Handout 3: Aspect-Based Sentiment Emotion Detection (Advanced Level)

Machine Learning for Smarter Innovation

1 Handout 3: Aspect-Based Sentiment & Emotion Detection (Advanced Level)

1.1 Week 3 - Advanced NLP Techniques for Deep User Understanding

1.1.1 Introduction to Aspect-Based Sentiment Analysis (ABSA)

Traditional sentiment analysis tells us if a review is positive or negative overall. ABSA goes deeper, identifying sentiment for specific aspects of a product or service.

Example: - **Overall**: "Mixed feelings about this laptop" → Neutral - **Aspect-based**: - Performance: "blazing fast processor" → Positive - Battery: "dies after 2 hours" → Negative - Design: "sleek and professional" → Positive - Price: "overpriced for the specs" → Negative

1.1.2 Multi-Aspect Sentiment Analysis Implementation

Aspect Extraction

```
import spacy
from transformers import pipeline
import pandas as pd

class AspectSentimentAnalyzer:
    def __init__(self):
        self.nlp = spacy.load("en_core_web_sm")
        self.sentiment_analyzer = pipeline("sentiment-analysis")
        self.aspect_keywords = {
            'performance': ['fast', 'slow', 'speed', 'quick', 'lag', 'smooth'
    ],
            'battery': ['battery', 'charge', 'power', 'lasting'],
            'display': ['screen', 'display', 'brightness', 'resolution'],
            'price': ['price', 'cost', 'expensive', 'cheap', 'value', 'worth'
    ],
            'quality': ['build', 'quality', 'durable', 'solid', 'flimsy'],
            'design': ['design', 'look', 'aesthetic', 'beautiful', 'ugly'],
            'support': ['support', 'service', 'help', 'response'],
        }

    def extract_aspects(self, text):
        """Extract aspect-specific sentences from text."""
        doc = self.nlp(text)
        aspect_sentences = {aspect: [] for aspect in self.aspect_keywords}

        for sent in doc.sents:
            sent_text = sent.text.lower()
            for aspect, keywords in self.aspect_keywords.items():
                if any(keyword in sent_text for keyword in keywords):
```

```
                    aspect_sentences[aspect].append(sent.text)

        return aspect_sentences

    def analyze_aspect_sentiment(self, text):
        """Analyze sentiment for each aspect in the text."""
        aspect_sentences = self.extract_aspects(text)
        aspect_sentiments = {}

        for aspect, sentences in aspect_sentences.items():
            if sentences:
                # Analyze sentiment for each sentence
                sentiments = []
                for sentence in sentences:
                    result = self.sentiment_analyzer(sentence)[0]
                    sentiments.append({
                        'sentence': sentence,
                        'sentiment': result['label'],
                        'score': result['score']
                    })

                # Aggregate sentiment for the aspect
                avg_score = sum(s['score'] if s['sentiment'] == 'POSITIVE'
    else -s['score']
                                for s in sentiments) / len(sentiments)

                aspect_sentiments[aspect] = {
                    'overall_sentiment': 'positive' if avg_score > 0 else '
    negative',
                    'confidence': abs(avg_score),
                    'details': sentiments
                }

        return aspect_sentiments
```

Advanced Aspect Extraction with Dependency Parsing

```
def extract_aspect_opinion_pairs(text):
    """Extract aspect-opinion pairs using dependency parsing."""
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(text)

    aspect_opinion_pairs = []

    for token in doc:
        # Find adjectives (opinions)
        if token.pos_ == "ADJ":
            # Find the aspect (noun) it modifies
            for child in token.children:
                if child.pos_ == "NOUN":
                    aspect_opinion_pairs.append({
                        'aspect': child.text,
                        'opinion': token.text,
                        'sentiment': analyze_opinion_sentiment(token.text)
                    })

            # Check if adjective modifies a noun through 'amod' relation
            if token.dep_ == "amod" and token.head.pos_ == "NOUN":
                aspect_opinion_pairs.append({
                    'aspect': token.head.text,
                    'opinion': token.text,
                    'sentiment': analyze_opinion_sentiment(token.text)
                })
```

```
    return aspect_opinion_pairs

def analyze_opinion_sentiment(opinion):
    """Simple sentiment scoring for opinion words."""
    positive_opinions = ['good', 'great', 'excellent', 'amazing', 'fast', '
    beautiful']
    negative_opinions = ['bad', 'poor', 'terrible', 'slow', 'ugly', 'broken']

    if opinion.lower() in positive_opinions:
        return 'positive'
    elif opinion.lower() in negative_opinions:
        return 'negative'
    else:
        return 'neutral'
```

1.1.3 Emotion Classification: Beyond Positive/Negative

8-Emotion Classification System

```
from transformers import pipeline

class EmotionDetector:
    def __init__(self):
        # Using a model trained on 8 emotions
        self.classifier = pipeline(
            "text-classification",
            model="j-hartmann/emotion-english-distilroberta-base",
            return_all_scores=True
        )
        self.emotions = ['anger', 'disgust', 'fear', 'joy',
                         'neutral', 'sadness', 'surprise', 'shame']

    def detect_emotions(self, text):
        """Detect multiple emotions with confidence scores."""
        results = self.classifier(text)[0]

        # Sort by score
        emotions = sorted(results, key=lambda x: x['score'], reverse=True)

        # Get primary and secondary emotions
        primary = emotions[0]
        secondary = emotions[1] if emotions[1]['score'] > 0.2 else None

        return {
            'primary_emotion': primary['label'],
            'primary_confidence': primary['score'],
            'secondary_emotion': secondary['label'] if secondary else None,
            'secondary_confidence': secondary['score'] if secondary else 0,
            'all_emotions': {e['label']: e['score'] for e in emotions}
        }

    def analyze_emotional_journey(self, reviews_timeline):
        """Track emotion evolution over time."""
        journey = []

        for timestamp, review in reviews_timeline:
            emotions = self.detect_emotions(review)
            journey.append({
                'timestamp': timestamp,
                'text': review,
                'emotion': emotions['primary_emotion'],
                'confidence': emotions['primary_confidence']
```

```
            })

        return pd.DataFrame(journey)
```

Multi-Label Emotion Detection

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

class MultiEmotionDetector:
    def __init__(self, threshold=0.3):
        self.threshold = threshold
        model_name = "SamLowe/roberta-base-go_emotions"
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForSequenceClassification.from_pretrained(
    model_name)

        # 28 emotions from GoEmotions dataset
        self.emotions = [
            'admiration', 'amusement', 'anger', 'annoyance', 'approval',
            'caring', 'confusion', 'curiosity', 'desire', 'disappointment',
            'disapproval', 'disgust', 'embarrassment', 'excitement', 'fear',
            'gratitude', 'grief', 'joy', 'love', 'nervousness', 'optimism',
            'pride', 'realization', 'relief', 'remorse', 'sadness',
            'surprise', 'neutral'
        ]

    def detect_multiple_emotions(self, text):
        """Detect multiple co-occurring emotions."""
        inputs = self.tokenizer(text, return_tensors="pt", truncation=True,
    padding=True)

        with torch.no_grad():
            outputs = self.model(**inputs)
            scores = torch.sigmoid(outputs.logits).squeeze().tolist()

        # Get emotions above threshold
        detected_emotions = []
        for emotion, score in zip(self.emotions, scores):
            if score > self.threshold:
                detected_emotions.append({
                    'emotion': emotion,
                    'confidence': score
                })

        return sorted(detected_emotions, key=lambda x: x['confidence'],
    reverse=True)
```

1.1.4 Sarcasm and Irony Detection

Rule-Based + ML Hybrid Approach

```
class SarcasmDetector:
    def __init__(self):
        self.sentiment_analyzer = pipeline("sentiment-analysis")
        self.sarcasm_indicators = [
            'yeah right', 'sure', 'oh great', 'wonderful', 'fantastic',
            'brilliant', 'genius', 'obviously', 'clearly'
        ]
        self.punctuation_patterns = ['!!!', '...', '?!', '!?']

    def detect_sarcasm(self, text, context=None):
```

```
        """Detect sarcasm using multiple signals."""
        text_lower = text.lower()
        sarcasm_score = 0

        # Check for sarcasm indicators
        for indicator in self.sarcasm_indicators:
            if indicator in text_lower:
                sarcasm_score += 0.3

        # Check for excessive punctuation
        for pattern in self.punctuation_patterns:
            if pattern in text:
                sarcasm_score += 0.2

        # Check for contradiction between words and sentiment
        if any(word in text_lower for word in ['love', 'great', 'amazing']):
            sentiment = self.sentiment_analyzer(text)[0]
            # If positive words but overall negative context
            if context and 'complaint' in context.lower():
                sarcasm_score += 0.4

        # Check for quotation marks around positive words
        import re
        quoted_positives = re.findall(r'"([^"]*)"', text)
        for quote in quoted_positives:
            if any(pos in quote.lower() for pos in ['great', 'wonderful', '
    amazing']):
                sarcasm_score += 0.3

        return {
            'is_sarcastic': sarcasm_score > 0.5,
            'confidence': min(sarcasm_score, 1.0),
            'indicators_found': self._get_found_indicators(text_lower)
        }

    def _get_found_indicators(self, text_lower):
        """Return which sarcasm indicators were found."""
        return [ind for ind in self.sarcasm_indicators if ind in text_lower]
```

Deep Learning Sarcasm Detection

```
from transformers import pipeline

def advanced_sarcasm_detection(texts):
    """Use a fine-tuned model for sarcasm detection."""
    # Load sarcasm detection model
    sarcasm_model = pipeline(
        "text-classification",
        model="mrm8488/t5-base-finetuned-sarcasm-twitter"
    )

    results = []
    for text in texts:
        prediction = sarcasm_model(text)[0]

        # Adjust interpretation based on context
        if prediction['label'] == 'SARCASM':
            # Double-check with sentiment analysis
            sentiment = pipeline("sentiment-analysis")(text)[0]

            # High confidence sarcasm if positive words but sarcasm detected
            contains_positive = any(word in text.lower()
```

```
                                for word in ['great', 'love', 'amazing', '
    wonderful'])

            confidence = prediction['score']
            if contains_positive:
                confidence = min(confidence * 1.2, 1.0)

            results.append({
                'text': text,
                'is_sarcastic': True,
                'confidence': confidence,
                'implied_sentiment': 'negative' if sentiment['label'] == '
    POSITIVE' else 'positive'
            })
        else:
            results.append({
                'text': text,
                'is_sarcastic': False,
                'confidence': prediction['score'],
                'implied_sentiment': None
            })

    return results
```

1.1.5 Building Production Pipelines

Complete NLP Pipeline

```
class ProductionNLPPipeline:
    def __init__(self):
        self.aspect_analyzer = AspectSentimentAnalyzer()
        self.emotion_detector = EmotionDetector()
        self.sarcasm_detector = SarcasmDetector()

    def full_analysis(self, text, user_context=None):
        """Complete analysis pipeline for production."""

        # 1. Preprocess
        cleaned_text = self.preprocess(text)

        # 2. Detect sarcasm first (affects interpretation)
        sarcasm = self.sarcasm_detector.detect_sarcasm(cleaned_text,
    user_context)

        # 3. Aspect-based sentiment
        aspects = self.aspect_analyzer.analyze_aspect_sentiment(cleaned_text)

        # 4. Emotion detection
        emotions = self.emotion_detector.detect_emotions(cleaned_text)

        # 5. Adjust interpretations if sarcastic
        if sarcasm['is_sarcastic']:
            aspects = self.invert_sentiments(aspects)
            emotions = self.adjust_emotions_for_sarcasm(emotions)

        # 6. Generate insights
        insights = self.generate_insights(aspects, emotions, sarcasm)

        return {
            'original_text': text,
            'preprocessed_text': cleaned_text,
            'sarcasm': sarcasm,
            'aspects': aspects,
```

```
            'emotions': emotions,
            'insights': insights,
            'priority_score': self.calculate_priority(aspects, emotions)
        }

    def preprocess(self, text):
        """Clean and prepare text for analysis."""
        # Remove URLs, mentions, etc.
        import re
        text = re.sub(r'http\S+', '', text)
        text = re.sub(r'@\w+', '', text)
        text = re.sub(r'#(\w+)', r'\1', text)  # Keep hashtag content
        text = re.sub(r'\s+', ' ', text)
        return text.strip()

    def invert_sentiments(self, aspects):
        """Invert sentiments if sarcasm detected."""
        inverted = {}
        for aspect, data in aspects.items():
            inverted[aspect] = data.copy()
            if data['overall_sentiment'] == 'positive':
                inverted[aspect]['overall_sentiment'] = 'negative'
                inverted[aspect]['sarcasm_adjusted'] = True
            elif data['overall_sentiment'] == 'negative':
                inverted[aspect]['overall_sentiment'] = 'positive'
                inverted[aspect]['sarcasm_adjusted'] = True
        return inverted

    def adjust_emotions_for_sarcasm(self, emotions):
        """Adjust emotion detection for sarcastic text."""
        # Sarcasm often indicates frustration or anger
        adjusted = emotions.copy()
        if emotions['primary_emotion'] == 'joy':
            adjusted['primary_emotion'] = 'anger'
            adjusted['sarcasm_adjusted'] = True
        return adjusted

    def generate_insights(self, aspects, emotions, sarcasm):
        """Generate actionable insights from analysis."""
        insights = []

        # Critical issues (negative sentiment + strong emotion)
        for aspect, sentiment_data in aspects.items():
            if sentiment_data.get('overall_sentiment') == 'negative':
                if sentiment_data.get('confidence', 0) > 0.8:
                    insights.append({
                        'type': 'critical_issue',
                        'aspect': aspect,
                        'action': f'Urgent attention needed for {aspect}',
                        'emotion': emotions['primary_emotion']
                    })

        # Sarcasm detection insight
        if sarcasm['is_sarcastic']:
            insights.append({
                'type': 'user_frustration',
                'indicator': 'Sarcasm detected',
                'action': 'User likely frustrated despite positive words'
            })

        # Emotional state insights
        if emotions['primary_emotion'] in ['anger', 'disgust', 'fear']:
            insights.append({
```

```
                'type': 'negative_emotion',
                'emotion': emotions['primary_emotion'],
                'action': 'Immediate response recommended'
            })

        return insights

    def calculate_priority(self, aspects, emotions):
        """Calculate priority score for response."""
        score = 0

        # Negative aspects increase priority
        negative_aspects = sum(1 for a in aspects.values()
                               if a.get('overall_sentiment') == 'negative')
        score += negative_aspects * 20

        # Strong negative emotions increase priority
        if emotions['primary_emotion'] in ['anger', 'disgust']:
            score += 30
        elif emotions['primary_emotion'] in ['sadness', 'fear']:
            score += 20

        # High confidence increases priority
        if emotions['primary_confidence'] > 0.9:
            score += 10

        return min(score, 100)  # Cap at 100
```

Batch Processing with Caching

```
import hashlib
import json
import redis

class CachedSentimentAnalyzer:
    def __init__(self, redis_host='localhost', redis_port=6379):
        self.pipeline = ProductionNLPPipeline()
        self.cache = redis.Redis(host=redis_host, port=redis_port, db=0)
        self.cache_ttl = 86400  # 24 hours

    def analyze_with_cache(self, text):
        """Analyze text with caching for efficiency."""
        # Generate cache key
        cache_key = hashlib.md5(text.encode()).hexdigest()

        # Check cache
        cached = self.cache.get(cache_key)
        if cached:
            return json.loads(cached)

        # Analyze if not cached
        result = self.pipeline.full_analysis(text)

        # Store in cache
        self.cache.setex(
            cache_key,
            self.cache_ttl,
            json.dumps(result, default=str)
        )

        return result

    def batch_analyze(self, texts, batch_size=32):
```

```
        """Efficiently process large batches."""
        results = []

        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]

            # Process batch in parallel (simplified)
            batch_results = [self.analyze_with_cache(text) for text in batch]
            results.extend(batch_results)

            # Log progress
            print(f"Processed {min(i+batch_size, len(texts))}/{len(texts)}")

        return results
```

1.1.6 Real-World Application: Customer Feedback Analysis

```
class CustomerFeedbackAnalyzer:
    def __init__(self):
        self.pipeline = ProductionNLPPipeline()

    def analyze_product_feedback(self, reviews_df):
        """Comprehensive analysis of product reviews."""
        results = {
            'aspect_summary': {},
            'emotion_distribution': {},
            'critical_issues': [],
            'positive_highlights': [],
            'improvement_suggestions': []
        }

        for _, review in reviews_df.iterrows():
            analysis = self.pipeline.full_analysis(review['text'])

            # Aggregate aspect sentiments
            for aspect, data in analysis['aspects'].items():
                if aspect not in results['aspect_summary']:
                    results['aspect_summary'][aspect] = {'positive': 0, '
   negative': 0}

                if data['overall_sentiment'] == 'positive':
                    results['aspect_summary'][aspect]['positive'] += 1
                else:
                    results['aspect_summary'][aspect]['negative'] += 1

            # Track emotions
            emotion = analysis['emotions']['primary_emotion']
            results['emotion_distribution'][emotion] = \
                results['emotion_distribution'].get(emotion, 0) + 1

            # Identify critical issues
            if analysis['priority_score'] > 70:
                results['critical_issues'].append({
                    'review': review['text'][:200],
                    'aspects': list(analysis['aspects'].keys()),
                    'emotion': emotion,
                    'priority': analysis['priority_score']
                })

        # Generate improvement suggestions
        for aspect, counts in results['aspect_summary'].items():
```

```
            if counts['negative'] > counts['positive']:
                results['improvement_suggestions'].append({
                    'aspect': aspect,
                    'negative_ratio': counts['negative'] / (counts['negative']
    + counts['positive']),
                    'recommendation': f"Focus on improving {aspect} - {counts
    ['negative']} negative mentions"
                })

        return results

    def generate_report(self, analysis_results):
        """Generate executive summary from analysis."""
        report = []
        report.append("# Customer Feedback Analysis Report\n")

        # Aspect Summary
        report.append("## Aspect Performance\n")
        for aspect, counts in analysis_results['aspect_summary'].items():
            total = counts['positive'] + counts['negative']
            if total > 0:
                satisfaction = counts['positive'] / total * 100
                report.append(f"- **{aspect}**: {satisfaction:.1f}%
    satisfaction "
                              f"({counts['positive']} positive, {counts['
    negative']} negative)\n")

        # Emotion Distribution
        report.append("\n## Emotional Response\n")
        total_reviews = sum(analysis_results['emotion_distribution'].values())
        for emotion, count in sorted(analysis_results['emotion_distribution'].
    items(),
                                     key=lambda x: x[1], reverse=True):
            percentage = count / total_reviews * 100
            report.append(f"- {emotion}: {percentage:.1f}% ({count} reviews)\n
    ")

        # Critical Issues
        report.append("\n## Critical Issues Requiring Attention\n")
        for issue in analysis_results['critical_issues'][:5]:  # Top 5
            report.append(f"- Priority {issue['priority']}: {issue['review']}\
    n")
            report.append(f"  - Aspects: {', '.join(issue['aspects'])}\n")
            report.append(f"  - Emotion: {issue['emotion']}\n")

        # Recommendations
        report.append("\n## Recommendations\n")
        for suggestion in analysis_results['improvement_suggestions']:
            report.append(f"- {suggestion['recommendation']}\n")

        return ''.join(report)
```

1.1.7 Practice Exercise: Complete Sentiment System

Build an end-to-end sentiment analysis system:

```
# Assignment: Build a comprehensive feedback analyzer
class CompleteFeedbackSystem:
    """
    Your task:
    1. Implement aspect extraction for your domain
```

```
    2. Add emotion detection
    3. Include sarcasm detection
    4. Create priority scoring
    5. Generate actionable insights
    6. Build API endpoint
    7. Add monitoring and logging
    """

    def __init__(self, domain='general'):
        # Initialize your components
        pass

    def analyze(self, text):
        # Implement complete analysis
        pass

    def generate_insights(self, analysis):
        # Convert analysis to actionable insights
        pass

    def create_dashboard_data(self, batch_results):
        # Prepare data for visualization
        pass

# Test with real data
if __name__ == "__main__":
    system = CompleteFeedbackSystem(domain='e-commerce')

    # Test cases
    test_reviews = [
        "Love the camera quality but the battery life is terrible!",
        "Oh great, another 'premium' phone that crashes every hour. Fantastic!
    ",
        "The display is gorgeous, performance is smooth, but way overpriced.",
        "Worst purchase ever. Screen broke on day 2. Support was useless.",
        "Exceeds expectations! Fast delivery, great packaging, product works
    perfectly."
    ]

    for review in test_reviews:
        analysis = system.analyze(review)
        print(f"\nReview: {review}")
        print(f"Analysis: {analysis}")
        print(f"Insights: {system.generate_insights(analysis)}")
```

1.1.8 Best Practices for Production

1. **Performance Optimization**
   - Use model quantization for faster inference
   - Implement caching for repeated queries
   - Batch process when possible
2. **Accuracy Improvement**
   - Combine multiple models for consensus
   - Use domain-specific fine-tuning
   - Implement confidence thresholds
3. **Monitoring**
   - Track model drift over time
   - Log edge cases for retraining

- Monitor response times

4. **Error Handling**
   - Graceful degradation for failures
   - Fallback to simpler models
   - Human-in-the-loop for low confidence

1.1.9 Resources

- Aspect-Based Sentiment Papers: https://arxiv.org/abs/1805.01984
- Emotion Detection Datasets: https://github.com/google-research/google-research/tree/master/goemotions
- Sarcasm Detection: https://arxiv.org/abs/1704.05579
- Production Best Practices: https://mlops.community/

Remember: The goal is not just to detect sentiment, but to understand the complete emotional context and generate actionable insights for better design decisions!