# I210895 Muhammad shahnawaz

Gen AI # Q2

Semantic Product Search and Ranking System

Technical Documentation

Table of Contents

System Overview

The Semantic Product Search and Ranking system is designed to understand natural language queries and retrieve contextually relevant products based on their titles and descriptions. Unlike traditional keyword-based search systems, this solution leverages deep learning techniques to understand the semantic meaning behind user queries and match them with the most appropriate products.

Key Features

- Natural language query processing
- Semantic understanding of product information
- Contextual relevance ranking
- Real-time web interface for search
- Deep learning-based retrieval system

Target Use Case

This system is ideal for e-commerce platforms seeking to enhance their search capabilities beyond simple keyword matching. It addresses the gap between how users naturally express their product needs and how products are described in catalogs.

Requirements

Functional Requirements

1. Accept natural language queries from users
2. Retrieve candidate products based on semantic relevance
3. Rank products according to contextual relevance
4. Display results in real-time through a web interface
5. Process both product titles and descriptions for improved matching

Technical Requirements

1. **Data Processing**
   - Load product dataset with title and description fields
   - Apply text preprocessing (lowercase, stop word removal, lemmatization, special character removal)
   - Transform text into numerical representations
2. **Model Development**
   - Implement deep learning models for semantic matching
   - Train on query-product pairs dataset
   - Fine-tune hyperparameters for optimal performance
3. **Evaluation**
   - Measure effectiveness using standard ranking metrics
   - Visualize model performance
4. **Deployment**
   - Implement web interface for user interaction
   - Ensure responsive search results

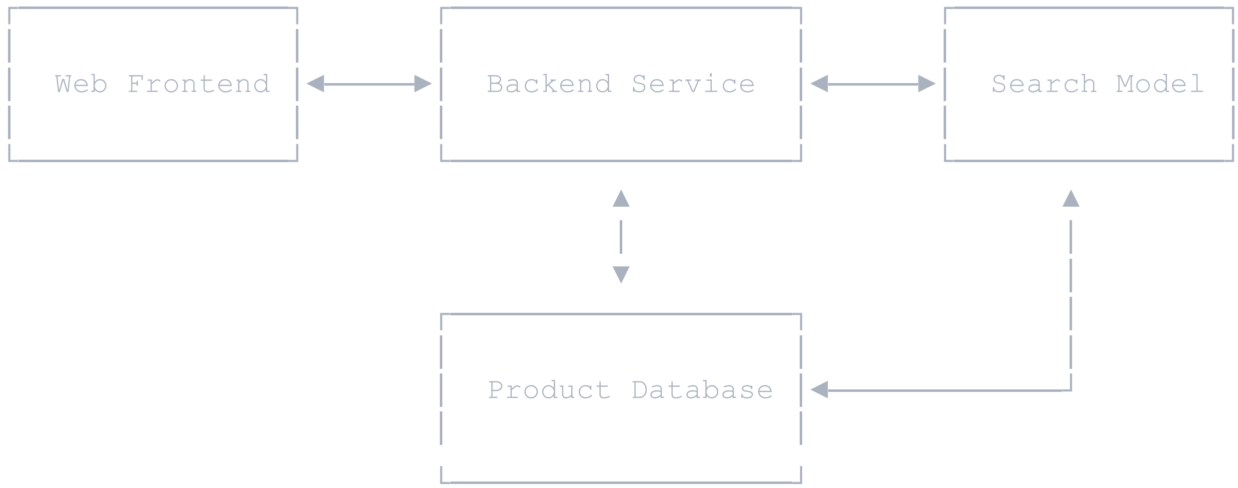
Dataset

The system uses the Amazon Shopping Queries Dataset, which contains:

- Query-product pairs labeled for semantic relevance
- Product information including title and description
- Source: [Amazon Science ESCI Data]

System Architecture

High-Level Architecture



Components

1. **Web Frontend**
    - User interface with search box
    - Results display with product information
    - Responsive design for various devices
2. **Backend Service**
    - API handling for search requests
    - Query preprocessing
    - Interface between frontend and search model
3. **Search Model**
    - Deep learning model for semantic matching
    - Text representation engine
    - Ranking algorithm
4. **Product Database**
    - Storage for product information
    - Indexed for efficient retrieval
    - Preprocessed text representations

Implementation Details

Text Preprocessing

1. **Tokenization**: Breaking text into individual words or tokens
2. **Normalization**:
    - Convert to lowercase
    - Remove special characters
    - Remove stop words
    - Apply lemmatization or stemming

3. **Numerical Representation**:
   - TF-IDF vectorization
   - Word embeddings (Word2Vec, GloVe, FastText)
   - Contextual embeddings (BERT, GPT)

Model Architecture

The system implements a deep learning model for semantic matching, with recommended architectures including:

Option 1: BERT-based Retrieval

```
Query Text → BERT Encoder → Query Embedding
                                  ↓
                              Similarity
                                  ↑
Product Text → BERT Encoder → Product Embedding
```

Option 2: Siamese Network

```
Query Text → Embedding Layer → Dense Layers
                                    ↓
                                Similarity
                                    ↑
Product Text → Embedding Layer → Dense Layers
```

Option 3: Cross-Encoder

```
[Query, Product] → BERT → Classification Layer → Relevance Score
```

Training Strategy

1. **Data Splitting**:
   - Training set: 70%
   - Validation set: 15%
   - Test set: 15%
2. **Training Procedure**:
   - Batch size optimization
   - Learning rate scheduling
   - Early stopping based on validation performance
   - Loss function: typically cross-entropy for relevance classification or triplet loss for embeddings
3. **Hyperparameter Tuning**:
   - Grid or random search for optimal parameters
   - Key parameters: learning rate, batch size, embedding dimension, model depth

Inference Pipeline

1. Receive user query
2. Preprocess query text
3. Generate query embedding
4. Compute similarity scores with product embeddings
5. Sort products by relevance score
6. Return top-K products to the user

Evaluation Methodology

Ranking Metrics

1. **NDCG (Normalized Discounted Cumulative Gain)**
   - Measures ranking quality considering position of relevant items
   - Formula: NDCG@k = DCG@k / IDCG@k
2. **MAP (Mean Average Precision)**
   - Average of precision values at relevant item positions
   - Considers both precision and recall
3. **Precision@K**
   - Proportion of relevant items in top-K results
   - Formula: P@k = (# relevant items in top k) / k
4. **Recall@K**
   - Proportion of all relevant items found in top-K results
   - Formula: R@k = (# relevant items in top k) / (total # relevant items)
5. **F1@K**
   - Harmonic mean of Precision@K and Recall@K
   - Formula: F1@k = 2 * (P@k * R@k) / (P@k + R@k)

Visualization

1. **Training and Validation Loss Curves**
   - Plot loss values against epochs
   - Identify overfitting or underfitting
2. **Confusion Matrix**
   - For relevance classification tasks
   - Visualize true positives, false positives, etc.
3. **Embedding Space Visualization**
   - t-SNE or PCA plots of query and product embeddings
   - Visualize clustering of related concepts

Deployment Guide

Web Application Setup

1. **Frontend Development**
    - HTML/CSS/JavaScript for user interface
    - Recommended frameworks: React, Vue.js, or Angular
    - Responsive design for various devices
2. **Backend Development**
    - API endpoints for search functionality
    - Query processing and model inference
    - Recommended frameworks: Flask, FastAPI, or Django

Model Deployment

1. **Model Serving**
    - Convert trained model to production format
    - Set up inference API
    - Optimize for latency
2. **Scaling Considerations**
    - Caching frequent queries
    - Load balancing for high traffic
    - Batch processing for efficiency

Environment Setup

1. **Development Environment**
    - Python 3.8+ with required libraries
    - Deep learning framework (PyTorch, TensorFlow)
    - Text processing libraries (NLTK, spaCy)
2. **Production Environment**
    - Docker containers for isolated deployment
    - Cloud hosting options (AWS, GCP, Azure)
    - Resource allocation based on expected traffic

Monitoring and Maintenance

1. **Performance Monitoring**
    - Track response times
    - Monitor resource utilization
    - Log search queries and results
2. **Regular Updates**
    - Retrain model with new data
    - Update product database
    - Implement user feedback

Troubleshooting

Common Issues and Solutions

1. **Slow Response Time**
   - Optimize model inference
   - Implement caching for frequent queries
   - Consider model quantization or distillation
2. **Poor Relevance Quality**
   - Retrain with more diverse data
   - Adjust hyperparameters
   - Implement user feedback loop
3. **Memory Issues**
   - Reduce embedding dimensions
   - Implement efficient indexing
   - Consider approximate nearest neighbor methods
4. **Integration Problems**
   - Ensure consistent API contracts
   - Implement proper error handling
   - Maintain comprehensive logs

Appendix

Recommended Tools

1. **Text Processing**
   - NLTK for tokenization and stemming
   - spaCy for advanced NLP tasks
   - TensorFlow Text or torchtext for deep learning integration
2. **Embedding Models**
   - Sentence-Transformers for BERT-based encodings
   - Gensim for Word2Vec, FastText, and GloVe
   - Hugging Face Transformers for various pre-trained models
3. **Deep Learning Frameworks**
   - PyTorch
   - TensorFlow
   - Keras
4. **Web Development**
   - Frontend: React, Vue.js
   - Backend: Flask, FastAPI
   - Deployment: Docker, Kubernetes
5. **Vector Storage**
   - FAISS for efficient similarity search

- Elasticsearch with vector search capabilities
- Pinecone for cloud-based vector database

Code Examples

Query Processing

python

```
def preprocess_text(text):
    # Convert to lowercase
    text = text.lower()

    # Remove special characters
    text = re.sub(r'[^a-z0-9\s]', '', text)

    # Tokenize
    tokens = word_tokenize(text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words]

    # Lemmatization
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]


    return ' '.join(tokens)
```

Model Training

python

```
def train_model(train_data, val_data, epochs=10):
    model =
BertForSequenceClassification.from_pretrained('bert-base-uncased')
    optimizer = AdamW(model.parameters(), lr=2e-5)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        # Training
        model.train()
        train_loss = 0
        for batch in train_data:
            optimizer.zero_grad()
            inputs = {
```

```
                'input_ids': batch['input_ids'],
                'attention_mask': batch['attention_mask'],
                'labels': batch['labels']
            }
            outputs = model(**inputs)
            loss = outputs.loss
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        train_losses.append(train_loss / len(train_data))

        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for batch in val_data:
                inputs = {
                    'input_ids': batch['input_ids'],
                    'attention_mask': batch['attention_mask'],
                    'labels': batch['labels']
                }
                outputs = model(**inputs)
                loss = outputs.loss
                val_loss += loss.item()

        val_losses.append(val_loss / len(val_data))

        print(f"Epoch {epoch+1}/{epochs} - Train Loss: {train_losses[-1]:.4f} - Val Loss: {val_losses[-1]:.4f}")

    return model, train_losses, val_losses
```

Search API

python

```
@app.route('/search', methods=['POST'])
def search():
    query = request.json.get('query', '')

    # Preprocess query
    processed_query = preprocess_text(query)

    # Generate embedding
    query_embedding = model.encode([processed_query])[0]
```

```python
    # Find similar products
    scores, indices = index.search(np.array([query_embedding]), k=10)

    # Get product details
    results = []
    for i, idx in enumerate(indices[0]):
        if idx != -1:  # FAISS returns -1 for invalid indices
            results.append({
                'id': products[idx]['id'],
                'title': products[idx]['title'],
                'description': products[idx]['description'],
                'score': float(scores[0][i])
            })

    return jsonify({'results': results})
```

References

1. Amazon Shopping Queries Dataset: https://github.com/amazon-science/esci-data/tree/main/shopping_queries_dataset
2. Dataset Description: https://github.com/amazon-science/esci-data
3. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
4. Learning to Rank for Information Retrieval
5. Semantic Product Search