Mathematical Models for AI News Bot

Dmitrii Bogolaev, Mark Lyumanov

May 2025

1 Mathematical Models

1.1 TF-IDF + KMeans Clustering

This model combines two powerful tools from natural language processing and unsupervised learning:

- **TF-IDF (Term Frequency–Inverse Document Frequency)** — transforms raw text into weighted vectors that emphasize rare but informative terms.
- **KMeans Clustering** — groups these vectors into clusters representing different topics.

The goal is to automatically select a small, diverse subset of articles that represent the main themes present in a larger set.

TF-IDF Explanation

TF-IDF evaluates how important a word is to a document in a collection. It is a product of two terms:

Term Frequency (TF):

$$\mathrm{TF}(t,d) = \frac{f_{t,d}}{\sum_k f_{k,d}}$$

Here, $f_{t,d}$ is the frequency of term t in document d, and the denominator sums frequencies of all terms in the same document.

Inverse Document Frequency (IDF):

$$\mathrm{IDF}(t) = \log\left(\frac{N}{1+n_t}\right)$$

Where N is the total number of documents and n_t is the number of documents in which term t appears. This gives higher weight to terms that are rare across the corpus.

TF-IDF Score:

$$\text{TF-IDF}(t,d) = \mathrm{TF}(t,d) \cdot \mathrm{IDF}(t)$$

The result is a high-dimensional vector for each document, with each component representing the TF-IDF weight of a term.

KMeans Clustering

KMeans is an iterative algorithm that aims to partition the dataset into k clusters by minimizing the variance within each cluster.

Given a set of vectors $x_1, x_2, ..., x_n$, the goal is to find cluster centroids $\mu_1, \mu_2, ..., \mu_k$ such that:

$$\arg\min_{\{\mu\}} \sum_{i=1}^{n} \|x_i - \mu_{C_i}\|^2$$

Where C_i denotes the cluster assignment of x_i.

Steps:

1. Initialize k random centroids.
2. Assign each point x_i to the nearest centroid.
3. Recompute each centroid as the mean of all points assigned to it.
4. Repeat steps 2 and 3 until convergence.

Implementation Code

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

def select_important(articles, n_clusters=3):
    if len(articles) == 0:
        return []

    texts = [article.get('title', '') for article in articles]

    if len(articles) < n_clusters:
        return articles

    vectorizer = TfidfVectorizer(stop_words='english')
    X = vectorizer.fit_transform(texts)

    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init='auto')
    labels = kmeans.fit_predict(X)

    selected = []
    seen_labels = set()

    for i, label in enumerate(labels):
        if label not in seen_labels:
            selected.append(articles[i])
            seen_labels.add(label)

    return selected
```

Why This Works

Articles that talk about similar topics tend to share key terms. TF-IDF helps identify these key terms, while KMeans groups together articles that emphasize the same vocabulary. By selecting one article per cluster, we create a concise and non-redundant summary of the broader information landscape.

This technique ensures topic diversity and avoids echoing similar headlines, which is crucial for a Telegram bot designed to inform rather than overwhelm.

1.2 Z-Score Statistical Selection

This model applies classical statistical hypothesis testing to detect news articles that significantly stand out in terms of their frequency or mention count.

Motivation

When a particular topic is mentioned far more frequently than others, it may indicate a trending or important story. To quantify how unusual a value is, we use the **z-score**.

Z-Score Formula

The z-score of an observation x is given by:

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x is the observed value (e.g., article mention count),
- μ is the mean of all values,
- σ is the standard deviation of values.

A z-score tells us how many standard deviations x is away from the mean.

Significance Threshold For a typical 95% confidence level, we consider an observation significant if:

$$z > 1.96$$

This means the value is in the top 2.5% of the normal distribution — statistically rare under the assumption of normality.

Practical Use Case

In our case, each article has a field `mention_count` (e.g., how many different sources mention it). We use z-scores to detect articles with unusually high mention counts and select them for summarization.

Implementation Code

```python
import numpy as np

def compute_z_score(x: float, mean: float, std: float) -> float:
    """
    Computes the z-score for a given value.
    """
    if std == 0:
        return 0.0
    return (x - mean) / std

def select_important(articles, threshold: float = 1.96):
    """
    Filters articles whose mention counts are significantly above the mean.
    """
    mention_counts = [a.get("mention_count", 1) for a in articles]

    if len(mention_counts) < 2:
        return articles

    mean = np.mean(mention_counts)
    std = np.std(mention_counts)

    filtered = []
    for article in articles:
        count = article.get("mention_count", 1)
        z = compute_z_score(count, mean, std)
        if z > threshold:
            filtered.append(article)

    return filtered
```

Why This Works

This method is simple, interpretable, and grounded in statistical theory. It allows the bot to focus on content that exhibits strong anomalies in attention — likely reflecting emerging or viral topics — without needing any labeled data or training.

It is especially useful when your system monitors hundreds of articles and needs a principled way to filter out noise.

1.3 Bayesian Importance Estimation

This model estimates the probability that an article is important given its observed characteristics, using Bayes' Theorem from probability theory.

Bayes' Theorem

Given a hypothesis H (the article is important) and observed data D (mention count), Bayes' rule defines:

$$P(H \mid D) = \frac{P(D \mid H) \cdot P(H)}{P(D)}$$

Where:

- $P(H)$ is the prior probability that a randomly selected article is important,
- $P(D \mid H)$ is the likelihood of observing the mention count if the article is important,
- $P(D)$ is the marginal probability of observing such data across all articles,
- $P(H \mid D)$ is the posterior — our updated belief in the article's importance.

Implementation Assumptions

For simplicity:

- The prior is fixed at $P(H) = 0.5$,
- The likelihood is approximated as a normalized count: $P(D \mid H) \approx \frac{\text{mention\_count}}{10}$,
- The evidence is fixed as $P(D) = 1$, acting as a scaling constant.

Python Code

```
def bayesian_importance(prior: float, likelihood: float, evidence: float) -> float:
    """
    Applies Bayes' theorem:
    P(H|D) = (P(D|H) * P(H)) / P(D)
    """
    if evidence == 0:
        return 0.0
    return (likelihood * prior) / evidence

def select_important(articles, threshold: float = 0.6):
    """
    Selects articles with posterior importance probability > threshold.
    """
    prior = 0.5        # Prior belief: 50% chance an article is important
    evidence = 1.0     # Simplified: fixed marginal likelihood

    selected = []
    for article in articles:
        mention_count = article.get("mention_count", 1)
        likelihood = min(mention_count / 10, 1.0)  # Normalize to [0, 1]
        posterior = bayesian_importance(prior, likelihood, evidence)
        if posterior > threshold:
            selected.append(article)

    return selected
```

Why This Works

This method formalizes a probabilistic belief update: the more often an article is mentioned, the higher the likelihood it is important. Bayes' Theorem combines this with a prior belief to yield a robust, interpretable metric.

It is particularly useful when explicit thresholds or supervised labels are not available but some notion of "probable importance" is still desired.

1.4 Logistic Regression Classification

This model uses supervised learning to predict whether a news article is important based on its numerical features. The method is based on **logistic regression**, a statistical model used for binary classification.

Mathematical Foundation

The logistic regression hypothesis function is defined as:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Where:

- x is the feature vector (e.g., title length, mention count),
- θ is the parameter vector (learned from data),
- $h_\theta(x) \in (0, 1)$ is the predicted probability that the article is important.

A threshold (e.g. 0.7) is used to convert the probability into a binary decision.

Training Data

To train the model, we need a set of labeled data:

- Each sample represents an article with features such as:
  - Title length
  - Mention count
  - Score from platform (optional)
  - Keyword presence
- Each sample has a binary label: 1 = important, 0 = not important

Example training matrix:

$$X = \begin{bmatrix} 30 & 2 \\ 70 & 6 \\ 20 & 0 \\ 90 & 8 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

Python Code for Training

```
from sklearn.linear_model import LogisticRegression
import numpy as np

# Training data
X_train = np.array([
    [30, 2],
    [70, 6],
    [20, 0],
    [90, 8]
])
y_train = np.array([0, 1, 0, 1])

# Train model
model = LogisticRegression()
model.fit(X_train, y_train)
```

Feature Extraction Example

```
def extract_features(article):
    title_length = len(article.get("title", ""))
    mentions = article.get("mention_count", 1)
    return [title_length, mentions]
```

Prediction and Filtering

Once trained, the model can be used to predict the probability that new articles are important:

```
features = extract_features(article)
prob = model.predict_proba([features])[0][1]
if prob >= 0.7:
    mark_as_important(article)
```

Why This Works

Logistic regression finds optimal weights for each feature, allowing us to build a probabilistic decision boundary. It's simple, fast, and interpretable — and can be retrained as new labeled data becomes available.

1.5 FDR-Based Selection (Benjamini–Hochberg Procedure)

This model uses the concept of **False Discovery Rate (FDR)** to control the number of false positives when selecting important articles. It is particularly useful when analyzing many items simultaneously — such as hundreds of news items — to avoid mistakenly selecting articles that appear important just by chance.

False Discovery Rate (FDR)

FDR is defined as the expected proportion of false discoveries (type I errors) among the set of accepted hypotheses:

$$\mathrm{FDR} = \mathbb{E}\left[\frac{V}{R}\right]$$

Where:

- V is the number of false positives,
- R is the total number of rejected null hypotheses (selected articles).

Instead of controlling the probability of even a single false positive (as in Bonferroni), FDR allows some false positives but keeps their rate under control.

Benjamini–Hochberg (BH) Procedure

The BH procedure operates as follows:

1. Sort the p-values $p_1 \leq p_2 \leq \cdots \leq p_n$.
2. For each p_i, compute the threshold $\frac{i}{n} \cdot \alpha$, where α is the desired FDR level.
3. Find the largest i such that $p_i \leq \frac{i}{n} \cdot \alpha$.
4. Reject all null hypotheses for $p_1, \ldots, p_i$.

Interpretation: This guarantees that the expected proportion of false positives among the selected articles does not exceed α.

Pseudo P-values

In our unsupervised setting, we do not have real hypothesis tests. Instead, we simulate p-values inversely from article importance, using:

$$p_i = \frac{1}{\text{mention\_count}_i + \varepsilon}$$

Where ε is a small constant to prevent division by zero.

Python Code

```
import numpy as np
from statsmodels.stats.multitest import multipletests

def compute_pvalues(articles):
     """
    Computes pseudo p-values for each article based on mention count.
    Lower p-value = more important.

    Parameters:
        articles (list): List of articles with 'mention_count'.

    Returns:
        list of float: Simulated p-values (1 / (mention_count + epsilon))
    """
    return [1.0 / (a.get("mention_count", 1) + 1e-8) for a in articles]


def select_important(articles, alpha=0.05):
    """
    Applies the Benjamini-Hochberg procedure to control the False Discovery Rate (FDR)
    when selecting significant articles.

    Parameters:
        articles (list): List of article dicts with 'mention_count'.
        alpha (float): Desired FDR level (default = 0.05).

    Returns:
        list: Filtered list of articles that pass FDR control.
    """
    if len(articles) == 0:
        return []

    # Compute pseudo p-values for each article
    pvals = compute_pvalues(articles)

    # Apply BenjaminiHochberg FDR correction
    reject, _, _, _ = multipletests(pvals, alpha=alpha, method='fdr_bh')

    # Return only those articles that passed the FDR test
    return [a for a, keep in zip(articles, reject) if keep]
```

Why This Works

The Benjamini–Hochberg method balances discovery and reliability. It allows us to select a wider range of potentially interesting articles while still controlling the false discovery rate — which is particularly valuable when monitoring large-scale, noisy news streams.

This is a statistically rigorous way to avoid overfitting to random spikes in mention counts.