Topic Modeling - Intermediate Handout

Machine Learning for Smarter Innovation

1 Topic Modeling - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from gensim import corpora, models
from gensim.models import CoherenceModel, Phrases
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.decomposition import LatentDirichletAllocation, NMF
import warnings
warnings.filterwarnings('ignore')

# Download NLTK resources
nltk.download('stopwords', quiet=True)
nltk.download('wordnet', quiet=True)
nltk.download('punkt', quiet=True)

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

This handout covers practical implementation of topic modeling algorithms for discovering themes in text collections. Topic modeling automatically identifies recurring themes across documents without requiring labeled data. Applications include customer feedback analysis, document categorization, content recommendation, and trend discovery.

1.2 1. Text Preprocessing for Topic Modeling

1.2.1 Concept Overview

Topic modeling quality depends heavily on preprocessing. The goal is to reduce vocabulary to meaningful words that distinguish topics while removing noise. Unlike sentiment analysis where negations matter, topic modeling focuses on content words (nouns, verbs, adjectives) that carry thematic meaning.

1.2.2 Comprehensive Preprocessing Pipeline

```
def preprocess_text(text, custom_stopwords=None, min_word_length=3):
    """
    Clean and prepare text for topic modeling.
    Returns list of tokens.
    """
    if pd.isna(text) or text is None:
        return []

    # Lowercase
    text = text.lower()

    # Remove URLs and email addresses
    text = re.sub(r'http\S+|www\.\S+', '', text)
    text = re.sub(r'\S+@\S+', '', text)

    # Remove special characters, keep only letters
    text = re.sub(r'[^a-z\s]', ' ', text)

    # Tokenize
    tokens = text.split()

    # Build stopwords set
    all_stopwords = stop_words.copy()
    if custom_stopwords:
        all_stopwords.update(custom_stopwords)

    # Filter tokens
    filtered_tokens = []
    for token in tokens:
        if (token not in all_stopwords and
            len(token) >= min_word_length):
            # Lemmatize
            lemma = lemmatizer.lemmatize(token, pos='v')  # verb form
            lemma = lemmatizer.lemmatize(lemma, pos='n')  # noun form
            filtered_tokens.append(lemma)

    return filtered_tokens

# Example usage with custom domain stopwords
domain_stopwords = {'product', 'item', 'thing', 'stuff', 'really', 'very'}

sample_texts = [
    "The new smartphone has amazing battery life and fast charging technology.
    ",
    "I love the sleek design, it's beautiful and modern looking.",
    "Customer service was helpful when I had issues with my order.",
    "The camera quality is excellent, photos come out crystal clear.",
    "Shipping was fast and the packaging was secure and professional.",
    "Great value for money, performance exceeds expectations.",
    "The interface is user-friendly and intuitive to navigate.",
    "Build quality feels premium with solid construction.",
]
```

```
# Preprocess all documents
processed_docs = [preprocess_text(text, domain_stopwords) for text in
    sample_texts]
print(f"Sample processed document: {processed_docs[0]}")
print(f"Total documents: {len(processed_docs)}")
print(f"Average tokens per doc: {np.mean([len(d) for d in processed_docs]):.1f
    }")
```

1.2.3 Adding Bigrams and Trigrams

```
def add_phrases(documents, min_count=5, threshold=100):
    """
    Detect and add common phrases (bigrams, trigrams).
    """
    # Train bigram detector
    bigram = Phrases(documents, min_count=min_count, threshold=threshold)
    bigram_mod = bigram.freeze()

    # Train trigram detector on bigrams
    trigram = Phrases(bigram_mod[documents], min_count=min_count, threshold=
    threshold)
    trigram_mod = trigram.freeze()

    # Apply to documents
    docs_with_phrases = [trigram_mod[bigram_mod[doc]] for doc in documents]

    return docs_with_phrases

# For larger datasets, phrases capture important multi-word concepts
# processed_docs = add_phrases(processed_docs)
```

1.3 2. LDA with Gensim

1.3.1 Concept Overview

Latent Dirichlet Allocation (LDA) is a probabilistic model that assumes documents are mixtures of topics, and topics are mixtures of words. Each document exhibits topics in different proportions, and each topic is characterized by a distribution over words. LDA learns both distributions simultaneously.

1.3.2 Building the Model

```
def build_lda_model(processed_docs, num_topics=5, passes=10, filter_extremes=
    True):
    """
    Build LDA model using Gensim.
    Returns model, corpus, and dictionary.
    """
    # Create dictionary
    dictionary = corpora.Dictionary(processed_docs)

    # Filter extremes
    if filter_extremes:
        dictionary.filter_extremes(
            no_below=2,      # Appear in at least 2 documents
```

```
            no_above=0.5,     # Appear in at most 50% of documents
            keep_n=10000      # Keep top N most frequent
        )

    print(f"Dictionary size: {len(dictionary)}")

    # Create bag-of-words corpus
    corpus = [dictionary.doc2bow(doc) for doc in processed_docs]

    # Build LDA model
    lda_model = models.LdaModel(
        corpus=corpus,
        id2word=dictionary,
        num_topics=num_topics,
        random_state=42,
        passes=passes,
        alpha='auto',
        eta='auto',
        per_word_topics=True
    )

    return lda_model, corpus, dictionary

# Build model
lda_model, corpus, dictionary = build_lda_model(processed_docs, num_topics=4)
```

1.3.3 Exploring Topics

```
def display_topics(model, num_words=10):
    """
    Display topics with their top words and probabilities.
    """
    print("\n" + "="*60)
    print("DISCOVERED TOPICS")
    print("="*60)

    for idx in range(model.num_topics):
        topic_words = model.show_topic(idx, num_words)
        print(f"\nTopic {idx}:")
        for word, prob in topic_words:
            print(f"  {word:<20} {prob:.4f}")

display_topics(lda_model)

def get_document_topics(model, corpus, doc_idx):
    """
    Get topic distribution for a specific document.
    """
    doc_topics = model.get_document_topics(corpus[doc_idx])

    print(f"\nDocument {doc_idx} topic distribution:")
    for topic_id, prob in sorted(doc_topics, key=lambda x: -x[1]):
        print(f"  Topic {topic_id}: {prob:.3f}")

    return doc_topics

# Analyze first document
get_document_topics(lda_model, corpus, 0)
```

1.4 3. Finding Optimal Number of Topics

1.4.1 Coherence Score Evaluation

```
def evaluate_topic_range(processed_docs, dictionary, corpus,
                         min_topics=2, max_topics=15):
    """
    Evaluate LDA models with different numbers of topics.
    Uses coherence score (c_v) for evaluation.
    """
    results = []

    for num_topics in range(min_topics, max_topics + 1):
        # Build model
        model = models.LdaModel(
            corpus=corpus,
            id2word=dictionary,
            num_topics=num_topics,
            random_state=42,
            passes=10,
            alpha='auto'
        )

        # Calculate coherence
        coherence_model = CoherenceModel(
            model=model,
            texts=processed_docs,
            dictionary=dictionary,
            coherence='c_v'
        )
        coherence = coherence_model.get_coherence()

        # Calculate perplexity (lower is better)
        perplexity = model.log_perplexity(corpus)

        results.append({
            'num_topics': num_topics,
            'coherence': coherence,
            'perplexity': perplexity
        })

        print(f"Topics: {num_topics:2d} | Coherence: {coherence:.4f} |
    Perplexity: {perplexity:.4f}")

    return pd.DataFrame(results)

# Evaluate range of topics
# results_df = evaluate_topic_range(processed_docs, dictionary, corpus)
```

1.4.2 Visualization of Results

```
def plot_topic_evaluation(results_df):
    """
    Plot coherence and perplexity vs number of topics.
    """
    fig, axes = plt.subplots(1, 2, figsize=(14, 5))

    # Coherence plot
    axes[0].plot(results_df['num_topics'], results_df['coherence'], 'bo-',
    linewidth=2)
```

```
    axes[0].set_xlabel('Number of Topics')
    axes[0].set_ylabel('Coherence Score (c_v)')
    axes[0].set_title('Topic Coherence (Higher = Better)')
    axes[0].grid(True, alpha=0.3)

    # Perplexity plot
    axes[1].plot(results_df['num_topics'], results_df['perplexity'], 'ro-',
    linewidth=2)
    axes[1].set_xlabel('Number of Topics')
    axes[1].set_ylabel('Log Perplexity')
    axes[1].set_title('Perplexity (Lower = Better)')
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('topic_evaluation.pdf', bbox_inches='tight')
    plt.show()

    # Find optimal
    best_idx = results_df['coherence'].idxmax()
    best_row = results_df.loc[best_idx]
    print(f"\nOptimal: {int(best_row['num_topics'])} topics (coherence: {
    best_row['coherence']:.4f})")

# plot_topic_evaluation(results_df)
```

1.5 4. LDA with Scikit-learn

1.5.1 Alternative Implementation

```
def sklearn_lda(texts, num_topics=5, max_features=5000):
    """
    LDA implementation using scikit-learn.
    Useful for integration with sklearn pipelines.
    """
    # Vectorize documents
    vectorizer = CountVectorizer(
        max_df=0.5,           # Ignore terms in >50% of docs
        min_df=2,             # Ignore terms in <2 docs
        max_features=max_features,
        stop_words='english'
    )
    doc_term_matrix = vectorizer.fit_transform(texts)

    # Build LDA model
    lda = LatentDirichletAllocation(
        n_components=num_topics,
        max_iter=20,
        learning_method='online',
        random_state=42,
        n_jobs=-1
    )
    lda.fit(doc_term_matrix)

    return lda, vectorizer, doc_term_matrix

def display_sklearn_topics(model, vectorizer, num_words=10):
    """
    Display topics from sklearn LDA model.
```

```
    """
    feature_names = vectorizer.get_feature_names_out()

    print("\n" + "="*60)
    print("SKLEARN LDA TOPICS")
    print("="*60)

    for topic_idx, topic in enumerate(model.components_):
        top_words_idx = topic.argsort()[:-num_words-1:-1]
        top_words = [feature_names[i] for i in top_words_idx]

        print(f"\nTopic {topic_idx}:")
        for word, weight in zip(top_words, topic[top_words_idx]):
            print(f"  {word:<20} {weight:.2f}")

# sklearn_model, vectorizer, dtm = sklearn_lda(sample_texts, num_topics=4)
# display_sklearn_topics(sklearn_model, vectorizer)
```

1.6 5. Non-Negative Matrix Factorization (NMF)

1.6.1 Concept Overview

NMF is an alternative to LDA that factorizes the document-term matrix into two non-negative matrices representing document-topic and topic-word relationships. NMF often produces more coherent topics on short texts and is faster to train.

1.6.2 Implementation

```
def nmf_topic_model(texts, num_topics=5, max_features=5000):
    """
    Topic modeling using NMF with TF-IDF.
    """
    # TF-IDF vectorization (better for NMF than raw counts)
    tfidf = TfidfVectorizer(
        max_df=0.5,
        min_df=2,
        max_features=max_features,
        stop_words='english'
    )
    tfidf_matrix = tfidf.fit_transform(texts)

    # Build NMF model
    nmf = NMF(
        n_components=num_topics,
        init='nndsvd',
        random_state=42,
        max_iter=500
    )
    nmf.fit(tfidf_matrix)

    return nmf, tfidf, tfidf_matrix

def compare_lda_nmf(texts, num_topics=4):
    """
    Compare LDA and NMF on the same data.
    """
    # LDA
```

```
    lda, lda_vec, _ = sklearn_lda(texts, num_topics)

    # NMF
    nmf, nmf_vec, _ = nmf_topic_model(texts, num_topics)

    print("LDA Topics:")
    display_sklearn_topics(lda, lda_vec, num_words=5)

    print("\n" + "-"*60)
    print("NMF Topics:")
    display_sklearn_topics(nmf, nmf_vec, num_words=5)

# compare_lda_nmf(sample_texts)
```

1.7 6. Document-Topic Assignment

1.7.1 Assigning Topics to Documents

```
def assign_topics_to_documents(model, vectorizer, texts, threshold=0.2):
    """
    Assign dominant topic(s) to each document.
    """
    # Transform documents
    doc_matrix = vectorizer.transform(texts)
    doc_topics = model.transform(doc_matrix)

    results = []
    for i, (text, topic_dist) in enumerate(zip(texts, doc_topics)):
        dominant_topic = topic_dist.argmax()
        dominant_prob = topic_dist[dominant_topic]

        # Get all topics above threshold
        significant_topics = [(j, prob) for j, prob in enumerate(topic_dist)
    if prob >= threshold]

        results.append({
            'doc_id': i,
            'text': text[:100] + '...' if len(text) > 100 else text,
            'dominant_topic': dominant_topic,
            'topic_prob': dominant_prob,
            'all_topics': significant_topics
        })

    return pd.DataFrame(results)

# topic_assignments = assign_topics_to_documents(sklearn_model, vectorizer,
    sample_texts)
# print(topic_assignments[['doc_id', 'dominant_topic', 'topic_prob']])
```

1.7.2 Topic Distribution Visualization

```
def visualize_topic_distribution(doc_topics_matrix, doc_labels=None):
    """
    Visualize topic distribution across documents.
    """
    num_docs, num_topics = doc_topics_matrix.shape
```

```
    fig, ax = plt.subplots(figsize=(12, 6))

    # Stacked bar chart
    bottom = np.zeros(num_docs)
    for topic_idx in range(num_topics):
        ax.bar(range(num_docs), doc_topics_matrix[:, topic_idx],
               bottom=bottom, label=f'Topic {topic_idx}')
        bottom += doc_topics_matrix[:, topic_idx]

    ax.set_xlabel('Document')
    ax.set_ylabel('Topic Proportion')
    ax.set_title('Topic Distribution Across Documents')
    ax.legend(loc='upper right')

    if doc_labels:
        ax.set_xticks(range(num_docs))
        ax.set_xticklabels(doc_labels, rotation=45, ha='right')

    plt.tight_layout()
    plt.savefig('topic_distribution.pdf', bbox_inches='tight')
    plt.show()
```

1.8 7. Interactive Visualization with pyLDAvis

1.8.1 Creating Visualizations

```
def create_lda_visualization(model, corpus, dictionary, output_file='lda_vis.
    html'):
    """
    Create interactive pyLDAvis visualization.
    """
    try:
        import pyLDAvis
        import pyLDAvis.gensim_models as gensimvis

        # Prepare visualization
        vis_data = gensimvis.prepare(model, corpus, dictionary)

        # Save as HTML
        pyLDAvis.save_html(vis_data, output_file)
        print(f"Visualization saved to: {output_file}")

        return vis_data

    except ImportError:
        print("pyLDAvis not installed. Install with: pip install pyLDAvis")
        return None

# create_lda_visualization(lda_model, corpus, dictionary)
```

1.9 Common Hyperparameters

Parameter	Algorithm	Typical Range	Notes
num_topics	LDA/NMF	5-50	Use coherence to optimize
alpha	LDA	'auto', 0.1-1.0	Document-topic density
eta/beta	LDA	'auto', 0.01-0.1	Topic-word density
passes	LDA (Gensim)	5-20	More = better but slower
max_iter	LDA/NMF (sklearn)	10-50	Increase if not converged
min_df	Vectorizer	2-10	Filter rare words
max_df	Vectorizer	0.5-0.9	Filter common words
max_features	Vectorizer	1000-10000	Vocabulary size

1.10 Practice Projects

1. **Customer Feedback Analysis**: Apply topic modeling to product reviews. Identify key themes (quality, price, service), track topic prevalence over time, and generate automated summaries.
2. **News Article Categorization**: Build a topic model on news articles. Use topics for automatic categorization and discover trending themes across time periods.
3. **Research Paper Discovery**: Model abstracts from academic papers in a field. Use topics to find related work and identify emerging research areas.
4. **Support Ticket Classification**: Apply topic modeling to support tickets to identify common issues, prioritize by topic, and route to appropriate teams.

1.11 Troubleshooting

() () ()
* * *

0.3 Solution

() () ()
* * *

0.3 Common-
co-pre- prove
her- pro- stop-
ent- cess- word
top- in- re-
ics gre- moval,
add
do-
main
words

() () ()
* * *

0.3 Common-
do- few- crease
sam- hop- num_topics
top- ics

Problem	Cause	Solution
Many similar topics	Too many topics	Decrease num_topics
Low coherence score	Poor hyperparameters	Tune alpha/beta, more passes
Slow training	Large vocabulary	Reduce max_features, filter extremes
Topics too general	High max_df	Lower max_df to 0.3-0.5
Topics too specific	Low min_df	Increase min_df
Random words in topics	Insufficient data	Need more documents

1.12 Next Steps

- Read the advanced handout for hierarchical topic models and dynamic topic modeling
- Experiment with BERTopic for neural topic modeling
- Apply topic modeling to your organization's document collections
- Build topic-based search and recommendation systems
- Implement topic drift detection for monitoring

Topic modeling reveals hidden themes in text. The goal is interpretable, actionable topics that domain experts recognize as meaningful. Always validate with human evaluation alongside coherence metrics.