AI Cure

-REPORT

By Team Pandas | 26th January, 2024

Contents

Aim

The aim is to construct an advanced model capable of accurately predicting an individual's heart rate using AI-ML. This report is based on the aicure_Pandas.ipynb notebook which can be found at https://github.com/prakriti16/aicure_Pandas.

Candidate/Alternate Models

Based on the data dictionary, there are a lot of features related to the RR intervals, which are the time intervals between consecutive heartbeats. These features can capture the variability and complexity of the heart rate, which are influenced by various factors such as physical activity, stress, emotions, and health conditions. Therefore, a model that can handle the nonlinearity of the heart rate time series is required.

One possible approach is to use a recurrent neural network (RNN), which is a type of deep learning model that can process sequential data and learn from the temporal dependencies. However, RNNs also have some limitations, such as the vanishing gradient problem, which makes it difficult to learn long-term dependencies.

To overcome this problem, a variant of RNN called long short-term memory (LSTM) can be used as it is designed to remember and forget information over long periods of time. LSTM has a special structure that consists of three gates: input gate, forget gate, and output gate. These gates control the flow of information and allow the LSTM to learn what to keep and what to discard from the previous states.

Another possible approach is to use a convolutional neural network (CNN), which is another type of deep learning model that can extract features from spatial data such as images and videos.

CODE SNIPPET

```
# Import the libraries
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Conv1D,
MaxPooling1D, Flatten
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Load the data
data = pd.read_csv("train_data.csv")
```

```
# Split the data into features and target
X = data.drop(["uuid", "datasetId", "condition", "HR"], axis=1) #
Features
y = data["HR"] # Target

# Normalize the features
X = (X - X.mean()) / X.std()

# Reshape the features for LSTM or CNN input
X = np.array(X).reshape(X.shape[0], X.shape[1], 1)

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# LSTM model
model = Sequential()
model.add(LSTM(64, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(32, activation="relu"))
model.add(Dense(1))

# CNN model
model = Sequential()
model.add(Conv1D(32, 3, activation="relu", input_shape=(X.shape[1],
X.shape[2])))
model.add(MaxPooling1D(2))
model.add(Flatten())
model.add(Dense(32, activation="relu"))
model.add(Dense(1))

# Compile the model
model.compile(loss="mse", optimizer=Adam(learning_rate=0.001),
metrics=["mae"])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2)

# Evaluate the model on the test set
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = tf.keras.losses.MeanAbsoluteError()(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
print(f"MSE: {mse:.2f}, MAE: {mae:.2f}, R2: {r2:.2f}")
```

The code uses TensorFlow, a popular deep learning framework, to build and train a LSTM or CNN model for heart rate prediction.

Firstly, the necessary libraries, such as NumPy, Pandas, TensorFlow, and SKLearn are imported.

The data is loaded from a csv file and splits it into features and target. The features are the variables that describe the RR intervals and other physiological signals, and the target is the heart rate.

The features are normalized by subtracting the mean and dividing by the standard deviation, to make the data more suitable for the neural network.

The features are reshaped into a three-dimensional array as the LSTM or CNN input expects a sequence of vectors.

The data is split into train and test sets, using 80% of it for training and 20% for testing.

Either the LSTM or the CNN model should be chosen and the code for the other model should be commented out. A LSTM model consists of a LSTM layer, which can process the sequential data and learn from the temporal dependencies, followed by two dense layers, which are fully connected layers that perform the regression task.

Whereas, the CNN model consists of a convolutional layer, which can extract local features from the spatial data, followed by a max pooling layer, which reduces the dimensionality and noise, and then a flatten layer, which converts the two-dimensional output into a one-dimensional vector, and finally two dense layers, which perform the regression task.

The model is compiled, specifying the loss function, the optimizer, and the metrics. The loss function is the mean squared error, which measures the difference between the predicted and actual heart rate. The optimizer is the Adam algorithm, which is a gradient-based optimization method that adapts the learning rate dynamically. The metric is the mean absolute error, which measures the average absolute difference between the predicted and actual heart rate.

The code then trains the model, using 10 epochs, which are the number of times the model sees the entire training data, and 32 batch size, which are the number of samples the model sees at each iteration. The code also uses 20% of the training data for validation, which is used to monitor the model performance and prevent overfitting.

The model is evaluated on the test set, using the predict method to generate the heart rate predictions. The mean squared error, the mean absolute error, and the R2 score, which are the metrics that measure the model performance are printed.

RESULTS:

LSTM: `MSE: 101.56, MAE: 8.30, R2: 0.06`

CNN: `MSE: 21.13, MAE: 11.94, R2: 0.80`

From the results we can see that the CNN model outperforms the LSTM model. However, it is worse than the final XGBoost model.

Final Model

The final model uses XGBoost, a popular machine learning library, to build and train a regression model for heart rate prediction.

Some of the advantages of XGBoost over TensorFlow are as follows:

XGBoost is a gradient boosting framework that primarily focuses on decision trees, which are simple and interpretable models that can handle both linear and nonlinear relationships.

TensorFlow is a deep learning framework that specializes in building and training neural networks, which are complex and black-box models that require more data and computation to achieve good results.

XGBoost employs a boosting algorithm that combines multiple weak models to create a strong model. It adds new models iteratively, with each subsequent model attempting to correct the mistakes made by the previous models.

This makes XGBoost more robust to overfitting and noise than TensorFlow, which performs training using gradient descent optimization algorithms that update the model's weights iteratively to minimize the loss function.

In summary, the advantages of XGBoost over TensorFlow lie in its simplicity, robustness, efficiency, and interpretability for heart rate prediction using tabular data. However, this does not mean that TensorFlow is inferior to XGBoost, as it may perform better on other tasks that require more complex and diverse data types, such as image recognition or natural language processing.

CODE EXPLANATION:

```
# Import necessary libraries
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score

# Load the training data
train_data = pd.read_csv('train_data.csv')

# Drop columns not needed for training (e.g., uuid, datasetId,
condition)
train_data = train_data.drop(['uuid', 'datasetId', 'condition'],
axis=1)

# Separate features and target variable
X = train_data.drop('HR', axis=1)
y = train_data['HR']

# Split the data into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

# Train the XGBoost model
model = XGBRegressor()
model.fit(X_train_scaled, y_train)

# Predict on the validation set
y_pred = model.predict(X_valid_scaled)

# Evaluate the model
rmse = np.sqrt(mean_squared_error(y_valid, y_pred))
mae = mean_absolute_error(y_valid, y_pred)
r2 = r2_score(y_valid, y_pred)
print(f"RMSE: {rmse:.2f}, MAE: {mae:.2f}, R2: {r2:.2f}")

# Save the trained model for later use
model.save_model('heart_rate_model.model')
```

The XGBRegressor class is imported from the xgboost module.

The csv file contains the training data for the heart rate prediction task, which has various features related to the RR intervals and other physiological signals, and the target variable, which is the heart rate. We use the drop method of the DataFrame object to drop the columns that are either not numerical (condition) or not needed (uuid, datasetId). The columns not relevant for predicting the heart rate may introduce errors, noise or bias to the model so we drop them.

The column containing the target variable (HR). The resulting DataFrame object, named X, contains only the features that describe the RR intervals and other physiological signals. The axis=1 argument specifies that the column is dropped, not the row.

80% of the data is used to train the model and the remaining 20% is used to evaluate the model and tune the hyperparameters. The random_state=42 argument specifies a seed for the random number generator, which ensures that the data is split in the same way every time the code is run. The function returns four objects: X_train, which is the DataFrame containing the features for the training set, X_valid, which is the DataFrame containing the features for the validation set, y_train, which is the Series containing the target variable for the training set, and y_valid, which is the Series containing the target variable for the validation set.

The fit method of the model object is used to train the XGBoost model on the training set. The method takes two arguments: X_train_scaled, which is the array containing the standardized features for the training set, and y_train, which is the Series containing the target variable for the training set. The method returns the trained model object, which can be used for prediction and evaluation.

predict method of the model object is used to generate predictions for the validation set. The method takes one argument: X_valid_scaled, which is the array containing the standardized features for the validation set. The method returns an array object, named y_pred, which contains the predicted heart rate values for the validation set.

RESULTS:

```
RMSE: 0.41, MAE: 0.21, R2: 1.00
```

To evaluate the model, different metrics that measure the accuracy and error of our predictions, were used:

Root mean squared error (RMSE), which is the square root of the average of the squared differences between the predicted and actual values. A lower RMSE indicates a better fit of

the model. We have an RMSE of 0.41 and considering the data values are between 50-100 the model has an error of just 0.82 percent on the training data!

Mean absolute error (MAE), which is the average of the absolute differences between the predicted and actual values. A lower MAE indicates a better fit of the model. We have an MAE of 0.21 or just 0.42 percent!

R-squared (R2), which is the proportion of the variance in the target variable that is explained by the model. A higher R2 indicates a better fit of the model. Our R2 value is 100%, thus the variables are perfectly correlated!

TESTING THE MODEL

```
# Load the test data
test_data = pd.read_csv('sample_test_data.csv')

# Drop columns not needed for prediction (e.g., datasetId,
condition)
test_data = test_data.drop(['datasetId', 'condition'], axis=1)

# Extract the 'uuid' column for later inclusion in the results
uuid_column = test_data['uuid']

# Drop 'uuid' column as it's not needed for prediction
test_data = test_data.drop(['uuid'], axis=1)

# Load the trained model
model = XGBRegressor()
model.load_model('heart_rate_model.model')

# Standardize features
scaler = StandardScaler()

# Exclude non-numeric columns before scaling
numeric_columns = test_data.select_dtypes(include=['float64',
'int64']).columns
test_data[numeric_columns] =
scaler.fit_transform(test_data[numeric_columns])

# Predict on the test set
predictions = model.predict(test_data)

# Create a DataFrame with 'uuid' and predicted heart rates
result_df = pd.DataFrame({'uuid': uuid_column, 'Predicted_HR':
predictions})
```

```
# Save the predictions to results.csv
result_df.to_csv('results.csv', index=False)
```

The read_csv method of the pandas module is used to read the data from a csv file that contains the test data for the heart rate prediction task.

An instance of the XGBRegressor model is initialized and the load_model method of the model object is used to load the trained model from a file named heart_rate_model.model. The file was saved previously using the save_model method of the model object.

The predict method of the model object is used to generate predictions for the test set. A new DataFrame object, named result_df, that contains the uuid (unique identifier) and the predicted heart rate values for each patient is created.

The predictions are saved to a csv file named results.csv.

SAMPLE TEST RESULT:

Comparing the HR column in results.csv and the sample_output_generated.csv it can be concluded that the RMSE on the testing data set is around 13.7%.

uuid	HR	Predicted_HR
1ae30e0b-098e-46fc-a897-0a6661f26370	75.20605	84.88413
428b41b3-9461-4c79-ab4e-d03b122b2553	80.87013	90.85277
88f82ac7-02dd-447e-a289-22e8e22432c2	62.31306	68.22056
1d09b18f-d82f-4c1a-bb2d-71fda6fea837	66.33692	74.494804
a6302640-f70a-4a3a-ad36-a8c3d5df9400	64.4226	71.14344
3f6508be-4b0a-4008-b701-49d8c2d5dd43	56.06109	59.71483
a07d84c8-fc44-45ef-bb85-f06f06b70e9f	75.54367	84.0379
f4a449db-a7ff-437b-852b-821a6e965f2f	62.45828	68.14893
94364ef1-12e2-4ddd-9f35-99e270547849	56.27188	60.131626
231d34f5-1028-4f2e-8e1d-00d086b0c218	71.2015	78.70497

Further Enhancement

The model can be further enhanced through hyperparameter tuning.

CODE SNIPPET

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from xgboost import XGBRegressor
from sklearn.metrics import mean_squared_error

# Load the training data
train_data = pd.read_csv('train_data.csv')

# Drop columns not needed for training (e.g., uuid, datasetId,
condition)
train_data = train_data.drop(['uuid', 'datasetId', 'condition'],
axis=1)

# Separate features and target variable
X = train_data.drop('HR', axis=1)
y = train_data['HR']

# Set a fixed value for the test size
test_size = 0.2

# Split the data into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(X, y,
test_size=test_size, random_state=42)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

# Define the hyperparameters to tune
params = {
    'max_depth': [3, 5, 7],
    'n_estimators': [100, 200, 300],
```

```
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.5, 0.8, 1.0],
    'colsample_bytree': [0.5, 0.8, 1.0],
    'reg_lambda': [0.1, 1.0, 10.0]
}

# Initialize the XGBoost model with the gpu_hist tree method
model = XGBRegressor(tree_method='hist')

# Perform grid search to find the best parameters
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(model, params,
scoring='neg_root_mean_squared_error', cv=5, verbose=1)
grid.fit(X_train_scaled, y_train, eval_set=[(X_valid_scaled,
y_valid)], early_stopping_rounds=10)

# Print the best parameters and score
print(grid.best_params_)
print(grid.best_score_)

# Save the best model for later use
grid.best_estimator_.save_model('heart_rate_model.model')

# Load the test data
test_data = pd.read_csv(sys.argv[1])

# Drop columns not needed for prediction (e.g., datasetId,
condition)
test_data = test_data.drop(['datasetId', 'condition'], axis=1)

# Extract the 'uuid' column for later inclusion in the results
uuid_column = test_data['uuid']

# Drop 'uuid' column as it's not needed for prediction
test_data = test_data.drop(['uuid'], axis=1)

# Load the trained model
model = XGBRegressor()
model.load_model('heart_rate_model.model')

# Standardize features
scaler = StandardScaler()

# Exclude non-numeric columns before scaling
```

```
numeric_columns = test_data.select_dtypes(include=['float64',
'int64']).columns
test_data[numeric_columns] =
scaler.fit_transform(test_data[numeric_columns])

# Predict on the test set
predictions = model.predict(test_data)

# Create a DataFrame with 'uuid' and predicted heart rates
result_df = pd.DataFrame({'uuid': uuid_column, 'Predicted_HR':
predictions})

# Save the predictions to results.csv
result_df.to_csv('results.csv', index=False)
```

Hyperparameters are the parameters that control the behavior and performance of the model. The code uses grid search which tries all possible combinations to arrive at the optimum parameters required for best results.

However, it takes hours to execute this using CPU. If we could access GPU computing the model could be further enhanced and modified to give more accurate predictions.