Mathematical Models for AI News Bot

Dmitrii Bogolaev, Mark Lyumanov

May 2025

1 Mathematical Models

1.1 Bayesian Importance Estimation

The Bayesian method uses Bayes' Theorem to calculate the probability of an article being important based on certain features such as the number of keywords it contains or its length. In this method, we calculate a **posterior** probability that an article is important, given observed data.

Bayes' Theorem

Bayes' Theorem provides a way to update our beliefs about the importance of an article based on new evidence:

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

Where:

- $P(H)$ is the prior probability that an article is important, which we set to 0.5 (indicating an equal chance for an article to be important or not).
- $P(E \mid H)$ is the likelihood of the observed data (e.g., the number of keywords or word count) given that the article is important.
- $P(E)$ is the evidence, or the probability of observing the data across all articles.
- $P(H \mid E)$ is the posterior, or the updated probability that the article is important after accounting for the evidence.

The threshold for considering an article important is set at 0.574, which means that if the posterior probability of an article being important is greater than this threshold, it is considered for selection.

Implementation Code

```
def bayesian_importance(prior: float, likelihood: float, evidence: float) -> float:
    """
    Computes Bayesian importance using the formula:
    P(H|E) = (P(E|H) * P(H)) / P(E)
    """
    if evidence == 0:
        return 0.0
    return (likelihood * prior) / evidence

def select_important(articles, threshold: float = 0.574):
    """
    Selects important articles based on Bayesian inference.
    """
    prior = 0.5  # Base probability that an article is important
    evidence = 1.0  # Simplified assumption
```

```
    selected = []
    keywords = {"AI", "OpenAI", "GPT", "Machine Learning", "Deep Learning"}

    for article in articles:
        word_count = len(article['text'].split())

        # Ignore very short articles
        if word_count < 20:
            continue

        # Calculate base likelihood
        likelihood = min(word_count / 150, 1.0)  # Smoother normalization

        # Boost for keywords, but with diminishing returns
        keyword_count = sum(article['text'].count(kw) for kw in keywords)

        # Adjust likelihood based on keyword occurrence (capped at 0.2)
        likelihood += 0.03 * min(keyword_count, 5)

        # Compute the posterior
        posterior = bayesian_importance(prior, likelihood, evidence)

        if posterior > threshold:
            selected.append(article)

    print(f"Model 'bayesian' returned {len(selected)} articles.")
    return selected
```

1.2 FDR (False Discovery Rate) Selection

The False Discovery Rate (FDR) is a method used in multiple hypothesis testing to control the proportion of false positives (incorrectly identifying an article as important). FDR applies a statistical correction to ensure that not too many articles are incorrectly flagged as important.

FDR Method

FDR is controlled by calculating p-values and adjusting them using the Benjamini-Hochberg procedure. P-values are calculated by comparing the observed word count of an article to the distribution of word counts in the corpus.

The threshold for selecting articles is determined by the FDR-corrected p-values, and articles with p-values below a certain threshold (e.g., $\alpha = 0.1$) are considered important.

Implementation Code

```
import numpy as np
from scipy.special import erf
from statsmodels.stats.multitest import multipletests

def compute_pvalues(articles):
    """
    Compute p-values based on the word count of the articles.
    """
    pvalues = []
    word_counts = [len(article['text'].split()) for article in articles]

    # Compute mean and std deviation
    mean_count = np.mean(word_counts)
    std_count = np.std(word_counts) + 1e-8  # To avoid division by zero

    for word_count in word_counts:
        # Z-score calculation
        z_score = (word_count - mean_count) / std_count
```

```
        # Convert to p-value using normal distribution (1 - CDF)
        pvalue = 1 - (0.5 * (1 + erf(z_score / np.sqrt(2))))
        pvalues.append(pvalue)

    return pvalues

def select_important(articles, alpha=0.1):
    """
    Selects important articles using FDR (False Discovery Rate) method.
    """
    if len(articles) == 0:
        return []

    pvals = compute_pvalues(articles)
    print(f"Computed p-values: {pvals}")

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

    # Return only the articles that pass the filter
    return [a for a, keep in zip(articles, reject) if keep]
```

1.3 Logistic Regression Classification

In Logistic Regression, we build a model to predict whether an article is important or not based on several features such as the title length, number of mentions, and word count. Logistic Regression outputs a probability value between 0 and 1, and we classify articles as important if this probability is greater than a set threshold (e.g., 0.8).

Logistic Regression Model

The logistic regression model uses the sigmoid function to map linear combinations of features to probabilities. The logistic regression hypothesis is defined as:

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

Where $h_\theta(x)$ is the predicted probability of an article being important given its feature vector x, and θ are the learned model parameters.

Implementation Code

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

# Training data
X_train = np.array([
    [30, 1, 150],    # [title_length, mention_count, word_count]
    [50, 3, 300],
    [20, 0, 80],
    [100, 7, 800],
    [80, 6, 600],
    [15, 0, 70],
    [60, 5, 400]
])

# Labels: 1 = important, 0 = not important
y_train = np.array([0, 1, 0, 1, 1, 0, 1])

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

```
def extract_features(article):
    title_length = len(article.get("text", ""))
    mentions = article.get("mention_count", 1)  # Now it is real
    word_count = len(article.get("text", "").split())
    return [title_length, mentions, word_count]

def select_important(articles, threshold=0.8):
    """
    Select important articles using Logistic Regression with a high probability threshold
        and softened penalties.
    """
    selected = []
    for article in articles:
        features = np.array(extract_features(article)).reshape(1, -1)
        prob = model.predict_proba(features)[0][1]

        # Print probability for debugging
        print(f"Article URL: {article['url']}, Initial probability: {prob:.4f}")

        # Soft penalty for missing keywords
        keywords = ["AI", "GPT", "OpenAI", "Machine Learning", "Deep Learning"]
        if not any(kw in article.get("text", "") for kw in keywords):
            prob *= 0.75  # Soft penalty for missing keywords

        # Soft penalty for articles with low mention count
        if article.get("mention_count", 1) < 3:  # Threshold for mentions
            prob *= 0.8  # Soft penalty for low mentions

        print(f"Adjusted probability: {prob:.4f}")

        # Only select articles if probability is above the threshold
        if prob >= threshold:
            selected.append(article)
    return selected
```

1.4 TF-IDF + KMeans Clustering

TF-IDF is used to transform articles into a high-dimensional vector space where each word is weighted by its importance. KMeans clustering is then used to group similar articles. After clustering, we select one article per cluster to represent the most important articles in that cluster.

TF-IDF Explanation

The TF-IDF (Term Frequency-Inverse Document Frequency) measure is used to represent documents as vectors where each dimension corresponds to a term. The weight of each term is computed based on how frequently it appears in the document and how rare it is across the entire corpus.

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 []

    # Replaced 'title' with 'text'
    texts = [article.get('text', '') for article in articles]

    # Filter empty texts
    texts = [t for t in texts if len(t.split()) > 3]

    if len(texts) < n_clusters:
```

```
        return articles  # No need to clusterize

    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
```

1.5 Z-Score Based Selection

In this method, articles are selected based on how unusual their word count is compared to the mean word count of all articles. A Z-score is computed for each article, and articles with a Z-score greater than a threshold (e.g., 1.96) are considered important.

Z-Score Formula

The Z-score is calculated as follows:

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

Where:

- x is the observed value (e.g., word count),
- μ is the mean word count,
- σ is the standard deviation of word counts.

Articles with a Z-score greater than 1.96 are considered to be significantly different from the mean and are selected.

Implementation Code

```
import numpy as np

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

def select_important(articles, threshold: float = 1.96):
    """
    Select important articles based on Z-Score of text length.
    Only articles with Z-Score > threshold are returned.
    """
    # Compute the word count for each article instead of mention_count
    word_counts = [len(a.get("text", "").split()) for a in articles]

    if len(word_counts) < 2:
        return articles
```

```
    mean = np.mean(word_counts)
    std = np.std(word_counts)

    filtered = []
    for article in articles:
        count = len(article.get("text", "").split())
        z = compute_z_score(count, mean, std)
        if z > threshold:
            filtered.append(article)
    return filtered
```