CS310 NLP - Assignment 1 Report

1. Data Processing (15 points)

Basic Tokenizer Implementation

We first implemented a basic character-level tokenizer that treats each Chinese character as an individual token and discards all non-Chinese characters:

```
class BasicTokenizer:
    def __init__(self):
        pass

    def tokenize(self, text):
        # Only keep Chinese characters
        chinese_chars = re.findall(r'[\u4e00-\u9fff]', text)
        return chinese_chars
```

This tokenizer uses a regular expression to match characters within the Unicode range \u4e00 to \u9fff, which corresponds to Chinese characters.

Improved Tokenizer

Next, we implemented an improved tokenizer that can recognize special patterns, including consecutive digits, English words, and punctuation:

```
class ImprovedTokenizer:
    def __init__(self):
        pass

    def tokenize(self, text):
        tokens = []
        # Match Chinese characters, consecutive digits, English words, and punctuation
        pattern = r'[\u4e00-\u9fff]|[0-9]+|[a-zA-Z]+|[^\w\s]'
        matches = re.finditer(pattern, text)
        for match in matches:
            tokens.append(match.group(0))
        return tokens
```

This tokenizer uses a more complex regular expression that matches:

- Chinese characters: [\u4e00-\u9fff]
- Consecutive digits: [0-9]+
- English words: [a-zA-Z]+
- Punctuation: [^\w\s]

Dataset Class Implementation

We created a custom HumorDataset class that inherits from PyTorch's Dataset class to load and process the data:

```
class HumorDataset(Dataset):
    def __init__(self, file_path, tokenizer, vocab=None, max_len=100):
        # Initialize data, labels, and tokenizer
        # Build vocabulary
        # ...

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # Convert tokens to indices
        # Pad sequences
        # ...
```

This class implements the following functionalities:

1. Reading JSONL files and processing text using the specified tokenizer
2. Building a vocabulary (if not provided)
3. Converting tokens to indices
4. Padding sequences to a fixed length

2. Building the Model (15 points)

We built a neural network model using PyTorch's nn.Module:

```
class HumorClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dims=[256, 128], num_classes=2):
        super(HumorClassifier, self).__init__()

        # Implement bag-of-words using EmbeddingBag
        self.embedding = nn.EmbeddingBag(vocab_size, embed_dim, mode='mean')

        # Build fully-connected layers with at least 2 hidden layers
        layers = []
        input_dim = embed_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(input_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))  # Add dropout to prevent overfitting
            input_dim = hidden_dim

        layers.append(nn.Linear(input_dim, num_classes))

        self.fc_layers = nn.Sequential(*layers)

        # Initialize weights
        self.init_weights()
```

The model includes:

1. Embedding layer: Converts vocabulary indices to dense vectors
2. EmbeddingBag: Implements the bag-of-words method with mean pooling
3. Fully-connected layers: At least two hidden layers with ReLU activation and Dropout regularization
4. Output layer: Binary classification (humor/non-humor)

3. Training and Evaluation (10 points)

We implemented training and evaluation functions:

```
def train_model(model, train_loader, test_loader, epochs=10, lr=0.001, device='cpu'):
    # Initialize optimizer and loss function
    # Training loop
    # Record training history
    # ...

def evaluate_model(model, data_loader, device='cpu'):
    # Model evaluation
    # Calculate accuracy, precision, recall, and F1 score
    # ...
```

During training, we:

1. Used cross-entropy loss function and Adam optimizer
2. Evaluated the model on the test set after each epoch
3. Recorded training and testing accuracy, loss, and other metrics
4. Calculated precision, recall, and F1 score

Finally, our model achieved approximately 68% accuracy and F1 score on the test set.

4. Exploring Word Segmentation (10 points)

We implemented Chinese word segmentation using the jieba package:

```
class JiebaTokenizer:
    def __init__(self):
        pass

    def tokenize(self, text):
        # Use jieba for Chinese word segmentation
        words = jieba.cut(text, cut_all=False)
        return list(words)
```

We compared the performance of three tokenization methods:

1. Basic character-level tokenizer
2. Improved tokenizer

3. Jieba tokenizer

The experimental results showed that:

字符级	2230	0.6743	0.6768	0.6743	0.6756
改进的	2294	0.6943	0.7078	0.6943	0.7003
jieba	6019	0.6482	0.6738	0.6482	0.6589

Contrary to our expectations, the Jieba word segmentation method performed worse than both character-level and improved tokenization methods. Here are several potential reasons for this unexpected result:

1. Vocabulary Size and Data Sparsity

Large Vocabulary: Jieba created a much larger vocabulary (6019 tokens) compared to the other methods (~2200 tokens). This significant increase leads to a more sparse representation.

Data Sparsity Problem: With a larger vocabulary, each token appears less frequently in the training data, making it harder for the model to learn meaningful representations for each token.

Limited Training Data: The humor detection dataset may not be large enough to effectively train embeddings for such a large vocabulary.

2. Domain Mismatch

General vs. Domain-Specific: Jieba is trained on general Chinese text, while humor often involves wordplay, puns, and domain-specific language.

Segmentation Errors: Jieba might incorrectly segment humor-specific expressions or colloquial language that are crucial for humor detection.

Loss of Character-Level Information: Some humor in Chinese relies on character-level wordplay, which might be lost when segmenting into words.

3. Model Architecture Considerations

Bag-of-Words Limitation: Our model uses a bag-of-words approach (EmbeddingBag), which disregards word order. This might be more problematic with word-level tokens than with character-level tokens.

Fixed Embedding Dimension: We used the same embedding dimension for all methods, but word-level embeddings might require higher dimensions to capture semantic information effectively.

4. Humor-Specific Linguistic Features

Character-Level Humor: Chinese humor often relies on character-level features, such as homophony (similar-sounding characters) or visual similarities between characters.

Contextual Understanding: Humor detection might benefit more from understanding the relationships between adjacent characters rather than pre-defined word boundaries.

Ambiguity Preservation: Character-level processing might better preserve the ambiguity that is often central to humor.

Recommendations for Improvement

1. Hybrid Approach: Combine character-level and word-level features to capture both granularities of information.
2. Custom Segmentation: Train a domain-specific word segmenter for humor text.
3. Contextual Embeddings: Use models like BERT that can better capture contextual information.
4. Feature Engineering: Add specific features that capture humor-related linguistic phenomena.
5. Reduce Vocabulary: Implement stricter frequency thresholds for Jieba tokens to reduce vocabulary size.

This analysis highlights that more sophisticated NLP techniques don't always yield better results - the choice of tokenization method should be tailored to the specific task and language characteristics.