ASDS 5305 Final Project Datasets

2/14/2025

Group Name: sp25_BerKyd

Group Members:
- Henry Berrios #1001392315
- LeMaur Kydd #1001767382

Step 1 – Open the Google Colab Links
Step 2 – Open the GitHub Repository
Step 3 – Navigate to the “FPA_2” directory
Step 4 - Refer to the README.md file for instructions on how to run each of the Google Colab Notebook successfully.

Dataset 1

Link to Google Colab

Title: Drug SMILES Strings and Classifications

Dataset Description

The SMILES Strings and Drug Classification dataset sourced from a paper is a compilation of a few different datasets, namely PubChem and ZINC. It contains SMILES Strings, which will function as the basis of our training data and drug classifications, which will be our target variable. There are 6 other chemical features that can help classification performance if necessary.

The features in dataset 1 are as follows:
- IsomericSMILES Strings
- De-salted SMILES Strings
- Drug Classification (Target)
- XLogP
- Molecular Weight
- CID (PubChem Molecular ID#)
- HBondAcceptorCount
- HBondDonorCount

Defining the ML Problem

- Supervised Learning Task: Classification
- Goal: Predict the Drug Classification of a given SMILES String using only the SMILES String data.

- Potential Use: Molecular structures can have their medicinal value estimated without costly lab research based on its SMILES String.
- Target variable: Drug Classification (categorical variable)

Tensorizing Commentary

In order to tensorize SMILES text data there are a few steps. I knew of this process as I have worked with text data in general before as well as this dataset in particular in the past. First you have to take each SMILES string and tokenize them. This involves splitting the string into either individual characters or atom-based splits that keep the molecule characters together. For this dataset we went with character based tokenization for the first round of modeling. The second step is considered sequencing but in the use case of TensorFlow's ***tf.keras.preprocessing.text.Tokenizer*** this step and the first one are done sequentially.

```
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
# Tokenize SMILES Strings
tokenizer = Tokenizer(char_level=True, filters="") # Initialize the
tokenizer
tokenizer.fit_on_texts(train_df['IsomericSMILES']) # Fit the tokenizer on
the training data
```

```
# SMILES Strings to sequences
x_train_sequences =
tokenizer.texts_to_sequences(train_df['IsomericSMILES']) # Convert the
training SMILES Strings to sequences
x_test_sequences = tokenizer.texts_to_sequences(test_df['IsomericSMILES'])
# Convert the testing SMILES Strings to sequences
```

The remaining steps involve padding and embedding. For padding we used and referenced ***tensorflow.keras.preprocessing.sequence.pad_sequence*** in order to standardize the length of each tokenized SMILES string. This makes sure our input layer is consistently receiving data with the same dimensions.

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Padding the sequences to the length of the longest SMILES String
max_length = max(map(len, x_train_sequences)) # Get max length
x_train_padded = pad_sequences(x_train_sequences, maxlen=max_length,
padding='post') # Pad the training sequences
x_test_padded = pad_sequences(x_test_sequences, maxlen=max_length,
padding='post')
```

After a bit of research on what to do next to the data we found out about embedding the vectors using an embedding layer from the ***torch.nn.Embedding*** docs. This embedding reduces the dimensionality of the data and keeps important structural information, which is great for LSTM's and Transformer model architectures.

```
import torch
import torch.nn as nn

# Now we will embed the SMILES tensors
embedding_dim = 128 #embedding dimension
embedding_layer =
torch.nn.Embedding(num_embeddings=len(tokenizer.word_index) + 1,
embedding_dim=embedding_dim) # Initialize the embedding layer
x_train_tensor = embedding_layer(x_train_tensor) # Embed the training
sequences
x_test_tensor = embedding_layer(x_test_tensor) # Embed the testing
sequences
```

Lastly, the embedded vectors are compiled into simple tensor data types and saved under appropriate file names.

Dataset 2

Link to Google Colab

Title: 911 Recordings: The First 6 Seconds

Data Description

The 911 Recordings: The First 6 Seconds dataset is a curated collection of emergency call recordings compiled from public sources. Originally collected by the late Gary Allen, former editor of Dispatch Monthly magazine and 911 Dispatch website, this dataset provides valuable insight into real-world emergency situations and dispatcher responses. The dataset contains audio recordings of the first 6 seconds of civilian-initiated 911 calls, along with metadata that describes each event.

Each recording begins in a similar way, where the caller starts describing the nature of the emergency, which is often after reporting their location. The dataset also includes manually labeled metadata, including:

- Date
- State
- Potential life-threatening situation
- False alarms

- Number of deaths reported
- Description of the event

Since the dataset focuses on real-world emergency scenarios, it presents a unique challenge in speech-based machine learning and audio classification.

Defining the ML Problem

- Supervised Learning Task: Binary Classification
- Goal: Predict whether a 911 call describes a life-threatening situation based on the first 6 seconds of audio.
- Potential Use: An AI-based triaging system to assist emergency dispatchers in identifying critical calls that require immediate attention.
- Target Variable: *potential_death* (Binary: 1 = Serious Incident, 0 = Not Serious)

Tensorizing Commentary

Deep learning models require numerical inputs in a structured format. Since raw audio data is *unstructured*, it must be transformed into numerical representations before being converted into tensors. Challenges can vary, but are not limited to:

- Handling raw audio data
- Extracting meaningful features from the audio (in the form of MFCCs in our case)
- Ensuring numerical consistency
- Balancing imbalance target classes before training

In dataset 2, the audio data was already limited to the first 6 seconds. However, the waveforms themselves vary in their sampling rate, and amplitude values. Because deep learning models expect fixed-length numerical inputs, we transformed raw waveforms into structured numerical data using feature extraction.

To transform our raw waveforms, we used Mel-Frequency Cepstral Coefficients (MFCCs). MFCCs are commonly used in speech recognition and help capture the spectral properties of human speech (Rabiner & Juang, 1993). MFCCs are useful in that they reduce dimensionality, mimic human auditory perception and are effective for classification tasks.

In our case, we first resampled our audio data to make sure our sample rate was unified. *Librosa*was used to resample the audio files to 16,000 hz for standardization, which ensures that the extracted features are computed consistently.

```
# code to check sample rates were adapted using OpenAI's ChatGPT

sample_rates = []

# looping through audio files
for filename in os.listdir(audio_folder_path):
  if filename.endswith('.wav'):
```

```
    file_path = os.path.join(audio_folder_path, filename)
    samplerate, _ = wavfile.read(file_path)
    sample_rates.append(samplerate)

# printing the unique sample rates
print(set(sample_rates))
```

```
#
```

```
 code to resample audio files to 16,000 Hz was generated using OpenAI's ChatGPT

# set target sample rate
TARGET_SAMPLE_RATE = 16000

# function to load and resample audio
def load_and_resample_audio(filepath, target_sr = TARGET_SAMPLE_RATE):
  signal, sr = librosa.load(filepath, sr = target_sr) # load and resample
  return sr, signal # return new sample rate and audio signal

# dictionary to store resampled audio data
audio_data = {}

# loop through all audio files and resample them
for filename in os.listdir(audio_folder_path):
  if filename.endswith('.wav'):
    file_path = os.path.join(audio_folder_path, filename)
    try:
      sr, signal = load_and_resample_audio(file_path)
      audio_data[filename] = (sr, signal) # store resmapled data

    except Exception as e:
      print(f"Error processing file {filename}: {e}")
```

After the resample, MFCCs were extracted to help capture the phonetic and tonal structure of speech, which is crucial in understanding call severity.

```
# code to extract MFCC feature was generated using OpenAI's ChatGPT
def extract_mfcc(filename, audio_data, n_mfcc = 13):
  samplerate, signal = audio_data[filename] # get resampled audio

  #extract MFCCs
  mfccs = librosa.feature.mfcc(y = signal, sr = samplerate, n_mfcc = n_mfcc)
```

```
  # return mean of each MFCC coefficient
  return np.mean(mfccs, axis = 1)

# apply MFCC extraction to all resampled audio files
mfcc_features = {filename: extract_mfcc(filename, audio_data) for filename in
audio_data}
```

The MFCC dataframe was merged with the metadata dataframe, and SMOTE was used to handle the class imbalance for our target variable. Standardization was applied shortly after, which is a necessary step in neural networks, since larger features with larger magnitudes could dominate smaller ones.

```
# apply SMOTE to the unbalanced data
smote = SMOTE(sampling_strategy = 'auto', random_state = 2)
X_balanced, y_balanced = smote.fit_resample(X, y)
```

```
# standardizing the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Finally, PyTorch models require tensor inputs instead of pandas dataframes of NumPy arrays. Converting data into PyTorch tensors allows for efficient computation, and compatibility with deep learning models. Converting the 911 Recordings dataset into tensors involved multiple preprocessing steps, including resampling, feature extraction, balancing and standardization, and tensor conversion. These steps are crucial to ensure the dataset was clean, balanced and formatted correctly for PyTorch models.

```
# convert train and test sets to tensors
# code adapted and assisted by OpenAi's ChatGPT (OpenAI, 2025) and from PyTorch
Documentation (Pazke et al., 2019)

X_train_tensor = torch.tensor(X_train_scaled, dtype = torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype = torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype = torch.float32).view(-1,
1)
y_test_tensor = torch.tensor(y_test.values, dtype = torch.float32).view(-1, 1)
```

Dataset 3

Link to Google Colab

Title: TBI-NDSC

Dataset Description

The Traumatic Brain Injury National Data and Statistical Center (TBI-NDSC) dataset contains extensive information on individuals who have sustained traumatic brain injuries (TBIs). It includes a wide range of variables such as demographic, injury-related, rehabilitation, and outcome variables. These variables help clinicians track recovery and functional outcomes.

For this project, dataset 3 focuses on the TBIMS Form 1 dataset, which provides admission/intake data including:
- Demographics
- Pre-injury conditions
- Acute hospital information
- Cognitive and functional scores
- Glasgow Coma Scale (GCS) Scores
- FIM Cognitive & Motor Scores
- Insurance and employment details

Since data collection forms have changed over time, columns contain a high percentage of missing values. Therefore, we will select only the most relevant columns.

Defining the ML Problem

- Supervised Learning Task: Regression
- Goal: Predict Disability Rating Scale (DRS) at discharge (DRSd) using only admission data.
- Potential Use: Early prognosis estimation to assist clinicians in treatment planning and rehabilitation resource allocation.
- Target variable: Disability Rating Scale at Discharge (DRSd) (continuous variable)

Tensorizing Commentary

Converting the TBI-NDSC dataset into tensor format required several key preprocessing steps. First, only important columns were kept that were deemed meaningful based on domain knowledge. Next, placeholder values (such as 66, 77 etc.) were removed from those columns to keep only the data that was known for each patient. Outliers were removed for continuous variables using IQR, and label encoding/one-hot encoding was applied to the nominal and ordinal variables respectively.

```
# using IQR for continous variables
Q1 = tbi1_adm[cont_vars].quantile(0.25)
Q3 = tbi1_adm[cont_vars].quantile(0.75)
IQR = Q3 - Q1
```

```
# defining upper and lower bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

```
# list of nominal columns for one-hot encoding that are not already binary
nom_cols = ['SexF','Race','Mar', 'Emp1', 'Cause', 'PTAMethod',
'Craniotomy', 'AcutePay1', 'RehabPay1']

#list of ordinal columns for label encoding:
ord_cols = ['CTComp', 'GCSEye', 'GCSVer', 'GCSMot', 'DRSEyeA', 'DRSVerA',
'DRSMotA', 'DRSFeedA', 'DRSToiletA', 'DRSGroomA', 'DRSFuncA', 'DRSEmpA',
'FIMCompA', 'FIMExpressA', 'FIMSocialA', 'FIMProbSlvA', 'FIMMemA']
```

```
# one-hot encoding for nominal columns
tbi1_adm_encoded = pd.get_dummies(tbi1_adm_filtered, columns = nom_cols,
drop_first = True)

# label encoding for ordinal columns
le = LabelEncoder()
for col in ord_cols:
  tbi1_adm_encoded[col] = le.fit_transform(tbi1_adm_encoded[col])
```

The dataset was then standardized by using sklearn's StandardScaler. As mentioned before, this helps deep learning networks perform better and allows PyTorch models to converge fast and more stably. Once it was standardized, it was converted into PyTorch tensors. In our code, we used .values for the y_train and y_test since they were Pandas series, and PyTorch does not accept Pandas objects. By using .values, we convert them to NumPy arrays, which PyTorch can use. .view(-1, 1) ensures that the target variable y is formatted as a column vector rather than a 1D array. This prevents the shape from becoming mismatched during training. Finally, dtype = torch.float32 was used to optimize the deep learning computations. Float32 reduces memory usage while still maintaining precision.

```
# initializing the scaler
scaler = StandardScaler()

# fit on training data, and transform the train and test
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# convert train and test sets to tensors
# code adapted and assisted by OpenAi's ChatGPT (OpenAI, 2025) and from
PyTorch Documentation (Pazke et al., 2019)
X_train_tensor = torch.tensor(X_train_scaled, dtype = torch.float32)
X_test_tensor = torch.tensor(X_test_scaled, dtype = torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype =
torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test.values, dtype =
torch.float32).view(-1,1)
```

References

- TensorFlow Developers. (2023). Tokenizer API: tf.keras.preprocessing.text.Tokenizer. Retrieved from https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer.
- RDKit Developers. (2023). RDKit: Open-source cheminformatics. Retrieved from https://www.rdkit.org/docs/.
- Krenn, M., Häse, F., Nigam, A., Friederich, P., & Aspuru-Guzik, A. (2020). SELFIES: A robust representation of semantically constrained graphs with an example application in chemistry. Machine Learning: Science and Technology, 1(4). Retrieved from https://iopscience.iop.org/article/10.1088/2632-2153/aba947/meta.
- NVIDIA. (2020). CUDA Programming Guide. Retrieved from https://developer.nvidia.com/cuda-toolkit.
- OpenAI. (2025). Response generated by ChatGPT [Large language model]. OpenAI. Retrieved from https://chat.openai.com
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). PyTorch: An imperative style, high-performance deep learning library. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.
- PyTorch Community. (2024). torch.Tensor.view. Retrieved from https://pytorch.org/docs/stable/generated/torch.Tensor.view.html/.
- PyTorch Developers. (2023). Data types in PyTorch. Retrieved from https://pytorch.org/docs/stable/tensors.html#torch-tensor.
- Raschka, S., Liu, Y., & Mirjalili, V. (2022). Machine Learning with PyTorch and Scikit-Learn. Packt Publishing.
- Scikit-learn Developers. (2023). Preprocessing data: StandardScaler. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.