Exercise 9

Repeat exercise 11 of Part 2 (text classification with mostly linear classifiers), now using an MLP classifier implemented (by you) in Keras/TensorFlow or PyTorch. 3 You may use different features in the MLP classifier than the ones you used in exercise 11 of Part 2. Tune the hyper-parameters (e.g., number of hidden layers, dropout probability) on the development subset of your dataset. Monitor the performance of the MLP on the development subset during training to decide how many epochs to use. Include experimental results of a baseline majority classifier, as well as experimental results of your best classifier from exercise 11 of Part 2, now treated as a second baseline. Include in your report:

• Curves showing the loss on training and development data as a function of epochs (slide 49).

• Precision, recall, F1, precision-recall AUC scores, for each class and classifier, separately for the training, development, and test subsets, as in exercise 11 of Part 2.

• Macro-averaged precision, recall, F1, precision-recall AUC scores (averaging the corresponding scores of the previous bullet over the classes), for each classifier, separately for the training, development, and test subsets, as in exercise 11 of Part 2.

• A short description of the methods and datasets you used, including statistics about the datasets (e.g., average document length, number of training/dev/test documents, vocabulary size) and a description of the preprocessing steps that you performed. You may optionally wish to try ensembles. One possibility is to use k separate MLP classifiers, corresponding to your k best checkpoints (k best epochs in terms of development loss), and aggregate their decisions by majority voting. Another possibility is to use temporal averaging, i.e., use a single MLP classifier, whose weights are the average of the weights of the k best checkpoints.

Mount Google Drive

To access files stored in Google Drive, we need to mount it in Google Colab.
The drive will be mounted at `/content/drive`, allowing seamless file access.

Explanation:

- **Mounting Drive**: Connects your Google Drive to Colab.
- **Forced Remount**: Ensures that the drive is remounted if already mounted.
- **Use Cases**: Useful for loading datasets, saving models, and storing experiment results persistently.

[ ] ↳ 1 κρυφό κελί

Import Required Libraries

This project utilizes various libraries for data processing, machine learning, and deep learning.

General Libraries

- **matplotlib.pyplot, numpy, pandas**: Essential for data manipulation, visualization, and numerical operations.

PyTorch (Deep Learning)

- **torch, torch.optim, torch.nn, torch.nn.functional**: Provides deep learning capabilities.
- **torch.utils.data**: Facilitates dataset handling and batching.
- **torch.device**: Checks for GPU availability to optimize training.

Scikit-learn (Machine Learning & Evaluation)

- **DummyClassifier**: Baseline majority classifier for comparison.
- **LogisticRegression**: Traditional ML classifier for sentiment analysis.
- **train_test_split**: Splits dataset into training, development, and test sets.
- **TfidfVectorizer**: Converts text into TF-IDF features.
- **Metrics (precision, recall, F1, etc.)**: Used for performance evaluation.

Word Embeddings

- **gensim.downloader**: Downloads pre-trained word embeddings (e.g., GloVe, Word2Vec).

NLP & Text Processing

- **re (Regular Expressions)**: Used for text cleaning.
- **nltk & word_tokenize**: Tokenizes text into words.

Performance Optimization

- **tqdm**: Displays progress bars for efficient iteration tracking.

Device Configuration

- The script checks for GPU availability and assigns computations to `cuda` if available, otherwise defaults to `cpu`.

NLTK Data Download

- Ensures necessary tokenization resources (`punkt`, `punkt_tab`) are available.

`[ ]` ↳ 1 κρυφό κελί

Load and Merge Dataset

Dataset Information

The dataset consists of two CSV files:

- **Fake.csv**: Contains fake news articles.
- **True.csv**: Contains real news articles.

Loading Data

- The datasets are read using **pandas** `read_csv` from Google Drive.
- Each dataset is stored as a DataFrame (`df_fake`, `df_true`).

Labeling Data

- A **binary label** is assigned:
  - **Fake news** → Label `1`
  - **True news** → Label `0`

Merging Data

- Both DataFrames are concatenated along axis `0` (rows), forming a combined dataset (`df_merge`).

Preview Data

- The first 10 rows of `df_merge` are displayed to verify the data structure.

`[ ]` ↳ 3 κρυφά κελιά

Data Preprocessing and Dataset Statistics

Removing Unnecessary Columns

- The columns `title`, `subject`, and `date` are dropped as they are not relevant to the analysis.
- The index is reset, and the old index column is dropped.

Preprocessing Text

A function is defined to preprocess text:

- **Lowercase conversion**: All text is converted to lowercase.
- **Removing numbers**: All digits are removed.
- **Punctuation removal**: Non-alphanumeric characters are discarded.
- **Tokenization**: The text is split into individual tokens (words) using NLTK's `word_tokenize`.

Splitting Data

The dataset is divided into:

- **Training set**: 80% of the data.
- **Validation set**: 10% of the data.
- **Test set**: 10% of the data.
- Stratified splitting ensures that each set maintains the proportion of fake and true news articles.

Checking Class Distribution

The class distribution (number of fake vs. true news) is displayed for each data split.

Preprocessing All Splits

The preprocessing function is applied to the text data in all splits (training, validation, test).

Dataset Statistics Function

The function `dataset_stats_and_preprocessing` provides key statistics about the dataset:

- **Number of documents** in each set.
- **Average document length** (word count) in each set.
- **Vocabulary size**: Unique words across the entire corpus.
- **Preprocessing steps** description: Lists the transformations applied to the text.

The function also prints these statistics for the user to review.

Example Output

The function outputs the following:

- Total number of documents.
- Distribution of fake and true news.
- Document counts for each split.
- Average document length for each split.
- Vocabulary size.

```
#Removing columns which are not required¶
df = df_merge.drop(["title", "subject","date"], axis = 1)

df.reset_index(inplace = True)
df.drop(["index"], axis = 1, inplace = True)
df.head()
```

	text	label
0	Donald Trump just couldn t wish all Americans ...	1
1	House Intelligence Committee Chairman Devin Nu...	1
2	On Friday, it was revealed that former Milwauk...	1
3	On Christmas day, Donald Trump announced that ...	1
4	Pope Francis used his annual Christmas Day mes...	1

```
# Preprocessing function
def preprocess_text(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r'\d+', '', text)  # Remove numbers
    text = re.sub(r'[^\w\s]', '', text)  # Remove punctuation
    tokens = word_tokenize(text)  # Tokenize text
    return ' '.join(tokens)
```

```
from sklearn.model_selection import train_test_split

# Split data into features (X) and target labels (y)
X = df['text'].values
y = df['label'].values

# Split into training, validation, and test sets with stratification
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.1, random_state=2025, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=2025, stratify=y_temp)

# Check the distribution of labels in each set
print("Training set class distribution:", np.bincount(y_train))
print("Validation set class distribution:", np.bincount(y_val))
print("Test set class distribution:", np.bincount(y_test))
```

```
Training set class distribution: [19275 21133]
Validation set class distribution: [1071 1174]
Test set class distribution: [1071 1174]
```

```
 # Preprocess all splits
 X_train = [preprocess_text(text) for text in X_train]
 X_val = [preprocess_text(text) for text in X_val]
 X_test = [preprocess_text(text) for text in X_test]
```

```
import numpy as np
from collections import Counter
import string

# Function to calculate dataset statistics and preprocessing steps
def dataset_stats_and_preprocessing(df, X_train, X_val, X_test):
    # Number of documents
    num_train = len(X_train)
    num_val = len(X_val)
    num_test = len(X_test)
```

```
    # Average document length (in terms of word count)
    avg_train_length = np.mean([len(doc.split()) for doc in X_train])
    avg_val_length = np.mean([len(doc.split()) for doc in X_val])
    avg_test_length = np.mean([len(doc.split()) for doc in X_test])

    # Vocabulary size (unique words in the whole corpus)
    all_text = ' '.join(df['text'].values)
    all_words = all_text.split()
    vocab_size = len(set(all_words))

    # Preprocessing steps description
    preprocessing_steps = """
    1. Text Tokenization: The text is split into words based on spaces. Punctuation is included as part of words (can be changed based on need).
    2. Stop-word Removal: If implemented, remove common words that don't carry useful information (e.g., "the", "and").
    3. Lowercasing: The text is converted to lowercase to ensure uniformity.
    """

    # Displaying the calculated statistics
    print(f"Total documents: {len(df)}")
    print(f"Fake news: {len(df[df['label'] == 1])}, True news: {len(df[df['label'] == 0])}")
    print(f"Number of training documents: {num_train}")
    print(f"Number of validation documents: {num_val}")
    print(f"Number of test documents: {num_test}")
    print(f"Average document length in training set (in words): {round(avg_train_length)}")
    print(f"Average document length in validation set (in words): {round(avg_val_length)}")
    print(f"Average document length in test set (in words): {round(avg_test_length)}")
    print(f"Vocabulary size (unique words): {vocab_size}")


    return {
        "num_train": num_train,
        "num_val": num_val,
        "num_test": num_test,
        "avg_train_length": avg_train_length,
        "avg_val_length": avg_val_length,
        "avg_test_length": avg_test_length,
        "vocab_size": vocab_size,
        "preprocessing_steps": preprocessing_steps
    }

# Example usage:
# Assuming df is your dataframe containing the text and labels
dataset_info = dataset_stats_and_preprocessing(df, X_train, X_val, X_test)
```

```
Total documents: 44898
Fake news: 23481, True news: 21417
Number of training documents: 40408
Number of validation documents: 2245
Number of test documents: 2245
Average document length in training set (in words): 399
Average document length in validation set (in words): 408
Average document length in test set (in words): 402
Vocabulary size (unique words): 397481
```

MLP Model Definition

The **MLP (Multilayer Perceptron)** model is a fully connected neural network used for binary classification.

Model Structure

- The model consists of an input layer, multiple hidden layers, and an output layer.
- Each hidden layer includes:
  - **Linear layer**: Applies a linear transformation (i.e., weighted sum of inputs).
  - **Batch Normalization** (optional): Normalizes the outputs of each hidden layer to improve training stability.
  - **ReLU Activation**: Adds non-linearity, allowing the network to learn more complex patterns.
  - **Dropout** (optional): Randomly drops a fraction of units during training to prevent overfitting.
- The output layer has one unit, which is used for binary classification (fake vs. true news). The output is passed through a **sigmoid activation function** to produce a probability value between 0 and 1.

Class Parameters:

- **`input_dim`**: Number of features in the input.
- **`hidden_dims`**: List defining the number of units in each hidden layer.
- **`dropout`**: Probability of dropout to prevent overfitting (default is 0.0).
- **`batch_norm`**: Boolean flag to include batch normalization (default is False).

Forward Pass

- The `forward` method takes an input tensor `x`, passes it through the defined network layers, and applies the sigmoid activation to the output layer to obtain probabilities.

This architecture is suitable for tasks like binary text classification, such as determining whether news articles are fake or true.

[ ] ↳ 1 κρυφό κελί

MLP Prediction Function

This function uses the trained MLP model to make predictions on a given dataset.

Function Overview

The `predict_mlp` function:

- Sets the model in **evaluation mode** (`model.eval()`) to disable certain features like dropout.
- Iterates over the `data_loader` to process input data in batches.
- Moves each batch of inputs to the appropriate device (CPU or GPU) using `inputs.to(device)`.
- Feeds the inputs through the model to get the outputs.
- Applies the **sigmoid function** to convert the raw model outputs into probabilities between 0 and 1.
- Collects the predictions and returns them as a flattened numpy array.

Parameters:

- **`model`**: The trained MLP model.
- **`data_loader`**: DataLoader containing the input data in batches.
- **`device`**: The device to run the model on (either "cuda" for GPU or "cpu").

Output:

- A numpy array of predicted probabilities for each sample, where values close to 1 indicate a higher likelihood of the positive class (fake news in this case).

[ ] ↳ 1 κρυφό κελί

Word Embeddings for Text Features

Embedding Models

The following pre-trained word embedding models are used:

- **Word2Vec (Google News)**: Trained on Google News dataset, providing 300-dimensional word vectors.
- **GloVe (Wikipedia + Gigaword)**: Trained on a combination of Wikipedia and the Gigaword dataset, also providing 300-dimensional word vectors.

Loading Embeddings

The embeddings are loaded using the **gensim API**:

- The `load_embeddings` function loads the specified pre-trained model (either Word2Vec or GloVe) from the gensim API.

Word2Vec Features Extraction

- **Function**: `get_word2vec_features`
- **Process**:
  - For each text in the dataset, it is tokenized into words.
  - Each word is converted into a 300-dimensional vector using the **Word2Vec model**.
  - The vectors for all words in the text are averaged to get a single feature vector for that text.
  - If no word from the text exists in the model's vocabulary, a zero vector of size 300 is used.

GloVe Features Extraction

- **Function**: `get_glove_features`
- **Process**: Similar to Word2Vec, but uses the **GloVe model** for word embeddings.

Training Set Features Extraction

- The **Word2Vec** and **GloVe** embeddings are applied to the training data (`X_train`), generating feature matrices:
  - `word2vec_train_features`: Contains the averaged Word2Vec vectors for each training sample.
  - `glove_train_features`: Contains the averaged GloVe vectors for each training sample.

These feature matrices can then be used as input for further model training or evaluation.

[ ] ↳ 1 κρυφό κελί

Plotting Loss Curves for Multiple Models and Representations

Purpose

The `plot_loss_curves` function visualizes the training and validation loss curves for multiple models and representations over multiple epochs. This is helpful in comparing the performance of different models and embeddings (such as TF-IDF, Word2Vec, and GloVe).

Parameters:

- **`results`**: A dictionary containing loss data for each model and representation combination. The dictionary should have the following structure:
  - Keys: Model and representation names, e.g., `"Simple Deep MLP (TF-IDF)"`.
  - Values: A dictionary with keys `"train_losses"` and `"val_losses"`, each containing a list of losses over the epochs.

Process:

- **Models**: The function considers three models ("Simple Deep MLP", "Wide MLP", and "Wide & Deep MLP"). You can modify the list to match the actual models in your results.
- **Representations**: The function also evaluates three representations ("TF-IDF", "Word2Vec", and "GloVe"), which can be adjusted based on your use case.
- **Subplots**: For each combination of model and representation, the function creates a subplot that shows:
  - **Training Loss** (solid blue line).
  - **Validation Loss** (dashed orange line).
  - **Annotations**: Each point on the curves is annotated with its respective loss value for clarity.
  - **Y-Axis Customization**: The y-axis ticks are set to show the min, midpoint, and max loss values for better readability.

Output:

- **Visualized Curves**: The function generates a grid of plots showing the loss curves for each model and representation. Each plot displays the corresponding loss values for training and validation across epochs.

`[ ]` ↳ 1 κρυφό κελί

EarlyStopping Class

Purpose

The `EarlyStopping` class helps prevent overfitting by stopping the training process early if the model's validation loss stops improving for a set number of epochs (defined by `patience`). It is commonly used in training neural networks to avoid wasting resources on unnecessary epochs when further improvements in the model's performance are unlikely.

Parameters:

- **`patience`**: The number of epochs to wait for improvement in the validation loss before stopping training. Default is 5.
- **`delta`**: The minimum change in validation loss to qualify as an improvement. If the change is smaller than `delta`, it is considered as no improvement. Default is 0.

Attributes:

- **`best_loss`**: Stores the best (lowest) validation loss observed so far.
- **`counter`**: Tracks the number of epochs without improvement in validation loss.
- **`stop_training`**: Boolean flag indicating whether training should stop.

Methods:

- **`__call__(self, val_loss)`**: This method is called at the end of each epoch with the current validation loss (`val_loss`).
  - If the `best_loss` is `None`, it sets it to the first observed `val_loss`.
  - If the current `val_loss` is not better than the `best_loss - delta`, it increments the counter.
  - If the counter exceeds the `patience` value, it sets `stop_training` to `True`, signaling the training process to halt.
  - If the current `val_loss` is better, it resets the counter and updates `best_loss`.

`[ ]` ↳ 1 κρυφό κελί

Model Training and Evaluation with Different Text Representations

The following code provides a comprehensive framework for training and evaluating multiple models using various text representations like TF-IDF, Word2Vec, and GloVe. The models include:

- **Dummy Majority Classifier**
- **Logistic Regression**
- **Multiple MLP (Multi-Layer Perceptron) models** with different architectures.

1. Text Representations

Three types of text representations are used in the model:

- **TF-IDF**: This uses `TfidfVectorizer` from `sklearn` to extract important features from the text.
- **Word2Vec**: Pre-trained embeddings from `word2vec-google-news-300` are used to convert words into vectors.
- **GloVe**: Pre-trained embeddings from `glove-wiki-gigaword-300` are used for word vectorization.

2. MLP Models

Three types of MLP models with varying complexities are defined:

- **Simple Deep MLP**: A basic MLP with 3 hidden layers.
- **Wide MLP**: A more complex MLP with a wider architecture.
- **Wide & Deep MLP**: A model with both a wider architecture and additional regularization (dropout and batch normalization).

3. Training Procedure

The training loop iterates through each representation and model:

- For **MLP models**, the loss function used is `BCEWithLogitsLoss` (Binary Cross Entropy with Logits), and the optimizer is `Adam`. Early stopping is applied to prevent overfitting.
- **Dummy Majority Classifier** and **Logistic Regression** are used as baselines for comparison.

4. Evaluation Metrics

The following evaluation metrics are calculated:

- **Precision-Recall AUC**: Calculated for each model to evaluate performance in terms of precision and recall.
- **Classification Reports**: Includes precision, recall, F1 score, and support for both classes (True News and Fake News).

5. Results

The results are stored in a dictionary and displayed after training. Each model's:

- Classification reports (for training, validation, and test sets)
- Precision-Recall AUC scores for training, validation, and test sets are printed.

Suggestions for Improvement

1. **Handling Imbalanced Data**: You can apply class weights in models like logistic regression or MLP to address class imbalance.
2. **Model Checkpointing**: Consider saving the best model based on validation performance to avoid overfitting.
3. **Hyperparameter Tuning**: Experiment with tuning hyperparameters like learning rate and layer sizes for the MLP models.
4. **Early Stopping Optimization**: Ensure that the `EarlyStopping` class is functioning as intended, resetting the counter on improvements.

[ ] ↳ 2 κρυφά κελιά

Performance Metrics Comparison Across Feature Representations (TF-IDF, Word2Vec, GloVe)

This code generates a series of bar plots to compare the performance of various models based on their feature representations: TF-IDF, Word2Vec, and GloVe. The three metrics used for comparison are:

1. **Accuracy**
2. **Weighted F1-score**
3. **PR AUC (Precision-Recall AUC)**

The performance metrics for each model are extracted from the classification report, and the following steps are performed:

1. Extracting Metrics:

The `extract_metrics` function is used to extract the accuracy and weighted F1-score from the classification report for each model. These metrics are parsed using regular expressions.

2. Grouping Models by Feature Representation:

Models are grouped based on the feature representation they use. This is done by creating a dictionary `feature_groups` where each key corresponds to a feature type (TF-IDF, Word2Vec, GloVe), and the value is another dictionary containing model names and their corresponding performance data.

3. Plotting the Results:

- For each feature representation (TF-IDF, Word2Vec, GloVe), the models' performance on the three metrics (accuracy, weighted F1-score, and PR AUC) is extracted.
- A set of bar plots is created for each feature representation. Each plot compares the performance of the models across three datasets: **Train**, **Validation**, and **Test**.

4. Plot Customization:

- Each bar in the plot represents one of the performance metrics (accuracy, F1-score, or PR AUC) for a specific model.
- The heights of the bars are annotated with their corresponding values for clarity.
- Separate colors are used for each dataset (Train, Validation, and Test) to distinguish between them.
- The x-axis displays the model names, and the y-axis shows the performance scores, with a fixed range of 0 to 1 for all metrics to ensure consistent scaling.

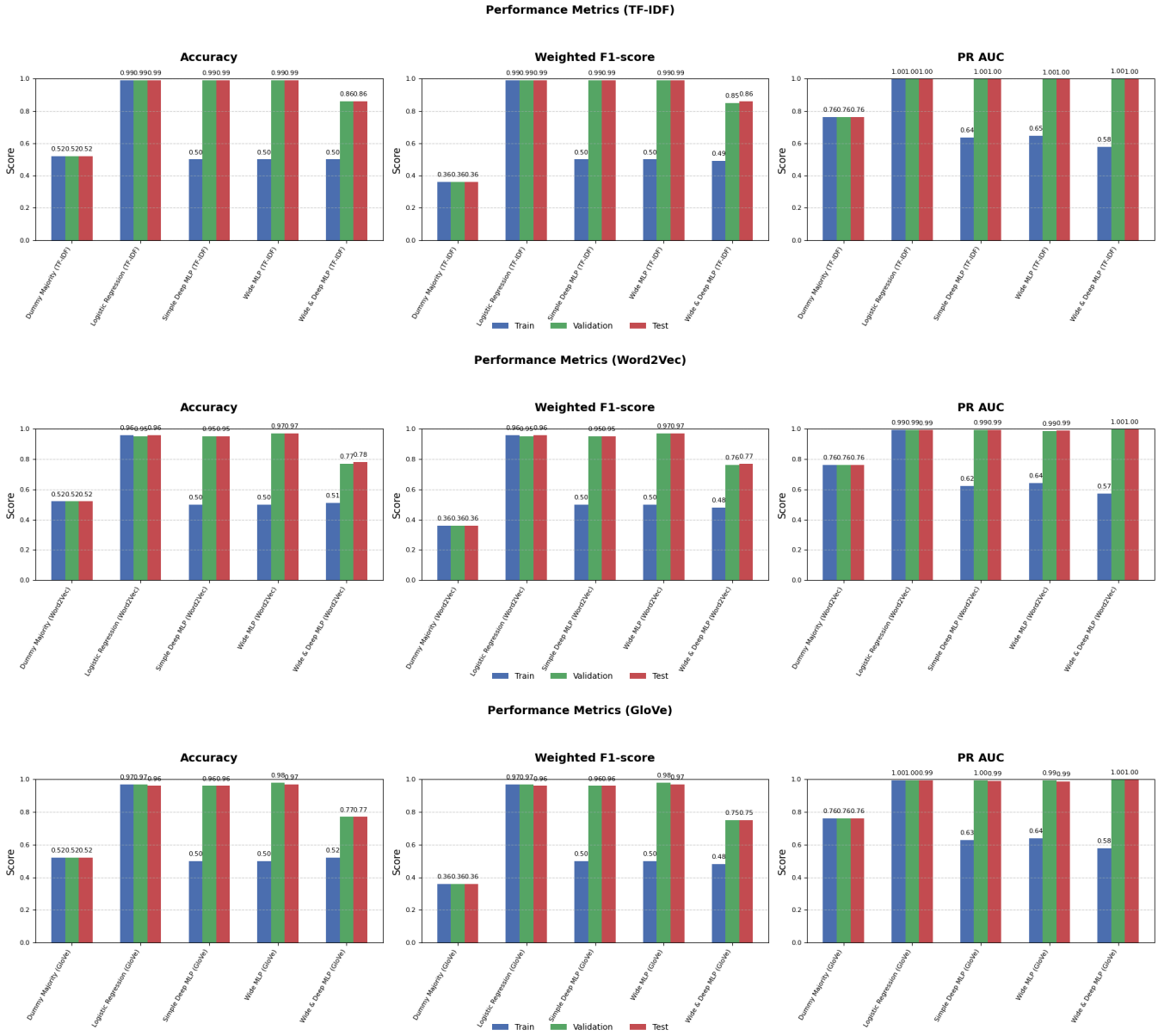
5. Final Visualization:

- The plots are arranged in a single row with three subplots, one for each metric.
- A combined legend is added below the plots to indicate which bars correspond to Train, Validation, and Test datasets.

These plots provide a visual comparison of how each model performs across different feature representations and datasets. This is helpful in selecting the best-performing model.

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import re

# Function to extract accuracy and weighted F1-score from classification report
def extract_metrics(report):
    accuracy_match = re.search(r'accuracy\s+([\d\.]+)', report)
    weighted_f1_match = re.search(r'weighted avg\s+[\d\.]+\s+[\d\.]+\s+([\d\.]+)', report)

    accuracy = float(accuracy_match.group(1)) if accuracy_match else None
    weighted_f1 = float(weighted_f1_match.group(1)) if weighted_f1_match else None

    return accuracy, weighted_f1

# Group models by feature representation (TF-IDF, Word2Vec, GloVe)
feature_groups = {"TF-IDF": {}, "Word2Vec": {}, "GloVe": {}}

for model_name, model_data in results.items():
    if "TF-IDF" in model_name:
        feature_groups["TF-IDF"][model_name] = model_data
    elif "Word2Vec" in model_name or "W2V" in model_name:
        feature_groups["Word2Vec"][model_name] = model_data
    elif "GloVe" in model_name:
        feature_groups["GloVe"][model_name] = model_data

# Use Seaborn color palette
sns_colors = sns.color_palette("deep")  # Seaborn deep palette
colors = [sns_colors[0], sns_colors[2], sns_colors[3]]  # Pick 3 distinct colors

# Iterate over each feature representation
for feature, models_dict in feature_groups.items():
    if not models_dict:  # Skip if no models exist for this feature
        continue

    models = list(models_dict.keys())
    train_acc, val_acc, test_acc = [], [], []
    train_f1, val_f1, test_f1 = [], [], []
    train_pr_auc, val_pr_auc, test_pr_auc = [], [], []

    for model in models:
        metrics = models_dict[model]['metrics']

        # Extract accuracy and F1-score
        acc_train, f1_train = extract_metrics(metrics['train_report'])
        acc_val, f1_val = extract_metrics(metrics['val_report'])
        acc_test, f1_test = extract_metrics(metrics['test_report'])

        # Append values
        train_acc.append(acc_train)
        val_acc.append(acc_val)
        test_acc.append(acc_test)

        train_f1.append(f1_train)
        val_f1.append(f1_val)
        test_f1.append(f1_test)

        # Extract PR AUC
        train_pr_auc.append(metrics['pr_auc']['train_pr_auc'])
        val_pr_auc.append(metrics['pr_auc']['val_pr_auc'])
        test_pr_auc.append(metrics['pr_auc']['test_pr_auc'])

    # Define the metric data for plotting
    metric_data = [
        ("\nAccuracy\n", train_acc, val_acc, test_acc),
        ("\nWeighted F1-score\n", train_f1, val_f1, test_f1),
        ("\nPR AUC\n", train_pr_auc, val_pr_auc, test_pr_auc)
```

```
]

# Create figure for this feature representation
fig, axes = plt.subplots(1, 3, figsize=(20, 6), sharey=False)  # Larger figure size
fig.suptitle(f"\nPerformance Metrics ({feature})\n", fontsize=14, fontweight="bold")

x = np.arange(len(models))
width = 0.2  # Smaller width for spacing

for i, (title, train_vals, val_vals, test_vals) in enumerate(metric_data):
    ax = axes[i]

    bars1 = ax.bar(x - width, train_vals, width, label="Train", color=colors[0])
    bars2 = ax.bar(x, val_vals, width, label="Validation", color=colors[1])
    bars3 = ax.bar(x + width, test_vals, width, label="Test", color=colors[2])

    # Add value labels on bars
    for bars in [bars1, bars2, bars3]:
        for bar in bars:
            height = bar.get_height()
            ax.annotate(f'{height:.2f}',
                        xy=(bar.get_x() + bar.get_width() / 2, height),
                        xytext=(0, 5),  # Offset for visibility
                        textcoords="offset points",
                        ha='center', va='bottom', fontsize=8)  # Smaller font size for annotations

    # Set labels and formatting
    ax.set_xticks(x)
    ax.set_xticklabels(models, rotation=60, ha="right", fontsize=10)  # Smaller font size for x labels
    ax.set_title(title, fontsize=14, fontweight="bold")
    ax.set_ylabel("Score", fontsize=12)
    ax.set_ylim(0, 1)  # Ensures consistent scaling
    ax.grid(axis='y', linestyle='--', alpha=0.6)

    # Adjust tick size
    ax.tick_params(axis='both', which='major', labelsize=8)  # Smaller tick marks

# Add a single legend for the whole row
handles, labels = axes[0].get_legend_handles_labels()
fig.legend(handles, labels, loc="lower center", ncol=3, fontsize=10, frameon=False)

plt.subplots_adjust(bottom=0.3, top=0.85)
plt.tight_layout()
plt.show()
```


Performance Metrics (TF-IDF)
Accuracy
Weighted F1-score
PR AUC
Score
Dummy Majority (TF-IDF)
Logistic Regression (TF-IDF)
Simple Deep MLP (TF-IDF)
Wide MLP (TF-IDF)
Wide & Deep MLP (TF-IDF)
Train
Validation
Test
Performance Metrics (Word2Vec)
Accuracy
Weighted F1-score
PR AUC
Score
Dummy Majority (Word2Vec)
Logistic Regression (Word2Vec)
Simple Deep MLP (Word2Vec)
Wide MLP (Word2Vec)
Wide & Deep MLP (Word2Vec)
Train
Validation
Test
Performance Metrics (GloVe)
Accuracy
Weighted F1-score
PR AUC
Score
Dummy Majority (GloVe)
Logistic Regression (GloVe)
Simple Deep MLP (GloVe)
Wide MLP (GloVe)
Wide & Deep MLP (GloVe)
Train
Validation
Test

Exercise 10

Develop a part-of-speech (POS) tagger for one of the languages of the Universal Dependencies treebanks (http://universaldependencies.org/), using an MLP operating on windows of words. Consider only the words, sentences, and POS tags of the treebanks (not the dependencies or other annotations).

Use Keras/TensorFlow or PyTorch to implement the MLP. You may use any types of word features you prefer, but it is recommended to use pre-trained word embeddings.

Make sure that you use separate training, development, and test subsets. Tune the hyper-parameters (e.g., number of hidden layers, dropout probability) on the development subset. Monitor the performance of the MLP on the development subset during training to decide how many epochs to use.

Include experimental results of a baseline that tags each word with the most frequent tag it had in the training data; for words that were not encountered in the training data, the baseline should return the most frequent tag (over all words) of the training data.

Include in your report:

• Curves showing the loss on training and development data as a function of epochs

• Precision, recall, F1, precision-recall AUC scores, for each class (tag) and classifier,

• Macro-averaged precision, recall, F1, precision-recall AUC scores (averaging the corresponding scores of the previous bullet over the classes), for each classifier, separately for the training, development, and test subsets.

• A short description of the methods and datasets you used, including statistics about the datasets (e.g., average sentence length, number of training/dev/test sentences and words, vocabulary size) and a description of the preprocessing steps.

Imports

[ ] ↳ 1 κρυφό κελί

Data Download

```
# URLs for the train, dev, and test data
urls = {
    "train": "https://raw.githubusercontent.com/UniversalDependencies/UD_English-EWT/master/en_ewt-ud-train.conllu",
    "dev": "https://raw.githubusercontent.com/UniversalDependencies/UD_English-EWT/master/en_ewt-ud-dev.conllu",
    "test": "https://raw.githubusercontent.com/UniversalDependencies/UD_English-EWT/master/en_ewt-ud-test.conllu"
}

# Function to download data from URLs
def download_data(url, filename):
    if os.path.exists(filename):
        print(f"{filename} already exists, skipping download.")
        return

    try:
        print(f"Downloading {filename}...")
        response = requests.get(url)
        response.raise_for_status()  # Raise an error for bad status codes
        with open(filename, 'w', encoding='utf-8') as file:
            file.write(response.text)
        print(f"Downloaded {filename}")
    except requests.exceptions.RequestException as e:
        print(f"Error downloading {filename}: {e}")

# Download train, dev, and test data
download_data(urls['train'], 'en_ewt-ud-train.conllu')
download_data(urls['dev'], 'en_ewt-ud-dev.conllu')
download_data(urls['test'], 'en_ewt-ud-test.conllu')
```

```
en_ewt-ud-train.conllu already exists, skipping download.
en_ewt-ud-dev.conllu already exists, skipping download.
en_ewt-ud-test.conllu already exists, skipping download.
```

Data Parsing and Preprocessing

```
def parse_conllu_file(filename):
    """
    Parse a .conllu file to extract words and their POS tags.
    Returns a list of sentences, where each sentence is a list of tuples (word, pos_tag).
    """
    sentences = []
    sentence = []

    with open(filename, 'r', encoding='utf-8') as file:
        for line in file:
            # Skip empty lines and comments
            if line.strip() == "" or line.startswith("#"):
                continue

            # Split each line into columns (word, pos_tag, etc.)
            columns = line.strip().split("\t")
            word = columns[1]
            pos_tag = columns[3]

            sentence.append((word, pos_tag))

            # Check if we reached the end of a sentence (a line with only space or empty)
            if len(columns) == 1:
                if sentence:
                    sentences.append(sentence)
                sentence = []

    # In case the last sentence doesn't have an empty line at the end
    if sentence:
        sentences.append(sentence)

    return sentences


train_data = parse_conllu_file('en_ewt-ud-train.conllu')
dev_data = parse_conllu_file('en_ewt-ud-dev.conllu')
test_data = parse_conllu_file('en_ewt-ud-test.conllu')

print(train_data[:1])  # Print first sentence for inspection
```

```
[[('Al', 'PROPN'), ('-', 'PUNCT'), ('Zaman', 'PROPN'), (':', 'PUNCT'), ('American', 'ADJ'), ('forces', 'NOUN'), ('killed
```

```
def display_words_per_tag(data):
    """
    Display 5 words for each POS tag.
    """
    tag_to_words = {tag: [] for tag in set(tag for sentence in data for _, tag in sentence)}

    # Collect words for each POS tag
    for sentence in data:
        for word, tag in sentence:
            tag_to_words[tag].append(word)

    # Display 5 random words for each tag
    for tag, words in tag_to_words.items():
        print(f"\n{tag}:")
        sample_words = random.sample(words, min(5, len(words)))  # Get 5 words or less if not enough words
        for word in sample_words:
            print(f"  - {word}")

# Display words for each tag in the training dataset
print("Sample Words for Each Tag:")
display_words_per_tag(train_data)
```

```
Sample Words for Each Tag:
Sample words for each POS tag:

AUX:
  - 'm
  - would
  - be
  - was
  - can

PROPN:
  - Gas
  - SAP
  - Maryam
  - Road
  - Qaeda

SCONJ:
```

```
        - than
        - of
        - while
        - since
        - that

      CCONJ:
        - and
        - and
        - and
        - and
        - or

      SYM:
        - /
        - %
        - :)
        - %
        - $

      PUNCT:
        - ?
        - .
        - .
        - ,
        - .

      NUM:
        - 24
        - 2.7
        - 2
        - 4
        - four

      X:
        - fall
        - -
        - day
        - forma
        - Pricing
```

```
def extract_context_windows(sentences, word2vec_model, window_size=3):
    data = []
    for sentence in sentences:
        # Loop over each word in the sentence, using it as the target word
        for i, (word, pos_tag) in enumerate(sentence):
            # Create a context window of size 3
            context = []
            for j in range(-window_size, window_size + 1):
                if i + j >= 0 and i + j < len(sentence) and j != 0:
                    context.append(sentence[i + j][0])  # Add context word (excluding the target word)

            # Convert the context words to embeddings (Word2Vec)
            context_embeddings = []
            for context_word in context:
                if context_word in word2vec_model:
                    context_embeddings.append(word2vec_model[context_word])
                else:
                    context_embeddings.append(np.zeros(word2vec_model.vector_size))  # For unknown words

            # Aggregate context embeddings into a single vector (average of all context word embeddings)
            context_vector = np.mean(context_embeddings, axis=0)  # Averaging the context words' embeddings
            target_word_embedding = word2vec_model[word] if word in word2vec_model else np.zeros(word2vec_model.vector_size)

            # Combine the context and target word embeddings into one feature vector
            feature_vector = np.concatenate([context_vector, target_word_embedding])

            # Append the feature vector along with the corresponding POS tag
            data.append((feature_vector, pos_tag))

    return data

# Load the pre-trained Word2Vec model (using the example from gensim)
import gensim.downloader as api
word2vec = api.load("word2vec-google-news-300")  # 300-dimensional Word2Vec embeddings

# Extract training, dev, and test data
train_features = extract_context_windows(train_data, word2vec)
dev_features = extract_context_windows(dev_data, word2vec)
test_features = extract_context_windows(test_data, word2vec)



# Prepare the data
def prepare_data(features):
    # Extract feature vectors and labels
    X = [f[0] for f in features]
    y = [f[1] for f in features]

    # Convert feature vectors into tensors
    X_tensor = torch.tensor(X, dtype=torch.float32)

    # Encode POS tags as integers
    label_encoder = LabelEncoder()
```

```
    y_encoded = label_encoder.fit_transform(y)
    y_tensor = torch.tensor(y_encoded, dtype=torch.long)

    return TensorDataset(X_tensor, y_tensor)

# Prepare the datasets for training, dev, and test
train_dataset = prepare_data(train_features)
dev_dataset = prepare_data(dev_features)
test_dataset = prepare_data(test_features)

# Create DataLoader for batching
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
<ipython-input-5-ce01f72b587b>:11: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely slow. Please
  X_tensor = torch.tensor(X, dtype=torch.float32)
```

MLPs Architecture, Training and Evaluation

```
class ShallowPOS_MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.3):
        super(ShallowPOS_MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)  # Single hidden layer
        self.dropout = nn.Dropout(dropout)
        self.fc2 = nn.Linear(hidden_dim, output_dim)  # Output layer
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))  # First layer with ReLU activation
        x = self.dropout(x)  # Apply dropout for regularization
        x = self.fc2(x)  # Output layer
        return x



class DeepPOS_MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.3):
        super(DeepPOS_MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)  # Second hidden layer
        self.fc3 = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))  # Second hidden layer with ReLU activation
        x = self.dropout(x)
        x = self.fc3(x)
        return x



class VeryDeepPOS_MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.3):
        super(VeryDeepPOS_MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))  # Third hidden layer with ReLU activation
        x = self.dropout(x)
        x = self.fc4(x)
        return x
```

Why We Picked These Specific Architectures

We chose three MLP architectures (Shallow, Deep, and Very Deep) to explore different levels of model complexity and expressiveness:

1. **Shallow MLP (1 hidden layer)**:
    - **Simplicity**: Serves as a baseline model, ideal for smaller datasets or simpler tasks.
    - **Efficiency**: Computationally inexpensive and faster to train.
    - **Lower risk of overfitting**: Suitable for less complex tasks.
2. **Deep MLP (2 hidden layers)**:
    - **Increased expressiveness**: Can capture more complex relationships in data.

- **Better performance**: Ideal for these datasets where complexity increases but overfitting needs to be avoided.
- **Balanced complexity**: More capable than shallow models while still being manageable.

3. **Very Deep MLP (3 hidden layers)**:
   - **High capacity**: Captures intricate and abstract features, beneficial for large or complex datasets.
   - **Better pattern recognition**: Useful for tasks where deep hierarchies and complex relationships are needed.

Adding Other Types of Layers

1. **Convolutional Layers**:
   - **Reason**: To capture local patterns (e.g., in text or time-series data).
   - **Benefit**: Helps identify local dependencies like neighboring word relationships in POS tagging.
2. **Recurrent Layers (LSTM/GRU)**:
   - **Reason**: Captures long-term dependencies in sequential data like POS tagging.
   - **Benefit**: Improves context-based predictions for tasks with temporal or sequential patterns.
3. **Attention Mechanisms**:
   - **Reason**: Focuses on important parts of the input for better predictions.
   - **Benefit**: Improves performance by emphasizing relevant parts of the data.
4. **Embedding Layers**:
   - **Reason**: Converts words into dense vector representations for better semantic understanding.
   - **Benefit**: Enhances model generalization and improves performance on NLP tasks.

Why We Didn't Add More Layers

The decision to keep the models simple with only fully connected (dense) layers was based on several key factors:

1. **Model Complexity and Overfitting**:
   - Adding too many layers can lead to overfitting, especially on smaller or simpler datasets. The more layers you introduce, the more parameters the model has, which increases the risk of memorizing the training data rather than learning generalizable patterns.
   - By limiting the layers, we maintain a balance between model complexity and generalization which is particularly important for avoiding overfitting.
2. **Computational Efficiency**:
   - The goal was to ensure efficient training and inference. Adding layers like convolutional or recurrent layers increases both the time and memory complexity of the model.
3. **Simplicity and Interpretability**:
   - Shallow architectures, like the one used here, are often easier to interpret and debug. By sticking to a basic MLP structure, we reduce the complexity of the model and make it easier to track how it's learning.
4. **Suitability for Task**:
   - For POS tagging, which can be performed well with dense, fully connected layers, we felt that additional layers such as convolutional, recurrent, or attention layers would not offer a significant improvement, especially in comparison to the added complexity and computational cost. These simpler MLP architectures are often sufficient for such tasks.
5. **Starting Point for Experimentation**:
   - The chosen architectures (Shallow, Deep, Very Deep) provide a clear and structured approach to explore model depth and performance. Introducing more types of layers could complicate early experimentation, and we wanted to first evaluate how increasing depth within fully connected layers impacts the model's performance before moving on to more complex architectures.

In conclusion, keeping the architecture focused on fully connected layers strikes a balance between performance and simplicity, avoiding unnecessary complexity for the given task and dataset size.

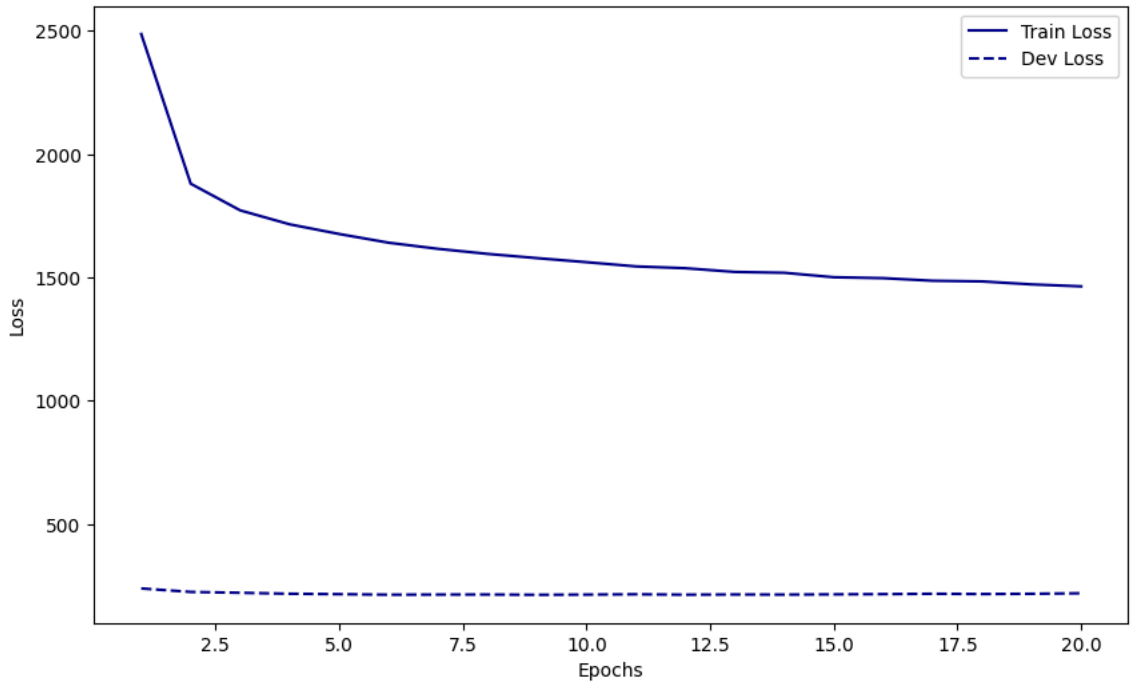
```
# Get the unique POS tags from the training data
pos_tags = list(set([tag for _, tag in train_features]))
pos_tags.sort()  # Sort for a consistent order

# Create a dictionary that maps index to POS tag
tag_to_idx = {pos_tags[i]: i for i in range(len(pos_tags))}
idx_to_tag = {i: pos_tags[i] for i in range(len(pos_tags))}
```

```
def train_and_evaluate(model_class, input_dim, hidden_dim, output_dim, num_epochs=20):
    # Initialize the model
    model = model_class(input_dim, hidden_dim, output_dim, dropout=0.5)
```

```
# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Lists to store loss during training
train_losses = []
dev_losses = []

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    total_train_loss = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()  # Zero the gradients. In PyTorch, gradients are accumulated by default during backpropagation. If you don't zero out the gr
        outputs = model(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Calculate loss
        loss.backward()  # Backpropagation
        optimizer.step()  # Optimize

        total_train_loss += loss.item()

    train_losses.append(total_train_loss)

    # Validation loop
    model.eval()  # Set the model to evaluation mode
    total_dev_loss = 0
    with torch.no_grad(): # During inference (model evaluation or testing), you don't need gradients because you're not performing backpropagation. Disabl
        for inputs, labels in dev_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            total_dev_loss += loss.item()

    dev_losses.append(total_dev_loss)

    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {total_train_loss}, Dev Loss: {total_dev_loss}")

# Plotting training and validation loss curves
plt.figure(figsize=(10, 6))
# plt.plot(range(1, num_epochs+1), train_losses, label="Train Loss",)
plt.plot(range(1, num_epochs+1), train_losses, label="Train Loss", color='#00008B')
plt.plot(range(1, num_epochs+1), dev_losses, label="Dev Loss", linestyle='--', color='#00008B')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()

# Evaluate on the test set
model.eval()
y_true = []
y_pred = []
y_prob = []  # To store the predicted probabilities

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        y_true.extend(labels.numpy())
        y_pred.extend(predicted.numpy())
        y_prob.extend(torch.softmax(outputs, dim=1).cpu().numpy())  # Get probabilities

# Convert numeric labels to POS tags using idx_to_tag
y_true_tags = [idx_to_tag[idx] for idx in y_true]
y_pred_tags = [idx_to_tag[idx] for idx in y_pred]

# Classification report with tags instead of numeric class labels
print(f"Classification Report for {model_class.__name__}:")
print(classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, zero_division=0))

# Calculate AUC scores for each class
auc_scores = {}
for i, tag in enumerate(pos_tags):
    # One-hot encode the true labels for the current tag
    true_binary = [1 if label == i else 0 for label in y_true]
    pred_prob = [prob[i] for prob in y_prob]  # Probabilities for the current tag

    auc_score = roc_auc_score(true_binary, pred_prob)
    auc_scores[tag] = auc_score

# Print AUC scores for each class
print("\nAUC Scores for Each Class:")
for tag, auc in auc_scores.items():
    print(f"{tag}: {auc:.4f}")

# Calculate Macro-Averaged Precision, Recall, F1, and Precision-Recall AUC
# Precision, recall, and F1
report = classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, output_dict=True, zero_division=0)
precision_macro = report["macro avg"]["precision"]
recall_macro = report["macro avg"]["recall"]
f1_macro = report["macro avg"]["f1-score"]

# Precision-Recall AUC (macro-average)
precision_recall_auc_macro = np.mean([roc_auc_score([1 if label == i else 0 for label in y_true],
                                                    [prob[i] for prob in y_prob]) for i in range(len(pos_tags))])

# Print Macro-Averaged Metrics
print(f"\nMacro-Averaged Precision: {precision_macro}")
print(f"Macro-Averaged Recall: {recall_macro}")
```

```
        print(f"Macro-Averaged F1: {f1_macro}")
        print(f"Macro-Averaged Precision-Recall AUC: {precision_recall_auc_macro}")

        return model


print("Training Shallow MLP:")
shallow_model = train_and_evaluate(ShallowPOS_MLP, input_dim=2*word2vec.vector_size, hidden_dim=128, output_dim=len(set([f[1] for f in train_features])), num_
```

```
Training Shallow MLP:
Epoch 1/20, Train Loss: 2487.628724694252, Dev Loss: 239.0158531665802
Epoch 2/20, Train Loss: 1880.5576054304838, Dev Loss: 224.8769073188305
Epoch 3/20, Train Loss: 1772.451532870531, Dev Loss: 221.4169827401638
Epoch 4/20, Train Loss: 1715.84065105021, Dev Loss: 217.5296540260315
Epoch 5/20, Train Loss: 1676.634464636445, Dev Loss: 215.8184494227171
Epoch 6/20, Train Loss: 1641.2918900251389, Dev Loss: 213.61387968063354
Epoch 7/20, Train Loss: 1616.4067249149084, Dev Loss: 214.1634572595358
Epoch 8/20, Train Loss: 1595.938673466444, Dev Loss: 214.38553020358086
Epoch 9/20, Train Loss: 1578.6978471428156, Dev Loss: 213.17859655618668
Epoch 10/20, Train Loss: 1562.2563051879406, Dev Loss: 214.02639035880566
Epoch 11/20, Train Loss: 1545.149732619524, Dev Loss: 215.34580318629742
Epoch 12/20, Train Loss: 1537.7930569946766, Dev Loss: 213.6443422138691
Epoch 13/20, Train Loss: 1522.7121153622866, Dev Loss: 214.53271222114563
Epoch 14/20, Train Loss: 1519.260555833578, Dev Loss: 213.98709635436535
Epoch 15/20, Train Loss: 1501.2115272134542, Dev Loss: 215.04362384974957
Epoch 16/20, Train Loss: 1497.3834756463766, Dev Loss: 216.0239826142788
Epoch 17/20, Train Loss: 1486.9624973237514, Dev Loss: 217.44726103544235
Epoch 18/20, Train Loss: 1484.2341464608908, Dev Loss: 216.33932879567146
Epoch 19/20, Train Loss: 1472.194731131196, Dev Loss: 217.37275847792625
Epoch 20/20, Train Loss: 1464.227485626936, Dev Loss: 219.50031086802483
```



Classification Report for ShallowPOS_MLP:

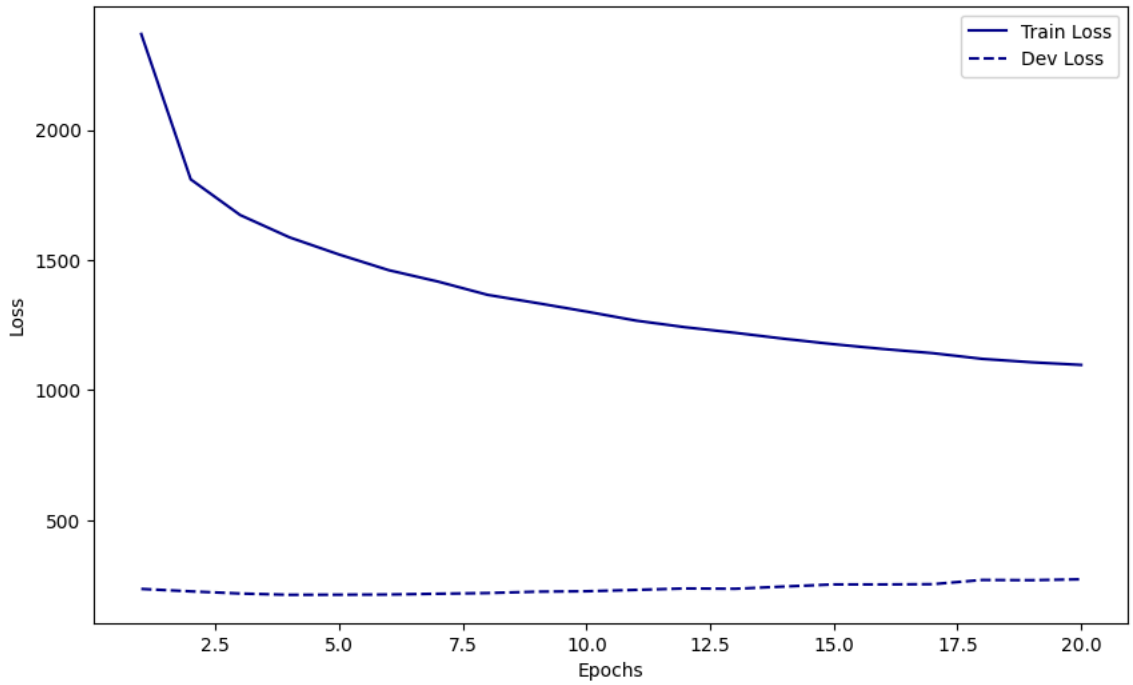
	precision	recall	f1-score	support
ADJ	0.92	0.88	0.90	1794
ADP	0.82	0.76	0.79	2030
ADV	0.91	0.86	0.88	1183
AUX	0.94	0.97	0.95	1543
CCONJ	0.75	0.34	0.46	736
DET	0.88	0.78	0.83	1896
INTJ	0.96	0.74	0.83	121
NOUN	0.90	0.89	0.90	4123
NUM	0.63	0.51	0.57	542
PART	0.67	0.61	0.64	649
PRON	0.97	0.96	0.97	2166
PROPN	0.88	0.81	0.84	2076
PUNCT	0.59	0.90	0.71	3096
SCONJ	0.77	0.65	0.71	384
SYM	0.94	0.61	0.74	109
VERB	0.93	0.89	0.91	2606
X	0.00	0.00	0.00	42
_	0.96	0.77	0.85	354
accuracy			0.84	25450
macro avg	0.80	0.72	0.75	25450
weighted avg	0.85	0.84	0.83	25450

```
AUC Scores for Each Class:
ADJ: 0.9886
ADP: 0.9806
ADV: 0.9937
AUX: 0.9985
CCONJ: 0.9561
DET: 0.9882
INTJ: 0.9887
NOUN: 0.9870
NUM: 0.9738
PART: 0.9825
PRON: 0.9994
```

```
PROPN: 0.9795
PUNCT: 0.9655
SCONJ: 0.9905
SYM: 0.9726
VERB: 0.9942
X: 0.8736
_: 0.9848

Macro-Averaged Precision: 0.8012228889153971
Macro-Averaged Recall: 0.7178828679587173
Macro-Averaged F1: 0.7489499163974004
Macro-Averaged Precision-Recall AUC: 0.9776463554203535
```

```
print("Training Somewhat Deep MLP:")
deep_model = train_and_evaluate(DeepPOS_MLP, input_dim=2*word2vec.vector_size, hidden_dim=128, output_dim=len(set([f[1] for f in train_features])), num_epochs
```

```
Training Somewhat Deep MLP:
Epoch 1/20, Train Loss: 2368.972687959671, Dev Loss: 236.01379914581776
Epoch 2/20, Train Loss: 1810.2979687601328, Dev Loss: 226.61035127937794
Epoch 3/20, Train Loss: 1673.139632500708, Dev Loss: 218.3626715540886
Epoch 4/20, Train Loss: 1587.1272729933262, Dev Loss: 213.34799091517925
Epoch 5/20, Train Loss: 1521.3918149471283, Dev Loss: 213.78560818731785
Epoch 6/20, Train Loss: 1461.3324899822474, Dev Loss: 214.49091900885105
Epoch 7/20, Train Loss: 1417.3827497288585, Dev Loss: 217.49580043554306
Epoch 8/20, Train Loss: 1366.458241418004, Dev Loss: 220.09760899841785
Epoch 9/20, Train Loss: 1334.866234742105, Dev Loss: 225.4418142735958
Epoch 10/20, Train Loss: 1302.147398263216, Dev Loss: 227.0092499256134
Epoch 11/20, Train Loss: 1267.4159463644028, Dev Loss: 232.31591953337193
Epoch 12/20, Train Loss: 1241.8476319536567, Dev Loss: 237.80643305182457
Epoch 13/20, Train Loss: 1220.5980688780546, Dev Loss: 236.52202494442463
Epoch 14/20, Train Loss: 1197.4676636755466, Dev Loss: 245.2383982092142
Epoch 15/20, Train Loss: 1176.6448509171605, Dev Loss: 253.3154240399599
Epoch 16/20, Train Loss: 1158.2532987594604, Dev Loss: 253.32407623529434
Epoch 17/20, Train Loss: 1142.2314057350159, Dev Loss: 254.2816606014967
Epoch 18/20, Train Loss: 1120.31521794945, Dev Loss: 270.50823614001274
Epoch 19/20, Train Loss: 1107.1822783350945, Dev Loss: 269.665175139904
Epoch 20/20, Train Loss: 1097.05962587893, Dev Loss: 273.0303966552019
```

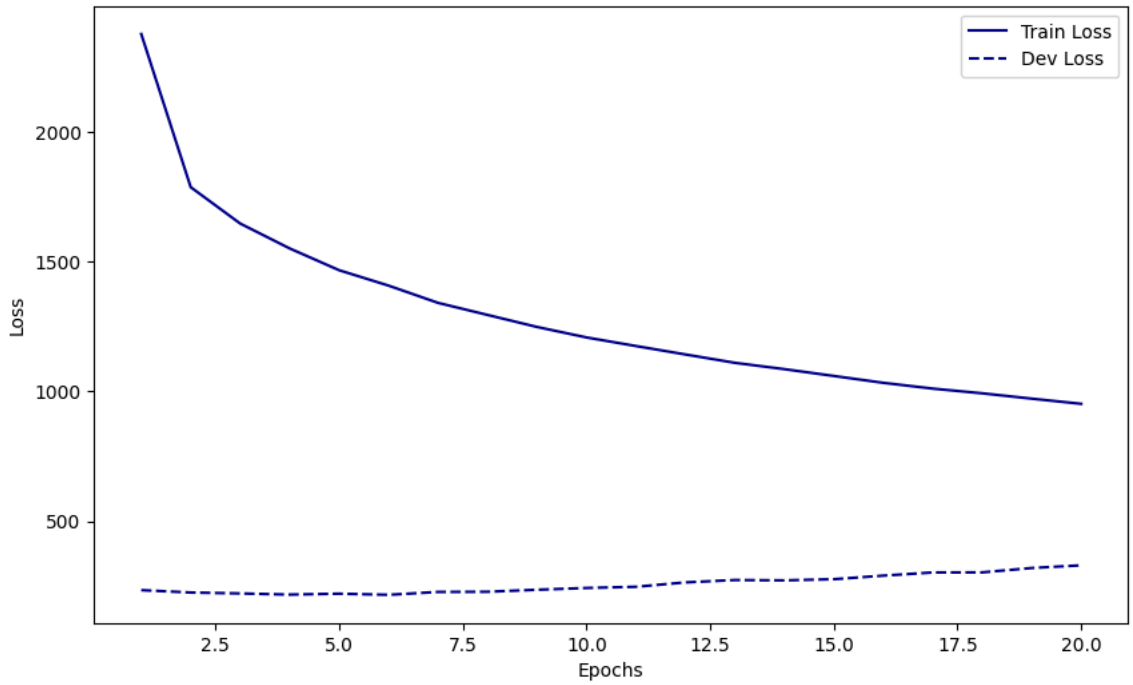



```
Classification Report for DeepPOS_MLP:
              precision    recall  f1-score   support

         ADJ       0.91      0.87      0.89      1794
         ADP       0.79      0.77      0.78      2030
         ADV       0.91      0.87      0.89      1183
         AUX       0.96      0.96      0.96      1543
       CCONJ       0.70      0.34      0.45       736
         DET       0.87      0.77      0.82      1896
        INTJ       0.93      0.68      0.78       121
        NOUN       0.89      0.89      0.89      4123
         NUM       0.56      0.56      0.56       542
        PART       0.65      0.60      0.62       649
        PRON       0.97      0.97      0.97      2166
       PROPN       0.89      0.79      0.84      2076
       PUNCT       0.60      0.86      0.70      3096
       SCONJ       0.80      0.65      0.72       384
         SYM       0.93      0.60      0.73       109
        VERB       0.92      0.91      0.91      2606
           X       0.00      0.00      0.00        42
           _       0.93      0.77      0.84       354

    accuracy                           0.83     25450
   macro avg       0.79      0.71      0.74     25450
weighted avg       0.84      0.83      0.83     25450


AUC Scores for Each Class:
ADJ: 0.9828
ADP: 0.9804
ADV: 0.9913
AUX: 0.9988
CCONJ: 0.9567
DET: 0.9867
INTJ: 0.9723
NOUN: 0.9837
NUM: 0.9749
PART: 0.9790
PRON: 0.9993
```

```
PROPN: 0.9728
PUNCT: 0.9626
SCONJ: 0.9860
SYM: 0.9708
VERB: 0.9914
X: 0.8282
_: 0.9846

Macro-Averaged Precision: 0.7893676067816789
Macro-Averaged Recall: 0.7129567005844416
Macro-Averaged F1: 0.7418362048925187
Macro-Averaged Precision-Recall AUC: 0.9723516238810017
```

```
print("Training Deep MLP:")
very_deep_model = train_and_evaluate(VeryDeepPOS_MLP, input_dim=2*word2vec.vector_size, hidden_dim=128, output_dim=len(set([f[1] for f in train_features])), n
```

```
Training Deep MLP:
Epoch 1/20, Train Loss: 2376.2586891055107, Dev Loss: 235.04231701791286
Epoch 2/20, Train Loss: 1786.6116228997707, Dev Loss: 225.92248821258545
Epoch 3/20, Train Loss: 1646.5832473039627, Dev Loss: 222.3802939504385
Epoch 4/20, Train Loss: 1550.7370206415653, Dev Loss: 217.8673692792654
Epoch 5/20, Train Loss: 1466.8394483178854, Dev Loss: 221.42730598151684
Epoch 6/20, Train Loss: 1407.4774709194899, Dev Loss: 217.03836742043495
Epoch 7/20, Train Loss: 1341.0882495641708, Dev Loss: 228.16171610355377
Epoch 8/20, Train Loss: 1294.6896402463317, Dev Loss: 228.74935557693243
Epoch 9/20, Train Loss: 1248.3857691660523, Dev Loss: 236.5930026769638
Epoch 10/20, Train Loss: 1207.998226299882, Dev Loss: 243.35564750432968
Epoch 11/20, Train Loss: 1174.933920301497, Dev Loss: 247.9553687274456
Epoch 12/20, Train Loss: 1142.4845060780644, Dev Loss: 264.8668832182884
Epoch 13/20, Train Loss: 1110.2738841623068, Dev Loss: 274.02378419041634
Epoch 14/20, Train Loss: 1085.9472921565175, Dev Loss: 272.7577810436487
Epoch 15/20, Train Loss: 1059.6684476025403, Dev Loss: 277.15677611529827
Epoch 16/20, Train Loss: 1033.265403226018, Dev Loss: 290.85540610551834
Epoch 17/20, Train Loss: 1010.9689370244741, Dev Loss: 303.14305797219276
Epoch 18/20, Train Loss: 992.8496845662594, Dev Loss: 303.4856590330601
Epoch 19/20, Train Loss: 972.3054616451263, Dev Loss: 320.2400607764721
Epoch 20/20, Train Loss: 952.4098950177431, Dev Loss: 330.64763954281807
```



Classification Report for VeryDeepPOS_MLP:

	precision	recall	f1-score	support
ADJ	0.91	0.87	0.89	1794
ADP	0.73	0.80	0.76	2030
ADV	0.90	0.84	0.87	1183
AUX	0.95	0.96	0.95	1543
CCONJ	0.72	0.30	0.42	736
DET	0.76	0.84	0.80	1896
INTJ	0.92	0.75	0.83	121
NOUN	0.88	0.89	0.89	4123
NUM	0.61	0.46	0.52	542
PART	0.62	0.65	0.63	649
PRON	0.97	0.96	0.97	2166
PROPN	0.88	0.78	0.82	2076
PUNCT	0.62	0.78	0.69	3096
SCONJ	0.82	0.62	0.71	384
SYM	0.93	0.61	0.74	109
VERB	0.92	0.90	0.91	2606
X	0.06	0.02	0.03	42
_	0.89	0.77	0.82	354
accuracy			0.82	25450
macro avg	0.78	0.71	0.74	25450
weighted avg	0.83	0.82	0.82	25450

```
AUC Scores for Each Class:
ADJ: 0.9807
ADP: 0.9775
ADV: 0.9901
AUX: 0.9977
CCONJ: 0.9482
DET: 0.9846
INTJ: 0.9729
NOUN: 0.9780
NUM: 0.9665
PART: 0.9748
PRON: 0.9991
```

```
PROPN: 0.9618
PUNCT: 0.9614
SCONJ: 0.9888
SYM: 0.9620
VERB: 0.9885
X: 0.7743
_: 0.9796

Macro-Averaged Precision: 0.7824470051285168
Macro-Averaged Recall: 0.7110433756807468
Macro-Averaged F1: 0.7367859446088755
Macro-Averaged Precision-Recall AUC: 0.9659193505126894
```

Baseline Tagger(s)

```
class BaselineTagger:
    def __init__(self, train_data):
        """
        Initializes the baseline tagger. It trains on the given training data
        by counting the most frequent tag for each word.
        """
        self.word_to_tag = defaultdict(lambda: None)  # Default None for unseen words
        self.most_frequent_tag = None
        self.train(train_data)

    def train(self, train_data):
        """
        Trains the baseline tagger by recording the most frequent tag for each word.
        """
        word_tag_counts = defaultdict(lambda: defaultdict(int))  # Count tags per word

        # Count occurrences of each word-tag pair in the training data
        for sentence in train_data:
            for word, tag in sentence:
                word_tag_counts[word][tag] += 1

        # For each word, find the most frequent tag
        for word, tag_counts in word_tag_counts.items():
            self.word_to_tag[word] = max(tag_counts, key=tag_counts.get)

        # Find the most frequent tag overall in the training data (for unseen words)
        tag_counts = defaultdict(int)
        for sentence in train_data:
            for _, tag in sentence:
                tag_counts[tag] += 1

        # Set the most frequent tag to handle unseen words
        self.most_frequent_tag = max(tag_counts, key=tag_counts.get)

    def predict(self, sentence):
        """
        Predicts the tags for the given sentence using the baseline strategy.
        For each word, it returns the most frequent tag it has in the training data,
        or the most frequent tag overall if the word is unseen.
        """
        return [self.word_to_tag.get(word, self.most_frequent_tag) for word, _ in sentence]

    def evaluate(self, test_data):
        """
        Evaluates the baseline tagger on the given test data and prints precision, recall, F1,
        and AUC scores.
        """
        y_true = []
        y_pred = []
        y_prob = []

        for sentence in test_data:
            true_tags = [tag for _, tag in sentence]
            predicted_tags = self.predict(sentence)

            y_true.extend(true_tags)
            y_pred.extend(predicted_tags)

        # Convert numeric labels to POS tags using idx_to_tag
        y_true_tags = [tag_to_idx[idx] for idx in y_true]
        y_pred_tags = [tag_to_idx[idx] for idx in y_pred]

        # Classification report with tags instead of numeric class labels
        print(f"Baseline Tagger Classification Report:")
        print(classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, zero_division=0))

        # Calculate AUC scores for each class
        auc_scores = {}
        for i, tag in enumerate(pos_tags):
            # One-hot encode the true labels for the current tag
            true_binary = [1 if label == i else 0 for label in y_true_tags]
            pred_binary = [1 if label == i else 0 for label in y_pred_tags]
            auc_score = roc_auc_score(true_binary, pred_binary)
```

```
            auc_scores[tag] = auc_score

        # Print AUC scores for each class
        print("\nAUC Scores for Each Class:")
        for tag, auc in auc_scores.items():
            print(f"{tag}: {auc:.4f}")

        # Calculate Macro-Averaged Precision, Recall, F1, and Precision-Recall AUC
        report = classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, output_dict=True, zero_division=0)
        precision_macro = report["macro avg"]["precision"]
        recall_macro = report["macro avg"]["recall"]
        f1_macro = report["macro avg"]["f1-score"]

        # Precision-Recall AUC (macro-average)
        precision_recall_auc_macro = np.mean([roc_auc_score([1 if label == i else 0 for label in y_true_tags],
                                                            [1 if label == i else 0 for label in y_pred_tags]) for i in range(len(pos_tags))])

        # Print Macro-Averaged Metrics
        print(f"\nMacro-Averaged Precision: {precision_macro:.4f}")
        print(f"Macro-Averaged Recall: {recall_macro:.4f}")
        print(f"Macro-Averaged F1: {f1_macro:.4f}")
        print(f"Macro-Averaged Precision-Recall AUC: {precision_recall_auc_macro:.4f}")



# Train the baseline tagger
baseline_tagger = BaselineTagger(train_data)

# Evaluate on the test set
baseline_tagger.evaluate(test_data)
```

```
Baseline Tagger Classification Report:
                precision    recall  f1-score   support

           ADJ       0.91      0.83      0.87      1794
           ADP       0.87      0.88      0.88      2030
           ADV       0.94      0.79      0.86      1183
           AUX       0.93      0.89      0.91      1543
         CCONJ       0.99      1.00      0.99       736
           DET       0.96      0.97      0.96      1896
          INTJ       0.97      0.69      0.80       121
          NOUN       0.67      0.93      0.78      4123
           NUM       0.91      0.61      0.73       542
          PART       0.69      0.99      0.81       649
          PRON       0.96      0.93      0.95      2166
         PROPN       0.91      0.51      0.66      2076
         PUNCT       0.99      0.99      0.99      3096
         SCONJ       0.62      0.60      0.61       384
           SYM       0.81      0.83      0.82       109
          VERB       0.89      0.82      0.85      2606
             X       1.00      0.00      0.00        42
             _       0.97      0.81      0.89       354

      accuracy                           0.86     25450
     macro avg       0.89      0.78      0.80     25450
  weighted avg       0.88      0.86      0.86     25450


AUC Scores for Each Class:
ADJ: 0.9111
ADP: 0.9350
ADV: 0.8924
AUX: 0.9422
CCONJ: 0.9985
DET: 0.9823
INTJ: 0.8429
NOUN: 0.9219
NUM: 0.8029
PART: 0.9902
PRON: 0.9629
PROPN: 0.7539
PUNCT: 0.9927
SCONJ: 0.7992
SYM: 0.9170
VERB: 0.9017
X: 0.5000
_: 0.9066

Macro-Averaged Precision: 0.8891
Macro-Averaged Recall: 0.7814
Macro-Averaged F1: 0.7974
Macro-Averaged Precision-Recall AUC: 0.8863
```

```
class DummyTagger:
    def __init__(self, train_data):
        """
        Initializes the dummy tagger. It trains on the given training data
        by finding the most frequent tag overall in the training data.
        """
        self.most_frequent_tag = None
        self.train(train_data)
```

```
    def train(self, train_data):
        """
        Trains the dummy tagger by finding the most frequent tag in the training data.
        """
        tag_counts = defaultdict(int)

        # Count occurrences of each tag in the training data
        for sentence in train_data:
            for _, tag in sentence:
                tag_counts[tag] += 1

        # Find the most frequent tag in the training data
        self.most_frequent_tag = max(tag_counts, key=tag_counts.get)

    def predict(self, sentence):
        """
        Predicts the tag for every word in the sentence using the dummy strategy.
        Every word gets the most frequent tag found during training.
        """
        return [self.most_frequent_tag for _ in sentence]

    def evaluate(self, test_data):
        """
        Evaluates the dummy tagger on the given test data and prints precision, recall, F1,
        and AUC scores.
        """
        y_true = []
        y_pred = []
        y_prob = []  # Dummy tagger does not return probabilities, so we'll skip this.

        for sentence in test_data:
            true_tags = [tag for _, tag in sentence]
            predicted_tags = self.predict(sentence)

            y_true.extend(true_tags)
            y_pred.extend(predicted_tags)

        # Convert numeric labels to POS tags using idx_to_tag
        y_true_tags = [tag_to_idx[idx] for idx in y_true]
        y_pred_tags = [tag_to_idx[idx] for idx in y_pred]

        # Classification report with tags instead of numeric class labels
        print(f"Dummy Tagger Classification Report:")
        print(classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, zero_division=0))

        # Calculate AUC scores for each class
        auc_scores = {}
        for i, tag in enumerate(pos_tags):
            # One-hot encode the true labels for the current tag
            true_binary = [1 if label == i else 0 for label in y_true_tags]
            pred_binary = [1 if label == i else 0 for label in y_pred_tags]

            auc_score = roc_auc_score(true_binary, pred_binary)
            auc_scores[tag] = auc_score

        # Print AUC scores for each class
        print("\nAUC Scores for Each Class:")
        for tag, auc in auc_scores.items():
            print(f"{tag}: {auc:.4f}")

        # Calculate Macro-Averaged Precision, Recall, F1, and Precision-Recall AUC
        report = classification_report(y_true_tags, y_pred_tags, target_names=pos_tags, output_dict=True, zero_division=0)
        precision_macro = report["macro avg"]["precision"]
        recall_macro = report["macro avg"]["recall"]
        f1_macro = report["macro avg"]["f1-score"]

        # Precision-Recall AUC (macro-average)
        precision_recall_auc_macro = np.mean([roc_auc_score([1 if label == i else 0 for label in y_true_tags],
                                                            [1 if label == i else 0 for label in y_pred_tags]) for i in range(len(pos_tags))])

        # Print Macro-Averaged Metrics
        print(f"\nMacro-Averaged Precision: {precision_macro:.4f}")
        print(f"Macro-Averaged Recall: {recall_macro:.4f}")
        print(f"Macro-Averaged F1: {f1_macro:.4f}")
        print(f"Macro-Averaged Precision-Recall AUC: {precision_recall_auc_macro:.4f}")



# Train the dummy tagger
dummy_tagger = DummyTagger(train_data)

# Evaluate on the test set
dummy_tagger.evaluate(test_data)
```

```
Dummy Tagger Classification Report:
              precision    recall  f1-score   support

         ADJ       0.00      0.00      0.00      1794
         ADP       0.00      0.00      0.00      2030
         ADV       0.00      0.00      0.00      1183
         AUX       0.00      0.00      0.00      1543
       CCONJ       0.00      0.00      0.00       736
         DET       0.00      0.00      0.00      1896
        INTJ       0.00      0.00      0.00       121
        NOUN       0.16      1.00      0.28      4123
```