Handout 3: Production Topic Modeling Advanced Techniques (Advanced Level)

Machine Learning for Smarter Innovation

1 Handout 3: Production Topic Modeling & Advanced Techniques (Advanced Level)

1.1 Building Production-Grade Topic Modeling Systems

This handout covers advanced techniques for deploying topic models at scale, including real-time processing, multi-lingual support, and integration with modern ML pipelines.

1.2 Advanced Algorithms

1.2.1 1. BERTopic - Transformer-Based Topic Modeling

```
from bertopic import BERTopic
from sentence_transformers import SentenceTransformer
from umap import UMAP
from hdbscan import HDBSCAN
from sklearn.feature_extraction.text import CountVectorizer

# Custom embedding model for domain-specific text
sentence_model = SentenceTransformer("all-MiniLM-L6-v2")

# Dimensionality reduction with UMAP
umap_model = UMAP(n_neighbors=15, n_components=5,
                  min_dist=0.0, metric='cosine', random_state=42)

# Clustering with HDBSCAN
hdbscan_model = HDBSCAN(min_cluster_size=10, metric='euclidean',
                        cluster_selection_method='eom', prediction_data=True)

# Custom vectorizer with n-grams
vectorizer_model = CountVectorizer(ngram_range=(1, 3), stop_words="english",
                                   min_df=2, max_features=5000)

# Create BERTopic model
topic_model = BERTopic(
    embedding_model=sentence_model,
    umap_model=umap_model,
    hdbscan_model=hdbscan_model,
    vectorizer_model=vectorizer_model,
    top_n_words=10,
    language='english',
    calculate_probabilities=True,
    verbose=True
)

# Fit the model
topics, probs = topic_model.fit_transform(documents)

# Dynamic topic modeling over time
topics_over_time = topic_model.topics_over_time(
    docs=documents,
    timestamps=timestamps,
    global_tuning=True,
    evolution_tuning=True,
    nr_bins=20
)
```

1.2.2 2. Hierarchical Topic Modeling

```
from gensim.models import HdpModel
import numpy as np

# Hierarchical Dirichlet Process (infinite topics)
hdp = HdpModel(corpus, dictionary)

# Get topic hierarchy
def extract_topic_hierarchy(hdp_model, num_words=10):
    """Extract hierarchical topic structure."""
    topics = []
    for topic_id in range(hdp_model.m_lambda.shape[0]):
        if hdp_model.m_lambda[topic_id].sum() > 0:
            top_words = hdp_model.show_topic(topic_id, topn=num_words)
            topics.append({
                'id': topic_id,
                'weight': hdp_model.m_lambda[topic_id].sum(),
                'words': top_words
            })

    # Sort by weight (importance)
    topics.sort(key=lambda x: x['weight'], reverse=True)
    return topics

hierarchy = extract_topic_hierarchy(hdp)
```

1.2.3 3. Correlated Topic Models (CTM)

```
import tomotopy as tp

# Correlated Topic Model for capturing topic relationships
ctm = tp.CTModel(k=10, corpus=corpus)

# Train model
for i in range(100):
    ctm.train(10)
    if i % 20 == 0:
        print(f"Iteration {i}, Log-likelihood: {ctm.ll_per_word}")

# Get topic correlations
correlations = ctm.get_correlation()
print("Topic correlation matrix shape:", correlations.shape)
```

```
# Find strongly correlated topics
import numpy as np
strong_correlations = np.where(np.abs(correlations) > 0.5)
for i, j in zip(strong_correlations[0], strong_correlations[1]):
    if i < j:  # Avoid duplicates
        print(f"Topics {i} and {j}: correlation = {correlations[i,j]:.3f}")
```

1.3 Production Architecture

1.3.1 1. Real-Time Topic Modeling Pipeline

```
from kafka import KafkaConsumer, KafkaProducer
import json
import redis
from datetime import datetime
import logging

class RealTimeTopicModeler:
    def __init__(self, model_path, redis_host='localhost'):
        """Initialize real-time topic modeling system."""
        self.model = self.load_model(model_path)
        self.redis_client = redis.Redis(host=redis_host, decode_responses=True
    )
        self.producer = KafkaProducer(
            value_serializer=lambda v: json.dumps(v).encode('utf-8')
        )
        self.consumer = KafkaConsumer(
            'text-stream',
            value_deserializer=lambda m: json.loads(m.decode('utf-8'))
        )
        logging.basicConfig(level=logging.INFO)

    def process_stream(self):
        """Process incoming text stream."""
        batch = []
        batch_size = 100

        for message in self.consumer:
            batch.append(message.value)

            if len(batch) >= batch_size:
                # Process batch
                topics = self.model.transform(batch)

                # Update model (online learning)
                self.model.partial_fit(batch)

                # Store results
                self.store_topics(batch, topics)

                # Send to downstream
                self.producer.send('topic-results', {
                    'timestamp': datetime.now().isoformat(),
                    'batch_size': len(batch),
                    'topics': topics.tolist()
                })

                # Reset batch
                batch = []
                logging.info(f"Processed batch of {batch_size} documents")
    def store_topics(self, documents, topics):
        """Store topics in Redis with TTL."""
        pipeline = self.redis_client.pipeline()

        for doc, topic_dist in zip(documents, topics):
            doc_id = hashlib.md5(doc.encode()).hexdigest()
            pipeline.setex(
                f"topic:{doc_id}",
                86400,  # 24 hour TTL
                json.dumps(topic_dist.tolist())
            )

        pipeline.execute()

# Deploy
modeler = RealTimeTopicModeler('models/production_lda.pkl')
modeler.process_stream()
```

1.3.2 2. Multi-Language Topic Modeling

```
from transformers import pipeline
import langdetect
from collections import defaultdict

class MultilingualTopicModeler:
    def __init__(self):
        """Initialize multilingual topic modeling."""
        self.models = {
            'en': BERTopic(language='english'),
            'es': BERTopic(language='spanish'),
            'fr': BERTopic(language='french'),
            'de': BERTopic(language='german'),
            'zh': BERTopic(language='chinese')
        }

        # Zero-shot topic classifier for cross-lingual topics
        self.classifier = pipeline(
            "zero-shot-classification",
            model="facebook/bart-large-mnli"
        )

    def process_multilingual_corpus(self, documents):
        """Process documents in multiple languages."""
        # Detect languages
        docs_by_lang = defaultdict(list)
        for doc in documents:
            lang = langdetect.detect(doc)
            if lang in self.models:
                docs_by_lang[lang].append(doc)

        # Model each language
        topics_by_lang = {}
        for lang, docs in docs_by_lang.items():
            if len(docs) > 10:  # Minimum documents
                topics, _ = self.models[lang].fit_transform(docs)
                topics_by_lang[lang] = topics

        # Align topics across languages
        aligned_topics = self.align_crosslingual_topics(topics_by_lang)
        return aligned_topics
    def align_crosslingual_topics(self, topics_by_lang):
        """Align topics across languages using embeddings."""
        from sentence_transformers import SentenceTransformer
        model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

        # Get topic representations for each language
        topic_embeddings = {}
        for lang, model in self.models.items():
            if lang in topics_by_lang:
                topic_words = model.get_topics()
                embeddings = []
                for topic_id, words in topic_words.items():
                    if topic_id != -1:  # Skip outliers
                        text = ' '.join([w[0] for w in words[:10]])
                        emb = model.encode(text)
                        embeddings.append(emb)
                topic_embeddings[lang] = np.array(embeddings)

        # Find similar topics across languages
        from sklearn.metrics.pairwise import cosine_similarity
        similarity_threshold = 0.7

        aligned = []
        for lang1 in topic_embeddings:
            for lang2 in topic_embeddings:
                if lang1 < lang2:
                    sims = cosine_similarity(
                        topic_embeddings[lang1],
                        topic_embeddings[lang2]
                    )
                    matches = np.where(sims > similarity_threshold)
                    for i, j in zip(matches[0], matches[1]):
                        aligned.append({
                            'lang1': lang1, 'topic1': i,
                            'lang2': lang2, 'topic2': j,
                            'similarity': sims[i, j]
                        })

        return aligned
```

1.3.3 3. Incremental Learning & Model Updates

```
class IncrementalTopicModel:
    def __init__(self, base_model_path):
        """Initialize incremental learning system."""
        self.base_model = self.load_model(base_model_path)
        self.update_buffer = []
        self.update_threshold = 1000
        self.version = 1

    def incremental_update(self, new_documents):
        """Update model incrementally with new documents."""
        self.update_buffer.extend(new_documents)

        if len(self.update_buffer) >= self.update_threshold:
            # Prepare data
            new_corpus = self.preprocess(self.update_buffer)

            # Online LDA update
            self.base_model.update(new_corpus)
            # Evaluate drift
            drift = self.evaluate_topic_drift()

            if drift > 0.3:  # Significant drift
                # Retrain from scratch with combined data
                self.full_retrain()
            else:
                # Save incremental update
                self.save_checkpoint()

            # Clear buffer
            self.update_buffer = []
            self.version += 1

            logging.info(f"Model updated to version {self.version}, drift: {
    drift:.3f}")

    def evaluate_topic_drift(self):
        """Measure topic drift using Jensen-Shannon divergence."""
        from scipy.stats import entropy
        from scipy.spatial.distance import jensenshannon

        # Get topic distributions before and after update
        old_topics = self.load_previous_topics()
        new_topics = self.base_model.get_topics()

        # Calculate average JS divergence
        divergences = []
        for old, new in zip(old_topics, new_topics):
            div = jensenshannon(old, new) ** 2
            divergences.append(div)

        return np.mean(divergences)

    def full_retrain(self):
        """Retrain model from scratch when drift is high."""
        # Load all historical data
        all_data = self.load_all_training_data()

        # Add recent data
        all_data.extend(self.update_buffer)

        # Train new model
        new_model = train_fresh_model(all_data)

        # A/B test new model
        if self.ab_test(self.base_model, new_model):
            self.base_model = new_model
            logging.info("Switched to retrained model")
        else:
            logging.info("Keeping existing model")
```

1.4 Advanced Evaluation Metrics

1.4.1 1. Topic Diversity & Coverage

```
def calculate_topic_diversity(model, top_n_words=10):
    """Calculate diversity of topics (unique words / total words)."""
    topics = model.get_topics()
    all_words = []
```

```python
    for topic_id, words in topics.items():
        if topic_id != -1:  # Skip outlier topic
            all_words.extend([w[0] for w in words[:top_n_words]])

    unique_words = len(set(all_words))
    total_words = len(all_words)

    diversity = unique_words / total_words
    return diversity

def calculate_topic_coverage(model, documents):
    """Calculate percentage of documents with confident topic assignment."""
    topic_probs = model.transform(documents)

    # Documents with max probability > threshold
    confident_assignments = np.max(topic_probs, axis=1) > 0.3
    coverage = np.mean(confident_assignments)

    return coverage
```

1.4.2 2. Stability Analysis

```python
def topic_stability_analysis(documents, n_runs=10):
    """Analyze topic stability across multiple runs."""
    from sklearn.metrics import adjusted_rand_score

    topic_assignments = []

    for run in range(n_runs):
        model = BERTopic(random_state=run)
        topics, _ = model.fit_transform(documents)
        topic_assignments.append(topics)

    # Calculate pairwise stability
    stability_scores = []
    for i in range(n_runs):
        for j in range(i+1, n_runs):
            score = adjusted_rand_score(
                topic_assignments[i],
                topic_assignments[j]
            )
            stability_scores.append(score)

    mean_stability = np.mean(stability_scores)
    std_stability = np.std(stability_scores)

    print(f"Topic Stability: {mean_stability:.3f}    {std_stability:.3f}")
    return mean_stability, std_stability
```

1.5 Deployment Best Practices

1.5.1 1. Model Versioning

```python
import mlflow

# Track experiments
mlflow.start_run()
mlflow.log_param("num_topics", 20)
mlflow.log_param("algorithm", "LDA")
mlflow.log_metric("coherence", 0.65)
mlflow.sklearn.log_model(model, "topic_model")
mlflow.end_run()
```

1.5.2 2. API Design

```
from fastapi import FastAPI, BackgroundTasks
from pydantic import BaseModel
from typing import List, Dict

app = FastAPI()

class TextRequest(BaseModel):
    texts: List[str]
    num_topics: int = 10

class TopicResponse(BaseModel):
    topics: List[Dict[str, float]]
    processing_time: float

@app.post("/topics", response_model=TopicResponse)
async def get_topics(request: TextRequest, background_tasks: BackgroundTasks):
    """API endpoint for topic extraction."""
    start_time = time.time()

    # Process
    topics = model.transform(request.texts)

    # Background task for model update
    background_tasks.add_task(update_model, request.texts)

    return TopicResponse(
        topics=topics.tolist(),
        processing_time=time.time() - start_time
    )
```

1.5.3 3. Monitoring & Alerts

```
from prometheus_client import Counter, Histogram, Gauge

# Metrics
topics_processed = Counter('topics_processed_total', 'Total topics processed')
processing_time = Histogram('processing_time_seconds', 'Processing time')
model_coherence = Gauge('model_coherence', 'Current model coherence')

def monitor_model_health():
    """Monitor model health metrics."""
    # Check coherence
    coherence = calculate_coherence(model)
    model_coherence.set(coherence)

    if coherence < 0.4:
        send_alert("Model coherence below threshold")

    # Check topic distribution
    topic_sizes = get_topic_distribution(model)
    if max(topic_sizes) / sum(topic_sizes) > 0.5:
        send_alert("Topic imbalance detected")
```

1.6 Exercise: Build Production Pipeline

1.6.1 Task

Create a complete production pipeline that: 1. Ingests real-time text data 2. Applies topic modeling 3. Stores results in database 4. Provides REST API 5. Monitors model quality

1.6.2 Evaluation Criteria

- Handles 1000+ documents/minute
- Coherence > 0.5
- API latency < 100ms
- Automatic model updates
- Comprehensive monitoring

Remember: Production systems require robustness, scalability, and maintainability beyond just model accuracy.