x20122136_ResearchProject_HKDataset_Part_1

Table of Contents

1. Objective

In this notebook applied deep learning algorithm on HK dataset to extend its accuracy using ANN.

2. Import packages

Here, we import common packages for deep learning.

- **pandas: for data reading and preprocessing**
- **tensorflow: for neural network construction**
- **sklearn.preprocessing: for data encoding**
- **sklearn.model_selection: it has convenient method for training/test data spliting**
- **matplotlib.pyplot: to plot performance of the training process.**
- **pandas_profiling: to perform EDA**

In [ ]:

```
!pip install pandas
!pip install numpy
!pip install tensorflow
!pip install scikit-learn
!pip install matplotlib
!pip install pandas-profiling
```

In [ ]:

```
import pandas as pd
import numpy as np
import tensorflow as tf
```

```
import sklearn.preprocessing as preprocessing
import sklearn.model_selection as model_selection
import matplotlib.pyplot as plt
from pandas_profiling import ProfileReport
```

3. Loading Runs and Races CSV's

Read CSV's nd run EDA on them to understand data

In [ ]:

```
races_df = pd.read_csv(r"races.csv", delimiter=",", header=0, index_col='race_id')
runs_df = pd.read_csv(r"runs.csv", delimiter=",", header=0)
```

4. EDA

4.1. EDA report on reaces Data

In [ ]:

```
# races_profile = ProfileReport(races_df, title="Pandas Profiling Races Report")

# races_profile.to_file("/kaggle/working/eda_races.html")

# races_profile.to_file("/kaggle/working/eda_races.json")
```

4.2 EDA report on runs Data

In [ ]:

```
# runs_profile = ProfileReport(runs_df, title="Pandas Profiling Races Report")

# runs_profile.to_file("/kaggle/working/eda_runs.html")

# runs_profile.to_file("/kaggle/working/eda_runs.json")
```

5. Data Pre-Processing

I found it makes more sense to predict winner horse for every race because winning is relative to other horses performance rather predict every single horse run.

5.1. Prepare races data from races.csv

Only select several columns that make sense for this kernel. Then, use different encoders for different types of attribute.

In [ ]:

```
races_df = races_df[['venue', 'config', 'surface', 'distance', 'going', 'race_class']]

# check to see if we have NaN, then drop NaN
print(races_df[races_df.isnull().any(axis=1)])
races_df = races_df.dropna()

print(races_df.shape)
print(races_df.head())
```

In [ ]:

```
# encode ordinal columns: config, going,
config_encoder = preprocessing.OrdinalEncoder()
races_df['config'] = config_encoder.fit_transform(races_df['config'].values.reshape(-1, 1
))
going_encoder = preprocessing.OrdinalEncoder()
races_df['going'] = going_encoder.fit_transform(races_df['going'].values.reshape(-1, 1))

# encode nominal column: venue
venue_encoder = preprocessing.LabelEncoder()
races_df['venue'] = venue_encoder.fit_transform(races_df['venue'])

print(races_df.dtypes)
print(races_df.shape)
print(races_df.head())
```

5.2. Prepare races data from runs.csv

Similar to races data, only select columns that are relevant to the model.

Data cleaning

- **two rows that includes NaN, so just drop them.**
- **strange data for 'draw', e.g. 15. As we only deal with standard 14 horses racing, so let's drop it.**

Encoding

Then, use label encoders for 'horse_country' and 'horse_type'.

In [ ]:

```
runs_df = runs_df[['race_id', 'draw',
                   'horse_age', 'horse_country', 'horse_type', 'horse_rating', 'declared
_weight', 'actual_weight', 'win_odds',
                   'result']]

# check to see if we have NaN, then drop NaN
print(runs_df[runs_df.isnull().any(axis=1)])
runs_df = runs_df.dropna()

# not sure why, but we got some strange draw in the dataset. Maximum shall be 14
strange_draw_index = runs_df[runs_df['draw'] > 14].index
# delete these row indexes from dataFrame
runs_df = runs_df.drop(strange_draw_index)

# encode nominal columns: horse_country, horse_type
horse_country_encoder = preprocessing.LabelEncoder()
runs_df['horse_country'] = horse_country_encoder.fit_transform(runs_df['horse_country'])
horse_type_encoder = preprocessing.LabelEncoder()
runs_df['horse_type'] = horse_type_encoder.fit_transform(runs_df['horse_type'])

print(runs_df.dtypes)
print(runs_df.shape)
print(runs_df.head())
```

5.3. Further preprocessing for runs data

We are targeting to put all the 14 horses' features into the one input, but it expands into multiple rows now. Luckily, pandas has a nice method called `pivot`. `pivot` aggregates horses data from multiple rows, which belongs to a single race, into one row.

After `pivot`, some races may not have 14 horses, so let's fill NaN with 0.

In [ ]:

```
def group_horse_and_result(element):
    if element[0] == 'result':
```

```
        return 100 + element[1] # to make sure results are put near the end
    else:
        return element[1]

runs_df = runs_df.pivot(index='race_id', columns='draw', values=runs_df.columns[2:])
rearranged_columns = sorted(list(runs_df.columns.values), key=group_horse_and_result)
runs_df = runs_df[rearranged_columns]
print(runs_df.head())

# quite some NaNs appreared in the dataframe, reason is some races didnt have full 14 hor
ses participating
# fill with 0
runs_df = runs_df.fillna(0)
```

6. Prepare training and test data

Here, we combine races data and runs data by `join` two data frames above.

Standardization

If you look at the data closely, if will find that features are not in the same scale, e.g. weight can go to 1000+. Standardize the data for to make training easier.

Select right columns for X, y

- **Select all the data except last 28 columns, because last 28 columns is about 'result' and 'won'**
- **Select last 14 columns for y_won. Each row shall have one '1.0' and rest are 0.**
- **Select second last 14 columns for y_top3. It used to the the column 'result', e.g. 1~14, which is horses' final positions when the race finishes. Apply a function to convert it to 1.0 if the horse is in top 3, else 0.**

Split data into train/test sets

sklearn comes with such a handy method `train_test_split`. We split the data as following:

- **80% for training**
- **20% for testing(validation)**

In [ ]:

```
data = races_df.join(runs_df, on='race_id', how='right')
X = data[data.columns[:-14]]
ss = preprocessing.StandardScaler()
X = pd.DataFrame(ss.fit_transform(X),columns = X.columns)

y_won = data[data.columns[-14:]].applymap(lambda x: 1.0 if 0.5 < x < 1.5 else 0.0)

print(X.shape)
print(y_won.shape)

# split data into train and test sets
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y_won, train_size
=0.8, test_size=0.2, random_state=1)
print('X_train', X_train.shape)
print('y_train', y_train.shape)
print('X_test', X_test.shape)
print('y_test', y_test.shape)
```

7. ANN algorithm

7.1. Build the model

Use keras to build the model with easy-to-use api `Sequential`.

Have to mention that input layer has 104 inputs. The calculation is following:

- **6 features from races dataframe: 'venue', 'config', 'surface', 'distance', 'going', 'race_class'**
- **14 horses per races, and each horse has 7 features; 'horse_age', 'horse_country', 'horse_type', 'horse_rating', 'declared_weight', 'actual_weight', 'win_odds'**
- **so total 104 features = 6 + 14 x 7**

Output layer has 14 nodes, as each node stands for each horse's result.

In [ ]:

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(96, activation='relu', input_shape=(104,)),
    tf.keras.layers.Dense(14, activation='softmax')
])
model.compile(optimizer=tf.keras.optimizers.Adam(5e-04),
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=[tf.keras.metrics.Precision(name='precision')])
```

7.2. Train the model

In [ ]:

```
dataset = tf.data.Dataset.from_tensor_slices((X_train.values, y_train.values))
train_dataset = dataset.shuffle(len(X_train)).batch(500)
dataset = tf.data.Dataset.from_tensor_slices((X_test.values, y_test.values))
validation_dataset = dataset.shuffle(len(X_test)).batch(500)

print("Start training..\n")
history = model.fit(train_dataset, epochs=200, validation_data=validation_dataset)
print("Done.")
```

8. Visualization

Plot the result

In [ ]:

```
precision = history.history['precision']
val_precision = history.history['val_precision']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(precision) + 1)

plt.plot(epochs, precision, 'b', label='Training precision')
plt.plot(epochs, val_precision, 'r', label='Validation precision')
plt.title('Training and validation precision')
plt.legend()
plt.figure()

plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

9. Conclusion

With the 2 layer nerual network, we reached 0.92 precision on the the training dataset. However, best precision on the testing dataset was about 0.3, which happened around epoch 70~80. Then overfitting happened.

precision = 0.3, means If we bet 'Win' 10 times based on the model's prediction, only 3 times is correct.