []

for deckList in deckLists:

    index = deckLists.index(deckList)

    oneHotDeck = np.zeros(len(cardNames))

    for i in range(len(cardNames)):

        if list(cardNames.keys())[i] in deckList:

            oneHotDeck[i] = deckList[list(cardNames.keys())[i]]

    unique_elem, elem_counts = np.unique(oneHotDeck,return_counts=True)

    cards = 0

    for card,count in zip(unique_elem,elem_counts):
        cards += card*count

    if cards == 30:

        oneHotDeckScores.append(float(deckScores[index]))

        oneHotDeckList.append(oneHotDeck)
```

[37]:
```
oneHotDeckScoresArr = np.array(oneHotDeckScores).reshape(-1,1)
oneHotDeckScoresArr.shape
```

[37]:
```
(392, 1)
```

[38]:
```
oneHotDeckListsArr = np.vstack(oneHotDeckList)
oneHotDeckListsArr.shape
```

[38]:
```
(392, 1254)
```

1.4 Simple Regression

[6]:
```
import tensorflow as tf
```

[9]:
```
x = tf.placeholder(tf.float32,[None,1254])
```

[10]:
```
y = tf.placeholder(tf.float32,[None,1])
```

[11]:
```
w = tf.Variable(tf.random_normal([1254,1]))
```

[12]:
```
b = tf.Variable(1.0)
```

[13]:
```
xw = tf.matmul(x,w)
y_pred = tf.add(xw,b)
```

[14]:
```
error = tf.reduce_mean(tf.square(y_pred-y))
```

[15]:
```
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
train = optimizer.minimize(error)
```

[16]:
```
init = tf.global_variables_initializer()
```

[17]:
```
saver = tf.train.Saver()
```

[85]:
```
with tf.Session() as sess:

    sess.run(init)

    for i in range(5000):
        sess.run(train,feed_dict={x:oneHotDeckListsArr,y:oneHotDeckScoresArr})

        if i % 100 == 0:

            mse = error.eval(feed_dict={x:oneHotDeckListsArr,y:
↪oneHotDeckScoresArr})
            print(i,"\tMSE",mse)

    W,B = sess.run([w,b])
    score = sess.run(tf.add(tf.matmul(oneHotDeckListsArr[0].astype('float32').
↪reshape(1,1254),W),B))
    saver.save(sess,"./models/simpleRegression/simple_regression.ckpt")
```

```
0       MSE 5561.6646
100     MSE 5113.707
200     MSE 4693.195
300     MSE 4299.14
400     MSE 3930.3481
500     MSE 3585.6719
600     MSE 3264.0107
700     MSE 2964.305
800     MSE 2685.5344
900     MSE 2426.7148
1000    MSE 2186.8965
1100    MSE 1965.1598
1200    MSE 1760.6135
1300    MSE 1572.3937
1400    MSE 1399.6587
1500    MSE 1241.5905
1600    MSE 1097.391
1700    MSE 966.28094
1800    MSE 847.49854
1900    MSE 740.29895
2000    MSE 643.95233
2100    MSE 557.74414
2200    MSE 480.97543
2300    MSE 412.96094
2400    MSE 353.0318
2500    MSE 300.535
2600    MSE 254.83495
2700    MSE 215.31444
2800    MSE 181.37689
2900    MSE 152.44815
3000    MSE 127.97916
3100    MSE 107.44762
3200    MSE 90.36096
3300    MSE 76.258255
```

```
3400     MSE 64.71287
3500     MSE 55.333855
3600     MSE 47.76738
3700     MSE 41.697292
3800     MSE 36.8453
3900     MSE 32.97028
4000     MSE 29.866693
4100     MSE 27.362608
4200     MSE 25.316895
4300     MSE 23.616106
4400     MSE 22.171028
4500     MSE 20.913136
4600     MSE 19.791079
4700     MSE 18.767366
4800     MSE 17.815432
4900     MSE 16.917015
```

[226]:

```
# Prediction by Linear Regressor
score
```

[226]:

```
array([[71.079025]], dtype=float32)
```

[227]:

```
# Actual score
oneHotDeckScoresArr[0]
```

[227]:

```
array([73.3])
```

[19]:

```
# Variables

with tf.Session() as sess:

    saver.restore(sess,"./models/simpleRegression/simple_regression.ckpt")
    W,B = sess.run([w,b])
    print(W,B)
```

```
INFO:tensorflow:Restoring parameters from
./models/simpleRegression/simple_regression.ckpt
[[0.42239386]
 [2.6245205 ]
 [2.014812  ]
 …
 [0.5103437 ]
 [1.136565  ]
 [0.5874951 ]] 3.4050324
```

1.5 Train & Evaluate Models

[75]:
```
# Feature columns

feature_cols = []

for card in cardNames:

    card = card.replace(' ','').replace("'",'').replace(',','').replace('!','').
 ↪replace(':','')
    feature_cols.append(tf.feature_column.numeric_column('{}'.
 ↪format(card),shape=[1]))
```

[39]:
```
numpy_dict = dict()

for i in range(len(cardNames)):

    card = list(cardNames.keys())[i].replace(' ','').replace("'",'').
 ↪replace(',','').replace('!','').replace(':','')
    numpy_dict.update([(card,oneHotDeckListsArr[:,i])])
```

[81]:
```
# Training input function

input_func = tf.estimator.inputs.numpy_input_fn(numpy_dict,np.
 ↪array(oneHotDeckScores),num_epochs=None,batch_size=10,shuffle=True)
```

[ ]:
```
# Evaluation input function

eval_input_func = tf.estimator.inputs.numpy_input_fn(numpy_dict,np.
 ↪array(oneHotDeckScores),num_epochs=35,batch_size=10,shuffle=False)
```

[ ]:
```
# Linear Regressor estimator model

model = tf.estimator.LinearRegressor(feature_columns=feature_cols,model_dir='./
 ↪models/linearRegressor')
```

[ ]:
```
model.train(input_func,steps=1000)
```

[91]:
```
model.evaluate(eval_input_func,steps=1000)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-01-18T21:36:50Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/linearRegressor\model.ckpt-1000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

```
INFO:tensorflow:Evaluation [100/1000]
INFO:tensorflow:Evaluation [200/1000]
INFO:tensorflow:Evaluation [300/1000]
INFO:tensorflow:Evaluation [400/1000]
INFO:tensorflow:Evaluation [500/1000]
INFO:tensorflow:Evaluation [600/1000]
INFO:tensorflow:Evaluation [700/1000]
INFO:tensorflow:Evaluation [800/1000]
INFO:tensorflow:Evaluation [900/1000]
INFO:tensorflow:Evaluation [1000/1000]
INFO:tensorflow:Finished evaluation at 2021-01-18-21:40:06
INFO:tensorflow:Saving dict for global step 1000: average_loss = 8.609376,
global_step = 1000, label/mean = 75.03246, loss = 86.09376, prediction/mean =
74.83777
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1000:
./models/linearRegressor\model.ckpt-1000
```

```
[91]: {'average_loss': 8.609376,
       'label/mean': 75.03246,
       'loss': 86.09376,
       'prediction/mean': 74.83777,
       'global_step': 1000}
```

```
[ ]: # DNN Regressor estimator model

     dnn_model = tf.estimator.
      ↪DNNRegressor(hidden_units=[1254,1254,1254],feature_columns=feature_cols,model_dir='.
      ↪/models/DNNRegressor')
```

```
[ ]: dnn_model.train(input_func,steps=1000)
```

```
[47]: dnn_model.evaluate(eval_input_func,steps=1000)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-01-17T15:52:18Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
C:\Users\Jarel\AppData\Local\Temp\tmptjl0jlds\model.ckpt-1000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [100/1000]
INFO:tensorflow:Evaluation [200/1000]
INFO:tensorflow:Evaluation [300/1000]
INFO:tensorflow:Evaluation [400/1000]
INFO:tensorflow:Evaluation [500/1000]
INFO:tensorflow:Evaluation [600/1000]
INFO:tensorflow:Evaluation [700/1000]
```

```
INFO:tensorflow:Evaluation [800/1000]
INFO:tensorflow:Evaluation [900/1000]
INFO:tensorflow:Evaluation [1000/1000]
INFO:tensorflow:Finished evaluation at 2021-01-17-15:54:26
INFO:tensorflow:Saving dict for global step 1000: average_loss = 3.021388e-05,
global_step = 1000, label/mean = 75.03246, loss = 0.0003021388, prediction/mean
= 75.031525
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1000:
C:\Users\Jarel\AppData\Local\Temp\tmptjl0jlds\model.ckpt-1000
```

[47]:
```
{'average_loss': 3.021388e-05,
 'label/mean': 75.03246,
 'loss': 0.0003021388,
 'prediction/mean': 75.031525,
 'global_step': 1000}
```

1.6 Deck Score Predictions

[92]:
```python
import random
```

[93]:
```python
classMappings =␣
 ↪dict([('demon-hunter',14),('druid',2),('hunter',3),('mage',4),('paladin',5),('priest',6),('
                ('shaman',8),('warlock',9),('warrior',10),('neutral',12)])
```

[95]:
```python
# Option 1:generate random deck

numCards = 0
testDeckList = dict()
testClassID = random.choice(list(classMappings.values()))

while numCards < 30:

    card, classIDs = random.choice(list(cardNames.items()))
    if testClassID in classIDs:

        testDeckList[card] = testDeckList.get(card,0) + 1
        numCards += 1
```

[ ]:
```python
# Option 2:use deck from training data for prediction

testDecks = []

for i in range(len(oneHotDeckList)):

    tempDict = dict()

    for j in range(len(cardNames)):
        tempDict.update([(list(cardNames.keys())[j],oneHotDeckList[i][j])])

    testDecks.append(tempDict)
```

[ ]:

```
# Encoding decklist as card counts

oneHotTestDeck = np.zeros(len(cardNames))

for i in range(len(cardNames)):

    if list(cardNames.keys())[i] in testDecks[0]:

        oneHotTestDeck[i] = testDecks[0][list(cardNames.keys())[i]]

unique_elem, elem_counts = np.unique(oneHotTestDeck,return_counts=True)

cards = 0

for card,count in zip(unique_elem,elem_counts):

    cards += card*count
```

[98]:

```
test_numpy_dict = dict()

for i in range(len(cardNames)):

    card = list(cardNames.keys())[i].replace(' ','').replace("'",'').
 ↪replace(',','').replace('!','').replace(':','')
    test_numpy_dict.update([(card,np.array([oneHotTestDeck[i]]))])
```

[99]:

```
# Prediction input function

pred_input_func = tf.estimator.inputs.
 ↪numpy_input_fn(test_numpy_dict,shuffle=False)
```

[100]:

```
# Prediction by Linear Regressor

list(model.predict(input_fn=pred_input_func))
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/linearRegressor\model.ckpt-1000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

[100]:

```
[{'predictions': array([67.52657], dtype=float32)}]
```

[101]:

```
# Prediction by DNN Regressor (much better performance!)

list(dnn_model.predict(input_fn=pred_input_func))
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from ./models/DNNRegressor\model.ckpt-1000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

[101]:

```
[{'predictions': array([73.30318], dtype=float32)}]
```

[41]:

```
# Actual score

oneHotDeckScoresArr[0]
```

[41]:

```
array([73.3])
```

2 Individual Card Score

2.1 Data Preparation

[131]:

```
runningCardCountsList = []
allChoicesNamesList = []
allChoicesScoreList = []

for dic in link_dict_list:

    pcl = dic['pickedCardsList']

    cards = 0
    for card in pcl:

        if card in list(cardNames.keys()):

            cards += 1

    if cards == 30:

        for i in range(30):

            runningCardList = pcl[:i+1]
            oneHotDeck = np.zeros(len(cardNames))

            for j in range(len(cardNames)):

                if list(cardNames.keys())[j] in runningCardList:
                    oneHotDeck[j] = pcl.count(list(cardNames.keys())[j])

            for k in range(3):

                runningCardCountsList.append(oneHotDeck)

        acn = dic['allChoicesNames']
        allChoicesNamesList.extend(acn)

        acs = dic['allChoicesScores']
        allChoicesScoreList.extend(acs)
```

[132]:

```
runningCardCountsListArr = np.vstack(runningCardCountsList)
runningCardCountsListArr.shape
```

[132]:

```
(35280, 1254)
```

[133]:

```
oneHotNamesList = []

for name in allChoicesNamesList:

    oneHotName = np.zeros(len(cardNames))

    for i in range(len(cardNames)):

        if list(cardNames.keys())[i] == name:

            oneHotName[i] = 1

    oneHotNamesList.append(oneHotName)
```

[134]:

```
oneHotNamesListArr = np.vstack(oneHotNamesList)
oneHotNamesListArr.shape
```

[134]:

```
(35280, 1254)
```

[ ]:

```
allChoicesScoreList = [float(i) for i in allChoicesScoreList]
```

[135]:

```
# Only run when sure output should be saved

%store runningCardCountsList
%store oneHotNamesList
%store allChoicesScoreList
```

```
Stored 'runningCardCountsList' (list)
Stored 'oneHotNamesList' (list)
Stored 'allChoicesScoreList' (list)
```

[40]:
```
# Run to restore output

%store -r runningCardCountsList
%store -r oneHotNamesList
%store -r allChoicesScoreList
```

2.2 Train & Evaluate Models

[156]:
```
# Feature columns

feat_cols = []

for card in cardNames:

    card = card.replace(' ','').replace("'",'').replace(',','').replace('!','').
 ↪replace(':','')
    feat_cols.append(tf.feature_column.numeric_column('{}'.
 ↪format(card),shape=[1]))

feat_cols.append(tf.feature_column.
 ↪categorical_column_with_hash_bucket('name',1254))
```

[ ]:
```
numpy_dict = dict()

for i in range(len(cardNames)):

    card = list(cardNames.keys())[i].replace(' ','').replace("'",'').
 ↪replace(',','').replace('!','').replace(':','')
    numpy_dict.update([(card,runningCardCountsListArr[:,i])])

numpy_dict.update([('name',np.array(allChoicesNamesList))])
```

[197]:
```
# Training input function

train_input_func = tf.estimator.inputs.numpy_input_fn(numpy_dict,np.
 ↪array(allChoicesScoreList),num_epochs=None,batch_size=10,shuffle=True)
```

[ ]:
```
# Evaluation input function

eval_input_func = tf.estimator.inputs.numpy_input_fn(numpy_dict,np.
 ↪array(allChoicesScoreList),num_epochs=1000,batch_size=10,shuffle=False)
```

[ ]:
```
# Linear Regressor estimator model

card_model = tf.estimator.LinearRegressor(feat_cols,model_dir='./models/
 ↪linearRegressor_card')
```

[ ]:

```
card_model.train(train_input_func,steps=1000)
```

[201]:

```
card_model.evaluate(eval_input_func,steps=1000)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-01-19T17:45:26Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/linearRegressor3\model.ckpt-2000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [100/1000]
INFO:tensorflow:Evaluation [200/1000]
INFO:tensorflow:Evaluation [300/1000]
INFO:tensorflow:Evaluation [400/1000]
INFO:tensorflow:Evaluation [500/1000]
INFO:tensorflow:Evaluation [600/1000]
INFO:tensorflow:Evaluation [700/1000]
INFO:tensorflow:Evaluation [800/1000]
INFO:tensorflow:Evaluation [900/1000]
INFO:tensorflow:Evaluation [1000/1000]
INFO:tensorflow:Finished evaluation at 2021-01-19-17:48:51
INFO:tensorflow:Saving dict for global step 2000: average_loss = 1083.2366,
global_step = 2000, label/mean = 61.46461, loss = 10832.366, prediction/mean =
45.357845
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 2000:
./models/linearRegressor3\model.ckpt-2000
```

[201]:

```
{'average_loss': 1083.2366,
 'label/mean': 61.46461,
 'loss': 10832.366,
 'prediction/mean': 45.357845,
 'global_step': 2000}
```

[218]:

```
# Updating feature columns for DNN regressor model

feat_cols.pop()

cat = tf.feature_column.categorical_column_with_hash_bucket('name',1254)
feat_cols.append(tf.feature_column.embedding_column(cat,1254))
```

[218]:

```
1255
```

[ ]:

```
# DNN Regressor estimator model
```

```
card_dnn_model = tf.estimator.
 ↪DNNRegressor([1255,1255,1255],feat_cols,model_dir='./models/
 ↪DNNRegressor_card')
```

[ ]:

```
card_dnn_model.train(train_input_func,steps=1000)
```

[221]:

```
card_dnn_model.evaluate(eval_input_func,steps=1000)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Starting evaluation at 2021-01-19T18:13:15Z
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/DNNRegressor_card\model.ckpt-1000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Evaluation [100/1000]
INFO:tensorflow:Evaluation [200/1000]
INFO:tensorflow:Evaluation [300/1000]
INFO:tensorflow:Evaluation [400/1000]
INFO:tensorflow:Evaluation [500/1000]
INFO:tensorflow:Evaluation [600/1000]
INFO:tensorflow:Evaluation [700/1000]
INFO:tensorflow:Evaluation [800/1000]
INFO:tensorflow:Evaluation [900/1000]
INFO:tensorflow:Evaluation [1000/1000]
INFO:tensorflow:Finished evaluation at 2021-01-19-18:15:30
INFO:tensorflow:Saving dict for global step 1000: average_loss = 315.33557,
global_step = 1000, label/mean = 61.46461, loss = 3153.3557, prediction/mean =
57.02432
INFO:tensorflow:Saving 'checkpoint_path' summary for global step 1000:
./models/DNNRegressor_card\model.ckpt-1000
```

[221]:

```
{'average_loss': 315.33557,
 'label/mean': 61.46461,
 'loss': 3153.3557,
 'prediction/mean': 57.02432,
 'global_step': 1000}
```

2.3 Card Score Predictions

[173]:

```
# Ooption 1:generate random (possibly partially-filled) deck

testDeckList = dict()

numCardsChosen = random.randint(0,29)
testClassID = random.choice(list(classMappings.values()))
```

```
while numCardsChosen > 0:

    card, classIDs = random.choice(list(cardNames.items()))
    if testClassID in classIDs:

        testDeckList[card] = testDeckList.get(card,0) + 1
        numCardsChosen -= 1

testDeckList
```

[173]:

```
{'Plagiarize': 1,
 'Candle Breath': 1,
 'Tak Nozwhisker': 1,
 'Cold Blood': 1,
 'Flik Skyshiv': 1,
 'Steeldancer': 1,
 "Togwaggle's Scheme": 1,
 'Plaguebringer': 1}
```

[202]:

```
# Option 2:use deck from training data for prediction

index = random.randint(0,11759)

cardCount = np.vstack(runningCardCountsList[index*3:index*3+3])
threeOneHotNames = allChoicesNamesList[index*3:index*3+3]

threeScores = allChoicesScoreList[index*3:index*3+3]
```

[ ]:

```
test_numpy_dict = dict()

for i in range(len(cardNames)):

    card = list(cardNames.keys())[i].replace(' ','').replace("'",'').
 ↪replace(',','').replace('!','').replace(':','')
    test_numpy_dict.update([(card,cardCount[:,i])])

test_numpy_dict.update([('name',np.array(threeOneHotNames))])
```

[206]:

```
# Prediction input function

pred_input_func = tf.estimator.inputs.
 ↪numpy_input_fn(test_numpy_dict,shuffle=False)
```

[223]:

```
# Actual scores

print(threeOneHotNames) ; print(threeScores)
```

```
['Air Raid', 'Guardian Augmerchant', 'Holy Light']
```

```
[64.56, 64.69, 15.85]
```

[209]:

```python
# Prediction by Linear Regressor (rather poor performance)

list(card_model.predict(pred_input_func))
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/linearRegressor_card\model.ckpt-2000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

[209]:

```
[{'predictions': array([79.04803], dtype=float32)},
 {'predictions': array([82.29574], dtype=float32)},
 {'predictions': array([77.59356], dtype=float32)}]
```

[225]:

```python
# Prediction by DNN Regressor (much better performance!)

list(card_dnn_model.predict(pred_input_func))
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from
./models/DNNRegressor_card\model.ckpt-2000
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
```

[225]:

```
[{'predictions': array([67.01109], dtype=float32)},
 {'predictions': array([82.476265], dtype=float32)},
 {'predictions': array([17.470245], dtype=float32)}]
```