NLP Sentiment Analysis - Advanced Handout

Machine Learning for Smarter Innovation

1 NLP Sentiment Analysis - Advanced Handout

Target Audience: Data scientists and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations and production systems)

1.1 Mathematical Foundations of Sentiment Analysis

1.1.1 Probabilistic Framework

Sentiment analysis maps text sequences to sentiment labels. Let x = (x_1, x_2, . . . , x_n) be a sequence of tokens and y be the sentiment label from a discrete set Y = {positive, negative, neutral}. The goal is to learn a function f: X -> Y that maximizes:

P(y|x) = exp(score(x, y)) / sum_{y' in Y} exp(score(x, y'))

where score(x, y) measures compatibility between input text and sentiment class. This softmax formulation enables probabilistic interpretation and gradient-based optimization.

1.1.2 Cross-Entropy Loss

For training, we minimize cross-entropy loss over labeled examples:

L = -sum_{i=1}^{N} sum_{c=1}^{C} y_{ic} log(p_{ic})

where y_{ic} is the indicator (1 if sample i belongs to class c) and p_{ic} is the predicted probability. For binary sentiment:

L = -sum_{i=1}^{N} [y_i log(p_i) + (1-y_i) log(1-p_i)]

This loss function is convex in the log-probabilities, ensuring well-defined optimization landscape for logistic models.

1.1.3 Information-Theoretic Interpretation

Cross-entropy measures bits needed to encode ground truth using predicted distribution. The KL divergence between true and predicted distributions:

D_KL(p || q) = sum_y p(y) log(p(y)/q(y)) = H(p,q) - H(p)

Since H(p) is constant for fixed training data, minimizing cross-entropy H(p,q) equals minimizing KL divergence. This connects sentiment classification to information compression.

1.2 Transformer Architecture for Sentiment

1.2.1 Self-Attention Mechanism

The attention function computes weighted combinations of values based on query-key similarity:

Attention(Q, K, V) = softmax(QKˆT / sqrt(d_k)) V

where Q (queries), K (keys), V (values) are linear projections of input embeddings, and d_k is key dimension. The scaling factor 1/sqrt(d_k) prevents softmax saturation for large dimensions.

For input sequence H = [h_1, . . . , h_n], we compute:

Q = H W_Q, K = H W_K, V = H W_V

The attention weight between positions i and j:

alpha_{ij} = exp(q_i . k_j / sqrt(d_k)) / sum_{m=1}ˆ{n} exp(q_i . k_m / sqrt(d_k))

This allows each position to attend to all other positions, capturing long-range dependencies crucial for sentiment that spans sentences.

1.2.2 Multi-Head Attention

Multiple attention heads learn different relationship types:

MultiHead(Q, K, V) = Concat(head_1, . . . , head_h) W_O

head_i = Attention(Q W_Qˆi, K W_Kˆi, V W_Vˆi)

With h=12 heads and d_model=768, each head operates on d_k = d_model/h = 64 dimensions. Different heads can specialize: some capture syntactic dependencies, others semantic relationships relevant to sentiment.

1.2.3 Position Encoding

Transformers lack inherent position information. BERT uses learned position embeddings:

E(x_i) = E_token(x_i) + E_position(i) + E_segment(s_i)

where E_token is token embedding, E_position is position embedding (learned for positions 0-511), and E_segment indicates sentence membership.

Alternative sinusoidal encoding:

PE(pos, 2i) = sin(pos / 10000ˆ{2i/d_model}) PE(pos, 2i+1) = cos(pos / 10000ˆ{2i/d_model})

This enables extrapolation to longer sequences than seen during training.

1.3 BERT Architecture and Pre-training

1.3.1 Model Architecture

BERT-base: 12 layers, 768 hidden dimensions, 12 attention heads (110M parameters) BERT-large: 24 layers, 1024 hidden dimensions, 16 attention heads (340M parameters)

Each transformer layer:

```
H' = LayerNorm(H + MultiHead(H, H, H))
H_out = LayerNorm(H' + FFN(H'))
```

where FFN is position-wise feed-forward network:

FFN(x) = max(0, x W_1 + b_1) W_2 + b_2

with inner dimension 4 * d_model = 3072 for BERT-base.

1.3.2 Pre-training Objectives

Masked Language Modeling (MLM): Randomly mask 15% of tokens. For masked position i: - 80%: replace with [MASK] - 10%: replace with random token - 10%: keep original

Objective: predict original token from context:

L_MLM = -sum_{i in M} log P(x_i | x_{})

where M is set of masked positions and x_{} is sequence with masks.

Next Sentence Prediction (NSP): Given sentence pair (A, B), predict if B follows A:

L_NSP = -[y log P(IsNext|A,B) + (1-y) log P(NotNext|A,B)]

where y=1 if B is actual next sentence. RoBERTa showed NSP provides minimal benefit; later models omit it.

1.3.3 Fine-tuning for Sentiment

For classification, use [CLS] token representation:

h_CLS = BERT(x)_0 logits = h_CLS W_c + b_c P(y|x) = softmax(logits)

Fine-tuning updates all parameters with small learning rate (2e-5 to 5e-5) for 2-4 epochs. Gradient accumulation enables effective batch sizes of 16-32 on limited GPU memory.

1.4 Aspect-Based Sentiment Analysis Theory

1.4.1 Problem Formulation

ABSA extracts (aspect, sentiment) pairs from text. Given sentence s, identify: 1. Aspect terms A = {a_1, . . . , a_k} 2. Sentiment polarity p(a_i) for each aspect

This decomposes into: - Aspect Term Extraction (ATE): sequence labeling with BIO tags - Aspect Sentiment Classification (ASC): given aspect, predict sentiment

1.4.2 Attention-Based Aspect Sentiment

For aspect term a in sentence s, compute aspect-specific context:

alpha_i = softmax(h_iˆT W_a h_a) c_a = sum_i alpha_i h_i

where h_a is aspect representation and h_i is context word representation. The attention weights alpha_i highlight words relevant to the specific aspect.

Final prediction:

P(p|a, s) = softmax(W_p [c_a; h_a] + b_p)

This allows different sentiments for different aspects in the same sentence.

1.4.3 Multi-Task Learning

Joint training on ATE and ASC improves both:

L = lambda_1 L_ATE + lambda_2 L_ASC + lambda_3 L_OE

where L_OE is opinion extraction loss. Shared encoder learns representations useful for all tasks. Typical lambda values: 1.0, 1.0, 0.5.

1.5 Emotion Detection Theory

1.5.1 Plutchik's Wheel and Dimensional Models

Discrete emotions (Ekman): anger, disgust, fear, happiness, sadness, surprise

Dimensional model (Russell): emotions in valence-arousal space: - Valence: negative to positive - Arousal: calm to excited

Mapping between models: - joy: high valence, medium arousal - anger: low valence, high arousal - sadness: low valence, low arousal

1.5.2 Multi-Label Emotion Classification

GoEmotions dataset: 28 fine-grained emotions. Use sigmoid outputs instead of softmax:

P(e_j|x) = sigmoid(W_j h_CLS + b_j)

Loss for multi-label:

L = -sum_{j=1}^{28} [y_j log(p_j) + (1-y_j) log(1-p_j)]

Threshold selection via validation F1 optimization. Different emotions may require different thresholds.

1.5.3 Emotion Intensity Regression

Beyond classification, predict intensity on continuous scale:

intensity = sigmoid(W h_CLS + b) * scale

where scale is maximum intensity (e.g., 4.0). Train with MSE loss:

L = sum_{i=1}^{N} (intensity_i - y_i)^2

Ordinal regression alternative: predict cumulative probabilities P(intensity >= k) for k in {1, 2, 3, 4}.

1.6 Implementation

1.6.1 Setup

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from transformers import (
    AutoTokenizer, AutoModel, AutoModelForSequenceClassification,
    AdamW, get_linear_schedule_with_warmup
)
from sklearn.metrics import classification_report, f1_score
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1.6.2 Custom Attention-Based Aspect Sentiment Model

```
class AspectSentimentModel(nn.Module):
    """Attention-based model for aspect-level sentiment."""

    def __init__(self, model_name='bert-base-uncased', num_classes=3, dropout
    =0.3):
        super().__init__()
        self.encoder = AutoModel.from_pretrained(model_name)
        self.hidden_size = self.encoder.config.hidden_size

        # Aspect attention
        self.aspect_attention = nn.Sequential(
            nn.Linear(self.hidden_size * 2, self.hidden_size),
            nn.Tanh(),
            nn.Linear(self.hidden_size, 1)
        )

        # Classification head
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(self.hidden_size * 2, self.hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(self.hidden_size, num_classes)
        )

    def forward(self, input_ids, attention_mask, aspect_mask):
        """
        Args:
            input_ids: [batch, seq_len]
            attention_mask: [batch, seq_len]
            aspect_mask: [batch, seq_len] - 1 for aspect tokens, 0 otherwise
        """
        # Encode full sequence
        outputs = self.encoder(input_ids, attention_mask=attention_mask)
        hidden = outputs.last_hidden_state  # [batch, seq_len, hidden]

        # Extract aspect representation (mean of aspect tokens)
        aspect_mask_expanded = aspect_mask.unsqueeze(-1).float()
        aspect_sum = (hidden * aspect_mask_expanded).sum(dim=1)
        aspect_count = aspect_mask_expanded.sum(dim=1).clamp(min=1)
        aspect_repr = aspect_sum / aspect_count  # [batch, hidden]

        # Compute attention weights
        batch_size, seq_len, _ = hidden.shape
        aspect_expanded = aspect_repr.unsqueeze(1).expand(-1, seq_len, -1)
        concat = torch.cat([hidden, aspect_expanded], dim=-1)  # [batch, seq,
    2*hidden]

        attention_scores = self.aspect_attention(concat).squeeze(-1)  # [batch
    , seq]
        attention_scores = attention_scores.masked_fill(attention_mask == 0,
    -1e9)
        attention_weights = F.softmax(attention_scores, dim=-1)

        # Compute context vector
        context = torch.bmm(attention_weights.unsqueeze(1), hidden).squeeze(1)

        # Classify using aspect and context
        combined = torch.cat([aspect_repr, context], dim=-1)
        logits = self.classifier(combined)
```

```
        return logits, attention_weights
```

1.6.3 Multi-Label Emotion Model

```
class MultiLabelEmotionModel(nn.Module):
    """Multi-label emotion classifier with calibrated outputs."""

    def __init__(self, model_name='roberta-base', num_emotions=28):
        super().__init__()
        self.encoder = AutoModel.from_pretrained(model_name)
        self.hidden_size = self.encoder.config.hidden_size

        # Per-emotion classifiers for better calibration
        self.emotion_heads = nn.ModuleList([
            nn.Sequential(
                nn.Dropout(0.2),
                nn.Linear(self.hidden_size, 256),
                nn.ReLU(),
                nn.Dropout(0.1),
                nn.Linear(256, 1)
            ) for _ in range(num_emotions)
        ])

        # Learnable thresholds per emotion
        self.thresholds = nn.Parameter(torch.zeros(num_emotions))

    def forward(self, input_ids, attention_mask):
        outputs = self.encoder(input_ids, attention_mask=attention_mask)
        pooled = outputs.last_hidden_state[:, 0]  # CLS token

        # Get logits from each head
        logits = torch.cat([head(pooled) for head in self.emotion_heads], dim
    =-1)

        return logits

    def predict(self, input_ids, attention_mask, use_thresholds=True):
        logits = self.forward(input_ids, attention_mask)
        probs = torch.sigmoid(logits)

        if use_thresholds:
            adjusted_thresholds = torch.sigmoid(self.thresholds)
            predictions = (probs > adjusted_thresholds).float()
        else:
            predictions = (probs > 0.5).float()

        return predictions, probs
```

1.6.4 Production Sentiment Pipeline

```
class ProductionSentimentPipeline:
    """Production-ready sentiment analysis with monitoring."""

    def __init__(self, model_name='cardiffnlp/twitter-roberta-base-sentiment')
    :
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForSequenceClassification.from_pretrained(
    model_name)
```

```
        self.model.to(device)
        self.model.eval()

        self.label_map = {0: 'negative', 1: 'neutral', 2: 'positive'}

        # Monitoring statistics
        self.inference_times = []
        self.confidence_distribution = []
        self.low_confidence_samples = []

    def preprocess(self, text):
        """Clean and normalize text."""
        import re
        text = re.sub(r'http\S+|www\S+', '[URL]', text)
        text = re.sub(r'@\w+', '[USER]', text)
        text = re.sub(r'#(\w+)', r'\1', text)
        text = re.sub(r'\s+', ' ', text).strip()
        return text[:512]  # Truncate for BERT

    @torch.no_grad()
    def analyze(self, text, return_confidence=True):
        """Analyze sentiment with confidence metrics."""
        import time
        start = time.time()

        cleaned = self.preprocess(text)
        inputs = self.tokenizer(
            cleaned, return_tensors='pt', truncation=True,
            padding=True, max_length=512
        ).to(device)

        outputs = self.model(**inputs)
        probs = F.softmax(outputs.logits, dim=-1)

        pred_idx = probs.argmax(dim=-1).item()
        confidence = probs[0, pred_idx].item()

        # Compute entropy as uncertainty measure
        entropy = -(probs * torch.log(probs + 1e-10)).sum().item()

        elapsed = time.time() - start
        self.inference_times.append(elapsed)
        self.confidence_distribution.append(confidence)

        if confidence < 0.6:
            self.low_confidence_samples.append({
                'text': text[:100], 'confidence': confidence
            })

        result = {
            'sentiment': self.label_map[pred_idx],
            'confidence': confidence,
            'entropy': entropy,
            'probabilities': {
                self.label_map[i]: probs[0, i].item()
                for i in range(3)
            }
        }

        if return_confidence:
            result['is_reliable'] = confidence > 0.7 and entropy < 0.8

        return result
```

```
    def analyze_batch(self, texts, batch_size=32):
        """Efficient batch processing."""
        results = []

        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]
            cleaned = [self.preprocess(t) for t in batch]

            inputs = self.tokenizer(
                cleaned, return_tensors='pt', truncation=True,
                padding=True, max_length=512
            ).to(device)

            with torch.no_grad():
                outputs = self.model(**inputs)
                probs = F.softmax(outputs.logits, dim=-1)

            for j, text in enumerate(batch):
                pred_idx = probs[j].argmax().item()
                results.append({
                    'text': text[:100],
                    'sentiment': self.label_map[pred_idx],
                    'confidence': probs[j, pred_idx].item()
                })

        return results

    def get_monitoring_stats(self):
        """Return monitoring statistics."""
        return {
            'total_inferences': len(self.inference_times),
            'avg_latency_ms': np.mean(self.inference_times) * 1000,
            'p95_latency_ms': np.percentile(self.inference_times, 95) * 1000,
            'avg_confidence': np.mean(self.confidence_distribution),
            'low_confidence_rate': len(self.low_confidence_samples) / max(len(
    self.inference_times), 1),
            'confidence_histogram': np.histogram(self.confidence_distribution,
     bins=10)[0].tolist()
        }
```

1.6.5 Sarcasm-Aware Sentiment Analysis

```
class SarcasmAwareSentimentAnalyzer:
    """Handles sarcasm to avoid misinterpretation."""

    def __init__(self):
        self.sentiment_pipeline = ProductionSentimentPipeline()
        self.sarcasm_patterns = [
            r'\b(oh\s+)?great\b.*[!.]{2,}',
            r'\b(yeah|sure)\s+right\b',
            r'\bwow\b.*\bso\b.*\b(helpful|useful|great)\b',
            r'\bthanks\s+(a\s+lot|so\s+much)\b.*[!]{2,}',
            r'"[^"]*\b(great|wonderful|amazing)\b[^"]*"'
        ]

    def detect_sarcasm_signals(self, text):
        """Detect linguistic markers of sarcasm."""
        import re

        signals = {
```

```
            'pattern_matches': [],
            'punctuation_emphasis': False,
            'quoted_positives': False,
            'contradiction_indicators': 0
        }

        text_lower = text.lower()

        for pattern in self.sarcasm_patterns:
            if re.search(pattern, text_lower):
                signals['pattern_matches'].append(pattern)

        if re.search(r'[!?]{3,}|\.{4,}', text):
            signals['punctuation_emphasis'] = True

        positive_quoted = re.findall(r'"([^"]*)"', text)
        for quote in positive_quoted:
            if any(word in quote.lower() for word in ['great', 'wonderful', '
amazing', 'love']):
                signals['quoted_positives'] = True

        # Check for contradiction (positive words in negative context)
        positive_count = sum(1 for w in ['love', 'great', 'amazing', '
wonderful'] if w in text_lower)
        negative_context = sum(1 for w in ['but', 'however', 'except', '
unfortunately'] if w in text_lower)
        signals['contradiction_indicators'] = min(positive_count,
negative_context)

        return signals

    def calculate_sarcasm_probability(self, signals):
        """Convert signals to sarcasm probability."""
        score = 0.0

        score += len(signals['pattern_matches']) * 0.25
        score += 0.15 if signals['punctuation_emphasis'] else 0
        score += 0.20 if signals['quoted_positives'] else 0
        score += signals['contradiction_indicators'] * 0.15

        return min(score, 0.95)  # Cap at 95%

    def analyze(self, text):
        """Analyze with sarcasm awareness."""
        base_sentiment = self.sentiment_pipeline.analyze(text)
        sarcasm_signals = self.detect_sarcasm_signals(text)
        sarcasm_prob = self.calculate_sarcasm_probability(sarcasm_signals)

        result = {
            'original_sentiment': base_sentiment['sentiment'],
            'confidence': base_sentiment['confidence'],
            'sarcasm_probability': sarcasm_prob,
            'sarcasm_signals': sarcasm_signals
        }

        # Adjust interpretation if sarcasm likely
        if sarcasm_prob > 0.5:
            sentiment_map = {'positive': 'negative', 'negative': 'positive', '
neutral': 'neutral'}
            result['adjusted_sentiment'] = sentiment_map[base_sentiment['
sentiment']]
            result['interpretation_note'] = 'Sarcasm detected; sentiment
inverted'
```

```
        else:
            result['adjusted_sentiment'] = base_sentiment['sentiment']
            result['interpretation_note'] = 'No significant sarcasm detected'

        return result
```

1.6.6 Training Utilities

```
class SentimentTrainer:
    """Training utilities for sentiment models."""

    def __init__(self, model, tokenizer, device='cuda'):
        self.model = model.to(device)
        self.tokenizer = tokenizer
        self.device = device

    def prepare_data(self, texts, labels, max_length=128):
        """Prepare dataset for training."""
        encodings = self.tokenizer(
            texts, truncation=True, padding=True,
            max_length=max_length, return_tensors='pt'
        )

        class SentimentDataset(Dataset):
            def __init__(self, encodings, labels):
                self.encodings = encodings
                self.labels = torch.tensor(labels)

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

            def __getitem__(self, idx):
                return {
                    'input_ids': self.encodings['input_ids'][idx],
                    'attention_mask': self.encodings['attention_mask'][idx],
                    'labels': self.labels[idx]
                }

        return SentimentDataset(encodings, labels)

    def train(self, train_data, val_data, epochs=3, lr=2e-5, batch_size=16):
        """Fine-tune model on sentiment data."""
        train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=
    True)
        val_loader = DataLoader(val_data, batch_size=batch_size)

        optimizer = AdamW(self.model.parameters(), lr=lr, weight_decay=0.01)
        total_steps = len(train_loader) * epochs
        scheduler = get_linear_schedule_with_warmup(
            optimizer, num_warmup_steps=int(0.1 * total_steps),
            num_training_steps=total_steps
        )

        best_f1 = 0
        history = {'train_loss': [], 'val_f1': []}

        for epoch in range(epochs):
            # Training
            self.model.train()
            total_loss = 0
```

```
            for batch in train_loader:
                batch = {k: v.to(self.device) for k, v in batch.items()}

                outputs = self.model(
                    input_ids=batch['input_ids'],
                    attention_mask=batch['attention_mask'],
                    labels=batch['labels']
                )

                loss = outputs.loss
                total_loss += loss.item()

                loss.backward()
                torch.nn.utils.clip_grad_norm_(self.model.parameters(), 1.0)
                optimizer.step()
                scheduler.step()
                optimizer.zero_grad()

            avg_loss = total_loss / len(train_loader)
            history['train_loss'].append(avg_loss)

            # Validation
            val_f1 = self.evaluate(val_loader)
            history['val_f1'].append(val_f1)

            print(f"Epoch {epoch+1}: Loss={avg_loss:.4f}, Val F1={val_f1:.4f}"
    )

            if val_f1 > best_f1:
                best_f1 = val_f1
                torch.save(self.model.state_dict(), 'best_model.pt')

        return history

    def evaluate(self, dataloader):
        """Evaluate model on validation data."""
        self.model.eval()
        all_preds, all_labels = [], []

        with torch.no_grad():
            for batch in dataloader:
                batch = {k: v.to(self.device) for k, v in batch.items()}

                outputs = self.model(
                    input_ids=batch['input_ids'],
                    attention_mask=batch['attention_mask']
                )

                preds = outputs.logits.argmax(dim=-1)
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(batch['labels'].cpu().numpy())

        return f1_score(all_labels, all_preds, average='macro')
```

1.7 Evaluation and Calibration

1.7.1 Confidence Calibration

Model confidence should match empirical accuracy. Measure calibration via Expected Calibration Error (ECE):

ECE = sum_{m=1}^{M} (n_m / N) |accuracy(B_m) - confidence(B_m)|

where B_m is the m-th bin of predictions grouped by confidence. Well-calibrated models have ECE $<$ 0.05.

Temperature Scaling calibrates post-hoc:

P(y|x) = softmax(logits / T)

where temperature T $>$ 1 softens predictions, T $<$ 1 sharpens. Optimize T on validation set to minimize NLL.

```
class TemperatureScaler(nn.Module):
    """Temperature scaling for calibration."""

    def __init__(self):
        super().__init__()
        self.temperature = nn.Parameter(torch.ones(1))

    def calibrate(self, logits, labels, lr=0.01, max_iter=100):
        """Learn optimal temperature."""
        optimizer = torch.optim.LBFGS([self.temperature], lr=lr, max_iter=
    max_iter)

        def closure():
            optimizer.zero_grad()
            scaled = logits / self.temperature
            loss = F.cross_entropy(scaled, labels)
            loss.backward()
            return loss

        optimizer.step(closure)
        return self.temperature.item()
```

1.7.2 Statistical Significance Testing

Compare models using paired bootstrap test:

```
def bootstrap_significance(preds_a, preds_b, labels, metric_fn, n_bootstrap
    =1000):
    """Test if model A significantly outperforms model B."""
    n = len(labels)
    score_a = metric_fn(labels, preds_a)
    score_b = metric_fn(labels, preds_b)
    observed_diff = score_a - score_b

    bootstrap_diffs = []
    for _ in range(n_bootstrap):
        indices = np.random.choice(n, n, replace=True)
        boot_a = metric_fn(labels[indices], preds_a[indices])
        boot_b = metric_fn(labels[indices], preds_b[indices])
        bootstrap_diffs.append(boot_a - boot_b)

    # Two-sided p-value
    p_value = np.mean(np.abs(bootstrap_diffs) >= np.abs(observed_diff))
    ci_95 = np.percentile(bootstrap_diffs, [2.5, 97.5])
```

```
    return {
        'score_a': score_a,
        'score_b': score_b,
        'difference': observed_diff,
        'p_value': p_value,
        'significant': p_value < 0.05,
        'ci_95': ci_95
    }
```

1.8 Common Parameters

() *	() *	() *	() *
0.25	5	0.3409	Typical Range
() *	() *	() *	() *
0.25 fine-tuning	5	5e-5	Higher rates cause catastrophic forgetting
() *	() *	() *	() *
0.25 fine-tuning	5	4	More risks overfitting
() *	() *	() *	() *
0.25 fine-tuning	5	32	Limited by GPU memory
() *	() *	() *	() *
0.25 tention	5	16	Many heads capture diverse patterns

() *	() *	() *	() *
0.25	5	50	Typical 0.3409 Range
0.25	5	50	Higher 0.3 for smaller datasets
0.25	5	50	Found per-a-ture 2.5 via val-i-da-tion
0.25	5	50	Per-label 0.5 class op-ti-miza-tion pre-ferred

1.9 Practice Problems

1. **Attention Visualization**: Implement attention weight extraction and visualization to interpret which words influence aspect sentiment predictions most strongly.
2. **Cross-Domain Transfer**: Fine-tune a sentiment model on product reviews and evaluate on social media data. Measure domain shift via distribution divergence.
3. **Calibration Analysis**: Compute ECE for a pre-trained sentiment model, apply temperature scaling, and verify improved calibration without accuracy loss.
4. **Multi-Task ABSA**: Implement joint aspect extraction and sentiment classification with shared encoder. Compare against separate models.
5. **Uncertainty Quantification**: Add MC Dropout to estimate prediction uncertainty. Flag samples where epistemic uncertainty exceeds threshold for human review.

1.10 References

1. Devlin, J., et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." NAACL-HLT.
2. Vaswani, A., et al. (2017). "Attention Is All You Need." NeurIPS.
3. Demszky, D., et al. (2020). "GoEmotions: A Dataset of Fine-Grained Emotions." ACL.

4. Pontiki, M., et al. (2016). "SemEval-2016 Task 5: Aspect Based Sentiment Analysis." SemEval.
5. Guo, C., et al. (2017). "On Calibration of Modern Neural Networks." ICML.
6. Sun, C., et al. (2019). "How to Fine-Tune BERT for Text Classification." CCL.
7. Tang, D., et al. (2016). "Aspect Level Sentiment Classification with Deep Memory Network." EMNLP.

Sentiment analysis bridges linguistics and machine learning. Theoretical understanding of attention mechanisms, pre-training dynamics, and calibration enables building systems that not only predict accurately but quantify uncertainty reliably.