Assignment_1_for_DS_207_(Intro_to_NLP)_Text_Classification

January 24, 2025

1 Assignment 1: Text Classification, Word Vectors (TA: Tarun Gupta)

The goal of this assignment is introduce the basics of text classification and word vectors.

Please make a copy of this notebook (locally or on Colab). Ensure you adhere to the guidelines and submission instructions (mentioned below) for attempting and submitting the assignment.

Given that the class has 150+ students, we will **NOT** entertain any requests for changing your notebooks after the submission deadline (especially in cases when the notebook fails to compile or run because you did not follow the instructions).

1.0.1 Guidelines for Attempting the Assignment

1. Write your logic in the cells **ONLY** which have the comment `# ADD YOUR CODE HERE`, between the `# BEGIN CODE` and `# END CODE` comments. These cells are also demarcated by the special start (`## ==== BEGIN EVALUATION PORTION`) and end (`## ==== END EVALUATION PORTION`) comments. Do **NOT** remove any of these comments from the designated cells, otherwise your assignment will not be evaluated correctly.
2. Write your code **ONLY** in the cells designated for auto-evaluation, between the `# BEGIN CODE` and `# END CODE` comments. Please don't write any extra code or comments anywhere else.
3. All imports that should be necessary are already provided as part of the notebook. You should **NOT** import any new libraries, otherwise your assignment will not be graded.
4. You need to install the libraries/imports used in this notebook yourself. Its recommended to use python version between 3.9 and 3.11 to attempt this assignment.
5. **If you encounter any errors in the supporting cells during execution, contact the respective TAs.**
6. **Please read the function docs and comments carefully**. They provide specific instructions and examples for implementing each function. Follow these instructions precisely - neither oversimplify nor overcomplicate your implementations. Deviating from the provided implementation guidelines may result in lost marks.
7. **Important**: Use of AI-assistive technologies such as ChatGPT or GitHub CoPilot is not permitted for this assignment. Ensure that all attempts are solely your own. Not following this

rule can incur a large penalty, including but not limited to scoring a zero for this assignment.

1.0.2 Submission Instructions

1. Ensure your code follows all guidelines mentioned above before submission.
2. Ensure you only add code in designated areas, otherwise you assignment will not evaluated.
3. When you have completely attempted the assignment, **export the current notebook as a `.py` file**, with the following name: `SAPName_SRNo_assignment1.py`, where `SAPName` would be your name as per SAP record, and `SRNo` will be the last 5 digits of your IISc SR number. For example, IISc student with SAP name Twyla Linda (SR no - 04-03-00-10-22-20-1-15329) would use `Twyla_Linda_15329_assignment1.py`.
4. You should put your assignment file `SAPName_SRNo_assignment1.py` inside a folder `SAPName_SRNo`. The folder structure looks as follows:

```
SAPName_SRNo
    SAPName_SRNo_assignment1.py
```

5. When you run the assignment code, it may download certain datasets and other artifacts. These should **NOT** be part of the above folder.
6. Once you have validated the folder structure as above, zip the folder and name it as `submission.zip` and submit this ZIP archive.

If you have any confusion regarding submission instructions, please ask the respective TAs.

1.0.3 Marks Distribution

- Generative Classification: 40 marks
- Word2Vec and Word Analogies: 30 marks
- Discriminative Classification: 30 marks

In the cell below, replace `SAPName` with your name as per SAP record, and `SRNo` with the last 5 digits of your IISc SR number. For example, IISc student with SAP name Twyla Linda (SR no - 04-03-00-10-22-20-1-15329) would use:

```
STUDENT_SAP_NAME  = "Twyla Linda"
STUDENT_SR_NUMBER = "15329"
```

[1]:
```
STUDENT_SAP_NAME  = "Yalla Mahanth"
STUDENT_SR_NUMBER = "24004"
```

2 Imports

[2]:
```
import requests
import numpy as np
import pandas as pd
import re
from pathlib import Path
```

```
import nltk
from gensim.models import KeyedVectors
import gensim.downloader as api
from sklearn.linear_model import LogisticRegression
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\myalla\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

```
[2]: True
```

3 Dataset

We will dive into a basic text-based sentiment classification task. The dataset consists of sentences with two different kinds of sentiments- `1` (`positive`), and `0` (`negative`) sentiments. Following are a set of examples,

- `1`: *I really like your new haircut!*
- `0`: *Your new haircut is awful!*

Below we download a training set (`train_data.csv`- provided) and a validation set (`val_data.csv`- provided). During evaluation we will use a blind test set.

```
[3]: def download_dataset(url, output_path):
         """
         Download a CSV file from a given URL and save it to the specified path.
         If the file already exists, skip the download.

         Parameters:
         url (str): URL of the CSV file to download
         output_path (str): Path where the file should be saved

         Returns:
         bool: True if download was successful or file already exists, False␣
      ↪otherwise
         """
         # Convert to Path object for easier path manipulation
         output_path = Path(output_path)

         # Check if file already exists
         if output_path.exists():
             print(f"File already exists: {output_path}")
             return True

         try:
             print(f"Downloading {output_path.name}...")
             response = requests.get(url)
```

```
        response.raise_for_status()  # Raise an exception for bad status codes

        # Create parent directories if they don't exist
        output_path.parent.mkdir(parents=True, exist_ok=True)

        with open(output_path, 'wb') as f:
            f.write(response.content)
        print(f"Successfully downloaded: {output_path}")
        return True

    except requests.exceptions.RequestException as e:
        print(f"Error downloading {output_path}: {e}")
        return False
    except IOError as e:
        print(f"Error saving file {output_path}: {e}")
        return False

# URLs for the datasets
urls = {
    'train': "https://docs.google.com/spreadsheets/d/
  ↪1pHK8joOen4R1KlhF5Re3LZFb4DtOn4jraHoTGMBgtF4/export?format=csv",
    'val': "https://docs.google.com/spreadsheets/d/
  ↪1t2J2EJPo-P2AlDOAybxq7nX61mbZwgMoQpR_J3FneJ4/export?format=csv",
}

# Download all datasets
for dataset_type, url in urls.items():
    output_path = f"downloaded_datasets/{dataset_type}_data.csv"
    download_dataset(url, output_path)
```

```
File already exists: downloaded_datasets\train_data.csv
```

[3]:
```
True
```

```
File already exists: downloaded_datasets\val_data.csv
```

[3]:
```
True
```

[4]:
```
df = pd.read_csv('downloaded_datasets/train_data.csv')
df_val = pd.read_csv('downloaded_datasets/val_data.csv')

df.head()
```

[4]:
```
                                              review  sentiment
0  I really liked this Summerslam due to the look…          1
1  Not many television shows appeal to quite as m…          1
2  The film quickly gets to a major chase scene w…          0
3  Jane Austen would definitely approve of this o…          1
```

```
4  Expectations were somewhat high for me when I …          0
```

[5]:
```python
sentiment_percentages = df_val['sentiment'].value_counts(normalize=True) * 100

print("Sentiment Distribution:")
print(f"Class 0: {sentiment_percentages[0]:.2f}%")
print(f"Class 1: {sentiment_percentages[1]:.2f}%")
```

```
Sentiment Distribution:
Class 0: 49.95%
Class 1: 50.05%
```

[6]:
```python
X_train, y_train = df.review.values.tolist(), df.sentiment.values.tolist()
X_val, y_val = df_val.review.values.tolist(), df_val.sentiment.values.tolist()
```

4 General evaluation function

[7]:
```python
def evaluate_classifier(y_true, y_pred):
    """
    Calculate classification accuracy.

    Args:
        y_true (list): True class labels
            Example: [0, 1, 0, 1]
        y_pred (list): Predicted class labels
            Example: [0, 1, 1, 1]

    Returns:
        float: Accuracy (proportion of correct predictions)
            Example: 0.75 (3 correct predictions out of 4)

    Note:
        - Accuracy = (number of correct predictions) / (total number of␣
 ↪predictions)
        - Raises ValueError if lengths of inputs don't match
    """
    if len(y_true) != len(y_pred):
        raise ValueError("Length of true and predicted labels must match")

    correct = sum(1 for t, p in zip(y_true, y_pred) if t == p)
    return correct / len(y_true)
```

5 Generative Classification

5.1 Naive Bayes Text Classification

This implementation covers a Naive Bayes classifier for text classification. The key mathematical foundation is Bayes' theorem:

P(class|document)   P(class) * P(document|class)

Where: - P(class|document) is the posterior probability - P(class) is the prior probability of the class - P(document|class) is the likelihood of the document given the class

Under the "naive" assumption of conditional independence: P(document|class) = P(word1|class) * P(word2|class) * … * P(wordN|class)

[8]:
```
# ==== BEGIN EVALUATION PORTION

class NaiveBayesClassifier:
    def __init__(self, min_freq=1):
        """
        Initialize the Naive Bayes classifier.

        Args:
            min_freq (int): Minimum frequency threshold for a word to be␣
 ↪included in vocabulary.
                               Words appearing less than min_freq times will be␣
 ↪treated as UNK token.
                               Default: 1 (include all words)

        Attributes:
            class_probs (dict): P(class) for each class
                Example: {0: 0.5, 1: 0.5}

            word_probs (dict): P(word|class) for each word and class
                Example: {
                    0: {'hello': 0.5, 'world': 0.4, '<UNK>': 0.1},
                    1: {'hello': 0.3, 'world': 0.5, '<UNK>': 0.2}
                }

            vocabulary (dict): Word to index mapping, including special UNK␣
 ↪token
                Example: {'<UNK>': 0, 'hello': 1, 'world': 2}

            min_freq (int): Minimum frequency threshold for vocabulary inclusion
                Example: If min_freq=2, words must appear at least twice to be␣
 ↪included

        Note:
            - Words appearing less than min_freq times in training data will be␣
↪mapped to <UNK>
            - <UNK> token is automatically added to vocabulary as first token␣
↪(index 0)
            - Probability for <UNK> is calculated during training based on rare␣
↪words
        """
        self.class_probs = None
        self.word_probs = None
        self.vocabulary = None
        self.min_freq = min_freq

    def preprocess_text(self, text):
        """
        Preprocess the input text by converting to lowercase, removing non-word␣
↪characters,
        and filtering out common stop words.

        Args:
            text (str): Raw input text
                Example: "Hello, World! How are you doing today?"

        Returns:
            list: List of cleaned, tokenized, and filtered words with stop␣
↪words removed
                Example: ['hello', 'world', 'doing', 'today']

        Note:
            - Converts all text to lowercase
            - Removes punctuation and special characters
            - Splits text into individual tokens
            - Removes common English stop words (e.g., 'a', 'an', 'the', 'is',␣
↪'are', 'how')
            - Stop words are removed using NLTK's English stop words list
        """
        # Import stop words from NLTK
        from nltk.corpus import stopwords
        stop_words = set(stopwords.words('english'))

        # Convert to lowercase
        text = text.lower()

        # Extract word characters only and split into tokens
        tokens = re.findall(r'\w+', text)

        # Remove stop words
        filtered_tokens = [token for token in tokens if token not in stop_words]

        return filtered_tokens

    def create_vocabulary(self, texts):
        """
        Create vocabulary from training texts by mapping unique words to␣
↪indices,
        considering minimum frequency threshold and adding UNK token.

        Args:
            texts (list): List of text documents
                Example: [
                    "Hello world hello",
                    "Hello there",
                    "World is beautiful"
                ]

        Returns:
            dict: Mapping of words to unique indices, including UNK token
                Example (with min_freq=2): {
                    '<UNK>': 0,     # Special token for rare/unseen words
                    'hello': 1,     # Frequency=3, included in vocab
                    'world': 2,     # Frequency=2, included in vocab
                    # 'there' and 'beautiful' not included (frequency=1 <␣
↪min_freq=2)
                }

        Note:
            - Always includes <UNK> token at index 0
            - Only includes words that appear >= min_freq times
            - Word frequency is counted across all documents
            - Uses preprocess_text function for preprocessing
            - Words below frequency threshold will be mapped to UNK during␣
↪feature extraction
        """

        # BEGIN CODE : naive_bayes.create_vocabulary

        self.UNK = '<UNK>'
        self.freqeuncies = {}
        for txt in texts:
            for word in self.preprocess_text(txt):
                self.freqeuncies[word] = self.freqeuncies.get(word, 0) + 1

        vocab = {self.UNK: 0}
        counter = 1
        for k,v in self.freqeuncies.items():
            if v >= self.min_freq:
                vocab[k] = counter
                counter += 1
        return vocab

        # END CODE

    def extract_features(self, texts, vocabulary):
        """
        Convert texts to bag-of-words feature vectors using the vocabulary,
        where each element represents the count of word occurrences (not binary␣
↪presence/absence).

        Args:
            texts (list): List of text documents
                Example: ["hello world hello", "world is beautiful"]
            vocabulary (dict): Word to index mapping with UNK token
                Example: {'<UNK>': 0, 'hello': 1, 'world': 2}

        Returns:
            np.array: Feature matrix where each row is a document vector
                Example: For the above input with min_freq=2:
                array([
                    [0, 2, 1],   # First doc: 0 UNKs, 2 'hello's, 1 'world'
                    [2, 0, 1]    # Second doc: 2 UNKs (one each for 'is' and␣
↪'beautiful'), 0 'hello's, 1 'world',
                ])

        Note:
            - Each row represents one document
            - Each column represents the count of a specific word
            - First column is always UNK token count
            - Words not in vocabulary are counted as UNK
            - Shape of output: (n_documents, len(vocabulary))
            - Uses preprocess_text function for preprocessing
        """

        # BEGIN CODE : naive_bayes.extract_features

        vocab_size = len(vocabulary)
        n_documents = len(texts)
        res = np.zeros((n_documents, vocab_size))
        for row, txt in enumerate(texts):
            for word in self.preprocess_text(txt):
                if word in vocabulary:
                    res[row][vocabulary[word]] += 1
                else:
                    res[row][vocabulary[self.UNK]] += 1
        return res

        # END CODE

    def calculate_class_probabilities(self, y):
        """
        Estimate probability P(class) for each class from training labels.

        Args:
            y (list): List of class labels
                Example: [0, 0, 1, 1, 0, 1]

        Returns:
            dict: Estimated probability for each class
                Example: {
                    0: 0.5,    # 3 out of 6 samples are class 0
                    1: 0.5     # 3 out of 6 samples are class 1
                }

        Note:
            - Probabilities sum to 1 across all classes
            - Handles any number of unique classes
        """
        # BEGIN CODE : naive_bayes.extract_features

        y_ = np.array(y)
        classes = sorted(set(y))
        res = {}
        for cls_idx , label in enumerate(classes):
            res[cls_idx] = np.sum(y_ == label) / y_.shape[0]
        return res

        # END CODE

    def calculate_word_probabilities(self, X, y, vocabulary, alpha=1.0):
        """
        Calculate conditional probability P(word|class) for each word and class,
        including probability for UNK token.

        Args:
            X (np.array): Document-term matrix (with UNK counts in first column)
                Example: array([
                    [0, 2, 1],  # Document 1: 0 UNKs, 2 of word 1, 1 of word 2
                    [1, 0, 1],  # Document 2: 1 UNK, 0 of word 1, 1 of word 2
                ])
            y (list): Class labels
                Example: [0, 1]
            vocabulary (dict): Word to index mapping with UNK token
                Example: {'<UNK>': 0, 'hello': 1, 'world': 2}
            alpha (float): Laplace smoothing parameter, default=1.0

        Returns:
            dict: Nested dict with P(word|class) for each word and class
                Example: {
                    0: {
                        '<UNK>': 0.167,     # P(word=UNK|class=0)
                        'hello': 0.5,      # P(word='hello'|class=0)
                        'world': 0.333      # P(word='world'|class=0)
                    },
                    1: {
                        '<UNK>': 0.4,     # P(word=UNK|class=1)
                        'hello': 0.2,      # P(word='hello'|class=1)
                        'world': 0.4       # P(word='world'|class=1)
                    }
                }

        Note:
            - Uses Laplace smoothing to handle unseen words
            - UNK token probability is learned from training data
            - Formula: P(word|class) = (count(word,class) + ) /␣
↪(total_words_in_class + |V|)
            - |V| is vocabulary size (including UNK token)
        """

        # BEGIN CODE : naive_bayes.calculate_word_probabilities

        classes = sorted(set(y))
        vocab_size = len(vocabulary)
        y_ = np.array(y)

        total_words_in_class = np.array([np.sum(X[y_ == cls]) for cls in␣
↪classes])
        den = total_words_in_class + alpha * vocab_size

        counts = np.array( [ np.sum(X[y_ == cls][:, np.arange(vocab_size)],␣
↪axis=0) for cls in classes])
        num = counts + alpha

        probs = num / den.reshape(-1, 1)

        words = list(vocabulary.keys())
        res = { cls: dict(zip(words, probs[cls])) for cls in classes}
        return res


        # END CODE

    def fit(self, X_text, y):
        """
        Train the Naive Bayes classifier on the provided text documents.

        Args:
            X_text (list): List of text documents
                Example: [
                    "hello world",
                    "beautiful world",
                    "hello there"
                ]
            y (list): Class labels
                Example: [0, 1, 0]

        Note:
            - Creates vocabulary from training texts
            - Calculates prior probabilities P(class)
            - Calculates conditional probabilities P(word|class)
            - Stores all necessary parameters for prediction
        """
        # Create vocabulary from training texts
        self.vocabulary = self.create_vocabulary(X_text)

        # Convert texts to feature vectors
        X = self.extract_features(X_text, self.vocabulary)

        # Calculate probabilities
        self.class_probs = self.calculate_class_probabilities(y)
        self.word_probs = self.calculate_word_probabilities(
            X, y, self.vocabulary)

    def predict(self, X_text):
        """
        Predict classes for new documents using Naive Bayes algorithm,
        handling unknown words using UNK token.

        Args:
            X_text (list): List of text documents
                Example: [
                    "hello world",
                    "beautiful day"  # 'day' is unknown, treated as UNK
                ]
        Returns:
            list: Predicted class labels
                Example: [0, 1]

        Theory:
            The standard Naive Bayes formula for text classification is:
            P(class|document)   P(class) *   P(word|class)

            For unknown words not in vocabulary:
            - They are mapped to UNK token
            - P(UNK|class) is used in probability calculation

            We use log space to prevent numerical underflow:
            log(P(class|document))   log(P(class)) + Σ log(P(word|class))

        Implementation:
            For each document:
            1. Preprocess and tokenize text
            2. Replace unknown words with UNK token
            3. Calculate log probabilities using appropriate word or UNK␣
↪probabilities
            4. Return class with highest log probability score

        Note:
            - Uses preprocess_text function for preprocessing
            - Words not in vocabulary are treated as UNK token
            - UNK probability is used for out-of-vocabulary words
        """
        # BEGIN CODE : naive_bayes.predict

        y_pred = []
        priors = np.log(np.array(list(self.class_probs.values())))
        for text in X_text:
            row = self.preprocess_text(text)
            probs = priors.copy()
            for cls in self.class_probs.keys():
                for word in row:
                    probs[cls] += np.log(self.word_probs[cls].get(word, self.
↪word_probs[cls][self.UNK]))
            y_pred.append(np.argmax(probs))
        return y_pred

        # END CODE

    def get_important_words(self, n=5, use_ratio=True):
        """
        Get the most important words for each class based either on their raw␣
↪conditional
        probabilities or their probability ratios between classes.

        Args:
            n (int): Number of top words to return for each class, default=5
            use_ratio (bool): If True, ranks words by probability ratio between␣
↪classes
                            If False, ranks words by raw conditional probability

        Returns:
            dict: Dictionary mapping class labels to lists of (word, score)␣
↪tuples,
                  where score is either probability or probability ratio
                Example with use_ratio=False: {
                    0: [('excellent', 0.014), ('great', 0.012), ('amazing', 0.
↪011),
                        ('<UNK>', 0.008), ('wonderful', 0.007)],  # Raw␣
↪probabilities
                    1: [('terrible', 0.015), ('bad', 0.012), ('<UNK>', 0.010),
                        ('boring', 0.008), ('awful', 0.007)]
                }
                Example with use_ratio=True: {
                    0: [('excellent', 7.5), ('amazing', 6.2), ('great', 5.8),
                        ('wonderful', 4.9), ('good', 4.1)],  # P(word|pos)/
↪P(word|neg)
                    1: [('terrible', 8.3), ('awful', 7.1), ('bad', 6.4),
                        ('boring', 5.2), ('waste', 4.8)]    # P(word|neg)/
↪P(word|pos)
                }

        Note:
            - When use_ratio=True:
                - For class 0: Returns words where P(word|class=0)/
↪P(word|class=1) is highest
                - For class 1: Returns words where P(word|class=1)/
↪P(word|class=0) is highest
                - Better at finding discriminative words that distinguish␣
↪between classes
                - Reduces overlap between top words of different classes
            - When use_ratio=False:
                - Returns words with highest raw P(word|class) for each class
                - May have significant overlap between classes
            - Includes UNK token only if it meets the ranking criteria
            - Small probabilities are handled safely to avoid division by zero
        """
        if not self.word_probs:
            raise ValueError("Classifier must be trained before getting␣
↪important words")

        important_words = {}
        classes = sorted(self.word_probs.keys())  # Get classes in consistent␣
↪order

        for cls in classes:
            other_cls = [c for c in classes if c != cls][0]  # Get the other␣
↪class

            if use_ratio:
                # Calculate probability ratios for all words
                word_scores = []
                for word in self.vocabulary:
                    # Add small epsilon to denominator to avoid division by zero
                    ratio = (self.word_probs[cls][word] /
                            (self.word_probs[other_cls][word] + 1e-4))
                    word_scores.append((word, ratio))
            else:
                # Use raw probabilities
                word_scores = list(self.word_probs[cls].items())

            # Sort by score (either ratio or probability) and take top n
            sorted_words = sorted(word_scores, key=lambda x: x[1], reverse=True)
            important_words[cls] = sorted_words[:n]

        return important_words

# ==== END EVALUATION PORTION
```

[9]:
```
def train_and_evaluate_naive_bayes_example():
    """
    Example demonstrating how to use the NaiveBayesClassifier.
    """
    # Sample training data
    X_example_train = [
        "I love this movie",
        "Great film, amazing actors",
        "Terrible waste of time",
        "Poor acting, bad script",
        "Excellent movie, highly recommend"
    ]
    y_example_train = [1, 1, 0, 0, 1]  # 1: positive, 0: negative

    # Sample validation data
    X_example_val = [
        "Really enjoyed this film",
        "Waste of money, terrible"
    ]
    y_example_val = [1, 0]

    # Train classifier
    nb_classifier = NaiveBayesClassifier(min_freq=1)
    nb_classifier.fit(X_example_train, y_example_train)

    # Make predictions
    predictions = nb_classifier.predict(X_example_val)

    # Evaluate
    accuracy = evaluate_classifier(y_example_val, predictions)
    print(f"Validation accuracy on this example dataset: {accuracy:.4f}")

    # Get and print important words
    important_words = nb_classifier.get_important_words(n=5)
    for class_label, words in important_words.items():
        sentiment = "Negative" if class_label == 0 else "Positive"
        print(f"\nTop words for {sentiment} sentiment:")
        for word, prob in words:
            print(f"{word}: {prob:.4f}")

train_and_evaluate_naive_bayes_example()
```

```
17
Validation accuracy on this example dataset: 1.0000

Top words for Negative sentiment:
terrible: 2.2439
waste: 2.2439
time: 2.2439
poor: 2.2439
acting: 2.2439

Top words for Positive sentiment:
movie: 2.6603
love: 1.7735
great: 1.7735
film: 1.7735
amazing: 1.7735
```

[10]:
```
def train_and_evaluate_naive_bayes_main():
    """
    Train and evaluate the Naive Bayes classifier.
    """
    nb_classifier = NaiveBayesClassifier(min_freq=3)
    nb_classifier.fit(X_train, y_train)

    predictions = nb_classifier.predict(X_val)
    accuracy = evaluate_classifier(y_val, predictions)
    print(f"Validation Accuracy: {accuracy:.4f}")

    # Get and print important words
    important_words = nb_classifier.get_important_words(n=10)
    for class_label, words in important_words.items():
        sentiment = "Negative" if class_label == 0 else "Positive"
        print(f"\nTop words for {sentiment} sentiment:")
        for word, prob in words:
            print(f"{word}: {prob:.4f}")

# Above 80% validation accuracy is good!
train_and_evaluate_naive_bayes_main()
```

```
22654
Validation Accuracy: 0.8433

Top words for Negative sentiment:
worst: 6.1628
awful: 5.3853
waste: 5.0829
stupid: 4.2850
bad: 3.9889
terrible: 3.8646
horrible: 3.8016
worse: 3.0935
crap: 3.0789
poor: 2.9945

Top words for Positive sentiment:
amazing: 2.9914
excellent: 2.7799
wonderful: 2.7467
fantastic: 2.6776
loved: 2.5074
great: 2.2642
favorite: 2.2347
best: 2.1258
today: 2.1067
superb: 2.0682
```

6 Word2Vec and Word Analogies: Understanding Semantic Relationships

Word embeddings have revolutionized NLP by capturing semantic relationships between words in dense vector spaces. Word2Vec, introduced by Mikolov et al. (2013), maps words to continuous vector representations where similar words cluster together and relationships between words are preserved as vector operations.

6.1 Key concepts

- Words are represented as dense vectors in high-dimensional space (typically 300D)
- Similar words have similar vector representations
- Vector arithmetic captures semantic relationships
- Famous example: king - man + woman queen

This notebook explores implementation and evaluation of word analogies using different similarity metrics.

6.2 Download word2vec

[11]:
```
def download_word2vec_model(model_name="word2vec-google-news-300"):
    """
    Download word2vec model using gensim's built-in downloader.

    Args:
        model_name (str): Name of the model to download.

    Returns:
        str: Path to the downloaded model

    Raises:
        ValueError: If the specified model is not available
        Exception: For other download or processing errors
    """
    try:
        # Check if model is available
        available_models = api.info()['models'].keys()
        if model_name not in available_models:
            raise ValueError(
                f"Model '{model_name}' not found. Available models: {', '.
↪join(available_models)}"
            )

        print(f"Downloading {model_name}...")
        model_path = api.load(model_name, return_path=True)
        print(f"Model downloaded successfully to: {model_path}")

        return model_path
    except Exception as e:
        print(f"Error downloading model: {str(e)}")
        raise

word2vec_path = download_word2vec_model()
```

```
Downloading word2vec-google-news-300…
Model downloaded successfully to: C:\Users\myalla/gensim-data\word2vec-google-
news-300\word2vec-google-news-300.gz
```

## Vector Operations and Similarity Metrics

We implement two key similarity metrics: 1. Cosine Similarity: Measures angle between vectors, normalized to [-1,1] 2. Euclidean Similarity: Based on straight-line distance between vectors

The class below handles vector operations and similarity computations.

[12]:
```
# ==== BEGIN EVALUATION PORTION

class WordEmbeddingOps:
    def __init__(self, word2vec_path):
        """
        Initialize the WordEmbeddings class with a pre-trained word2vec model.

        Args:
            word2vec_path (str): Path to the word2vec model file
                Example: 'path/to/GoogleNews-vectors-negative300.bin'

        Note:
            - Loads word vectors using gensim's KeyedVectors
        """
        self.word_vectors = KeyedVectors.load_word2vec_format(
            word2vec_path, binary=True)

    def cosine_similarity(self, vec1, vec2):
        """
        Calculate cosine similarity between two vectors.

        Args:
            vec1 (np.array): First vector
                Example: array([0.2, 0.5, -0.1])
            vec2 (np.array): Second vector
                Example: array([0.3, 0.4, -0.2])

        Returns:
            float: Cosine similarity between vectors
                Example: 0.95 (for above vectors)

        Note:
            - Cosine similarity = vec1 · vec2 / (||vec1|| ||vec2||)
            - Range: [-1, 1], where 1 means same direction
        """
        # BEGIN CODE : word_embedding_ops.cosine_similarity

        return vec1@vec2 / (np.linalg.norm(vec1)*np.linalg.norm(vec2))


        # END CODE

    def euclidean_similarity(self, vec1, vec2):
        """
        Calculate similarity based on Euclidean distance.

        Args:
            vec1 (np.array): First vector
                Example: array([0.2, 0.5, -0.1])
            vec2 (np.array): Second vector
                Example: array([0.3, 0.4, -0.2])

        Returns:
            float: Similarity score based on Euclidean distance
                Example: 0.85

        Note:
            - Converts Euclidean distance to similarity
            - similarity = 1 / (1 + distance)
            - Range: (0, 1], where 1 means identical vectors
        """
        # BEGIN CODE : word_embedding_ops.euclidean_similarity

        return 1 / (1 + np.linalg.norm(vec1 - vec2))


        # END CODE

    def find_analogies(self, word1, word2, word3, similarity_func='cosine',␣
↪num_results=5):
        """
        Find the words that complete the analogy: word1 : word2 :: word3 : ?

        Args:
            word1 (str): First word in the analogy
                Example: 'king'
            word2 (str): Second word in the analogy
                Example: 'man'
            word3 (str): Third word in the analogy
                Example: 'queen'
            num_results (int): Number of top results to return
                Example: 5
            similarity_func (str): Similarity function to use ('cosine' or␣
↪'euclidean')

        Returns:
            list: List of tuples (word, similarity_score) for top num_results␣
↪matches
                Example: [('woman', 0.95), ('girl', 0.82), ('lady', 0.78), ...]

        Note:
            - Uses vector arithmetic: word2 - word1 + word3
            - Excludes input words from results
            - Returns empty list if any input word not in vocabulary
            - Implementation iterates through all words in vocabulary using:
              for word in self.word_vectors.index_to_key
              This is necessary to compare the target vector against every
              possible word in the model's vocabulary
        """
        # BEGIN CODE : word_embedding_ops.find_analogies

        ex_words = [word1, word2, word3]
        try:
            word4_vec = self.word_vectors[word2] - self.word_vectors[word1] +␣
↪self.word_vectors[word3]
        except KeyError:
            return []
        if similarity_func == 'cosine':
            scrs = (self.word_vectors.vectors @ word4_vec.reshape(-1,1))
            w4_norm = np.linalg.norm(word4_vec)
            scrs /= w4_norm
            vec_norms = np.linalg.norm(self.word_vectors.vectors,axis = 1).
↪reshape(-1,1)
            scrs /= vec_norms
            scores = [ (word,score) for word,score in zip(self.word_vectors.
↪index_to_key,scrs.reshape(-1).tolist()) if word not in ex_words]
        else:
            scrs = np.linalg.norm(self.word_vectors.vectors - word4_vec,axis =␣
↪1)
            scrs = 1 / (1 + scrs)
            scores = [ (word,score) for word,score in zip(self.word_vectors.
↪index_to_key,scrs) if word not in ex_words]
        return sorted(scores,key=lambda x: x[1], reverse=True)[:num_results]
        # END CODE

    def find_similar_words(self, word, num_results=5, similarity_func='cosine'):
        """
        Find the most similar words to a given word.

        Args:
            word (str): Input word to find similar words for
                Example: 'computer'
            num_results (int): Number of similar words to return
                Example: 5
            similarity_func (str): Similarity function to use ('cosine' or␣
↪'euclidean')

        Returns:
            list: List of tuples (word, similarity_score) for top num_results␣
↪matches
                Example: [('laptop', 0.89), ('pc', 0.87), ('desktop', 0.85), ...
↪]

        Note:
            - Returns empty list if input word not in vocabulary
            - Excludes the input word from results
            - Implementation requires iterating through entire vocabulary using:
              for word in self.word_vectors.index_to_key
              This exhaustive search is needed to find the most similar words
              by comparing the target word's vector against all known words
        """
        # BEGIN CODE : word_embedding_ops.find_similar_words

        try:
            word_vec = self.word_vectors[word]
        except KeyError:
            return []

        if similarity_func == 'cosine':
            scrs = (self.word_vectors.vectors @ word_vec.reshape(-1,1))
            word_norm = np.linalg.norm(word_vec)
            scrs /= word_norm
            vec_norms = np.linalg.norm(self.word_vectors.vectors,axis = 1).
↪reshape(-1,1)
            scrs /= vec_norms
            scores = [ (word_i,score) for word_i,score in zip(self.word_vectors.
↪index_to_key,scrs.reshape(-1).tolist()) if word_i != word]
        else:
            scrs = np.linalg.norm(self.word_vectors.vectors - word_vec,axis = 1)
            scrs = 1 / (1 + scrs)
            scores = [ (word_i,score) for word_i,score in zip(self.word_vectors.
↪index_to_key,scrs) if word_i != word]
        return sorted(scores,key=lambda x: x[1], reverse=True)[:num_results]

        # END CODE

# ==== END EVALUATION PORTION
```

[13]:
```
word_embedding_ops = WordEmbeddingOps(word2vec_path)
```

## The Classic King-Man-Woman-Queen Analogy

This famous analogy demonstrates how Word2Vec captures gender relationships: - king is to man as queen is to woman - Mathematically: king - man + woman   queen

This relationship emerged naturally during training, showing how embeddings learn semantic patterns.

[14]:
```
def demonstrate_king_man_queen_analogy():
    """
    Demonstrate the famous king:man::queen:woman analogy.

    Note:
        - Shows results using both cosine and euclidean similarity
        - Prints intermediate vectors and calculations
        - Useful for understanding how word analogies work
    """
    print("Testing famous analogy: king:man::queen:?")

    # Try with cosine similarity
    results_cos = word_embedding_ops.find_analogies(
        "king", "man", "queen", similarity_func="cosine")
    print("\nUsing cosine similarity:")
    for word, score in results_cos:
        print(f"  {word}: {score:.3f}")

    # Try with euclidean similarity
    results_euc = word_embedding_ops.find_analogies(
        "king", "man", "queen", similarity_func="euclidean")
    print("\nUsing euclidean similarity:")
    for word, score in results_euc:
        print(f"  {word}: {score:.3f}")

demonstrate_king_man_queen_analogy()
```

```
Testing famous analogy: king:man::queen:?

Using cosine similarity:
  woman: 0.719
  girl: 0.588
  lady: 0.575
  teenage_girl: 0.570
  teenager: 0.538

Using euclidean similarity:
  woman: 0.303
  girl: 0.261
  lady: 0.258
  teenager: 0.255
  vivacious_blonde: 0.253
```

## Examining Gender Bias in Word Embeddings

Word embeddings can reflect and amplify societal biases present in training data. Bolukbasi et al. (2016) in "Man is to Computer Programmer as Woman is to Homemaker?" demonstrated systematic gender biases in word embeddings.

Common problematic analogies: - man:doctor :: woman:nurse - father:businessman :: mother:housewife

These biases can propagate through NLP systems, affecting downstream applications.

[15]:
```python
def demonstrate_gender_bias():
    """
    Demonstrate the famous man:doctor::woman:nurse analogy.

    Note:
        - Shows results using both cosine and euclidean similarity
        - Prints intermediate vectors and calculations
        - Useful for understanding how word analogies work
    """

    examples = [
        ("man", "doctor", "woman"),
        ("father", "doctor", "mother"),
    ]

    for word1, word2, word3 in examples:
        print(f"\nTesting: {word1}:{word2}::{word3}:?")

        # Try with cosine similarity
        results_cos = word_embedding_ops.find_analogies(
            word1, word2, word3, similarity_func="cosine")
        print("\nUsing cosine similarity:")
        for word, score in results_cos:
            print(f"  {word}: {score:.3f}")
        # Try with euclidean similarity
        results_euc = word_embedding_ops.find_analogies(
            word1, word2, word3, similarity_func="euclidean")
        print("\nUsing euclidean similarity:")
        for word, score in results_euc:
            print(f"  {word}: {score:.3f}")

demonstrate_gender_bias()
```

```
Testing: man:doctor::woman:?

Using cosine similarity:
  gynecologist: 0.728
  nurse: 0.670
  physician: 0.667
  doctors: 0.665
  pediatrician: 0.640

Using euclidean similarity:
  gynecologist: 0.272
  doctors: 0.267
  nurse: 0.266
  physician: 0.264
  prenatal_checkup: 0.248

Testing: father:doctor::mother:?

Using cosine similarity:
  nurse: 0.717
  doctors: 0.680
  physician: 0.667
  gynecologist: 0.663
  nurse_practitioner: 0.642

Using euclidean similarity:
  nurse: 0.287
  doctors: 0.278
  physician: 0.270
  CVS_pharmacist: 0.256
  prenatal_checkup: 0.256
```

## Word Similarity and Semantic Clustering

Beyond analogies, word embeddings cluster semantically similar words. The example below shows example of similar word finding.

```
[16]: def demonstrate_similar_words():
          """
          Demonstrate finding similar words for multiple example words.

          Note:
              - Tests similarity for words: cat, india, book, computer, phone
              - Shows results using both cosine and euclidean similarity
              - Prints top 5 similar words for each test word
          """
          test_words = ['india', 'book']

          for word in test_words:
              print(f"\nFinding similar words for: {word}")

              # Try with cosine similarity
              cos_similar = word_embedding_ops.find_similar_words(
                  word, similarity_func='cosine')
              print("Using cosine similarity:")
              for similar_word, score in cos_similar:
                  print(f"  {similar_word}: {score:.3f}")

              # Try with euclidean similarity
              euc_similar = word_embedding_ops.find_similar_words(
                  word, similarity_func='euclidean')
              print("\nUsing euclidean similarity:")
              for similar_word, score in euc_similar:
                  print(f"  {similar_word}: {score:.3f}")

      demonstrate_similar_words()
```

```
Finding similar words for: india
Using cosine similarity:
  indian: 0.697
  usa: 0.684
  pakistan: 0.682
  chennai: 0.668
  america: 0.659

Using euclidean similarity:
  indian: 0.264
  chennai: 0.257
  usa: 0.257
  sri_lanka: 0.255
  modi: 0.252

Finding similar words for: book
Using cosine similarity:
  tome: 0.749
  books: 0.738
  memoir: 0.730
  paperback_edition: 0.687
  autobiography: 0.674

Using euclidean similarity:
  books: 0.351
  Booklocker.com: 0.331
  hardbound_edition: 0.329
  Kimberla_Lawson_Roby: 0.328
  Darin_Strauss: 0.326
```

7 Discriminative Classification

## Bag of Words (BoW) Text Classifier

This class implements text classification using the Bag of Words approach: 1. Convert text to word count vectors 2. Train logistic regression on these vectors 3. Make predictions on new text

Features: - Text preprocessing (lowercase, remove punctuation, stop words) - Vocabulary creation from training data - Word count vectorization - Classification using logistic regression

[17]:
```
# ==== BEGIN EVALUATION PORTION

class BagOfWordsClassifier:
    def __init__(self, min_freq=1):
        """
        Initialize the Bag of Words classifier.

        Args:
            min_freq (int): Minimum frequency threshold for a word to be␣
↪included in vocabulary.
                           Words appearing less than min_freq times will be␣
↪treated as UNK token.
                           Default: 1 (include all words)

        Attributes:
            vocabulary (dict): Word to index mapping, including special UNK␣
↪token
                Example: {'<UNK>': 0, 'good': 1, 'movie': 2}
            classifier: Trained logistic regression model
            min_freq (int): Minimum frequency threshold for vocabulary inclusion
                Example: If min_freq=2, words must appear at least twice to be␣
↪included

        Note:
            - Words appearing less than min_freq times will be mapped to <UNK>
            - <UNK> token is automatically added to vocabulary as first token␣
↪(index 0)
            - Logistic regression is used as the underlying classifier
        """
        self.vocabulary = None
        self.classifier = LogisticRegression(random_state=42)
        self.min_freq = min_freq

    def preprocess_text(self, text):
        """
        Preprocess text by converting to lowercase, removing punctuation,
        and filtering stop words.

        Args:
            text (str): Raw input text
                Example: "This movie was really good!"

        Returns:
            list: Cleaned and tokenized words
                Example: ['movie', 'really', 'good']

        Note:
            - Converts all text to lowercase
            - Removes punctuation and special characters
            - Splits text into individual tokens
            - Removes common English stop words
            - Stop words are removed using NLTK's English stop words list
        """
        from nltk.corpus import stopwords
        stop_words = set(stopwords.words('english'))

        text = text.lower()
        tokens = re.findall(r'\w+', text)
        return [token for token in tokens if token not in stop_words]

    def create_vocabulary(self, texts):
        """
        Create vocabulary from training texts by mapping each unique word to an␣
↪index,
        considering minimum frequency threshold and adding UNK token.

        Args:
            texts (list): List of text documents
                Example: [
                    "good movie good",
                    "bad movie",
                    "great action movie"
                ]

        Returns:
            dict: Word to index mapping, including UNK token
                Example (with min_freq=2): {
                    '<UNK>': 0,     # Special token for rare/unseen words
                    'movie': 1,     # Frequency=3, included in vocab
                    'good': 2,      # Frequency=2, included in vocab
                    # 'bad', 'great', 'action' not included (frequency=1 <␣
↪min_freq=2)
                }

        Note:
            - Always includes <UNK> token at index 0
            - Only includes words that appear >= min_freq times
            - Word frequency is counted across all documents
            - Uses preprocess_text function for preprocessing
            - Words below frequency threshold will be mapped to UNK during␣
↪feature extraction
        """
        # BEGIN CODE : bow.create_vocabulary

        self.UNK = '<UNK>'
        self.freqeuncies = {self.UNK: self.min_freq + 2 }
        for txt in texts:
            for word in self.preprocess_text(txt):
                self.freqeuncies[word] = self.freqeuncies.get(word, 0) + 1
        return {k: i for i, (k, v) in enumerate(self.freqeuncies.items()) if v␣
↪>= self.min_freq}

        # END CODE

    def text_to_bow(self, texts):
        """
        Convert texts to bag-of-words feature vectors using the vocabulary,
        where each element represents the count of word occurrences (not binary␣
↪presence/absence).
        Words not in vocabulary are mapped to UNK token.

        Args:
            texts (list): List of text documents
                Example: ["good movie good watch", "bad movie skip"]

        Returns:
            np.array: Document-term matrix with UNK handling
            Example: For vocabulary {'<UNK>':0, 'movie':1, 'good':2} with␣
↪min_freq=2:
            array([[1, 1, 2],    # First doc: 1 UNK ('watch'), 1 'movie', 2␣
↪'good'
                   [2, 1, 0]])   # Second doc: 2 UNKs ('bad','skip'), 1␣
↪'movie', 0 'good'

        Note:
            - First column represents count of UNK tokens
            - Words not in vocabulary are mapped to UNK token (index 0)
            - Uses preprocess_text function for preprocessing
            - Shape of output: (n_documents, len(vocabulary))
        """
        # BEGIN CODE : bow.text_to_bow

        vocab_size = len(self.vocabulary)
        n_documents = len(texts)
        res = np.zeros((n_documents, vocab_size))
        for row, txt in enumerate(texts):
            for word in self.preprocess_text(txt):
                if word in self.vocabulary:
                    res[row][self.vocabulary[word]] += 1
                else:
                    res[row][self.vocabulary[self.UNK]] += 1
        return res

        # END CODE

    def fit(self, X_text, y):
        """
        Train the classifier on text documents.

        Args:
            X_text (list): List of text documents
                Example: ["good movie", "bad film", "great movie"]
            y (list): Class labels
                Example: [1, 0, 1]  # 1=positive, 0=negative

        Note:
            - Creates vocabulary from training texts using min_freq threshold
            - Converts texts to BoW features with UNK handling
            - Trains logistic regression classifier on features
        """
        # Create vocabulary from training texts
        self.vocabulary = self.create_vocabulary(X_text)

        # Convert texts to BoW features
        X_bow = self.text_to_bow(X_text)

        # Train classifier
        self.classifier.fit(X_bow, y)

    def predict(self, X_text):
        """
        Predict classes for new documents.

        Args:
            X_text (list): List of text documents
                Example: ["amazing film", "terrible movie"]

        Returns:
            list: Predicted class labels
                Example: [1, 0]  # 1=positive, 0=negative

        Note:
            - Unknown words in test documents are mapped to UNK token
            - Uses the same preprocessing as training
        """
        # Convert texts to BoW features
        X_bow = self.text_to_bow(X_text)

        # Make predictions
        return self.classifier.predict(X_bow)

    def get_class_probabilities(self, X_text):
        """
        Calculate prediction confidence scores for each class.

        Args:
            X_text (list): List of text documents
                Example: ["amazing film", "terrible movie"]

        Returns:
            np.array: Confidence scores for each class (values 0-1)
                Example: array([[0.1, 0.9],   # 90% confidence for positive class
                               [0.8, 0.2]])   # 20% confidence for positive class

        Note:
            - Returns probability distribution over classes
            - Each row sums to 1.0
            - For binary classification:
                - First column: confidence for negative class (0)
                - Second column: confidence for positive class (1)
            - Unknown words are handled via UNK token
        """
        X_bow = self.text_to_bow(X_text)
        return self.classifier.predict_proba(X_bow)

# ==== END EVALUATION PORTION
```

## Word2Vec Text Classifier

This class implements text classification using Word2Vec embeddings: 1. Load pre-trained word vectors 2. Represent each document as average of its word vectors 3. Train logistic regression on these dense vectors

Features: - Text preprocessing - Document representation using word embeddings - Classification using logistic regression - Support for different similarity metrics

[18]:
```
# ==== BEGIN EVALUATION PORTION

class Word2VecClassifier:
    def __init__(self, word2vec_path):
        """
        Initialize Word2Vec classifier.

        Args:
            word2vec_path (str): Path to pre-trained word2vec model
                Example: 'path/to/GoogleNews-vectors-negative300.bin'

        Attributes:
            word_vectors: Loaded word vectors
            classifier: Trained logistic regression model
        """
        self.word_vectors = KeyedVectors.load_word2vec_format(
            word2vec_path, binary=True)
        self.classifier = LogisticRegression(random_state=42)

    def preprocess_text(self, text):
        """
        Preprocess text by converting to lowercase, removing punctuation,
        and filtering stop words.

        Args:
            text (str): Raw input text
                Example: "This movie was really good!"

        Returns:
            list: Cleaned and tokenized words
                Example: ['movie', 'really', 'good']
        """
        from nltk.corpus import stopwords
        stop_words = set(stopwords.words('english'))

        text = text.lower()
        tokens = re.findall(r'\w+', text)
        return [token for token in tokens if token not in stop_words]

    def text_to_vec(self, texts):
        """
        Convert texts to document vectors by averaging word embeddings.

        Args:
            texts (list): List of text documents
                Example: ["good movie", "bad film"]

        Returns:
            np.array: Document vectors where each vector is average of its word␣
↪vectors
                Example shape: array([[0.2, 0.3, ..., -0.1],  # 300D vector for␣
↪doc1
                                      [0.1, 0.4, ..., -0.2]])   # 300D vector for␣
↪doc2

        Process:
            1. For each document:
                a. Split into words and preprocess
                b. Look up word2vec vector for each word
                c. Calculate mean of all word vectors in document
                   - e.g., if doc has words [w1, w2, w3]:
                     doc_vector = (vector(w1) + vector(w2) + vector(w3)) / 3
                d. If no words found in vocabulary, vector remains zero

        Note:
            - Implementation hint: Vector size can be obtained using self.
↪word_vectors.vector_size
            - Each document vector has same dimensions as word vectors (e.g.,␣
↪300)
            - Words not in word2vec vocabulary are skipped
            - Document vector is average of all found word vectors
            - Documents with no known words get zero vectors
            - You must use the preprocess_text function for pre-processing
        """

        # BEGIN CODE : word2vec.text_to_vec

        res = []
        vec_size = self.word_vectors.vector_size
        for txt in texts :
            vecs = []
            for word in self.preprocess_text(txt):
                try :
                    vecs.append(self.word_vectors[word])
                except KeyError :
                    continue
            res.append(np.mean(np.array(vecs),axis = 0) if len(vecs) > 0 else␣
↪np.zeros((vec_size)))

        return np.array(res)

        # END CODE

    def fit(self, X_text, y):
        """
        Train classifier on text documents.

        Args:
            X_text (list): List of text documents
                Example: ["good movie", "bad film", "great movie"]
            y (list): Class labels
                Example: [1, 0, 1]  # 1=positive, 0=negative
        """
        # Convert texts to document vectors
        X_vecs = self.text_to_vec(X_text)

        # Train classifier
        self.classifier.fit(X_vecs, y)

    def predict(self, X_text):
        """
        Predict classes for new documents.

        Args:
            X_text (list): List of text documents
                Example: ["amazing film", "terrible movie"]

        Returns:
            list: Predicted class labels
                Example: [1, 0]  # 1=positive, 0=negative
        """
        X_vecs = self.text_to_vec(X_text)
        return self.classifier.predict(X_vecs)

    def get_class_probabilities(self, X_text):
        """
        Calculate prediction confidence scores for each class.

        Args:
            X_text (list): List of text documents
                Example: ["amazing film", "terrible movie"]

        Returns:
            np.array: Confidence scores for each class (values 0-1)
                Example: array([[0.1, 0.9],  # 90% confidence for positive class
                               [0.8, 0.2]])   # 20% confidence for positive class

        Note:
            - Returns probability distribution over classes
            - Each row sums to 1.0
            - For binary classification:
                - First column: confidence for negative class (0)
                - Second column: confidence for positive class (1)
        """
        X_vecs = self.text_to_vec(X_text)
        return self.classifier.predict_proba(X_vecs)

# ==== END EVALUATION PORTION
```

## Training and Evaluation

Now we'll train both classifiers and evaluate their performance on the validation set.

[19]:
```
# Train and evaluate BoW classifier
print("Training Bag of Words classifier...")
bow_clf = BagOfWordsClassifier()
bow_clf.fit(X_train, y_train)

bow_predictions = bow_clf.predict(X_val)
bow_accuracy = evaluate_classifier(y_val, bow_predictions)

# Above 80% validation accuracy is good!
print(f"BoW Validation Accuracy: {bow_accuracy:.4f}")
```

```
Training Bag of Words classifier…
BoW Validation Accuracy: 0.8635
```

[20]:
```
print("\nTraining Word2Vec classifier...")
word2vec_path = download_word2vec_model()
w2v_clf = Word2VecClassifier(word2vec_path)
w2v_clf.fit(X_train, y_train)

w2v_predictions = w2v_clf.predict(X_val)
w2v_accuracy = evaluate_classifier(y_val, w2v_predictions)
```

```python
# Above 80% validation accuracy is good!
print(f"Word2Vec Validation Accuracy: {w2v_accuracy:.4f}")
```

```
Training Word2Vec classifier…
Downloading word2vec-google-news-300…
Model downloaded successfully to: C:\Users\myalla/gensim-data\word2vec-google-
news-300\word2vec-google-news-300.gz
Word2Vec Validation Accuracy: 0.8460
```