**Pseudo-code**

Step 1 : Read the data

- train\_file = instances.jsonl

- test\_file = truth.jsonl

- df\_train, df\_test = read train, read\_test

- size = train.shape[0]

Step 2 : Create the dictionary

- truth\_id, truth\_mean = list test(id), list test(mean)

- truth\_dict = truth\_id[i]:truth\_mean[i] for all I

- train\_id, train\_post, train\_text = list(id, heading, content)

- creating corpus = join(id, post , text) with truth\_dict

Step 3 : Cleaning of data (discard tweets with 0.3 < score < 0.7)

- initial\_length = size = 19538

- cleaned\_web17 = new List [ ]

- iterating from i = 0 to size and if condition match append in cleaned\_web17

- condition = 0.3 < mean score < 0.7

- new\_final\_length = 12963

Step 4 : Bert Embedding

Download BERT

bert\_tokenizer = BertTokenizer.from\_pretrained("bert-base-uncased")

bert\_model = BertModel.from\_pretrained("bert-base-uncased")

save BertTokenizer, BertModel.

*now encode text into sequence of IDs.*

encode1 = torch.tensor(bert\_tokenizer.encode(web17.corpus[0][0]))

print(encode1.shape)

Step 5 : Data Profiling

- extract data; title\_all,content\_all,score\_all = [data in web17.corpus]

-title\_all\_token = bert\_tokenizer()

- Print the mean no. Of token, ID etc.

-Average # of tokens = 17.628058143105743

- content\_all\_token = bert\_tokenizer()

- Print the mean no. Of token, ID etc.

-Average # of tokens = 791.2599037772546

Step 6 : Extract embeddings & divide train/val/test set

-title\_all\_tokenized = bert\_tokenizer()

-Save it as a Py Torch file

-Give train\_size and val\_size

-shape gives batch size of 800

Step 7 : Process by patches and combine

-import torch and gc

-extract\_size=800 // One batch

-for loop form I =0 to num\_data//800

-outputs = bert\_model()

-check the shape

-save the title\_all\_embed as per the shape

-save the last 501 content and title separately (from 19200 to 19538)

Step 8 : Loading Data

- importing TensorDataset and DataLoader

- Xt\_all = torch.load ('titles\_all.pt')

- yt\_all = torch.load ('scores.pt')

- diving training, validation size = 10000, 2000

- test size = total - training - val = 963

- batch\_size = 64

- using TensorDataset on train, val and test

- using DataLoader on train, val and test

Step 9 : Defining LSTM Model Architecture

- importing torch.nn

- class LSTM inherit base class nn.Module

- \_init\_ constructor : batch size , num\_tokens, embed\_dim, hidden\_dim, n\_layers, dropout

- self.LSTM = embed\_dim, hidden\_dim, n\_layers, batch\_first=True, dropout=dropout, bidirectional=True

- defining two fully-connected layers :

self.fc1=nn.Linear(2\*hidden\_dim, 64)

self.fc2=nn.Linear(64, 1)

- defining forward function LSTM

- lstm\_out, hidden = self.lstm(x.unsqueeze(1), hidden)

flat = lstm\_out.squeeze()

out1 = self.fc1(flat)

out2 = self.fc2(torch.relu(out1))

out = torch.sigmoid(out2)

- iterating over the parameters

- defining init\_hidden(batch\_size) :

hidden = (weight.new(self.n\_layers\*2, batch\_size, self.hidden\_dim).zero\_()

weight.new(self.n\_layers\*2, batch\_size, self.hidden\_dim).zero\_()

- initializing the weights using xavier uniform (normal)

- torch.nn.init.xavier\_uniform\_(m.weight)

m.bias.data.fill\_(0.0)

Step 10 : Hyper-parameters Initialization

- hidden\_dim = 10

- dropout = 0.2

- optimizer = Adam Optimizer

- learning rate = 3e-4

- n\_layers = 2

- importing learning rate scheduler

- hyper-parameters of lr\_scheduler :

optimizer, 'min', factor=0.25, patience=0, threshold=0.05,min\_lr=3e-5, verbose=True

Step 11 : Training and Testing (**KD**)

- define training function with parameters as :

train\_dataloader, y\_truth, model, loss\_fn, optimizer, mute = False

- y\_pred\_train = []

- enumerate over train\_dataloader

- for batch, (X, y) in enumerate(train\_dataloader):

- Compute prediction error

pred, hidden = model(X, hidden)

y\_pred\_train.extend(pred.squeeze().cpu())

loss = loss\_fn(pred.squeeze(), y)

- Backpropagation

loss.backward( )

optimizer.step( )

- define testing function with parameters same as training function and mode

- mode = 0: validation when training (lr\_scheduler)

mode = 1: validation

mode = 2: test

- evaluating the model using four metrices

Loss, Accuracy, F1Score, Pearson Coefficient

Step 12 : Running the model for 5 epochs

- epochs = 5

- model.train()

- best\_val\_performance = 1.0

- train(train\_dataloader, yt\_all[:train\_size], model, loss\_fn, optimizer)

- val\_performance = test(val\_dataloader, yt\_all[train\_size:train\_size+val\_size], model, loss\_fn, lr\_scheduler)

Step 13 : Loss Function Optimizer and Accuracy

- hidden\_dim = 10 # num of tokens is typically 20

- \_ , num\_tokens, embed\_dim = Xt\_all.shape

- dropout = 0.2

- Using MSELoss as a loss function

loss\_fn = nn.MSELoss()

- Using Adam Optimizer with learning rate 3e-4

optimizer = torch.optim.Adam (model.parameters(), lr=3e-4)

- using learning rate scheduler

lr\_scheduler = ReduceLROnPlateau(optimizer, 'min', factor=0.25, patience=0, threshold=0.05, min\_lr=3e-5, verbose=True)

Step 14 : Testing on validation and test data

- \_ = test(val\_dataloader, yt\_all[train\_size: train\_size+val\_size], model, loss\_fn, lr\_scheduler, mode = 1)

- \_ = test(test\_dataloader, yt\_all[train\_size+val\_size:], model, loss\_fn, lr\_scheduler, mode = 2)