

## ✓ Assignment 1 for FIT5212, Semester 1

**Student Name:** Anh Huy Phung

**Student ID:** 34140298

In this task, I run eight configurations to perform a binary classification on the *ComputationalLinguistics* column, treating it as a 0/1 label and ignoring the other two classes for simplicity.

## ✓ Loading dataset and import library for part 1

```
from google.colab import drive
drive.mount('/content/drive')
```

Mounted at /content/drive

```
# Run the following cmd if you have not installed torch and torchtext
# !pip install torch==2.0.0 --index-url https://download.pytorch.org/whl/cu118
# !pip install torchtext==0.15.1 -q
```

```
2.0/2.0 MB 22.9 MB/s eta 0:00:00
619.9/619.9 MB 1.4 MB/s eta 0:00:00
4.6/4.6 MB 50.9 MB/s eta 0:00:00
317.1/317.1 MB 4.2 MB/s eta 0:00:00
11.8/11.8 MB 107.5 MB/s eta 0:00:00
21.0/21.0 MB 83.0 MB/s eta 0:00:00
849.3/849.3 kB 43.6 MB/s eta 0:00:00
557.1/557.1 MB 1.3 MB/s eta 0:00:00
168.4/168.4 MB 7.0 MB/s eta 0:00:00
54.6/54.6 MB 12.3 MB/s eta 0:00:00
102.6/102.6 MB 8.3 MB/s eta 0:00:00
173.2/173.2 MB 6.9 MB/s eta 0:00:00
177.1/177.1 MB 6.3 MB/s eta 0:00:00
98.6/98.6 kB 6.9 MB/s eta 0:00:00
63.3/63.3 MB 12.6 MB/s eta 0:00:00
96.4/96.4 kB 7.4 MB/s eta 0:00:00
```

ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is torchvision 0.21.0+cu124 requires torch==2.6.0, but you have torch 2.0.0 which is incompatible.  
torchaudio 2.6.0+cu124 requires torch==2.6.0, but you have torch 2.0.0 which is incompatible.

```
import torch
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.data.functional import to_map_style_dataset
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence
from torchtext import data
import torch.optim as optim
import time
```

```
import csv
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer, PorterStemmer
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score, accuracy_score, matthews_corrcoef, precis
import seaborn as sns
import numpy as np
```

```
import torch
from torch.utils.data import DataLoader, SubsetRandomSampler
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import accuracy_score, classification_report
```

## ✓ Part 1: Text Classification

In this task, I removed only *stopwords*, as they offer little semantic value and tend to introduce noise. Since both the *title* and *abstract* are short and centered on key concepts, preserving full word forms is important. I avoided stemming and lemmatization to retain meaningful variations in academic language—terms like *learn*, *learning*, and *learned* may reflect different contexts or stages of a concept. Reducing them to a root form could weaken interpretability and reduce classification accuracy.

Below code defines a preprocessing pipeline for text data. It includes a customizable *tokenize\_text* function that can remove stopwords and return either tokens or joined text. The *load\_data* function reads a CSV file, processes text from a specified column using the tokenizer and extracts labels.

```
# lots of Python code here
tokenizer = get_tokenizer('basic_english')
```

```
# Ensure required NLTK resources are available
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')
```

```
# Initialize tools
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
stemmer = PorterStemmer()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

```
def tokenize_text(text, remove_stopwords=False, lemmatize=False, stem=False, pre_process=False, is_return_text=False):
    # Tokenize the text
    tokens = tokenizer(text)
```

```
    # Preprocess = remove stopwords and keep only alphabetic tokens
    if pre_process or remove_stopwords:
        tokens = [t for t in tokens if t not in stop_words and t.isalpha()]
```

```
    # Lemmatization
    if lemmatize:
        tokens = [lemmatizer.lemmatize(t) for t in tokens]
```

```
    # Stemming
    if stem:
        tokens = [stemmer.stem(t) for t in tokens]
```

```
    if is_return_text:
        return ' '.join(tokens)
```

```
    return tokens
```

```
def load_data(file_path, col_name, label_name=None, remove_stopwords=True, lemmatize=False, stem=False):
    """
```

```
    Load and preprocess text data from a CSV file. Always processes the text using tokenize_text.
```

```
    Parameters:
```

- file\_path (str): Path to the CSV file.
- col\_name (str): Column name containing the text.
- label\_name (str, optional): Column name for the label. If None, labels are not used.
- remove\_stopwords (bool): Whether to remove stopwords.
- lemmatize (bool): Whether to apply lemmatization.
- stem (bool): Whether to apply stemming.

```
    Returns:
```

```

- pd.DataFrame: DataFrame with columns ['label', 'processed_text'] or ['original_text', 'processed_text']
"""
data = []

with open(file_path, 'r', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    headers = next(reader)

    col_index = headers.index(col_name)
    label_index = headers.index(label_name) if label_name else None

    for row in reader:
        text = row[col_index].strip()
        processed_text = tokenize_text(
            text,
            remove_stopwords=remove_stopwords,
            lemmatize=lemmatize,
            stem=stem,
            is_return_text=True
        )

        if label_name:
            label = int(row[label_index])
            data.append((label, processed_text))
        else:
            data.append((text, processed_text))

    if label_name is None:
        return pd.DataFrame(data, columns=['text', 'processed_text'])
    return data

# train_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/train_set.csv'
# dev_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/dev_set.csv'
# test_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/test_set.csv'
train_url = 'train_set.csv'
dev_url = 'dev_set.csv'
test_url = 'test_set.csv'

train_title_comp_data = load_data(train_url, 'title', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)
dev_title_comp_data = load_data(dev_url, 'title', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)
test_title_comp_data = load_data(test_url, 'title', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)

train_abs_comp_data = load_data(train_url, 'abstract', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)
dev_abs_comp_data = load_data(dev_url, 'abstract', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)
test_abs_comp_data = load_data(test_url, 'abstract', 'ComputationalLinguistics', remove_stopwords=True, lemmatize=False, stem=False)

print(f'Number of training examples: {len(train_title_comp_data)}')
print(f'Number of validation examples: {len(dev_title_comp_data)}')
print(f'Number of testing examples: {len(test_title_comp_data)}')

```

```

↗ Number of training examples: 217058
  Number of validation examples: 27132
  Number of testing examples: 27133

```

After loading the data of both *abstract* and *title*, I will visualize the word frequency distribution and determine an appropriate cutoff point based on percentiles.

```

from collections import Counter
import numpy as np
import matplotlib.pyplot as plt

def yield_tokens(data_iter, frequent_tokens = None):
    for _, text in data_iter:
        tokens = tokenizer(text)
        if frequent_tokens is not None:
            yield [t for t in tokens if t in frequent_tokens]
        else:
            yield tokenizer(text)

```

```
# Step 1: Count token frequencies
token_counts = Counter()
for tokens in yield_tokens(train_title_comp_data):
    token_counts.update(tokens)

# Step 2: Calculate frequency percentiles
frequencies = np.array(list(token_counts.values()))
low_percentile = 10 # remove bottom 10% of rare tokens
high_percentile = 99 # optionally remove top 1% most common tokens

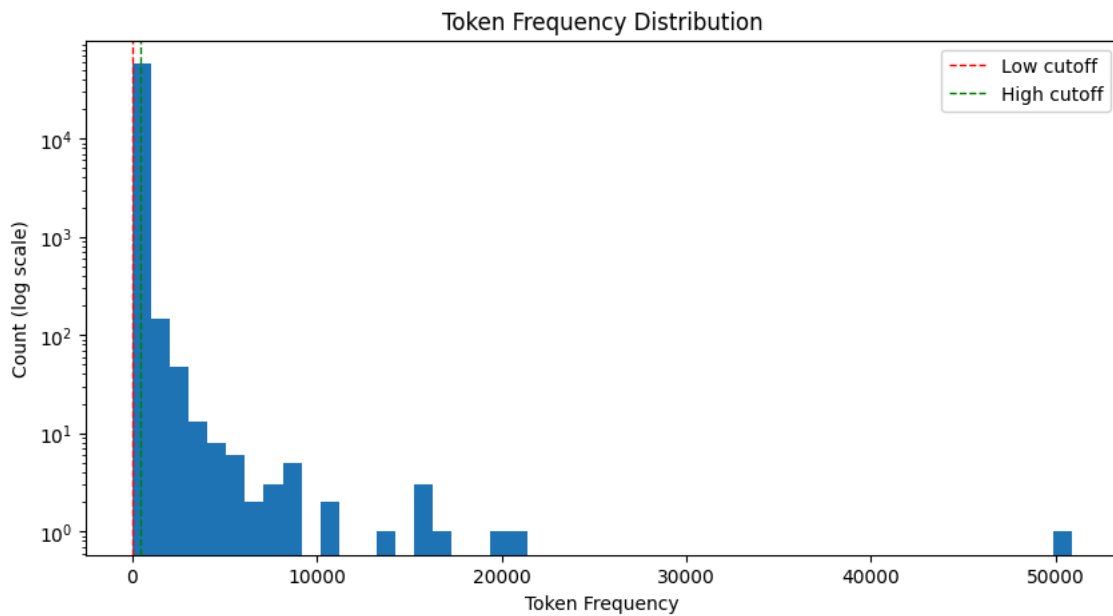
low_cutoff = np.percentile(frequencies, low_percentile)
high_cutoff = np.percentile(frequencies, high_percentile)

total_tokens = sum(token_counts.values())
print(f"Total number of tokens: {total_tokens}")
print(f"Keeping tokens with frequency between {low_cutoff:.2f} and {high_cutoff:.2f}")

# Optional: visualize frequency distribution
plt.figure(figsize=(10, 5))
plt.hist(frequencies, bins=50, log=True)
plt.axvline(low_cutoff, color='red', linestyle='dashed', linewidth=1, label='Low cutoff')
plt.axvline(high_cutoff, color='green', linestyle='dashed', linewidth=1, label='High cutoff')
plt.title("Token Frequency Distribution")
plt.xlabel("Token Frequency")
plt.ylabel("Count (log scale)")
plt.legend()
plt.show()

# Step 3: Calculate and print percentiles from 10 to 90 (step 10)
print("Percentiles (Token Frequency):")
for p in range(10, 100, 10):
    cutoff = np.percentile(frequencies, p)
    print(f"{p}th percentile: {cutoff:.2f}")
```

↻ Total number of tokens: 1485660  
Keeping tokens with frequency between 1.00 and 446.00



Percentiles (Token Frequency):

10th percentile: 1.00  
15th percentile: 1.00  
20th percentile: 1.00  
25th percentile: 1.00  
30th percentile: 1.00  
35th percentile: 1.00  
40th percentile: 1.00  
45th percentile: 1.00  
50th percentile: 1.00  
55th percentile: 1.00  
60th percentile: 1.00  
65th percentile: 2.00  
70th percentile: 2.00  
75th percentile: 3.00  
80th percentile: 5.00  
85th percentile: 8.00  
90th percentile: 17.00

Based on the percentile distribution, I set the cutoff at a token frequency of 90 (frequency 17) to filter out extremely rare and overly common tokens. This ensures a more balanced and meaningful vocabulary for RNN vocab modeling.

```
cutoff_75 = np.percentile(frequencies, 90)
print(f"90th percentile frequency cutoff: {cutoff_75:.2f}")
frequent_tokens = [token for token, freq in token_counts.items() if freq > cutoff_75]
```

```
vocab = build_vocab_from_iterator(
    yield_tokens(train_title_comp_data, frequent_tokens),
    specials=["<unk>"]
)
```

```
vocab.set_default_index(vocab["<unk>"])
```

```
vocab_size = len(set(list(vocab.get_itos())))
print(f"Unique tokens in TEXT vocabulary: {vocab_size}")
```

↻ 90th percentile frequency cutoff: 17.00  
Unique tokens in TEXT vocabulary: 5751

```
# Print the top 20 most common words in the vocabulary
top_n = 20
common_vocab = list(vocab.get_itos())[:top_n]
print(f"Top {top_n} most common words in the vocabulary: {common_vocab}")
```

↻ Top 20 most common words in the vocabulary: ['<unk>', 'learning', 'neural', 'models', 'using', 'deep', 'language', 'networks']

```
# Data processing
def text_pipeline(x):
    return vocab(tokenizer(x)) # return index of vocab

def collate_batch(batch):
    label_list, text_list = [], []
    for _label, _text in batch:
        label_list.append(_label)
        processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
        text_list.append(processed_text)
    return torch.tensor(label_list, dtype=torch.int64), pad_sequence(text_list, padding_value=vocab["<unk>"])
```

## ✓ Part 2. Create the Dataset and DataLoader

Due to Google Colab's runtime constraints—and after testing several subsets—I used only 10% of the original training data to ensure reasonable training time per epoch. In this context, “full training data” refers to this 10% subset. I also apply the code from tutorials for using torch to create TextDataset and DataLoader for RNN training preparation.

```
from torch.utils.data import Dataset

class TextDataset(Dataset):
    def __init__(self, data):
        self.data = data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

# Create TextDataset instances for abstract data
train_abs_dataset = TextDataset(train_abs_comp_data) # Use abstract data for training
test_abs_dataset = TextDataset(test_abs_comp_data) # Use abstract data for testing
validation_abs_dataset = TextDataset(dev_abs_comp_data) # Use abstract data for validation

# Create TextDataset instances
train_title_dataset = TextDataset(train_title_comp_data) # Use title data for training
test_title_dataset = TextDataset(test_title_comp_data) # Use title data for testing
validation_title_dataset = TextDataset(dev_title_comp_data) # Use title data for validation

# Ensure reproducibility
np.random.seed(42)
# Function to get DataLoader for the first 1000 cases or the full dataset
def get_dataloader(dataset, batch_size=32, shuffle=True, collate_fn=None, num_samples=None):
    """
    Function to get a DataLoader for a given dataset.
    If `num_samples` is provided, it will use only the first `num_samples` from the dataset.

    Args:
        dataset: The dataset to load.
        batch_size: The batch size to use in DataLoader.
        shuffle: Whether to shuffle the data (for training set).
        collate_fn: The function to collate data into batches.
        num_samples: Number of samples to use from the dataset (None for full dataset).

    Returns:
        A DataLoader instance for the dataset.
    """
    # Slice the dataset if num_samples is specified
    if num_samples is not None:
        dataset = dataset[:num_samples]

    # Create DataLoader
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle, collate_fn=collate_fn)

def get_random_subset_dataloader(dataset, batch_size=32, shuffle=True, collate_fn=None, subset_percentage=0.5):
    """
    Function to get a DataLoader for a random subset (50% by default) of the dataset.
    If `subset_percentage` is provided, it will randomly select that percentage of the dataset.
    """
```

```

Args:
    dataset: The dataset to load.
    batch_size: The batch size to use in DataLoader.
    shuffle: Whether to shuffle the data (for training set).
    collate_fn: The function to collate data into batches.
    subset_percentage: The percentage of the dataset to randomly select (default is 50%).

Returns:
    A DataLoader instance for the random subset of the dataset.
"""
# Get the indices of the full dataset
indices = list(range(len(dataset)))

# Shuffle the indices if shuffle is True
if shuffle:
    np.random.shuffle(indices)

# Select the subset percentage of the data
subset_size = int(len(indices) * subset_percentage)
subset_indices = indices[:subset_size]

# Create a SubsetRandomSampler with the selected indices
subset_sampler = SubsetRandomSampler(subset_indices)

# Create DataLoader for the subset
return DataLoader(dataset, batch_size=batch_size, sampler=subset_sampler, collate_fn=collate_fn)

"""
Take 10% of full data
"""
# For Abstract Data
train_abs_dataloader_1000 = get_dataloader(train_abs_dataset, batch_size=32, shuffle=True, collate_fn=collate_batch, num_samples=1000)
train_abs_dataloader_full = get_random_subset_dataloader(train_abs_dataset, batch_size=32, shuffle=True, collate_fn=collate_batch, subset_percentage=0.1)

validation_abs_dataloader = get_dataloader(validation_abs_dataset, batch_size=32, shuffle=False, collate_fn=collate_batch)
test_abs_dataloader = get_dataloader(test_abs_dataset, batch_size=32, shuffle=False, collate_fn=collate_batch)

# For Title Data
train_title_dataloader_1000 = get_dataloader(train_title_dataset, batch_size=32, shuffle=True, collate_fn=collate_batch, num_samples=1000)
train_title_dataloader_full = get_random_subset_dataloader(train_title_dataset, batch_size=32, shuffle=True, collate_fn=collate_batch, subset_percentage=0.1)

validation_title_dataloader = get_dataloader(validation_title_dataset, batch_size=32, shuffle=False, collate_fn=collate_batch)
test_title_dataloader = get_dataloader(test_title_dataset, batch_size=32, shuffle=False, collate_fn=collate_batch)

```

## ✓ Part 3: Model Configuration and result comprehension

### ✓ Part 3A: RNN Method

In this part, I configure and train an RNN model following the class tutorial. With approximately 5,800 unique tokens, setting the embedding dimension to 100 and the hidden dimension to 256 provides a balanced and efficient representation for the task.

```

import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.rnn = nn.RNN(embedding_dim, hidden_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        #text = [sent len, batch size]

        embedded = self.embedding(text)

        #embedded = [sent len, batch size, emb dim]

```

```

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1, :, :], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

INPUT_DIM = vocab_size
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1

RNN_model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
print(f'The model has {count_parameters(RNN_model):,} trainable parameters')
optimizer = optim.SGD(RNN_model.parameters(), lr=1e-3)
criterion = nn.BCEWithLogitsLoss()

```

## ✓ Train RNN model

```

def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8, NOT 8
    """

    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc

def train(model, dataloader, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for labels, texts in dataloader:

        optimizer.zero_grad()

        predictions = model(texts).squeeze(1)

        loss = criterion(predictions, labels.float()) #

        acc = binary_accuracy(predictions, labels)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(dataloader), epoch_acc / len(dataloader)

def evaluate(model, dataloader, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for labels, texts in dataloader:

            predictions = model(texts).squeeze(1)

```



```

    loss = criterion(predictions, labels.float())

    acc = binary_accuracy(predictions, labels)

    epoch_loss += loss.item()
    epoch_acc += acc.item()

    return epoch_loss / len(dataloader), epoch_acc / len(dataloader)

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs

abs_dic = {
    'train': {
        'abs_1000': train_abs_dataloader_1000,
        'abs_full': train_abs_dataloader_full
    },
    'validation': validation_abs_dataloader,
    'test': test_abs_dataloader
}

title_dic = {
    'train': {
        'title_1000': train_title_dataloader_1000,
        'title_full': train_title_dataloader_full
    },
    'validation': validation_title_dataloader,
    'test': test_title_dataloader
}

def train_rnn_on_data loaders(RNN_model, train_dict, val_loader, optimizer, criterion, n_epochs=5, model_prefix='RNN'):
    """
    Train an RNN model on multiple training dataloaders and save the best model per config.

    Args:
        RNN_model: The RNN model architecture (to be re-initialized per config).
        train_dict (dict): Dictionary of training dataloaders (e.g., {'abs_1000': train_loader_1000}).
        val_loader: Validation dataloader.
        optimizer: Optimizer instance.
        criterion: Loss function.
        n_epochs (int): Number of training epochs.
        model_prefix (str): Prefix for saved model filenames.
    """
    for dataloader_name, train_loader in train_dict.items():
        print(f'\n Start training for {dataloader_name} data:')
        best_valid_loss = float('inf')

        for epoch in range(n_epochs):
            start_time = time.time()

            train_loss, train_acc = train(RNN_model, train_loader, optimizer, criterion)
            valid_loss, valid_acc = evaluate(RNN_model, val_loader, criterion)

            end_time = time.time()
            epoch_mins, epoch_secs = epoch_time(start_time, end_time)

            model_filename = f'{model_prefix}_{dataloader_name}_model.pt'
            if valid_loss < best_valid_loss:
                best_valid_loss = valid_loss
                torch.save(RNN_model.state_dict(), model_filename)

            print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
            print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
            print(f'\tVal. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc*100:.2f}%')

    train_rnn_on_data loaders(RNN_model, abs_dic['train'], abs_dic['validation'], optimizer, criterion, n_epochs=5, model_prefix='RNN')
    train_rnn_on_data loaders(RNN_model, title_dic['train'], title_dic['validation'], optimizer, criterion, n_epochs=5, model_prefix='

```



```
val. Loss: 0.594 | val. Acc: 71.78%
```

```
Start training for abs_full data:
```

```
Epoch: 01 | Time: 14m 46s
  Train Loss: 0.588 | Train Acc: 72.35%
  Val. Loss: 0.594 | Val. Acc: 71.81%
Epoch: 02 | Time: 14m 35s
  Train Loss: 0.588 | Train Acc: 72.33%
  Val. Loss: 0.594 | Val. Acc: 71.85%
Epoch: 03 | Time: 14m 31s
  Train Loss: 0.589 | Train Acc: 72.32%
  Val. Loss: 0.594 | Val. Acc: 71.88%
Epoch: 04 | Time: 14m 30s
  Train Loss: 0.588 | Train Acc: 72.43%
  Val. Loss: 0.593 | Val. Acc: 71.91%
Epoch: 05 | Time: 14m 30s
  Train Loss: 0.588 | Train Acc: 72.50%
  Val. Loss: 0.594 | Val. Acc: 71.96%
```

```
Start training for title_1000 data:
```

```
Epoch: 01 | Time: 0m 3s
  Train Loss: 0.575 | Train Acc: 73.34%
  Val. Loss: 0.592 | Val. Acc: 71.87%
Epoch: 02 | Time: 0m 3s
  Train Loss: 0.571 | Train Acc: 74.02%
  Val. Loss: 0.593 | Val. Acc: 71.88%
Epoch: 03 | Time: 0m 4s
  Train Loss: 0.582 | Train Acc: 72.75%
  Val. Loss: 0.592 | Val. Acc: 71.89%
Epoch: 04 | Time: 0m 3s
  Train Loss: 0.576 | Train Acc: 74.12%
  Val. Loss: 0.592 | Val. Acc: 71.89%
Epoch: 05 | Time: 0m 3s
  Train Loss: 0.575 | Train Acc: 73.34%
  Val. Loss: 0.592 | Val. Acc: 71.89%
```

```
Start training for title_full data:
```

```
Epoch: 01 | Time: 0m 10s
  Train Loss: 0.588 | Train Acc: 72.29%
  Val. Loss: 0.591 | Val. Acc: 71.95%
Epoch: 02 | Time: 0m 10s
  Train Loss: 0.588 | Train Acc: 72.27%
  Val. Loss: 0.591 | Val. Acc: 72.00%
Epoch: 03 | Time: 0m 9s
  Train Loss: 0.588 | Train Acc: 72.36%
  Val. Loss: 0.591 | Val. Acc: 72.04%
Epoch: 04 | Time: 0m 10s
  Train Loss: 0.587 | Train Acc: 72.41%
  Val. Loss: 0.591 | Val. Acc: 72.05%
Epoch: 05 | Time: 0m 10s
  Train Loss: 0.587 | Train Acc: 72.51%
  Val. Loss: 0.591 | Val. Acc: 72.09%
```

### Training Summary

All four RNN configurations showed relatively stable training and validation performance across 5 epochs. Models trained on the full dataset (10% subset) slightly outperformed those using only 1,000 samples. Abstract-based models took significantly longer to train than title-based ones, but all models achieved similar validation accuracy, hovering around 71.8%–72.1%, with minor variations in training loss and accuracy.

### Part 3B: Statistical Method

In this section, I use\*\* Logistic Regression\*\* as the chosen statistical method. I implemented functions to train and build the model using **only the training data**, unlike the tutorial where both training and development sets were combined.

```
import numpy as np
```

```
def get_random_subset_log(data, subset_percentage=0.1, seed=42):
```

```
    """
```

```
    Returns a random subset of the input list-like data.
```

```
    Args:
```

```
        data (list-like): The input data to sample from.
```

```
        subset_percentage (float): Fraction of the data to include in the subset (e.g., 0.1 for 10%).
```

```
        seed (int): Random seed for reproducibility.
```

```
    Returns:
```

```
        list: A random subset of the input data.
```

```
    """
```

```

np.random.seed(seed)
indices = list(range(len(data)))
np.random.shuffle(indices)

subset_size = int(len(data) * subset_percentage)
subset_indices = indices[:subset_size]

return [data[i] for i in subset_indices]

def train_and_evaluate_logreg(train_title_comp_data, dev_title_comp_data = None, max_features=5000):
    """
    Train a Logistic Regression model on the training data and evaluate it on the development set.

    Args:
        train_title_comp_data (list of tuples): The list of text data (e.g., titles) for training with their labels.
        dev_title_comp_data (list of tuples): The list of text data (e.g., titles) for evaluation with their labels.
        max_features (int, optional): The maximum number of features (words) to consider for TF-IDF. Default is 5000.

    Returns:
        logreg: The trained Logistic Regression model.
    """
    # Extract titles and labels from the training and development data
    train_titles = [text for _, text in train_title_comp_data]
    train_labels = [label for label, _ in train_title_comp_data]
    # If dev data is provided, combine with training
    if dev_title_comp_data is not None:
        dev_titles = [text for _, text in dev_title_comp_data]
        dev_labels = [label for label, _ in dev_title_comp_data]
        combined_titles = train_titles + dev_titles
        combined_labels = train_labels + dev_labels
    else:
        combined_titles = train_titles
        combined_labels = train_labels
    # Step 1: Convert the text data into numerical features using TF-IDF
    vectorizer = TfidfVectorizer(stop_words='english', max_features=max_features)
    X_train_tfidf = vectorizer.fit_transform(combined_titles)

    # Step 2: Train Logistic Regression model
    logreg = LogisticRegression() #max_iter=1000

    start_time = time.time()
    logreg.fit(X_train_tfidf, combined_labels)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    print(f'Training time: {epoch_mins}m {epoch_secs}s')
    return logreg, vectorizer

```

### ✓ Part 3C: Configurations of all models

In this section, I evaluate multiple configurations of Logistic Regression and RNN models using both the title and abstract fields as inputs. After that I will evaluate metrics include accuracy, precision, recall, F1-score, and MCC, along with precision-recall curves. All results are also compiled into a single DataFrame for later comparison.

```

def get_predictions(model, test_data_loader):
    y_predict = []
    y_test = []
    y_probs = []

    model.eval()
    with torch.no_grad():
        for labels, texts in test_data_loader:
            predictions = model(texts).squeeze(1)
            probs = torch.sigmoid(predictions)
            rounded_preds = torch.round(probs)

            y_predict += rounded_preds.tolist()
            y_test += labels.tolist()
            y_probs += probs.tolist()

    return np.array(y_predict), np.array(y_test), np.array(y_probs)

```

```

def get_important_metrics(y_test, y_predict, y_probs):
    accuracy = accuracy_score(y_test, y_predict)
    precision = precision_score(y_test, y_predict, average='macro')
    recall = recall_score(y_test, y_predict, average='macro')
    f1score = f1_score(y_test, y_predict, average='macro')
    matthews = matthews_corrcoef(y_test, y_predict)
    precision_curve, recall_curve, _ = precision_recall_curve(y_test, y_probs)
    return accuracy, precision, recall, f1score, matthews, precision_curve, recall_curve

def run_configurations(configs):
    results = []
    for cfg in configs:
        model_type = cfg['model_type']
        input_type = cfg['input_type']
        train_data = cfg['train']
        dev_data = cfg['dev']
        test_data = cfg['test']
        test_loader = cfg.get('test_loader') # Only needed for RNN
        name = cfg['name']

        print(f"Running model {name}: ")

        if model_type == 'logreg':
            model, vectorizer = train_and_evaluate_logreg(train_data, dev_data)
            X_test = vectorizer.transform([text for _, text in test_data])
            y_test = [label for label, _ in test_data]
            start_time = time.time()
            y_pred = model.predict(X_test)
            y_probs = model.predict_proba(X_test)[: , 1]
            end_time = time.time()
        elif model_type == 'rnn':
            model = cfg['model']
            model.load_state_dict(torch.load(cfg['model_path']))
            start_time = time.time()
            y_pred, y_test, y_probs = get_predictions(model, test_loader)
            end_time = time.time()
        else:
            raise ValueError(f"Unknown model_type: {model_type}")

        mins, secs = epoch_time(start_time, end_time)
        print(f" Training time: {mins}m {secs}s")

        metrics = get_important_metrics(y_test, y_pred, y_probs)
        print(f" Acc: {metrics[0]:.4f}, Prec: {metrics[1]:.4f}, Rec: {metrics[2]:.4f}, "
              f"F1: {metrics[3]:.4f}, MCC: {metrics[4]:.4f}")
        print('\n')
        results.append((name, *metrics))

    return pd.DataFrame(results, columns=[
        'model_name', 'accuracy', 'precision', 'recall', 'f1score', 'matthews', 'precision_curve', 'recall_curve'
    ])

configs = [
    {
        'name': 'logreg_title_full',
        'model_type': 'logreg',
        'input_type': 'title',
        'train': get_random_subset_log(train_title_comp_data),
        'dev': None, # dev_title_comp_data,
        'test': test_title_comp_data
    },
    {
        'name': 'logreg_abs_full',
        'model_type': 'logreg',
        'input_type': 'abstract',
        'train': get_random_subset_log(train_abs_comp_data) ,
        'dev': None, # dev_abs_comp_data,
        'test': test_abs_comp_data
    },
    {
        'name': 'logreg_title_1000',
        'model_type': 'logreg',
        'input_type': 'title',
        'train': train_title_comp_data[:1000],
        'dev': None, #dev_title_comp_data,
        'test': test_title_comp_data
    }
]

```

```

    },
    {
        'name': 'logreg_abs_1000',
        'model_type': 'logreg',
        'input_type': 'abstract',
        'train': train_abs_comp_data[:1000],
        'dev': None, # dev_abs_comp_data,
        'test': test_abs_comp_data
    },
    {
        'name': 'RNN_title_1000',
        'model_type': 'rnn',
        'input_type': 'title',
        'model': RNN_model,
        'model_path': 'RNN_title_1000_model.pt',
        'train': None,
        'dev': None,
        'test': None,
        'test_loader': test_title_dataloader
    },
    {
        'name': 'RNN_title_full',
        'model_type': 'rnn',
        'input_type': 'title',
        'model': RNN_model,
        'model_path': 'RNN_title_full_model.pt',
        'train': None,
        'dev': None,
        'test': None,
        'test_loader': test_title_dataloader
    },
    {
        'name': 'RNN_abs_1000',
        'model_type': 'rnn',
        'input_type': 'abstract',
        'model': RNN_model,
        'model_path': 'RNN_abs_1000_model.pt',
        'train': None,
        'dev': None,
        'test': None,
        'test_loader': test_abs_dataloader
    },
    {
        'name': 'RNN_abs_full',
        'model_type': 'rnn',
        'input_type': 'abstract',
        'model': RNN_model,
        'model_path': 'RNN_abs_full_model.pt',
        'train': None,
        'dev': None,
        'test': None,
        'test_loader': test_abs_dataloader
    }
}
cv_df = run_configurations(configs)

➡ Running model logreg_title_full:
Training time: 0m 0s
Training time: 0m 0s
Acc: 0.8395, Prec: 0.8120, Rec: 0.7694, F1: 0.7860, MCC: 0.5798

Running model logreg_abs_full:
Training time: 0m 0s
Training time: 0m 0s
Acc: 0.8741, Prec: 0.8483, Rec: 0.8316, F1: 0.8393, MCC: 0.6796

Running model logreg_title_1000:
Training time: 0m 0s
Training time: 0m 0s
Acc: 0.7469, Prec: 0.7307, Rec: 0.5619, F1: 0.5456, MCC: 0.2391

Running model logreg_abs_1000:
Training time: 0m 0s
Training time: 0m 0s
Acc: 0.7703, Prec: 0.7877, Rec: 0.6024, F1: 0.6063, MCC: 0.3432

```

```

Running model RNN_title_1000:
Training time: 0m 4s
Acc: 0.7185, Prec: 0.5193, Rec: 0.5011, F1: 0.4299, MCC: 0.0090

Running model RNN_title_full:
Training time: 0m 3s
Acc: 0.7201, Prec: 0.5265, Rec: 0.5010, F1: 0.4270, MCC: 0.0101

Running model RNN_abs_1000:
Training time: 0m 29s
Acc: 0.7159, Prec: 0.4568, Rec: 0.4977, F1: 0.4243, MCC: -0.0201

Running model RNN_abs_full:
Training time: 0m 30s
Acc: 0.7177, Prec: 0.4666, Rec: 0.4986, F1: 0.4240, MCC: -0.0137

```

### ▼ Part 3D: Results for Methods

From the metrics dataframe, I will visualize the performance of each model with F1, precision, recall, accuracy and precision-recall curve

```

import matplotlib.pyplot as plt

metrics = ['accuracy', 'precision', 'recall', 'f1score']

# Create larger figure
plt.figure(figsize=(16, 9))

# Plot metric lines
cv_df.set_index('model_name')[metrics].plot(kind='line', figsize=(16, 9), legend=False)

plt.title('Model Performance Comparison', fontsize=20)
plt.ylabel('Score', fontsize=14)
plt.xlabel('Model Name', fontsize=14)
plt.ylim(0, 1.0)
plt.xticks(rotation=0)

# Increase tick value font sizes
plt.xticks(rotation=0, fontsize=14)
plt.yticks(fontsize=14)

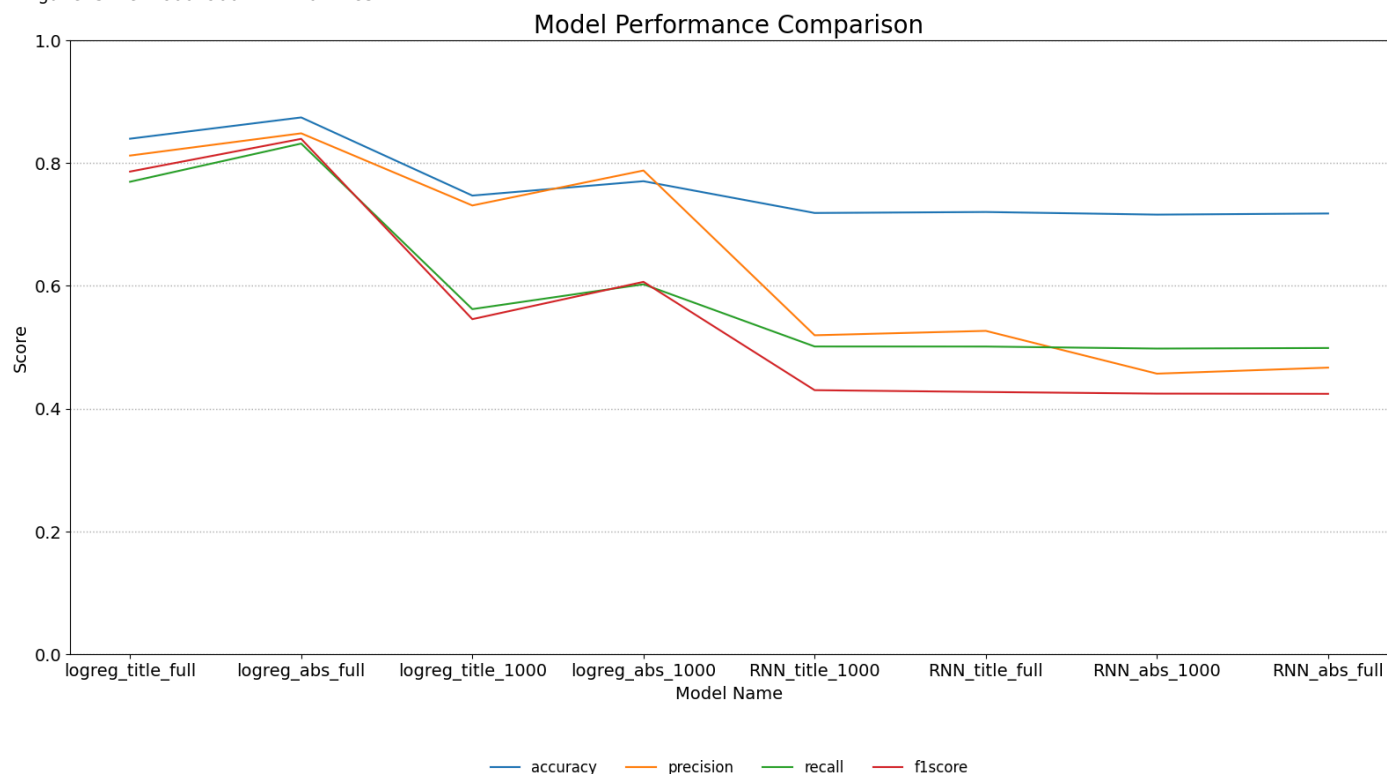
plt.grid(axis='y', linestyle=':', linewidth=1)

# Legend BELOW the chart
plt.legend(
    loc='upper center',
    bbox_to_anchor=(0.5, -0.15),
    ncol=4,
    fontsize=12,
    frameon=False
)

plt.tight_layout()
plt.show()

```

&lt;Figure size 1600x900 with 0 Axes&gt;



```
import matplotlib.pyplot as plt

plt.figure(figsize=(16, 9))

# Plot PR curves
for i in range(len(cv_df)):
    model_name = cv_df.iloc[i]['model_name']
    precision = cv_df.iloc[i]['precision_curve']
    recall = cv_df.iloc[i]['recall_curve']
    plt.plot(recall, precision, label=model_name)

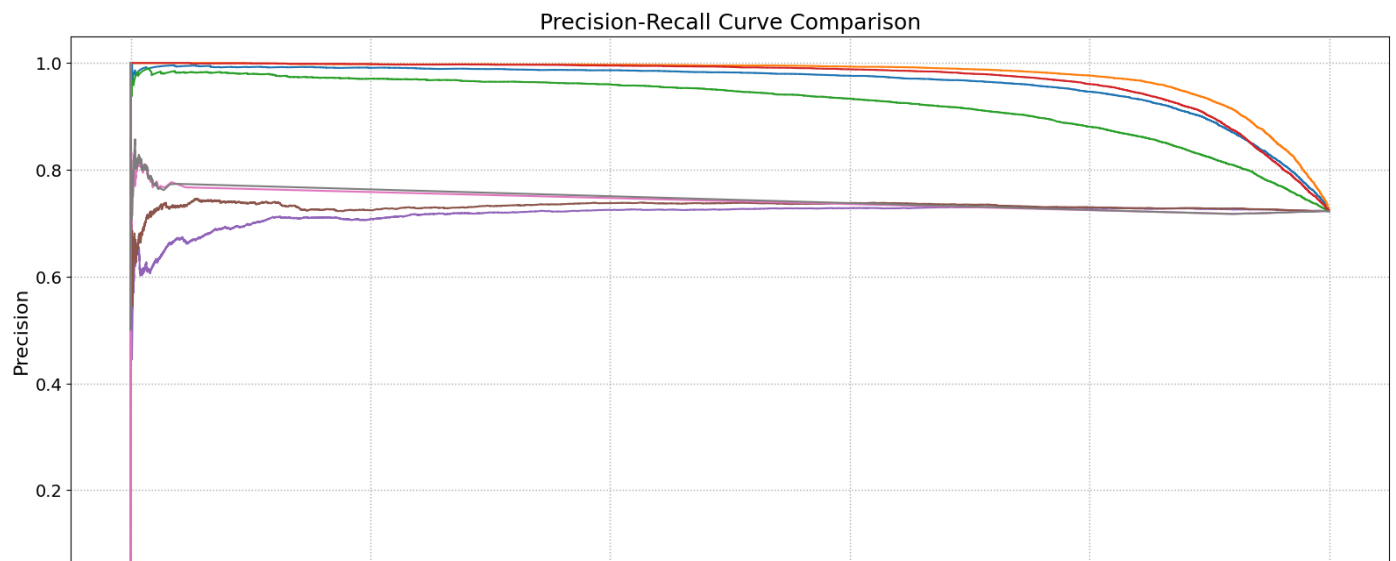
plt.title('Precision-Recall Curve Comparison', fontsize=18)
plt.xlabel('Recall', fontsize=16)
plt.ylabel('Precision', fontsize=16)

# Increase axis tick label sizes
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)

plt.grid(True, linestyle=':', linewidth=1)

# Legend BELOW the chart
plt.legend(
    loc='upper center',
    bbox_to_anchor=(0.5, -0.15),
    ncol=4,
    fontsize=13,
    frameon=False
)

plt.tight_layout()
plt.show()
```





**Student Name:** Anh Huy Phung

**Student ID:** 34140298

## ✓ Task 2

In this task, I will run two variations of the LDA model, with the key **difference being the inclusion or exclusion of bigrams** in the training data (for both 1000 and 20000 articles).

### ✓ Connect to google drive and install necessary library

```
from google.colab import drive
drive.mount('/content/drive')
```

↗ Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force\_remount=True)

```
import warnings
warnings.filterwarnings('ignore')
```

```
# !pip3 install scikit-learn plotly gensim -q
# !pip3 uninstall patsy -y
# !pip3 install patsy
# !pip3 uninstall seaborn -y
# !pip3 install seaborn
import seaborn as sns
import re
import nltk
import spacy
import string
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time
import csv
```

```
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score, classification_report, confusion_matrix,
    precision_score, recall_score, f1_score, matthews_corrcoef,
    precision_recall_curve
)
```

```
import plotly.graph_objects as go
import plotly.express as px
```

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer, PorterStemmer
```

```
import copy # Import the copy module for deep copy
```

```
# !pip3 install pyldavis
import pyLDavis
import pyLDavis.lda_model
```

↗ Requirement already satisfied: pyldavis in /usr/local/lib/python3.11/dist-packages (3.4.1)  
 Requirement already satisfied: numpy>=1.24.2 in /usr/local/lib/python3.11/dist-packages (from pyldavis) (1.26.4)  
 Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from pyldavis) (1.13.1)  
 Requirement already satisfied: pandas>=2.0.0 in /usr/local/lib/python3.11/dist-packages (from pyldavis) (2.2.2)  
 Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from pyldavis) (1.4.2)  
 Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from pyldavis) (3.1.6)  
 Requirement already satisfied: numexpr in /usr/local/lib/python3.11/dist-packages (from pyldavis) (2.10.2)  
 Requirement already satisfied: funcy in /usr/local/lib/python3.11/dist-packages (from pyldavis) (2.0)

```
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from pyldavis) (1.6.1)
Requirement already satisfied: gensim in /usr/local/lib/python3.11/dist-packages (from pyldavis) (4.3.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from pyldavis) (75.2.0)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=2.0.0->pyldavis) (2025.
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=2.0.0->pyldavis) (202
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.0.0->py
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.11/dist-packages (from gensim->pyldavis) (7.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->pyldavis) (3.0.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=2.0
Requirement already satisfied: wrapt in /usr/local/lib/python3.11/dist-packages (from smart-open>=1.8.1->gensim->pyldavis) (
```

```
# !pip3 uninstall -y torch torchtext torchvision torchaudio
# !pip3 install torch==2.0.0+cu118 torchvision==0.15.1+cu118 torchaudio==2.0.0+cu118 torchtext==0.15.1 --index-url https://downl
```

```
import torch
print("torch version:", torch.__version__)
import torchtext
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator
from torchtext.data.functional import to_map_style_dataset
```

```
🔗 torch version: 2.0.0+cu118
```

## ✓ 1. Data Preprocessing and loading data from file

Since the topics in the dataset—like Computational Linguistics, Machine Learning, and Human-Computer Interaction—share a lot of overlapping words (like "computer" and "learning"), using the Title field would be too vague and not help in distinguishing the topics well. Personally, I feel that using the Abstract field makes more sense, as it provides much more detailed and specific information, making it easier to separate the topics clearly.

In this task, I will use **nlTK** and **spaCy** for text preprocessing, including:

- Tokenization
- Stopword removal
- Lemmatization

The reason is that I want to select words that are not stopwords, as stopwords typically do not carry meaningful information for topic analysis. Additionally, I want to ensure the remaining words have grammatically accurate and meaningful base forms, which lemmatization provides. Unlike stemming, lemmatization maps words to their actual dictionary forms, preserving semantic meaning, improving interpretability that are suitable the experiment set up.

```
# lots of Python code here
tokenizer = get_tokenizer('basic_english')

# Ensure required NLTK resources are available
nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')

# Initialize tools
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()
stemmer = PorterStemmer()
```

```
🔗 [nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data] Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
```

```
# Tokenize the text with 2 options: reuturn tokens list or texts that are processed after being removed stopwords and got lemmat
def tokenize_text(text, remove_stopwords=False, lemmatize=False, stem=False, pre_process=False, is_return_text=False):
```

```
    # Tokenize the text
    tokens = tokenizer(text)
```

```
    # Preprocess = remove stopwords and keep only alphabetic tokens
```

```

    if pre_process or remove_stopwords:
        tokens = [t for t in tokens if t not in stop_words and t.isalpha()]

    # Lemmatization
    if lemmatize:
        tokens = [lemmatizer.lemmatize(t) for t in tokens]

    # Stemming
    if stem:
        tokens = [stemmer.stem(t) for t in tokens]
    tokens = [token for token in tokens if len(token) > 2]
    if is_return_text:
        return ' '.join(tokens)

    return tokens

# text = "" Molecule discovery is a pivotal research field, impacting everything from the
# medicines we take to the materials we use. Recently, Large Language Models
# (LLMs) have been widely adopted in molecule understanding and generation, yet
# the alignments between molecules and their corresponding captions remain a
# significant challenge. Previous endeavours often treat the molecule as a
# general SMILES string or molecular graph, neglecting the fine-grained
# alignments between the molecular sub-structures and the descriptive textual
# phrases, which are crucial for accurate and explainable predictions. In this
# case, we introduce MolReFlect, a novel teacher-student framework designed to
# contextually perform the molecule-caption alignments in a fine-grained way. Our
# approach initially leverages a larger teacher LLM to label the detailed
# alignments by directly extracting critical phrases from molecule captions or
# SMILES strings and implying them to corresponding sub-structures or
# characteristics. To refine these alignments, we propose In-Context Selective
# Reflection, which retrieves previous extraction results as context examples for
# teacher LLM to reflect and lets a smaller student LLM select from in-context
# reflection and previous extraction results. Finally, we enhance the learning
# process of the student LLM through Chain-of-Thought In-Context Molecule Tuning,
# integrating the fine-grained alignments and the reasoning processes within the
# Chain-of-Thought format. Our experimental results demonstrate that MolReFlect
# enables LLMs like Mistral-7B to significantly outperform the previous
# baselines, achieving SOTA performance on the ChEBI-20 dataset. This advancement
# not only enhances the generative capabilities of LLMs in the molecule-caption
# translation task, but also contributes to a more explainable framework.
# ""
# a = processed_text = tokenize_text(
#     text,
#     remove_stopwords=True,
#     lemmatize=True,
#     stem=False,
#     is_return_text=False
# )
# a

```

After that, we load the file and process it as described above. The result would be new dataframe that has column named **processed\_abstract** with list of tokens.

```

def load_data2(file_path, col_name, label_name=None, remove_stopwords=True, lemmatize=False, stem=False, first_row_num=None):
    """
    Load and preprocess text data from a CSV file. Always processes the text using tokenize_text.

    Parameters:
    - file_path (str): Path to the CSV file.
    - col_name (str): Column name containing the text.
    - label_name (str, optional): Column name for the label. If None, labels are not used.
    - remove_stopwords (bool): Whether to remove stopwords.
    - lemmatize (bool): Whether to apply lemmatization.
    - stem (bool): Whether to apply stemming.
    - first_row_num (int, optional): The number of rows to retrieve. If None, all rows are used.

    Returns:
    - pd.DataFrame: DataFrame with columns ['label', 'processed_text'] or ['original_text', 'processed_text']
    """
    data = []

    with open(file_path, 'r', newline='', encoding='utf-8') as f:

```

```

reader = csv.reader(f)
headers = next(reader)

col_index = headers.index(col_name)
label_index = headers.index(label_name) if label_name else None

# Counter to limit rows
row_count = 0

for row in reader:
    if first_row_num and row_count >= first_row_num:
        break # Stop reading after first_row_num rows

    text = row[col_index].strip()
    processed_text = tokenize_text(
        text,
        remove_stopwords=remove_stopwords,
        lemmatize=lemmatize,
        stem=stem,
        is_return_text=False
    )

    if label_name:
        label = int(row[label_index])
        data.append((label, processed_text))
    else:
        data.append((text, processed_text))

    row_count += 1

if label_name is None:
    return pd.DataFrame(data, columns=['abstract', 'processed_abstract'])
return data

```

```

# train_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/train_set.csv'
# dev_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/dev_set.csv'
# test_url = '/content/drive/MyDrive/FIT5212/Ass1/Dataset_Assignment1/test_set.csv'
train_url = 'train_set.csv'
dev_url = 'dev_set.csv'
test_url = 'test_set.csv'

```

```

train_abs_1000_df = load_data2(train_url, 'abstract', remove_stopwords = True, lemmatize = True, first_row_num = 1000)
train_abs_20000_df = load_data2(train_url, 'abstract', remove_stopwords = True, lemmatize = True, first_row_num = 20000)

```

```
train_abs_1000_df.head(3)
```

	abstract	processed_abstract
0	Molecule discovery is a pivotal research field...	[molecule, discovery, pivotal, research, field...
1	Counterfactual (CF) explanations for machine l...	[counterfactual, explanation, machine, learnin...

Next steps: [Generate code with train\\_abs\\_1000\\_df](#) [View recommended plots](#) [New interactive sheet](#)

```

docs_train_abs_1000 = train_abs_1000_df['processed_abstract'].tolist()
docs_train_abs_20000 = train_abs_20000_df['processed_abstract'].tolist()
docs_train_abs_1000_raw = copy.deepcopy(train_abs_1000_df['abstract'].tolist())
docs_train_abs_20000_raw = copy.deepcopy(train_abs_20000_df['abstract'].tolist())

```

```
docs_train_abs_20000_raw[1][:50]
```

## 2. Vocab selection

In this section, we aim to build vocabularies by experimenting with the inclusion or exclusion of bigrams for two sets of training data. For the vocabulary consisting of single words, since we are using data from the abstract field, it is expected that many tokens will appear only once. Therefore, we will define a threshold of appear more than **20 times** to filter out infrequent tokens. For the bigram vocabulary, we will apply a minimum frequency threshold of more than **10 occurrences**. This occurrence is relative small for all of size of datasets (1000 and 20000) so it would be a general view for examination

```
# !pip3 install gensim
from gensim.models import Phrases
def add_bigram(docs, docs_name):
    print(f"Processing document: {docs_name}")
    bigram = Phrases(docs, min_count=10)
    print(f'Total bigrams (vocab size): {len(bigram.vocab)}')
    #bigram = Phrases(docs)
    for idx in range(len(docs)):
        for token in bigram[docs[idx]]:
            if '_' in token:
                # Token is a bigram, add to document.
                docs[idx].append(token)
    return docs

docs_train_abs_1000_bi = add_bigram(copy.deepcopy(docs_train_abs_1000), 'docs_train_abs_1000')
docs_train_abs_20000_bi = add_bigram(copy.deepcopy(docs_train_abs_20000), 'docs_train_abs_20000')

train_list = [docs_train_abs_1000, docs_train_abs_1000_bi, docs_train_abs_20000, docs_train_abs_20000_bi]
train_list_with_name = ["docs_train_abs_1000", "docs_train_abs_1000_bi", "docs_train_abs_20000", "docs_train_abs_20000_bi"]
```

```
↗ Processing document: docs_train_abs_1000
Total bigrams (vocab size): 86472
Processing document: docs_train_abs_20000
Total bigrams (vocab size): 992550
```

```
import numpy as np
from collections import Counter
from gensim.corpora import Dictionary

def dictionary_with_doc_percentile(train_docs, doc_names):
    for docs, doc_name in zip(train_docs, doc_names):
        # Create a dictionary for the documents
        dictionary = Dictionary(docs)
        # Filter out words that occur in fewer than 20 documents
        dictionary.filter_extremes(no_below=20)

        # Initialize a Counter for document frequency
        doc_freqs = Counter()

        # Count document frequency: in how many documents each word appears
        for doc in docs:
            word_ids = set(dictionary.doc2idx(doc, unknown_word_index=-1))
            word_ids.discard(-1) # remove unknowns
            for word_id in word_ids:
                doc_freqs[word_id] += 1

        # Calculate document frequency percentiles
        doc_freq_values = list(doc_freqs.values())
        percentile_levels = [5, 20, 30, 40, 50, 60, 70, 80, 95]
        percentiles = np.percentile(doc_freq_values, percentile_levels)

        # Display results
        print(f'Train data: {doc_name}')
        print(f' Unique tokens in corpus: {len(dictionary)}')
        print(f' Document frequency percentiles:')
        for level, value in zip(percentile_levels, percentiles):
            print(f' {level}th percentile: {value:.1f}')
        print()

# Example usage
dictionary_with_doc_percentile(train_list, train_list_with_name)
```

```
↗ Train data: docs_train_abs_1000
Unique tokens in corpus: 860
Document frequency percentiles:
5th percentile: 21.0
20th percentile: 24.0
```

```

30th percentile: 28.0
40th percentile: 31.6
50th percentile: 37.0
60th percentile: 44.0
70th percentile: 55.0
80th percentile: 73.0
95th percentile: 171.1

```

```

Train data: docs_train_abs_1000_bi
Unique tokens in corpus: 902
Document frequency percentiles:
5th percentile: 21.0
20th percentile: 24.0
30th percentile: 27.0
40th percentile: 31.0
50th percentile: 36.5
60th percentile: 43.6
70th percentile: 54.0
80th percentile: 72.0
95th percentile: 166.7

```

```

Train data: docs_train_abs_20000
Unique tokens in corpus: 5304
Document frequency percentiles:
5th percentile: 22.0
20th percentile: 32.0
30th percentile: 42.0
40th percentile: 56.0
50th percentile: 79.0
60th percentile: 111.0
70th percentile: 173.0
80th percentile: 296.0
95th percentile: 1068.0

```

```

Train data: docs_train_abs_20000_bi
Unique tokens in corpus: 7314
Document frequency percentiles:
5th percentile: 21.0
20th percentile: 28.0
30th percentile: 35.0
40th percentile: 44.0
50th percentile: 59.0
60th percentile: 82.0
70th percentile: 124.0
80th percentile: 212.0
95th percentile: 866.0

```

I chose *no\_above*=0.6 based on an analysis of document frequency percentiles across all datasets. Specifically, the 60th percentile in each case corresponds to tokens that appear in approximately **4–5% of the total documents**. This makes it a consistent and objective threshold for filtering out extremely common terms while retaining those that are still broadly representative of meaningful content.

By setting *no\_above*=0.6, we strike a balanced trade-off:

- I exclude only the most frequent words, which often lack topic-distinguishing power.
- I retain a richer and more diverse vocabulary, especially important when working with smaller datasets where overly strict filtering could overly limit the token set.
- It ensures comparability (**around 4-5%**) across datasets of different sizes (1000 vs. 20000), providing a uniform criterion for vocabulary pruning.

Overall, this slightly looser cutoff supports both topic interpretability and model flexibility, particularly when experimenting with different LDA configurations.

Furthermore, we also save all variations into dictionary name **dic\_with\_corpus\_globdaldic** for later usage

```

from gensim.corpora import Dictionary
def dictionary_for_doc(train_docs, doc_names):
    """
    Create a dictionary representation of the documents and filter rare and common tokens.

    Args:
    train_docs (list of list): List of tokenized documents (list of words for each document).
    doc_names (list of str): List of names or identifiers for each document.

    Returns:
    dic_with_doc2bow (dict): Dictionary mapping document names to their Bag-of-Words representation and the dictionary.
    """
    dic_with_corpus_globdaldic = {}

```

```

# Iterate over documents and their names
for doc_name, docs in zip(doc_names, train_docs):
    # Create a global dictionary for all documents
    dictionary = Dictionary(docs)

    # Filter out words that occur in less than 20 documents or more than 50% of the documents
    dictionary.filter_extremes(no_below=20, no_above=0.6)

    # Create Bag-of-Words representation for the document
    corpus = [dictionary.doc2bow(doc) for doc in docs]
    # Print document-wise statistics (optional for debugging)
    print(f'Train data: {doc_name}')
    print(f'  Number of unique tokens in corpus: {len(dictionary)}')
    print('\n')

    # Store both the corpus and the dictionary for each document in a list
    dic_with_corpus_globdaldic[doc_name] = [corpus, dictionary]

return dic_with_corpus_globdaldic
dic_with_corpus_globdaldic = dictionary_for_doc(train_list, train_list_with_name)

➦ Train data: docs_train_abs_1000
  Number of unique tokens in corpus: 860

Train data: docs_train_abs_1000_bi
  Number of unique tokens in corpus: 902

Train data: docs_train_abs_20000
  Number of unique tokens in corpus: 5304

Train data: docs_train_abs_20000_bi
  Number of unique tokens in corpus: 7314

```

### ✓ 3. Training

We are ready to train the LDA model with number of topic is 10. The rest of the set up is similar to tutorial. After that we will save LDA model of each variations

```

from gensim.models import LdaModel
def LDA_model(corpus, dictionary, doc_name):
    # Set training parameters.
    NUM_TOPICS = 10
    chunksize = 5000
    passes = 20
    iterations = 400
    eval_every = None # Don't evaluate model perplexity, takes too much time.

    # Make an index to word dictionary.
    temp = dictionary[0] # This is only to "load" the dictionary.
    id2word = dictionary.id2token

    model = LdaModel(
        corpus=corpus,
        id2word=id2word,
        chunksize=chunksize,
        alpha='auto',
        eta='auto',
        iterations=iterations,
        num_topics=NUM_TOPICS,
        passes=passes,
        eval_every=eval_every
    )

    # Define the output file name based on document name
    outputfile = f'{doc_name}.gensim'
    print(f'Saving model for {doc_name} in {outputfile}')
    model.save(outputfile)

```

```

return model

# Assume dic_with_corpus_gloabaldic contains the corpus and dictionary for each document
# Loop through each document in train_list_with_name and its corresponding corpus/dictionary
for doc_name, (corpus, dictionary) in dic_with_corpus_gloabaldic.items():
    # Train and save the model for each document
    model = LDA_model(corpus, dictionary, doc_name)
    dic_with_corpus_gloabaldic[doc_name].append(model)

Saving model for docs_train_abs_1000 in docs_train_abs_1000.gensim
Saving model for docs_train_abs_1000_bi in docs_train_abs_1000_bi.gensim
Saving model for docs_train_abs_20000 in docs_train_abs_20000.gensim
Saving model for docs_train_abs_20000_bi in docs_train_abs_20000_bi.gensim

# Print the top words for each topic
num_topics = 10
for topic_id in range(num_topics):
    print(f"Topic {topic_id}:")
    print(model.print_topic(topic_id, topn=15)) # topn=10 for top 10 words per topic
    print("\n")

Topic 0:
0.019*"language" + 0.009*"research" + 0.009*"system" + 0.008*"study" + 0.007*"text" + 0.006*"paper" + 0.006*"human" + 0.006*"o

Topic 1:
0.023*"algorithm" + 0.018*"problem" + 0.014*"function" + 0.013*"learning" + 0.009*"method" + 0.009*"distribution" + 0.008*"o

Topic 2:
0.021*"llm" + 0.020*"task" + 0.015*"learning" + 0.011*"performance" + 0.011*"agent" + 0.010*"policy" + 0.010*"method" + 0.00

Topic 3:
0.013*"system" + 0.013*"data" + 0.013*"time" + 0.009*"approach" + 0.009*"dynamic" + 0.008*"prediction" + 0.007*"method" + 0.

Topic 4:
0.024*"graph" + 0.016*"representation" + 0.015*"task" + 0.013*"method" + 0.012*"information" + 0.009*"feature" + 0.009*"prop

Topic 5:
0.023*"network" + 0.016*"training" + 0.015*"neural" + 0.015*"method" + 0.009*"parameter" + 0.009*"performance" + 0.008*"arch

Topic 6:
0.032*"image" + 0.030*"data" + 0.016*"method" + 0.015*"training" + 0.011*"domain" + 0.011*"label" + 0.010*"learning" + 0.010

Topic 7:
0.029*"learning" + 0.025*"data" + 0.014*"user" + 0.011*"system" + 0.011*"algorithm" + 0.008*"machine" + 0.008*"machine_learn

Topic 8:
0.024*"attack" + 0.024*"adversarial" + 0.020*"detection" + 0.018*"data" + 0.011*"privacy" + 0.009*"robustness" + 0.008*"tra

Topic 9:
0.028*"network" + 0.020*"learning" + 0.019*"neural" + 0.017*"deep" + 0.014*"feature" + 0.012*"method" + 0.011*"data" + 0.010

```

## 4. Experiment with document within each topic

In this section, we also try to find document topic of all variations in the experiment and find the document and topic's words in each variation

```

def get_document_topics(ldamodel, corpus, texts, topn=15):
    # Init output
    document_topics_df = pd.DataFrame()
    data = []

    # Get main topic in each document
    for i, row in enumerate(ldamodel[corpus]): # iteration of main topic of each doc
        row = sorted(row, key=lambda x: (x[1]), reverse=True)

```



```
# Get the Dominant topic, Perc Contribution and Keywords for each document
for j, (topic_num, prop_topic) in enumerate(row):
    if j == 0: # => dominant topic
        wp = ldamodel.show_topic(topic_num, topn=topn)
        topic_keywords = ", ".join([word for word, prop in wp]) # get all word
        data.append([int(topic_num), round(prop_topic,4), topic_keywords])
    else:
        break

document_topics_df = pd.DataFrame(data, columns=['Dominant_Topic', 'Perc_Contribution', 'Topic_Keywords'])

# Add original text to the end of the output
document_topics_df['Original_Text'] = pd.Series(texts)

return document_topics_df

def find_top_k_doc(doc_topic_df, k=5):

    doc_topics_sorted_df = pd.DataFrame()

    doc_topic_df_grpd = doc_topic_df.groupby('Dominant_Topic')

    for i, grp in doc_topic_df_grpd:
        doc_topics_sorted_df = pd.concat([doc_topics_sorted_df,
                                           grp.sort_values(['Perc_Contribution'], ascending=[0]).head(k),
                                           axis=0)

    doc_topics_sorted_df.reset_index(drop=True, inplace=True)
    doc_topics_sorted_df.columns = ['Topic_Num', "Topic_Perc_Contrib", "Keywords", "Text"]
    return doc_topics_sorted_df

train_raw_list = [docs_train_abs_1000_raw, docs_train_abs_1000_raw, docs_train_abs_20000_raw, docs_train_abs_20000_raw]


# Initialize the dictionary to store results
document_topics_df_dic = {}

# Iterate over both `dic_with_corpus_globdaldic.items()` and `train_raw_list` simultaneously using zip
for (doc_name, (corpus, dictionary, model)), raw_doc in zip(dic_with_corpus_globdaldic.items(), train_raw_list):
    # Initialize an empty list for each doc_name in the dictionary
    if doc_name not in document_topics_df_dic:
        document_topics_df_dic[doc_name] = []

    # Get document topics
    document_topics_df = get_document_topics(model, corpus, raw_doc)
    document_topics_sorted_df = find_top_k_doc(document_topics_df, 5)

    # Append the result to the dictionary
    document_topics_df_dic[doc_name].append(document_topics_df)
    document_topics_df_dic[doc_name].append(document_topics_sorted_df)
```

```
document_topics_df_dic['docs_train_abs_20000'][0].head(5)
```



	Dominant_Topic	Perc_Contribution	Topic_Keywords	Original_Text
0	5	0.6800	language, task, llm, data, performance, method...	Molecule discovery is a pivotal research field...
1	4	0.2917	algorithm, learning, problem, optimization, me...	Counterfactual (CF) explanations for machine l...
2	2	0.5669	data, method, distribution, function, problem,...	The gauge function, closely related to the ato...
3	4	0.5186	algorithm, learning, problem, optimization, me...	Reinforcement learning (RL) is a promising met...

```
document_topics_df_dic['docs_train_abs_20000'][1].head(50)
```

	Topic_Num	Topic_Perc_Contrib	Keywords	Text	
0	0	0.9925	performance, training, architecture, method, m...	The Mixture of Experts (MoE) framework has bec...	
1	0	0.9908	performance, training, architecture, method, m...	Designing accurate and efficient convolutional...	
2	0	0.9897	performance, training, architecture, method, m...	Ensembling is a simple and popular technique f...	
3	0	0.9817	performance, training, architecture, method, m...	We study the problem of compressing recurrent ...	
4	0	0.9735	performance, training, architecture, method, m...	In this work we introduce a new transformer ar...	
5	1	0.9911	network, neural, training, data, learning, dee...	In this paper, we address the issue of how to ...	
6	1	0.9901	network, neural, training, data, learning, dee...	It is well-known that a deep neural network ha...	
7	1	0.9898	network, neural, training, data, learning, dee...	Adversarial attacks hamper the functionality a...	
8	1	0.9842	network, neural, training, data, learning, dee...	The commercialization of deep learning creates...	
9	1	0.9797	network, neural, training, data, learning, dee...	Increasingly machine learning systems are bein...	
10	2	0.9915	data, method, distribution, function, problem,...	Solving analytically intractable partial diffe...	
11	2	0.9904	data, method, distribution, function, problem,...	We study the concentration of random kernel ma...	
12	2	0.9897	data, method, distribution, function, problem,...	In this paper we study the asymptotics of line...	
13	2	0.9884	data, method, distribution, function, problem,...	In this work we consider the regularization of...	
14	2	0.9880	data, method, distribution, function, problem,...	We study three notions of uncertainty quantifi...	
15	3	0.9792	user, human, system, agent, interaction, bias,...	Self-reinforcing feedback loops are both cause...	
16	3	0.9748	user, human, system, agent, interaction, bias,...	Crowd feedback systems have the potential to s...	
17	3	0.9384	user, human, system, agent, interaction, bias,...	How can multiple humans interact with multiple...	
18	3	0.9004	user, human, system, agent, interaction, bias,...	How do people build up trust with artificial a...	
19	3	0.8998	user, human, system, agent, interaction, bias,...	Home assistant chat-bots, self-driving cars, d...	
20	4	0.9946	algorithm, learning, problem, optimization, me...	Minimax optimal convergence rates for classes ...	
21	4	0.9930	algorithm, learning, problem, optimization, me...	Stochastic model-based methods have received i...	
22	4	0.9922	algorithm, learning, problem, optimization, me...	Stochastic Gradient Descent (SGD) is a popular...	
23	4	0.9918	algorithm, learning, problem, optimization, me...	We address a generalization of the bandit with...	
24	4	0.9918	algorithm, learning, problem, optimization, me...	In this paper, we consider non-convex stochast...	
25	5	0.9940	language, task, llm, data, performance, method...	Recently, natural language generation (NLG) ev...	
26	5	0.9932	language, task, llm, data, performance, method...	Large language models (LLMs) have made signifi...	
27	5	0.9928	language, task, llm, data, performance, method...	Incorporating external knowledge into dialogue...	
28	5	0.9926	language, task, llm, data, performance, method...	In information retrieval (IR), domain adaptati...	
29	5	0.9923	language, task, llm, data, performance, method...	Retrieval-augmented generation (RAG) is an eff...	
30	6	0.9898	graph, representation, learning, method, netwo...	Heterogeneous graph convolutional networks hav...	
31	6	0.9870	graph, representation, learning, method, netwo...	Unsupervised homogeneous network embedding (NE...	
32	6	0.9869	graph, representation, learning, method, netwo...	In this paper, a novel unsupervised low-rank r...	
33	6	0.9863	graph, representation, learning, method, netwo...	Dynamic graphs (DG) describe dynamic interacti...	
34	6	0.9807	graph, representation, learning, method, netwo...	The ability of a graph neural network (GNN) to...	
35	7	0.9891	word, using, language, classification, feature...	Recent approaches for dialogue act recognition...	
36	7	0.9814	word, using, language, classification, feature...	Simultaneous translation is a task in which tr...	
37	7	0.9799	word, using, language, classification, feature...	A comparison of formulaic sequences in human a...	
38	7	0.9794	word, using, language, classification, feature...	In the current work, we explore the enrichment...	
39	7	0.9770	word, using, language, classification, feature...	In this paper, we demonstrate speech recogniti...	
40	8	0.9831	data, research, learning, machine, system, stu...	As machine learning systems become ubiquitous,...	
41	8	0.9781	data, research, learning, machine, system, stu...	Enterprise financial risk analysis aims at pre...	
42	8	0.9746	data, research, learning, machine, system, stu...	Artificial Intelligence (AI) and its data-cent...	
43	8	0.9535	data, research, learning, machine, system, stu...	With the rapid development of information and ...	
44	8	0.9496	data, research, learning, machine, system, stu...	Data visualization is becoming an increasingly...	
45	9	0.9928	image, data, method, time, detection, approach...	Aircraft landing time (ALT) prediction is cruc...	

<b>46</b>	9	0.9897	image, data, method, time, detection, approach...	Anomaly detection for indoor air quality (IAQ)...
<b>47</b>	9	0.9894	image, data, method, time, detection, approach...	To ensure the safety of railroad operations, i...
<b>48</b>	9	0.9646	image, data, method, time, detection, approach...	Recently, breakthroughs in video modeling have...
<b>49</b>	9	0.9605	image, data, method, time, detection, approach...	What if our clothes could capture our body mot...

```
def split_topic_words(df, k = 15):
    """
    This function splits the 'Top_Words' column in the DataFrame into separate columns based on a given number of top words.

    Parameters:
    - df: DataFrame containing 'Topic' and 'Top_Words' columns
    - k: Number of top words to display for each topic

    Returns:
    - DataFrame with each word in 'Top_Words' as a separate column
    """
    df = df.copy()
    # Split the 'Top_Words' column by comma and space, then expand it into separate columns
    words_split = df['Keywords'].str.split(' ', expand=True)

    # If there are more words than 'k', truncate the words to 'k' columns
    words_split = words_split.iloc[:, :k] # Keep only the first 'k' words

    # Concatenate the original 'Topic' column with the new words columns
    df_words = pd.concat([df['Topic_Num'], words_split], axis=1)

    # Rename columns to match the word columns (e.g., Word 0, Word 1, etc.)
    df_words.columns = ['Topic_Num'] + [f'Word {i}' for i in range(df_words.shape[1] - 1)]
    df_words = df_words.drop_duplicates(subset='Topic_Num', keep='first') # Remove duplicates
    return df_words

# Iterate through the dictionary items to retrieve the data
for train_name, topic_data_list in document_topics_df_dic.items():
    topic_word_df = split_topic_words(topic_data_list[1], 15)
    topic_data_list.append(topic_word_df)

# Print out the topic words of each topic in each variation
for train_name, topic_data_list in document_topics_df_dic.items():
    print(f'Train data: {train_name}:')

    # Select the desired DataFrame (assuming the 2nd index contains the sorted topics)
    topic_df = topic_data_list[2].head(10)

    # Display the dataframe
    display(topic_df)
    print('\n')
```

 Train data: docs\_train\_abs\_1000:

	Topic_Num	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9
0	0	network	data	image	neural	method	learning	deep	training	approach	proposed
5	1	language	llm	task	large	knowledge	text	data	information	method	performance
10	2	learning	data	prediction	uncertainty	accuracy	deep	method	query	system	challenge
15	3	task	agent	learning	policy	reward	environment	reinforcement	human	action	system
20	4	data	learning	problem	datasets	training	representation	method	online	performance	feature
25	5	user	data	system	method	study	task	information	approach	based	recommendation
30	6	training	network	architecture	performance	neural	paper	transformer	show	data	inference
35	7	algorithm	learning	problem	method	function	optimization	network	result	show	approach
40	8	method	graph	feature	learning	task	network	performance	propose	classification	image
45	9	entity	data	bias	word	approach	technique	used	sparse	game	embeddings

Train data: docs\_train\_abs\_1000\_bi:

	Topic_Num	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9
0	0	learning	image	method	domain	time	agent	brain	series	training	imagined
5	1	llm	language	large	large_language	task	performance	text	response	however	approach
10	2	image	patient	method	dataset	word	learning	clinical	task	information	segmentation
15	3	learning	data	graph	task	result	algorithm	system	performance	method	use
20	4	network	neural	proposed	result	neural_network	method	data	using	feature	learning
25	5	task	method	representation	data	learning	propose	knowledge	graph	performance	information
30	6	system	user	human	data	research	paper	feedback	study	learning	work
35	7	data	method	network	training	learning	neural	loss	performance	graph	dataset
40	8	algorithm	learning	problem	method	function	policy	show	optimization	network	sample
45	9	system	approach	attack	prediction	object	network	context	dynamic	using	method

Train data: docs\_train\_abs\_20000:

	Topic_Num	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9
0	0	performance	training	architecture	method	memory	time	parameter	transformer	computational	inference
5	1	network	neural	training	data	learning	deep	attack	adversarial	method	performance
10	2	data	method	distribution	function	problem	show	space	approach	result	set
15	3	user	human	system	agent	interaction	bias	behavior	robot	study	feedback
20	4	algorithm	learning	problem	optimization	method	policy	gradient	function	reinforcement	show
25	5	language	task	llm	data	performance	method	large	knowledge	generation	domain
30	6	graph	representation	learning	method	network	feature	information	structure	task	node
35	7	word	using	language	classification	feature	result	text	dataset	medical	translation
40	8	data	research	learning	machine	system	study	application	paper	analysis	tool
45	9	image	data	method	time	detection	approach	using	object	system	prediction

Train data: docs\_train\_abs\_20000\_bi:

	Topic_Num	Word 0	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9
0	0	language	research	system	study	text	paper	human	dataset	question	evaluation
5	1	algorithm	problem	function	learning	method	distribution	optimization	show	bound	
10	2	llm	task	learning	performance	agent	policy	method	knowledge	language	reinforcement
15	3	system	data	time	approach	dynamic	prediction	method	using	process	
20	4	graph	representation	task	method	information	feature	propose	speech	approach	

25	5	network	training	neural	method	parameter	performance	architecture	neural_network	layer	in
30	6	image	data	method	training	domain	label	learning	datasets	performance	
35	7	learning	data	user	system	algorithm	machine	machine_learning	device	communication	fran
40	8	attack	adversarial	detection		data	privacy	robustness	training	anomaly	perturbation
45	9	network	learning	neural	deep	feature	method		data	prediction	classification

Next steps:

Generate code with topic\_df

View recommended plots

New interactive sheet

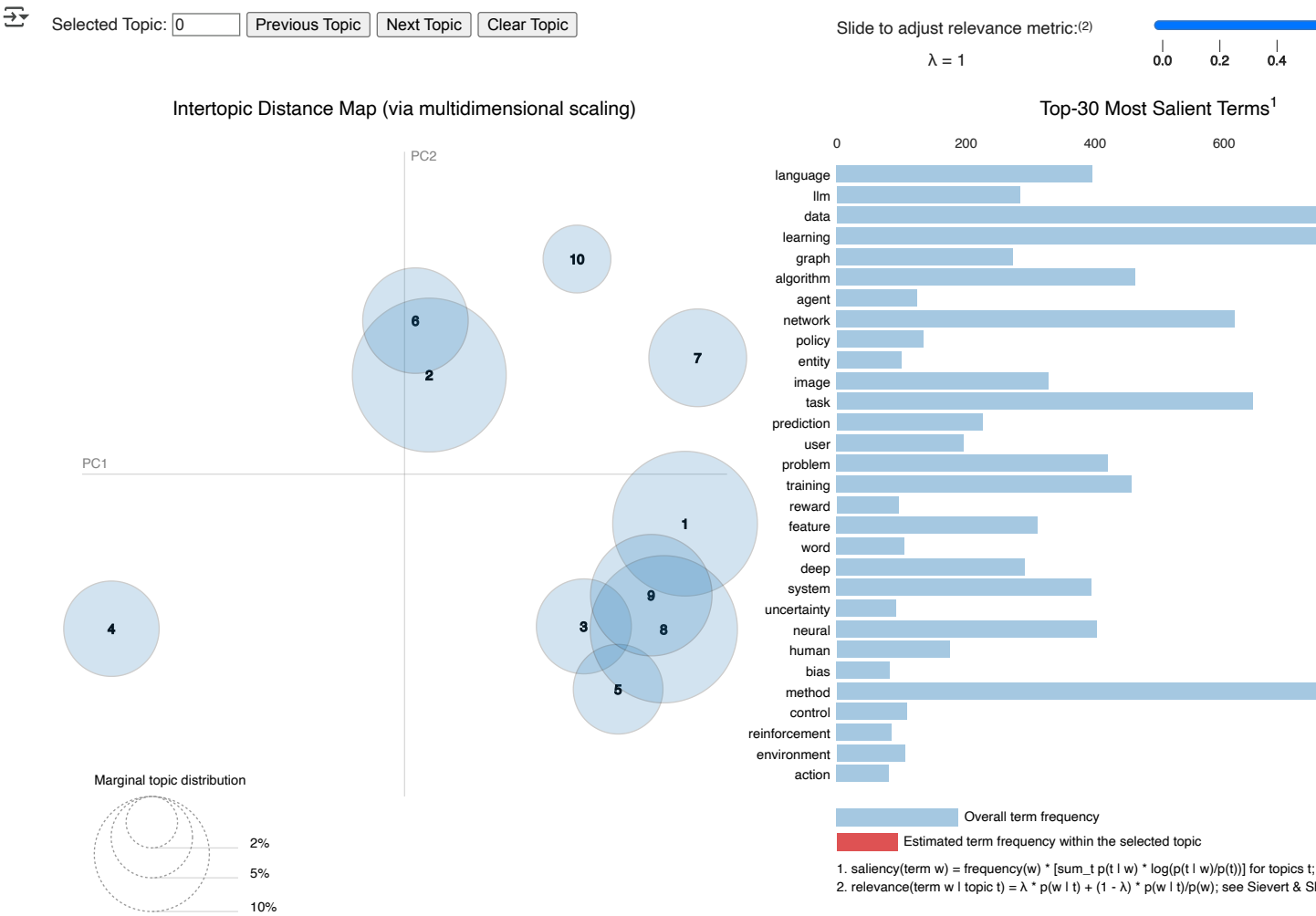
Generate code with topic\_df

View recommended plots

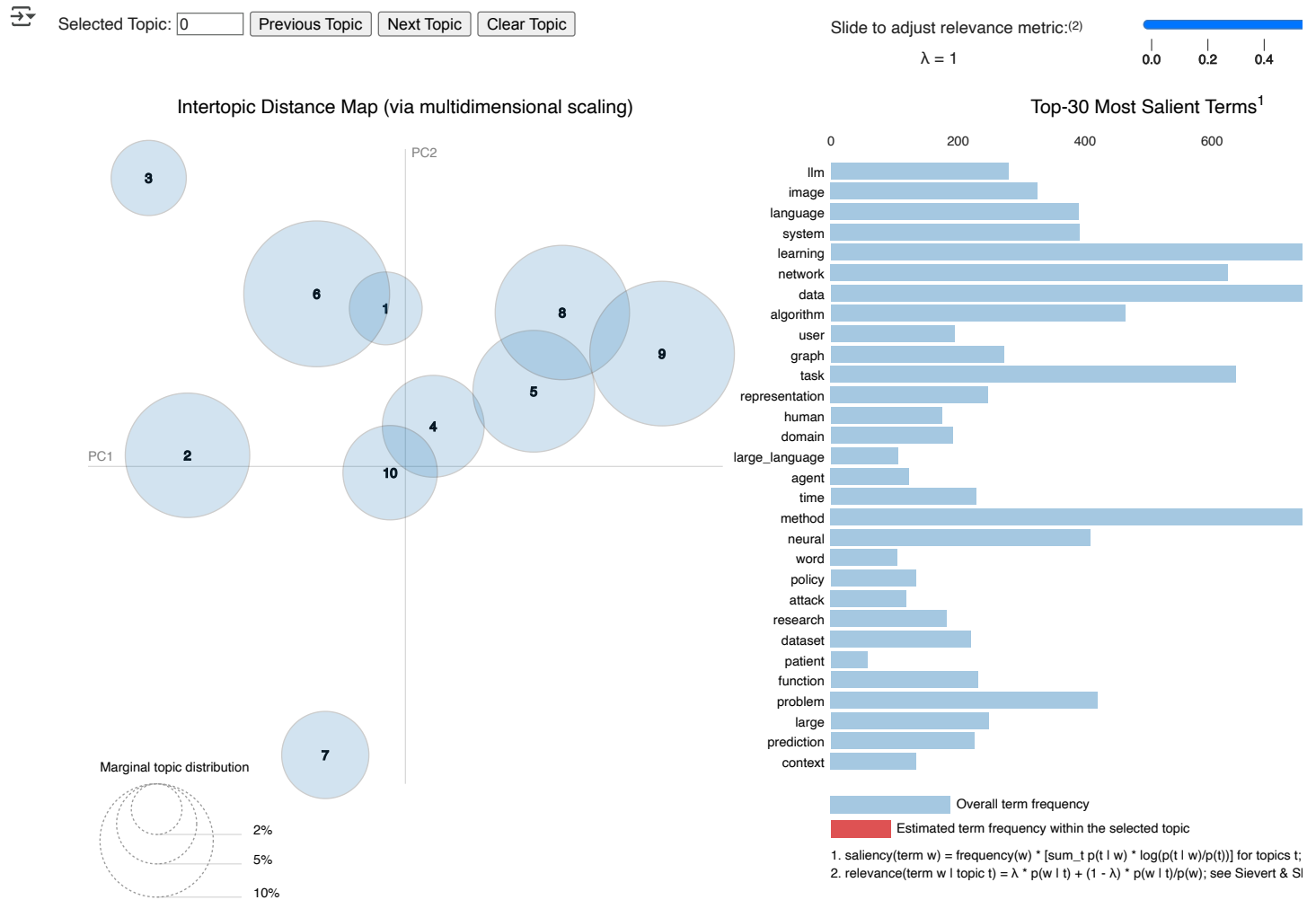
Visualization with pyLDAvis

In this section, we use pyLDAvis to visualize all topics, including word frequency distributions and the intertopic distance map, to better understand topic coherence and separation of all variations

```
import pyLDAvis.gensim
corpus, dictionary, model= dic_with_corpus_glodbaldic['docs_train_abs_1000']
lda_display = pyLDAvis.gensim.prepare(model, corpus, dictionary, sort_topics=False)
pyLDAvis.display(lda_display)
```



```
corpus, dictionary, model= dic_with_corpus_gloabaldic['docs_train_abs_1000_bi']
lda_display = pyLDAvis.gensim.prepare(model, corpus, dictionary, sort_topics=False)
pyLDAvis.display(lda_display)
```



```
corpus, dictionary, model= dic_with_corpus_gloabaldic['docs_train_abs_20000']
lda_display = pyLDAvis.gensim.prepare(model, corpus, dictionary, sort_topics=False)
pyLDAvis.display(lda_display)
```



Selected Topic: 0

Previous Topic

Next Topic

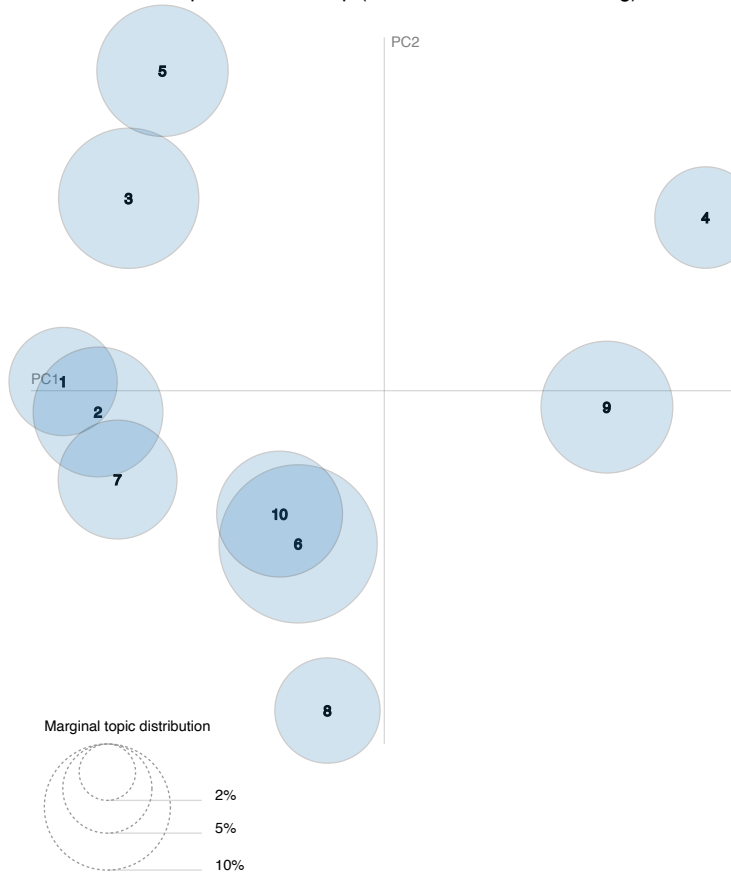
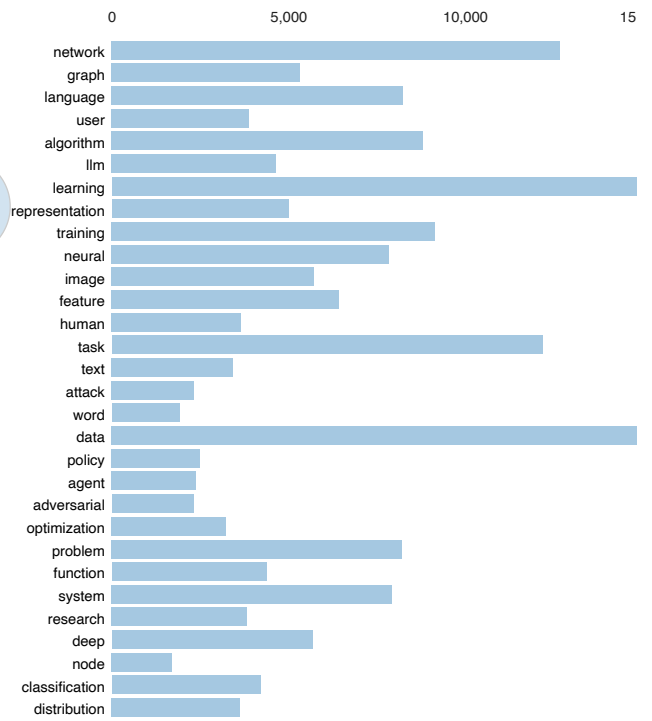
Clear Topic

Slide to adjust relevance metric:(2)

 $\lambda = 1$ 

0.0 0.2 0.4

Intertopic Distance Map (via multidimensional scaling)

Top-30 Most Salient Terms<sup>(1)</sup>

Overall term frequency

Estimated term frequency within the selected topic

1. saliency(term  $w$ ) = frequency( $w$ ) \* [sum<sub>t</sub> p( $t$  |  $w$ ) \* log(p( $t$  |  $w$ )/p( $t$ ))]
2. relevance(term  $w$  | topic  $t$ ) =  $\lambda$  \* p( $w$  |  $t$ ) + (1 -  $\lambda$ ) \* p( $w$  |  $t$ )/p( $w$ ); see Sievert & Si

```
corpus, dictionary, model= dic_with_corpus_gloabaldic['docs_train_abs_20000_bi']
lda_display = pyLDAvis.gensim.prepare(model, corpus, dictionary, sort_topics=False)
pyLDAvis.display(lda_display)
```