Natural Language Processing: Word Embeddings, Text Classification and Sentiment Analysis

Introduction

Welcome to the third module of Week 3 in our AI course! Today we're exploring Natural Language Processing (NLP), which focuses on how computers can **understand, interpret, and generate human language**. By the end of this module, you'll understand the fundamentals of **representing text as numbers**, **classifying text into categories**, and **determining the emotional tone of text**.

1. What is Natural Language Processing?

Natural Language Processing (NLP) is a field at the intersection of computer science, artificial intelligence, and linguistics. It focuses on enabling computers to process, understand, and generate human language in a way that is both meaningful and useful.

NLP brings together:

- **Linguistics**: Understanding the **structure and rules of language**
- **Computer Science**: Building algorithms that can **process text efficiently**
- **Machine Learning**: Creating models that **learn patterns from text data**
- **Statistics**: Analyzing **word frequencies and relationships**

Real-Life Examples

You use NLP technologies every day! For example:

- When you ask Siri or Alexa a question
- When Gmail suggests how to complete your sentences
- When you use Google Translate
- When spam filters keep junk mail out of your inbox
- When you use a spell checker in your word processor

💡 **Interactive Question #1**: Which of these is an example of NLP in daily life?

A) A calculator performing addition

B) A smartphone's facial recognition unlock feature

C) Autocorrect changing "teh" to "the"

D) A thermometer measuring temperature

2. Text Pre-processing: Getting Text Ready

Before we can analyze text, we need to clean and prepare it. This process includes:

2.1 Tokenization

Breaking **text into smaller units** (tokens):

- **Word tokenization**: Splitting text into individual words
- **Sentence tokenization**: Splitting text into sentences

Example:

```
# Simple word tokenization
text = "Natural language processing is fascinating!"
tokens = text.split()
print(tokens)  # ['Natural', 'language', 'processing', 'is', 'fascinating!']

# Using NLTK library
import nltk
nltk.download('punkt')
tokens = nltk.word_tokenize(text)
print(tokens)  # ['Natural', 'language', 'processing', 'is', 'fascinating', '!']
```

Image Source: https://medium.com/@utkarsh.kant/tokenization-a-complete-guide-3f2dd56c0682

2.2 Removing Punctuation and Special Characters

Cleaning text by **removing unnecessary elements**.

2.3 Converting to Lowercase

Making all text consistent **by converting to lowercase**.

2.4 Removing Stop Words

Eliminating common words (like "the", "is", "and") that don't add much meaning.

2.5 Stemming and Lemmatization

Reducing words to their base or root form:

- **Stemming**: Using simple rules to cut off word endings (faster but less accurate)
- **Lemmatization**: Using vocabulary and morphological analysis (slower but more accurate)

Example:

```
# Stemming example
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
words = ["running", "runs", "ran", "runner"]
stemmed = [stemmer.stem(word) for word in words]
print(stemmed)  # ['run', 'run', 'ran', 'runner']

# Lemmatization example
from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')
lemmatizer = WordNetLemmatizer()
lemmatized = [lemmatizer.lemmatize(word, pos='v') for word in words]
print(lemmatized)  # ['run', 'run', 'run', 'run']
```

💡 **Interactive Question #2**: What is the main purpose of removing stop words from text?

A) To make processing faster by reducing the amount of text

B) To eliminate words that carry little meaningful information

C) To fix spelling mistakes in the original text

D) To make the text more grammatically correct

3. Word Embeddings: Representing Words as Numbers

Computers **can't understand words** directly—they need **numbers**. Word embeddings are techniques that **convert words into numerical vectors** in a way that captures their meaning.

3.1 One-Hot Encoding

The simplest way to represent words:

- Create a vector with length equal to vocabulary size
- **Set 1 at the position corresponding to the word**
- **Set 0 everywhere else**

Limitations:

- Very sparse (mostly zeros)
- No relationship between similar words
- Very high-dimensional for large vocabularies

3.2 Word2Vec

This popular technique creates dense vector representations of words:

- Words with **similar meanings** have similar vectors
- **Vector relationships can capture semantic relationships**
- Example: vector("King") - vector("Man") + vector("Woman") ≈ vector("Queen")

Two main approaches:

- **Continuous Bag of Words (CBOW)**: Predicts a word based on its context
- **Skip-gram**: Predicts context words based on a given word

```python
# Simple Word2Vec example using Gensim
from gensim.models import Word2Vec

# Sample sentences
sentences = [
    ["natural", "language", "processing", "is", "fascinating"],
    ["machine", "learning", "models", "process", "language"],
    ["word", "embeddings", "represent", "words", "as", "vectors"]
]

# Train model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, sg=0)  # CBOW mo

# Find similar words
similar_words = model.wv.most_similar("language")
print(similar_words)
```

3.3 GloVe (Global Vectors for Word Representation)

Another popular word embedding technique:

- Uses word co-occurrence statistics from a large corpus
- Pre-trained versions are available for immediate use

3.4 FastText

An extension of Word2Vec that:

- Represents words as bags of character n-grams
- Can create embeddings for words not seen during training
- Works well with morphologically rich languages

💡 **Interactive Question #3**: What is a key advantage of word embedding techniques like Word2Vec over one-hot encoding?

A) They require less memory space

B) They capture semantic relationships between words

C) They're easier to implement

D) They work better for very small datasets

4. Text Classification: Categorizing Documents

Text classification is the task of assigning predefined categories to text documents. Common applications include:

- Spam detection
- Topic categorization
- Language identification
- Author attribution

4.1 Bag of Words (BoW)

A simple representation that:

- Counts word occurrences in documents
- Ignores word order
- Creates a document-term matrix

```python
# Bag of Words example using scikit-learn
from sklearn.feature_extraction.text import CountVectorizer

documents = [
    "I love natural language processing",
    "Machine learning is amazing",
    "NLP combines linguistics and AI"
]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(documents)
print(vectorizer.get_feature_names_out())
print(X.toarray())
```

4.2 TF-IDF (Term Frequency-Inverse Document Frequency)

An improvement over BoW that:

- Gives higher weight to important words
- Reduces the impact of common words
- Often improves classification performance

```python
# TF-IDF example
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()
X_tfidf = tfidf_vectorizer.fit_transform(documents)
print(X_tfidf.toarray())
```

4.3 Classification Algorithms

Several machine learning algorithms can be used for text classification:

- **Naive Bayes**: Fast and works well for text
- **Support Vector Machines (SVM)**: Effective for high-dimensional data
- **Decision Trees and Random Forests**: Good with mixed feature types
- **Neural Networks**: Can capture complex patterns but need more data

Example of a simple text classifier:

```python
# Simple text classification example
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline

# Sample data
X_train = ["This movie is excellent", "The film was terrible", "I enjoyed this show",
y_train = ["positive", "negative", "positive", "negative"]

# Create pipeline
text_clf = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', MultinomialNB()),
])

# Train the model
text_clf.fit(X_train, y_train)

# Predict new examples
X_test = ["This was a great movie", "I didn't like it at all"]
predicted = text_clf.predict(X_test)
print(predicted)  # ['positive', 'negative']
```


💡 **Interactive Question #4**: In a Bag of Words representation, what important feature of language is completely lost?

A) Word frequency

B) Word meaning

C) Word order

D) Word length

5. Sentiment Analysis: Understanding Emotions in Text

Sentiment analysis determines the emotional tone behind text. It can identify:

- **Polarity**: Positive, negative, or neutral sentiment
- **Emotions**: Joy, anger, sadness, fear, etc.
- **Intensity**: How strong the sentiment is

5.1 Rule-Based Approaches

Simple methods that:

- Use dictionaries of positive and negative words
- Count sentiment words
- Apply simple rules for negation

```python
# Simple rule-based sentiment analysis
def simple_sentiment(text):
    positive_words = ["good", "great", "excellent", "wonderful", "happy", "love"]
    negative_words = ["bad", "terrible", "awful", "horrible", "sad", "hate"]

    words = text.lower().split()
    pos_count = sum(1 for word in words if word in positive_words)
    neg_count = sum(1 for word in words if word in negative_words)

    if pos_count > neg_count:
        return "positive"
    elif neg_count > pos_count:
        return "negative"
    else:
        return "neutral"

print(simple_sentiment("I love this wonderful product"))  # positive
print(simple_sentiment("This was a terrible experience"))  # negative
```


5.2 Machine Learning Approaches

More sophisticated methods that:

- Train on labeled datasets
- Can capture context and nuance
- Often outperform rule-based approaches

```python
# ML-based sentiment analysis example
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.pipeline import Pipeline

# Sample data
X_train = [
    "This product is amazing and works great",
    "Terrible customer service, very disappointing",
    "I love this app, it's so useful",
    "Completely failed to meet expectations"
]
y_train = ["positive", "negative", "positive", "negative"]

# Create pipeline
sentiment_clf = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('clf', LinearSVC()),
])

# Train model
sentiment_clf.fit(X_train, y_train)

# Test on new examples
X_test = ["The quality exceeded my expectations", "Not worth the money"]
predicted = sentiment_clf.predict(X_test)
print(predicted)  # ['positive', 'negative']
```

5.3 Challenges in Sentiment Analysis

Sentiment analysis faces several challenges:

- **Sarcasm and irony**: "Oh, great, another delay. Just what I needed."
- **Negation**: "This movie is not bad" (actually positive)
- **Context dependence**: "The villain was terrifying" (positive for a horror movie)
- **Domain specificity**: Different vocabularies in different fields

💡 **Interactive Question #5**: Why is sarcasm particularly challenging for sentiment analysis systems?

A) Sarcastic text uses uncommon vocabulary

B) Sarcastic text is usually very short

C) The literal meaning often contradicts the intended sentiment

D) Sarcasm only appears in informal texts

Summary

Natural Language Processing is a fascinating field that enables computers to work with human language. In this module, we covered:

- Text pre-processing techniques that clean and prepare text data
- Word embeddings like Word2Vec that represent words as meaningful vectors
- Text classification methods that categorize documents
- Sentiment analysis approaches that determine the emotional tone of text

These technologies power many applications we use daily, from virtual assistants to recommendation systems. As NLP continues to advance, particularly with transformer-based models, we're approaching systems that can understand and generate language with remarkable human-like qualities.

Glossary

- **Corpus**: A large collection of texts used for linguistic analysis
- **Lemmatization**: The process of reducing words to their base or dictionary form
- **Named Entity Recognition (NER)**: Identifying entities like people, places, and organizations in text
- **n-gram**: A contiguous sequence of n items from a text sample
- **Polarity**: The positive or negative orientation of sentiment
- **Stemming**: Reducing words to their word stem or root form
- **Stop Words**: Common words like "the", "is", "in" that are often filtered out
- **Tokenization**: Breaking text into smaller units like words or sentences
- **TF-IDF**: A numerical statistic reflecting the importance of a word to a document
- **Word Embedding**: A learned representation for text where words with similar meaning have similar representations

Multiple Choice Answers

1. C) Autocorrect changing "teh" to "the"
2. B) To eliminate words that carry little meaningful information
3. B) They capture semantic relationships between words
4. C) Word order
5. C) The literal meaning often contradicts the intended sentiment
6. B) They can process the entire sequence in parallel rather than sequentially

Thank you for completing the Natural Language Processing module!