Topic Modeling - Advanced Handout

Machine Learning for Smarter Innovation

1 Topic Modeling - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations and production systems)

1.1 Mathematical Foundations

1.1.1 Latent Dirichlet Allocation Generative Model

LDA models documents as mixtures of topics, where each topic is a distribution over words. The generative process for corpus D with K topics, vocabulary V, and N_d words in document d:

1. For each topic k in {1,...,K}:
   - Draw word distribution phi_k ~ Dirichlet(beta)
2. For each document d in {1,...,D}:
   - Draw topic distribution theta_d ~ Dirichlet(alpha)
   - For each word position n in {1,...,N_d}:
     - Draw topic assignment z_{d,n} ~ Categorical(theta_d)
     - Draw word w_{d,n} ~ Categorical(phi_{z_{d,n}})

The joint probability:

P(W, Z, theta, phi | alpha, beta) = prod_k P(phi_k|beta) * prod_d [P(theta_d|alpha) * prod_n P(z_{d,n}|theta_d) P(w_{d,n}|phi_{z_{d,n}})]

1.1.2 Dirichlet Distribution

The Dirichlet distribution over K-dimensional simplex with parameter alpha:

P(theta | alpha) = Gamma(sum_k alpha_k) / prod_k Gamma(alpha_k) * prod_k theta_kˆ{alpha_k - 1}

Key properties: - E[theta_k] = alpha_k / sum_j alpha_j - Var(theta_k) = alpha_k(sum_j alpha_j - alpha_k) / [(sum_j alpha_j)ˆ2 (sum_j alpha_j + 1)]

Symmetric alpha = (alpha_0/K, ..., alpha_0/K): smaller alpha_0 yields sparser topic distributions.

1.1.3 Likelihood and Evidence

The marginal likelihood (evidence) integrates out latent variables:

P(W | alpha, beta) = int P(W, Z, theta, phi | alpha, beta) dtheta dphi dZ

This integral is intractable due to coupling between theta, phi, and Z. Two inference strategies: variational inference and Gibbs sampling.

1.2 Variational Inference

1.2.1 Mean-Field Approximation

Approximate the true posterior with factorized distribution:

q(Z, theta, phi | gamma, phi_hat, lambda) = prod_d q(theta_d|gamma_d) * prod_k q(phi_k|lambda_k) * prod_{d,n} q(z_{d,n}|phi_hat_{d,n})

The Evidence Lower Bound (ELBO):

L(q) = E_q[log P(W, Z, theta, phi | alpha, beta)] - E_q[log q(Z, theta, phi)] = E_q[log P(W, Z, theta, phi | alpha, beta)] + H[q]

Since log P(W) >= L(q), maximizing ELBO tightens the bound on log-likelihood.

1.2.2 Coordinate Ascent Updates

Optimizing ELBO yields update equations:

Topic assignment: phi_hat_{d,n,k} proportional to exp(E_q[log theta_{d,k}] + E_q[log phi_{k,w_{d,n}}])

where: - E_q[log theta_{d,k}] = psi(gamma_{d,k}) - psi(sum_j gamma_{d,j}) - E_q[log phi_{k,v}] = psi(lambda_{k,v}) - psi(sum_w lambda_{k,w}) - psi() is the digamma function

Document-topic: gamma_{d,k} = alpha_k + sum_n phi_hat_{d,n,k}

Topic-word: lambda_{k,v} = beta_v + sum_d sum_{n: w_{d,n}=v} phi_hat_{d,n,k}

Iterate until convergence. The algorithm is guaranteed to increase ELBO at each step.

1.2.3 Online Variational Inference

For large corpora, online learning processes documents in mini-batches:

1. E-step: Update local parameters (gamma, phi_hat) for mini-batch
2. M-step: Update global parameters (lambda) using stochastic gradient:

lambda_new = (1 - rho_t) * lambda_old + rho_t * lambda_tilde

where rho_t = (tau_0 + t)^{-kappa} is learning rate, and lambda_tilde is the natural gradient estimate from mini-batch.

This achieves O(D) memory instead of O(D*K) for batch inference.

1.3 Gibbs Sampling

1.3.1 Collapsed Gibbs Sampler

Integrate out theta and phi analytically, sample only Z:

P(z_{d,n} = k | Z_{-d,n}, W, alpha, beta) proportional to (n_{d,k,-n} + alpha_k) * (n_{k,w_{d,n},-n} + beta_{w_{d,n}}) / (n_{k,.,-n} + sum_v beta_v)

where: - n_{d,k,-n}: count of topic k in document d, excluding position n - n_{k,v,-n}: count of word v assigned to topic k, excluding position n - n_{k,.,-n}: total words assigned to topic k, excluding position n

1.3.2 Sampling Procedure

1. Initialize Z randomly
2. For each iteration t:
    - For each document d, word position n:
        - Decrement counts for current z_{d,n}
        - Sample new z_{d,n} from conditional
        - Increment counts for new z_{d,n}
3. After burn-in, collect samples for inference

Point estimates from samples:

theta_{d,k} = (n_{d,k} + alpha_k) / (N_d + sum_j alpha_j) phi_{k,v} = (n_{k,v} + beta_v) / (n_{k,.} + sum_w beta_w)

1.3.3 Convergence Diagnostics

Log-likelihood monitoring: Track P(W|Z,alpha,beta) over iterations. Should stabilize after burn-in.

Effective Sample Size (ESS): Measures independent samples accounting for autocorrelation:

ESS = n / (1 + 2 * sum_{t=1}^{inf} rho_t)

where rho_t is autocorrelation at lag t. ESS $<$ 100 suggests more samples needed.

Gelman-Rubin diagnostic: Run multiple chains, compare within-chain and between-chain variance. R-hat $<$ 1.1 indicates convergence.

1.4 Hierarchical Dirichlet Process

1.4.1 Infinite Topic Model

HDP extends LDA to automatically infer the number of topics. The generative process:

1. Draw global topic distribution G_0 ~ DP(gamma, H)
2. For each document d:
    - Draw document-topic distribution G_d ~ DP(alpha, G_0)
    - For each word position n:
        - Draw topic z_{d,n} ~ G_d
        - Draw word w_{d,n} ~ F(z_{d,n})

The Dirichlet Process DP(alpha, G_0) is a distribution over distributions with concentration parameter alpha and base distribution G_0.

1.4.2 Chinese Restaurant Franchise

HDP admits a constructive representation called the Chinese Restaurant Franchise:

- Restaurants (documents) share dishes (topics) from a global menu
- Customer n in restaurant d sits at table t with probability:
    - Existing table: proportional to n_{d,t} (customers at table)
    - New table: proportional to alpha * P(dish at new table | global)

New tables draw dishes from the global distribution: - Existing dish k: proportional to m_k (tables serving dish k) - New dish: proportional to gamma * H(new dish)

This naturally allows the number of topics to grow with data.

1.5 Evaluation Metrics

1.5.1 Perplexity

Perplexity measures generalization to held-out documents:

perplexity(D_test) = exp(-1/N * sum_d log P(w_d | model))

where N is total words in test set. Lower perplexity indicates better fit.

For LDA with variational inference:

log P(w_d) >= L(q_d) = E_q[log P(w_d, z_d, theta_d | alpha, phi)] - E_q[log q(z_d, theta_d)]

Compute bound using held-out variational parameters.

1.5.2 Coherence Metrics

Pointwise Mutual Information (PMI):

PMI(w_i, w_j) = log P(w_i, w_j) / (P(w_i) * P(w_j))

where P(w_i, w_j) is co-occurrence probability in sliding window.

Normalized PMI (NPMI):

NPMI(w_i, w_j) = PMI(w_i, w_j) / (-log P(w_i, w_j))

Ranges from -1 (never co-occur) to +1 (always co-occur).

C_V coherence (state of art): 1. Segment top words using sliding window 2. Compute NPMI for all word pairs 3. Aggregate using normalized vector similarity

C_V > 0.5 indicates coherent topics; C_V > 0.6 is excellent.

UMass coherence:

C_UMass = (2 / (n*(n-1))) sum_{i<j} log((D(w_i, w_j) + 1) / D(w_i))

where D(w_i, w_j) is document co-occurrence count. Uses only corpus statistics, no external data.

1.6 Implementation

1.6.1 Setup

```
import numpy as np
import pandas as pd
from scipy.special import psi, gammaln
from scipy.stats import entropy
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
from gensim.models import LdaModel, CoherenceModel
from gensim.corpora import Dictionary
import warnings
warnings.filterwarnings('ignore')
```

1.6.2 Variational LDA from Scratch

```
class VariationalLDA:
    """LDA with variational inference implementation."""

    def __init__(self, n_topics, alpha=None, beta=None, max_iter=100, tol=1e
    -4):
        self.n_topics = n_topics
        self.alpha = alpha if alpha is not None else np.ones(n_topics) /
    n_topics
        self.beta = beta
        self.max_iter = max_iter
        self.tol = tol

    def fit(self, corpus):
        """Fit LDA using coordinate ascent variational inference."""
        self.vocab_size = corpus.max() + 1
        self.n_docs = len(corpus)

        if self.beta is None:
            self.beta = np.ones(self.vocab_size) / self.vocab_size

        # Initialize global parameters
        self.lambda_ = np.random.gamma(100, 1/100,
                                       (self.n_topics, self.vocab_size))

        # Document-level parameters
        self.gamma = np.zeros((self.n_docs, self.n_topics))

        for iteration in range(self.max_iter):
            elbo_old = self._compute_elbo(corpus)

            # E-step: update document-level parameters
            for d, doc in enumerate(corpus):
                self._update_document(d, doc)

            # M-step: update global parameters
            self._update_global(corpus)

            elbo_new = self._compute_elbo(corpus)

            if abs(elbo_new - elbo_old) < self.tol:
                print(f"Converged at iteration {iteration}")
                break

        return self

    def _update_document(self, d, doc):
        """Update variational parameters for single document."""
        word_counts = np.bincount(doc, minlength=self.vocab_size)

        # Initialize gamma
        gamma_d = self.alpha + word_counts.sum() / self.n_topics

        for _ in range(20):  # Local iterations
            # E[log theta]
            e_log_theta = psi(gamma_d) - psi(gamma_d.sum())

            # E[log phi]
            e_log_phi = psi(self.lambda_) - psi(self.lambda_.sum(axis=1,
    keepdims=True))

            # Update phi (topic assignment probabilities)
            phi = np.zeros((len(doc), self.n_topics))
            for n, w in enumerate(doc):
```

```
                log_phi_n = e_log_theta + e_log_phi[:, w]
                phi[n] = np.exp(log_phi_n - log_phi_n.max())
                phi[n] /= phi[n].sum()

            # Update gamma
            gamma_new = self.alpha + phi.sum(axis=0)

            if np.allclose(gamma_d, gamma_new):
                break
            gamma_d = gamma_new

        self.gamma[d] = gamma_d
        return phi

    def _update_global(self, corpus):
        """Update global topic-word parameters."""
        lambda_new = np.tile(self.beta, (self.n_topics, 1))

        for d, doc in enumerate(corpus):
            phi = self._update_document(d, doc)
            for n, w in enumerate(doc):
                lambda_new[:, w] += phi[n]

        self.lambda_ = lambda_new

    def _compute_elbo(self, corpus):
        """Compute Evidence Lower Bound."""
        elbo = 0

        # E[log p(phi|beta)]
        for k in range(self.n_topics):
            elbo += gammaln(self.beta.sum()) - gammaln(self.beta).sum()
            elbo += ((self.beta - 1) * (psi(self.lambda_[k]) -
                     psi(self.lambda_[k].sum()))).sum()

        # E[log p(theta|alpha)] - E[log q(theta)]
        for d in range(self.n_docs):
            elbo += gammaln(self.alpha.sum()) - gammaln(self.alpha).sum()
            elbo += ((self.alpha - 1) * (psi(self.gamma[d]) -
                     psi(self.gamma[d].sum()))).sum()
            elbo -= gammaln(self.gamma[d].sum()) - gammaln(self.gamma[d]).sum
()
            elbo -= ((self.gamma[d] - 1) * (psi(self.gamma[d]) -
                     psi(self.gamma[d].sum()))).sum()

        return elbo

    def get_topic_words(self, n_words=10):
        """Get top words for each topic."""
        topics = {}
        phi = self.lambda_ / self.lambda_.sum(axis=1, keepdims=True)

        for k in range(self.n_topics):
            top_indices = phi[k].argsort()[-n_words:][::-1]
            topics[k] = [(idx, phi[k, idx]) for idx in top_indices]

        return topics

    def transform(self, corpus):
        """Get topic distributions for documents."""
        theta = self.gamma / self.gamma.sum(axis=1, keepdims=True)
        return theta
```

1.6.3 Production Topic Model

```
class ProductionTopicModel:
    """Production-ready topic model with monitoring."""

    def __init__(self, n_topics=20, coherence_threshold=0.5):
        self.n_topics = n_topics
        self.coherence_threshold = coherence_threshold
        self.model = None
        self.dictionary = None
        self.vectorizer = None
        self.coherence_history = []

    def preprocess(self, documents):
        """Preprocess documents for topic modeling."""
        import re
        from nltk.corpus import stopwords
        from nltk.stem import WordNetLemmatizer

        stop_words = set(stopwords.words('english'))
        lemmatizer = WordNetLemmatizer()

        processed = []
        for doc in documents:
            # Lowercase and remove punctuation
            doc = re.sub(r'[^\w\s]', '', doc.lower())

            # Tokenize and filter
            tokens = [lemmatizer.lemmatize(w) for w in doc.split()
                      if w not in stop_words and len(w) > 2]
            processed.append(tokens)

        return processed

    def fit(self, documents, passes=10):
        """Fit topic model with quality checks."""
        processed = self.preprocess(documents)

        # Create dictionary and corpus
        self.dictionary = Dictionary(processed)
        self.dictionary.filter_extremes(no_below=5, no_above=0.5)
        corpus = [self.dictionary.doc2bow(doc) for doc in processed]

        # Train model
        self.model = LdaModel(
            corpus=corpus,
            id2word=self.dictionary,
            num_topics=self.n_topics,
            passes=passes,
            alpha='auto',
            eta='auto',
            random_state=42
        )

        # Evaluate coherence
        coherence_model = CoherenceModel(
            model=self.model,
            texts=processed,
            dictionary=self.dictionary,
            coherence='c_v'
        )
        coherence = coherence_model.get_coherence()
        self.coherence_history.append(coherence)
```

```
        if coherence < self.coherence_threshold:
            print(f"Warning: Coherence {coherence:.3f} below threshold")

        return self

    def find_optimal_topics(self, documents, topic_range=range(5, 30, 5)):
        """Find optimal number of topics using coherence."""
        processed = self.preprocess(documents)
        self.dictionary = Dictionary(processed)
        self.dictionary.filter_extremes(no_below=5, no_above=0.5)
        corpus = [self.dictionary.doc2bow(doc) for doc in processed]

        coherence_scores = []

        for n_topics in topic_range:
            model = LdaModel(
                corpus=corpus, id2word=self.dictionary,
                num_topics=n_topics, passes=5, random_state=42
            )

            coherence_model = CoherenceModel(
                model=model, texts=processed,
                dictionary=self.dictionary, coherence='c_v'
            )
            score = coherence_model.get_coherence()
            coherence_scores.append((n_topics, score))
            print(f"Topics: {n_topics}, Coherence: {score:.4f}")

        # Select optimal
        optimal = max(coherence_scores, key=lambda x: x[1])
        self.n_topics = optimal[0]
        print(f"\nOptimal topics: {optimal[0]} with coherence {optimal[1]:.4f}
")

        return coherence_scores

    def get_topics(self, n_words=10):
        """Get human-readable topic descriptions."""
        topics = {}
        for idx in range(self.n_topics):
            topic_words = self.model.show_topic(idx, topn=n_words)
            topics[idx] = {
                'words': [w for w, p in topic_words],
                'weights': [p for w, p in topic_words]
            }
        return topics

    def transform(self, documents):
        """Get topic distributions for new documents."""
        processed = self.preprocess(documents)
        corpus = [self.dictionary.doc2bow(doc) for doc in processed]

        distributions = []
        for doc_bow in corpus:
            doc_topics = self.model.get_document_topics(doc_bow)
            dist = np.zeros(self.n_topics)
            for topic_id, prob in doc_topics:
                dist[topic_id] = prob
            distributions.append(dist)

        return np.array(distributions)
```

```
    def compute_perplexity(self, documents):
        """Compute perplexity on held-out documents."""
        processed = self.preprocess(documents)
        corpus = [self.dictionary.doc2bow(doc) for doc in processed]

        log_likelihood = self.model.log_perplexity(corpus)
        perplexity = np.exp(-log_likelihood)

        return perplexity
```

1.6.4 Dynamic Topic Modeling

```
class DynamicTopicModel:
    """Track topic evolution over time."""

    def __init__(self, n_topics=10):
        self.n_topics = n_topics
        self.time_slices = []
        self.models = []

    def fit(self, documents_by_time):
        """Fit dynamic topic model on time-sliced documents."""
        from gensim.models import LdaSeqModel

        all_docs = []
        self.time_slices = []

        for time_docs in documents_by_time:
            processed = self._preprocess(time_docs)
            all_docs.extend(processed)
            self.time_slices.append(len(processed))

        # Create corpus
        self.dictionary = Dictionary(all_docs)
        self.dictionary.filter_extremes(no_below=5, no_above=0.5)
        corpus = [self.dictionary.doc2bow(doc) for doc in all_docs]

        # Fit sequential LDA
        self.model = LdaSeqModel(
            corpus=corpus,
            id2word=self.dictionary,
            time_slice=self.time_slices,
            num_topics=self.n_topics,
            passes=10
        )

        return self

    def get_topic_evolution(self, topic_id, n_words=5):
        """Get how a topic's words change over time."""
        evolution = []

        for t in range(len(self.time_slices)):
            topic_words = self.model.print_topic(topic_id, t, n_words)
            evolution.append({
                'time': t,
                'words': topic_words
            })

        return evolution
```

```
    def _preprocess(self, documents):
        """Basic preprocessing."""
        import re
        processed = []
        for doc in documents:
            tokens = re.findall(r'\b[a-z]{3,}\b', doc.lower())
            processed.append(tokens)
        return processed
```

1.7 Production Deployment

1.7.1 Model Serving

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List, Dict
import joblib

app = FastAPI()

class TopicRequest(BaseModel):
    documents: List[str]

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

@app.post("/predict", response_model=TopicResponse)
async def predict_topics(request: TopicRequest):
    """Predict topics for documents."""
    try:
        distributions = model.transform(request.documents)
        top_words = model.get_topics()

        return TopicResponse(
            topics=[{f"topic_{i}": float(d[i]) for i in range(len(d))}
                    for d in distributions],
            top_words=[top_words[i]['words'] for i in range(model.n_topics)]
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/health")
async def health_check():
    """Check model health."""
    return {
        "status": "healthy",
        "n_topics": model.n_topics,
        "coherence": model.coherence_history[-1] if model.coherence_history
    else None
    }
```

1.7.2 Monitoring

```
class TopicModelMonitor:
    """Monitor topic model in production."""
```

```
    def __init__(self, model, reference_coherence):
        self.model = model
        self.reference_coherence = reference_coherence
        self.drift_threshold = 0.1
        self.alerts = []

    def check_topic_drift(self, new_documents):
        """Check for topic distribution drift."""
        new_dist = self.model.transform(new_documents)
        mean_entropy = np.mean([entropy(d) for d in new_dist])

        # High entropy indicates uncertain assignments
        if mean_entropy > 2.0:
            self.alerts.append({
                'type': 'high_entropy',
                'value': mean_entropy,
                'action': 'Consider retraining'
            })

        return mean_entropy

    def check_coherence_degradation(self, documents):
        """Check if model coherence has degraded."""
        current_coherence = self._compute_coherence(documents)
        degradation = self.reference_coherence - current_coherence

        if degradation > self.drift_threshold:
            self.alerts.append({
                'type': 'coherence_drop',
                'reference': self.reference_coherence,
                'current': current_coherence,
                'action': 'Trigger retraining pipeline'
            })

        return current_coherence

    def _compute_coherence(self, documents):
        """Compute coherence on new documents."""
        processed = self.model.preprocess(documents)
        coherence_model = CoherenceModel(
            model=self.model.model,
            texts=processed,
            dictionary=self.model.dictionary,
            coherence='c_v'
        )
        return coherence_model.get_coherence()
```

1.8 Common Parameters

Algorithm	Parameter	Typical Range	Notes
LDA	n_topics	10-100	Use coherence to optimize
LDA	alpha	0.01-1.0 or 'auto'	Lower = sparser document-topic
LDA	eta (beta)	0.01-1.0 or 'auto'	Lower = sparser topic-word
LDA	passes	5-20	More for small corpora
HDP	gamma	0.1-2.0	Global concentration
HDP	alpha	0.1-2.0	Document concentration

Algorithm	Parameter	Typical Range	Notes
BERTopic	min_cluster_size	10-50	Minimum documents per topic
BERTopic	n_neighbors	10-30	UMAP parameter

1.9 Practice Problems

1. **Inference Comparison:** Implement both variational inference and Gibbs sampling for the same corpus. Compare convergence speed, topic quality, and scalability.
2. **Coherence Analysis:** Train LDA with different numbers of topics (5-50). Plot coherence curves and identify the elbow point. Verify that perplexity and coherence may disagree.
3. **Dynamic Modeling:** Collect news articles from multiple time periods. Fit a dynamic topic model and visualize how topic prominence changes over time.
4. **Hyperparameter Sensitivity:** Study the effect of alpha and beta on topic sparsity. Generate synthetic data with known structure and recover the generating parameters.
5. **Production Pipeline:** Build an end-to-end system that ingests documents, trains incrementally, serves predictions via API, and monitors coherence drift.

1.10 References

1. Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). "Latent Dirichlet Allocation." JMLR, 3, 993-1022.
2. Hoffman, M. D., Blei, D. M., & Bach, F. (2010). "Online Learning for Latent Dirichlet Allocation." NeurIPS.
3. Teh, Y. W., Jordan, M. I., Beal, M. J., & Blei, D. M. (2006). "Hierarchical Dirichlet Processes." JASA, 101(476), 1566-1581.
4. Roder, M., Both, A., & Hinneburg, A. (2015). "Exploring the Space of Topic Coherence Measures." WSDM.
5. Grootendorst, M. (2022). "BERTopic: Neural Topic Modeling with a Class-Based TF-IDF Procedure." arXiv:2203.05794.
6. Mimno, D., et al. (2011). "Optimizing Semantic Coherence in Topic Models." EMNLP.

Topic modeling uncovers latent structure in document collections. Mathematical understanding of generative processes, inference algorithms, and evaluation metrics enables building interpretable models that reveal meaningful themes in text data.