SuperBPE for Entity Recognition and Product-Feature Disambiguation in Samsung Search

Entity Recognition Using SuperBPE

SuperBPE can be extended beyond basic tokenization to perform entity recognition within search queries—particularly valuable for distinguishing between products, features, accessories, and other entities in Samsung's ecosystem.

How SuperBPE Enables Entity Recognition

Traditional entity recognition systems typically rely on pre-tokenized input, making them vulnerable to spacing errors. SuperBPE changes this paradigm by integrating space prediction into the tokenization process, which creates several advantages:

1. **Joint optimization** of tokenization and entity boundaries
2. **Context-sensitive entity detection** that considers the entire query
3. **Error-resilient recognition** that works even with misspelled entities

Entity Type Classification with SuperBPE

The SuperBPE model can be enhanced to classify tokens into entity types by adding an entity classification layer that operates on the token embeddings. This creates a multi-task learning system that simultaneously:

1. Predicts subword units
2. Predicts space locations
3. Classifies entity types

python Copy

```python
class SuperBPEWithEntityRecognition:
    def __init__(self, vocab_size=50000, entity_types=["PRODUCT", "FEATURE", "ACCESSOR
        self.tokenizer = SuperBPETokenizer(vocab_size)
        self.entity_classifier = EntityClassifier(len(entity_types), entity_types)

    def process_query(self, query):
        # Generate multiple segmentation hypotheses with their probabilities
        segmentation_hypotheses = self.tokenizer.generate_hypotheses(query)

        results = []
        for tokens, probability in segmentation_hypotheses:
            # For each segmentation, classify entities
            entity_spans = self.entity_classifier.classify(tokens)

            # Combine segmentation and entity information
            enriched_result = {
                'tokens': tokens,
                'probability': probability,
                'entities': entity_spans
            }
            results.append(enriched_result)

        # Return the most likely segmentation with entity information
        return sorted(results, key=lambda x: x['probability'], reverse=True)[0]
```

Application to Samsung Product Search

Let's examine concrete examples of how this entity recognition approach works:

Example 1: Mixed Entity Query with Errors

Input: `"samsunggalexybuds2procaseblack"`

SuperBPE processing:

1. Tokenization: `samsung` `galexy` `buds` `2` `pro` `case` `black`
2. Error correction: `samsung` `galaxy` `buds` `2` `pro` `case` `black`
3. Entity classification:
   - `samsung galaxy buds 2 pro` → PRODUCT
   - `case` → ACCESSORY
   - `black` → COLOR

Final structured output:

json Copy

```
{
  "product": "Samsung Galaxy Buds 2 Pro",
  "accessory": "case",
  "color": "black"
}
```

Example 2: Ambiguous Entity Boundaries

Input: `"galaxys23ultracameraupgrade"`

Potential segmentations:

1. `galaxy` `s23` `ultra` `camera` `upgrade`
2. `galaxy` `s23` `ultracamera` `upgrade`

The SuperBPE entity recognizer would favor the first segmentation because:

- "s23 ultra" is a known product model
- "camera upgrade" is a common feature phrase

Entity classification:

- `galaxy s23 ultra` → PRODUCT
- `camera upgrade` → FEATURE

This produces a structured understanding of the query as asking about camera upgrades for the Galaxy S23 Ultra, rather than interpreting "ultracamera" as a single entity.

Detailed Analysis of SuperBPE Advantages

1. Product Name Disambiguation

Product name disambiguation is critical in Samsung's ecosystem where naming conventions follow patterns with subtle variations (S22, S22+, S22 Ultra, S22 FE).

SuperBPE approaches this challenge using both statistical and contextual information:

Statistical Knowledge: The model learns the frequency and likelihood of different product name combinations in the training corpus. For example, it learns that "S22" is more likely to be followed by "Ultra" than by "Note".

Contextual Understanding: The model considers surrounding tokens when disambiguating. If "5G" appears after a model name, it's likely a feature, not part of the product name.

python Copy

```python
def disambiguate_product_names(tokens, product_catalog):
    candidates = []

    # Generate candidate product spans
    for i in range(len(tokens)):
        for j in range(i+1, min(i+5, len(tokens)+1)):  # Look at spans up to 4 tokens
            span = " ".join(tokens[i:j])

            # Check if this span matches known products
            product_matches = fuzzy_match_products(span, product_catalog)
            if product_matches:
                candidates.append({
                    "span": (i, j),
                    "tokens": tokens[i:j],
                    "product_matches": product_matches,
                    "confidence": calculate_confidence(span, product_matches)
                })

    # Resolve overlapping spans by taking highest confidence non-overlapping set
    return resolve_overlapping_spans(candidates)
```

Example: Handling Ambiguous Product References

Input: `"galaxys22anotes22ultracomparison"`

SuperBPE processing:

1. Initial segmentation hypotheses:
   - `galaxy` `s22` `a` `note` `s22` `ultra` `comparison`
   - `galaxy` `s22` `and` `s22` `ultra` `comparison`
   - `galaxy` `s22a` `notes` `22` `ultra` `comparison`
2. Product disambiguation:
   - Recognizes "Galaxy S22" as a product
   - Identifies ambiguity with "a note" vs "and" vs "a notes"
   - Applies contextual understanding that "Galaxy S22" and "S22 Ultra" are likely being compared
   - Favors the second hypothesis with "and" as connector
3. Final interpretation: "Comparison between Galaxy S22 and S22 Ultra"

This example showcases how SuperBPE uses its understanding of product relationships to disambiguate between similar-sounding segments.

2. Compound Error Correction

The ability to handle multiple types of errors simultaneously is perhaps SuperBPE's most powerful feature. Traditional correction systems typically handle either spacing errors or character errors—but not both simultaneously.

SuperBPE's approach combines:

1. **Character-level error models** that capture substitution, insertion, and deletion patterns
2. **Space probability models** that learn where spaces are likely to occur
3. **Joint optimization** that finds the globally optimal correction

python Copy

```python
def correct_compound_errors(query, model):
    # Generate character-level variations for each potential token
    char_variations = generate_character_variations(query)

    # For each variation, generate space placement hypotheses
    segmentation_hypotheses = []
    for variation in char_variations:
        segmentation_hypotheses.extend(model.generate_space_hypotheses(variation))

    # Score each hypothesis using the joint model
    scored_hypotheses = []
    for hypothesis in segmentation_hypotheses:
        score = model.score_joint_hypothesis(hypothesis)
        scored_hypotheses.append((hypothesis, score))

    # Return the highest scoring correction
    return sorted(scored_hypotheses, key=lambda x: x[1], reverse=True)[0][0]
```

Example: Multi-Error Correction

Input: `"samsugqlaxys22ultrgreen"`

SuperBPE correction process:

1. Character-level error detection:
   - `samsug` → potential corrections: `samsung` (missing 'n')
   - `qlaxy` → potential corrections: `galaxy` (substitution 'q' for 'g', 'x' for 'ax')
   - `ultr` → potential corrections: `ultra` (missing 'a')
   - `green` → likely correct
2. Space prediction:

- High probability spaces after: `samsung`, `galaxy`, `s22`, `ultra`
- Low probability space between `ultr` and `green` (if uncorrected)

3. Joint optimization:
   - Considers all combinations of character corrections and space placements
   - Scores using learned model of Samsung product naming patterns
   - Selects optimal solution: `samsung galaxy s22 ultra green`

This example demonstrates how SuperBPE handles the interdependence of character-level and space-level errors—the correction of "ultr" to "ultra" affects where the space should be placed.

3. Feature vs. Product Distinction

One of the most challenging aspects of product search is distinguishing between product names and feature descriptions, especially when both use similar technical terminology.

SuperBPE approaches this by learning the contextual patterns that differentiate products from features:

python Copy

```python
class EntityTypeClassifier:
    def __init__(self, model_path):
        self.model = load_model(model_path)

    def classify_spans(self, tokens, spans):
        results = []
        for start, end in spans:
            span_tokens = tokens[start:end]

            # Get embedding for this span
            span_embedding = self.get_span_embedding(span_tokens)

            # Classify span type
            entity_type = self.model.predict_entity_type(span_embedding)
            confidence = self.model.predict_confidence(span_embedding)

            results.append({
                "start": start,
                "end": end,
                "text": " ".join(span_tokens),
                "entity_type": entity_type,
                "confidence": confidence
            })

        return results
```

Example: Product vs. Feature Disambiguation

Input: `"galaxybookproactivenoisecancelling"`

SuperBPE processing:

1. Initial segmentation: `galaxy` `book` `pro` `active` `noise` `cancelling`
2. Entity span identification:
   - `galaxy book pro` → matches known product pattern
   - `active noise cancelling` → matches known feature pattern
3. Classification:
   - `galaxy book pro` → classified as PRODUCT (high confidence)
   - `active noise cancelling` → classified as FEATURE (high confidence)
4. Structured interpretation:

json Copy

```
{
  "intent": "PRODUCT_WITH_FEATURE",
  "product": "Galaxy Book Pro",
  "feature": "Active Noise Cancelling"
}
```

What makes this challenging is that "Pro" could be either part of the product name or a feature descriptor. SuperBPE resolves this by:

1. Learning that "Galaxy Book Pro" is a coherent product entity
2. Recognizing that "Active Noise Cancelling" is a standard feature description
3. Using contextual patterns to determine where the product name ends and feature description begins

Another Complex Example:

Input: `"galaxywatchwithblodoxygenlteversion"`

SuperBPE processing:

1. Initial segmentation: `galaxy` `watch` `with` `blod` `oxygen` `lte` `version`
2. Error detection and correction: `blod` → `blood` (likely typo)
3. Entity recognition:
   - `galaxy watch` → PRODUCT
   - `blood oxygen` → FEATURE
   - `lte` → CONNECTIVITY
   - `version` → PRODUCT_QUALIFIER
4. Structured interpretation:

json Copy

```
{
  "product": "Galaxy Watch",
  "features": ["Blood Oxygen"],
  "connectivity": "LTE",
  "qualifier": "version"
}
```

This demonstrates SuperBPE's ability to not just correct and segment text, but to provide a structured understanding of different entity types in the query.

Practical Implementation

Implementing SuperBPE for Samsung's product search would involve these components:

Training Data Preparation

The system requires training data that captures:

1. Complete Samsung product catalog with proper categorization
2. Common misspellings and errors
3. Feature descriptions and technical specifications
4. Annotated queries with entity labels

Model Architecture

python Copy

```python
class SuperBPEForSamsungSearch:
    def __init__(self):
        # Base SuperBPE tokenizer with space prediction
        self.tokenizer = SuperBPETokenizer()

        # Error correction component
        self.error_corrector = ErrorCorrector()

        # Entity recognition component
        self.entity_recognizer = EntityRecognizer()

        # Product knowledge graph for validation
        self.product_knowledge = ProductKnowledgeGraph()

    def process_query(self, raw_query):
        # Generate tokenization hypotheses
        tokenization_hypotheses = self.tokenizer.generate_hypotheses(raw_query)

        # For top hypotheses, apply error correction
        corrected_hypotheses = []
        for tokens, score in tokenization_hypotheses[:5]:  # Consider top 5
            corrected_tokens = self.error_corrector.correct(tokens)
            corrected_score = self.tokenizer.score_tokens(corrected_tokens)
            corrected_hypotheses.append((corrected_tokens, corrected_score))

        # Select best corrected hypothesis
        best_tokens, _ = max(corrected_hypotheses, key=lambda x: x[1])

        # Perform entity recognition on best tokens
        entities = self.entity_recognizer.recognize_entities(best_tokens)

        # Validate against product knowledge
        validated_entities = self.product_knowledge.validate(entities)

        # Structure the final response
        return self.structure_response(best_tokens, validated_entities)
```

Inference Pipeline

During runtime, the system would:

1. Take a raw query like `"samsunggalaxys22ultrapromax5g"`
2. Generate multiple segmentation hypotheses
3. Apply error correction to promising candidates

4. Perform entity recognition on corrected tokens
5. Validate against product knowledge graph
6. Return structured understanding with entity types

This enables not just better search matching, but more intelligent responses that understand what the user is looking for at an entity level.

Conclusion

SuperBPE's approach to entity recognition and disambiguation offers a significant advancement for Samsung's product search system. By jointly modeling tokenization, error correction, and entity recognition, it can handle the complex challenges of:

1. Disambiguating between similar product names
2. Correcting multiple types of errors simultaneously
3. Distinguishing between products and features

The resulting system doesn't just match keywords but understands the structural relationships between entities in Samsung's product ecosystem, allowing for more precise, intelligent responses to user queries.