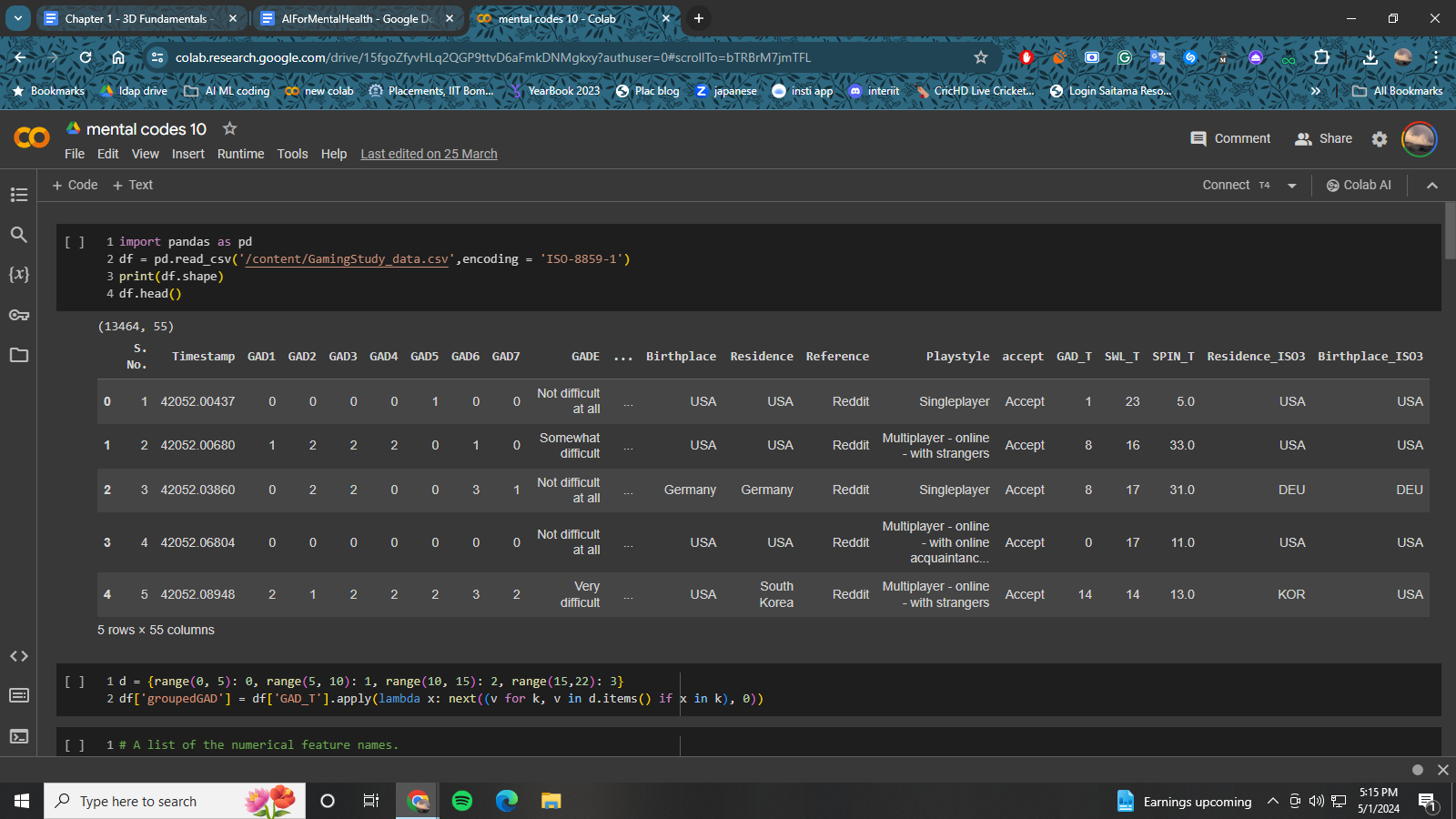
# Risk prediction model, anxiety stage prediction

Anxiety is very common nowadays so in this part we try to build a model which predicts the level of anxiety based on the psychometric tests conducted without any doctoral supervision. We use the Generalised Anxiety Disorder(GAD) [dataset](https://www.kaggle.com/datasets/divyansh22/online-gaming-anxiety-data) based on proper official test questionnaire given to the subjects.



First we begin by loading the data and separating it into categorical and continuous features. Categorical feature are those which have classes or which are quantised whereas continuous features are not quantised as the name suggests. We convert the categorical features to integers using the labelencoder from sklearn. We create the target label ‘GroupedGAD’ which basically divides total GAD score into 4 categories 0-5, 5-10, 10-15, 15-22 which are the actual categories prescribed in the test and using which the doctors assess the patient’s anxiety level. This is what our model will finally predict.

For this task, we will use the TabTransformer model which is a very popular and widely used tabular model which has the power and prowess to take into account the tabular relations and the connections between various features. We implement this model from scratch so it can be a little difficult to follow in between but make sure to read the comments

## CODE:

## Installation

Pip install pandas keras tensorflow sklearn functools numpy matplotlib

## Imports

Import pandas as pd

import keras

from keras import layers

from keras import ops

import math

import numpy as np

from tensorflow import data as tf\_data

import matplotlib.pyplot as plt

from functools import partial

from sklearn.preprocessing import LabelEncoder

## Data Loading

df = pd.read\_csv('/content/GamingStudy\_data.csv',encoding = 'ISO-8859-1')

## Data processing

d = {range(0, 5): 0, range(5, 10): 1, range(10, 15): 2, range(15,22): 3}

df['groupedGAD'] = df['GAD\_T'].apply(lambda x: next((v for k, v in d.items() if x in k), 0))

NUMERIC\_FEATURE\_NAMES = df.columns[df.dtypes=='int64'].tolist() + df.columns[df.dtypes=='float64'].tolist()

NUMERIC\_FEATURE\_NAMES.remove('groupedGAD')

cat\_cols = df.columns[df.dtypes=='object'].tolist()

for col in df.columns[df.dtypes == object]:

df[col] = df[col].fillna("NA")

l\_enc = LabelEncoder()

df[col] = l\_enc.fit\_transform(df[col].values)

for col in df.columns[df.dtypes == 'float64']:

df.fillna(df.loc[:, col].mean(), inplace=True)

for col in df.columns[df.dtypes == 'int64']:

df.fillna(df.loc[:, col].min(), inplace=True)

# A dictionary of the categorical features and their vocabulary.

CATEGORICAL\_FEATURES\_WITH\_VOCABULARY = {}

for col in cat\_cols:

CATEGORICAL\_FEATURES\_WITH\_VOCABULARY[col] = sorted(list(df[col].unique()))

CATEGORICAL\_FEATURE\_NAMES = list(CATEGORICAL\_FEATURES\_WITH\_VOCABULARY.keys())

# A list of all the input features.

FEATURE\_NAMES = NUMERIC\_FEATURE\_NAMES + CATEGORICAL\_FEATURE\_NAMES

# The name of the target feature.

TARGET\_FEATURE\_NAME = "groupedGAD"

# A list of the labels of the target features.

TARGET\_LABELS = [0,1,2,3]

## Hyperparameters

LEARNING\_RATE = 0.001

WEIGHT\_DECAY = 0.0001

DROPOUT\_RATE = 0.2

BATCH\_SIZE = 265

NUM\_EPOCHS = 15

NUM\_TRANSFORMER\_BLOCKS = 3 # Number of transformer blocks.

NUM\_HEADS = 4 # Number of attention heads.

EMBEDDING\_DIMS = 16 # Embedding dimensions of the categorical features.

MLP\_HIDDEN\_UNITS\_FACTORS = [2,1,

] # MLP hidden layer units, as factors of the number of inputs.

NUM\_MLP\_BLOCKS = 2 # Number of MLP blocks in the baseline model.

## Model building

def create\_model\_inputs():

inputs = {}

for feature\_name in FEATURE\_NAMES:

if feature\_name in NUMERIC\_FEATURE\_NAMES:

inputs[feature\_name] = layers.Input(

name=feature\_name, shape=(), dtype="float32")

else:

inputs[feature\_name] = layers.Input(

name=feature\_name, shape=(), dtype="float32")

return inputs

def encode\_inputs(inputs, embedding\_dims):

encoded\_categorical\_feature\_list = []

numerical\_feature\_list = []

for feature\_name in inputs:

if feature\_name in CATEGORICAL\_FEATURE\_NAMES:

vocabulary = CATEGORICAL\_FEATURES\_WITH\_VOCABULARY[feature\_name]

embedding = layers.Embedding(

input\_dim=len(vocabulary), output\_dim=embedding\_dims)

# Convert the index values to embedding representations.

encoded\_categorical\_feature = embedding(inputs[feature\_name])

encoded\_categorical\_feature\_list.append(encoded\_categorical\_feature)

else:

# Use the numerical features as-is.

numerical\_feature = ops.expand\_dims(inputs[feature\_name], -1)

numerical\_feature\_list.append(numerical\_feature)

return encoded\_categorical\_feature\_list, numerical\_feature\_list

def create\_mlp(hidden\_units, dropout\_rate, activation, normalization\_layer, name=None):

mlp\_layers = []

for units in hidden\_units:

mlp\_layers.append(normalization\_layer()),

mlp\_layers.append(layers.Dense(units, activation=activation))

mlp\_layers.append(layers.Dropout(dropout\_rate))

return keras.Sequential(mlp\_layers, name=name)

def create\_tabtransformer\_classifier(

num\_transformer\_blocks, num\_heads,embedding\_dims,

mlp\_hidden\_units\_factors, dropout\_rate,

use\_column\_embedding=False,):

# Create model inputs.

inputs = create\_model\_inputs()

# encode features.

encoded\_categorical\_feature\_list, numerical\_feature\_list = encode\_inputs(inputs, embedding\_dims)

print(encoded\_categorical\_feature\_list)

# Stack categorical feature embeddings for the Tansformer.

encoded\_categorical\_features = ops.stack(encoded\_categorical\_feature\_list, axis=1)

# Concatenate numerical features.

numerical\_features = layers.concatenate(numerical\_feature\_list)

# Add column embedding to categorical feature embeddings.

if use\_column\_embedding:

num\_columns = encoded\_categorical\_features.shape[1]

column\_embedding = layers.Embedding(

input\_dim=num\_columns, output\_dim=embedding\_dims)

column\_indices = ops.arange(start=0, stop=num\_columns, step=1)

encoded\_categorical\_features = encoded\_categorical\_features + column\_embedding(

column\_indices)

# Create multiple layers of the Transformer block.

for block\_idx in range(num\_transformer\_blocks):

# Create a multi-head attention layer.

attention\_output = layers.MultiHeadAttention(

num\_heads=num\_heads,key\_dim=embedding\_dims,

dropout=dropout\_rate,

name=f"multihead\_attention\_{block\_idx}",

)(encoded\_categorical\_features, encoded\_categorical\_features)

# Skip connection 1.

x = layers.Add(name=f"skip\_connection1\_{block\_idx}")(

[attention\_output, encoded\_categorical\_features])

# Layer normalization 1.

x = layers.LayerNormalization(name=f"layer\_norm1\_{block\_idx}", epsilon=1e-6)(x)

# Feedforward.

feedforward\_output = create\_mlp(

hidden\_units=[embedding\_dims],

dropout\_rate=dropout\_rate,

activation=keras.activations.gelu,

normalization\_layer=partial(

layers.LayerNormalization, epsilon=1e-6

), # using partial to provide keyword arguments before initialization

name=f"feedforward\_{block\_idx}")(x)

# Skip connection 2.

x = layers.Add(name=f"skip\_connection2\_{block\_idx}")([feedforward\_output, x])

# Layer normalization 2.

encoded\_categorical\_features = layers.LayerNormalization(

name=f"layer\_norm2\_{block\_idx}", epsilon=1e-6

)(x)

# Flatten the "contextualized" embeddings of the categorical features.

categorical\_features = layers.Flatten()(encoded\_categorical\_features)

# Apply layer normalization to the numerical features.

numerical\_features = layers.LayerNormalization(epsilon=1e-6)(numerical\_features)

# Prepare the input for the final MLP block.

features = layers.concatenate([categorical\_features, numerical\_features])

# Compute MLP hidden\_units.

mlp\_hidden\_units = [

factor \* features.shape[-1] for factor in mlp\_hidden\_units\_factors]

# Create final MLP.

features = create\_mlp(

hidden\_units=mlp\_hidden\_units,

dropout\_rate=dropout\_rate,

activation=keras.activations.selu,

normalization\_layer=layers.BatchNormalization,

name="MLP")(features)

# Add a sigmoid as a binary classifer.

outputs = layers.Dense(units=1, activation="sigmoid", name="sigmoid")(features)

model = keras.Model(inputs=inputs, outputs=outputs)

return model

tabtransformer\_model = create\_tabtransformer\_classifier(

num\_transformer\_blocks=NUM\_TRANSFORMER\_BLOCKS,

num\_heads=NUM\_HEADS, embedding\_dims=EMBEDDING\_DIMS,

mlp\_hidden\_units\_factors=MLP\_HIDDEN\_UNITS\_FACTORS,

dropout\_rate=DROPOUT\_RATE)

## Model Training

def run\_experiment(

model, num\_epochs, learning\_rate, weight\_decay, batch\_size):

optimizer = keras.optimizers.AdamW(

learning\_rate=learning\_rate, weight\_decay=weight\_decay)

model.compile(optimizer=optimizer,

loss=keras.losses.BinaryCrossentropy(),

metrics=[keras.metrics.BinaryAccuracy(name="accuracy")])

print("Start training the model...")

history = model.fit( [df[col] for col in FEATURE\_NAMES], df[TARGET\_FEATURE\_NAME], epochs=num\_epochs)

print("Model training finished")

return history,model

history,model = run\_experiment(

model=tabtransformer\_model,

num\_epochs=NUM\_EPOCHS,

learning\_rate=LEARNING\_RATE,

weight\_decay=WEIGHT\_DECAY,

batch\_size=BATCH\_SIZE,

)

## Prediction

model.predict([df[col] for col in FEATURE\_NAMES])