NLP Sentiment Analysis - Intermediate Handout

Machine Learning for Smarter Innovation

1 NLP Sentiment Analysis - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import re
import torch
from transformers import (
    pipeline,
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer
)
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.metrics import (
    accuracy_score,
    precision_recall_fscore_support,
    confusion_matrix,
    classification_report
)
from torch.utils.data import Dataset, DataLoader
import warnings
warnings.filterwarnings('ignore')

# Check GPU availability
device = 0 if torch.cuda.is_available() else -1
print(f"Using device: {'GPU' if device == 0 else 'CPU'}")
```

This handout covers practical implementation of sentiment analysis using transformer-based models like BERT. Sentiment analysis automatically determines whether text expresses positive, negative, or neutral opinions. Modern approaches using pre-trained language models achieve state-of-the-art performance and can be fine-tuned for specific domains with relatively small amounts of labeled data.

1.2 1. Understanding Transformer-Based Sentiment Analysis

1.2.1 Concept Overview

Traditional sentiment analysis used bag-of-words or simple word embedding approaches. These methods struggled with context, negation, and nuanced language. BERT (Bidirectional Encoder Representations from Transformers) revolutionized NLP by reading text bidirectionally, understanding context from both directions simultaneously. This enables much more nuanced sentiment detection.

1.2.2 Key Transformer Concepts

```
# Demonstrate why bidirectional understanding matters
examples = [
    "The product is not bad",           # Double negation = positive
    "I don't hate it",                  # Litotes = mild positive
    "Best disaster I've ever bought",   # Sarcasm = negative
    "The service was anything but helpful"  # Complex negation = negative
]

# Traditional approach would fail on these
# BERT captures the full context
```

1.2.3 Quick Start with Pipelines

```
def quick_sentiment_demo():
    """
    Fastest way to get started with BERT sentiment analysis.
    Uses Hugging Face pipelines for simple inference.
    """
    # Load pre-trained sentiment pipeline
    classifier = pipeline(
        "sentiment-analysis",
        model="cardiffnlp/twitter-roberta-base-sentiment-latest",
        device=device
    )

    # Test on various examples
    test_texts = [
        "This product exceeded all my expectations!",
        "Terrible customer service, never buying again.",
        "It works as described, nothing special.",
        "The quality is not what I expected, but the price was right."
    ]

    results = classifier(test_texts)
    for text, result in zip(test_texts, results):
        print(f"Text: {text[:50]}...")
        print(f"  Sentiment: {result['label']}, Score: {result['score']:.3f}\n
    ")

    return classifier

classifier = quick_sentiment_demo()
```

1.3 2. Working with Pre-trained Models

1.3.1 Available Models for Different Domains

```
# Pre-trained models optimized for different use cases
SENTIMENT_MODELS = {
    'general': 'cardiffnlp/twitter-roberta-base-sentiment-latest',
    'finance': 'ProsusAI/finbert',
    'multilingual': 'nlptown/bert-base-multilingual-uncased-sentiment',
    'emotion': 'j-hartmann/emotion-english-distilroberta-base',
    'product_reviews': 'nlptown/bert-base-multilingual-uncased-sentiment',
    'social_media': 'cardiffnlp/twitter-roberta-base-sentiment'
}

def load_domain_classifier(domain='general'):
    """
    Load a domain-specific sentiment classifier.
    """
    model_name = SENTIMENT_MODELS.get(domain, SENTIMENT_MODELS['general'])
    print(f"Loading model: {model_name}")

    classifier = pipeline(
        "sentiment-analysis",
        model=model_name,
        device=device
    )
    return classifier

# Example: Finance-specific sentiment
finance_classifier = load_domain_classifier('finance')
finance_texts = [
    "Stock prices soared after earnings beat expectations",
    "The company announced major layoffs and restructuring",
    "Revenue remained flat compared to last quarter"
]

for text in finance_texts:
    result = finance_classifier(text)[0]
    print(f"{text[:40]}... -> {result['label']} ({result['score']:.3f})")
```

1.3.2 Custom Model Loading for More Control

```
class SentimentAnalyzer:
    """
    Wrapper class for more control over sentiment analysis.
    """

    def __init__(self, model_name="bert-base-uncased", num_labels=3):
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model = AutoModelForSequenceClassification.from_pretrained(
            model_name,
            num_labels=num_labels
        )
        self.model.eval()
        self.labels = ['negative', 'neutral', 'positive']

    def predict(self, text, return_all_scores=False):
        """
        Predict sentiment for a single text.
        """
```

```
        inputs = self.tokenizer(
            text,
            return_tensors="pt",
            truncation=True,
            padding=True,
            max_length=512
        )

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

        scores = probs.numpy()[0]

        if return_all_scores:
            return {label: float(score) for label, score in zip(self.labels,
    scores)}

        predicted_idx = scores.argmax()
        return {
            'label': self.labels[predicted_idx],
            'score': float(scores[predicted_idx])
        }

    def predict_batch(self, texts, batch_size=32):
        """
        Efficient batch prediction.
        """
        results = []
        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]
            inputs = self.tokenizer(
                batch,
                return_tensors="pt",
                truncation=True,
                padding=True,
                max_length=512
            )

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

            for prob in probs:
                scores = prob.numpy()
                predicted_idx = scores.argmax()
                results.append({
                    'label': self.labels[predicted_idx],
                    'score': float(scores[predicted_idx])
                })

        return results

# Usage example
# analyzer = SentimentAnalyzer()
# result = analyzer.predict("Great product!", return_all_scores=True)
```

1.4 3. Text Preprocessing for Sentiment Analysis

1.4.1 Concept Overview

While BERT handles most preprocessing internally, cleaning text appropriately can improve results, especially for social media and user-generated content where noise is common.

1.4.2 Preprocessing Pipeline

```
def preprocess_text(text, remove_urls=True, remove_mentions=True,
                    remove_hashtags=False, lowercase=True):
    """
    Preprocess text for sentiment analysis.
    BERT is case-sensitive for some models, so lowercase is optional.
    """
    if text is None or pd.isna(text):
        return ""

    text = str(text)

    # Remove URLs
    if remove_urls:
        text = re.sub(r'http\S+|www\.\S+', '', text)

    # Remove mentions
    if remove_mentions:
        text = re.sub(r'@\w+', '', text)

    # Remove or replace hashtags
    if remove_hashtags:
        text = re.sub(r'#\w+', '', text)
    else:
        # Keep hashtag content, remove # symbol
        text = re.sub(r'#(\w+)', r'\1', text)

    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text)

    # Optionally lowercase
    if lowercase:
        text = text.lower()

    return text.strip()

def preprocess_dataframe(df, text_column, **kwargs):
    """
    Apply preprocessing to a DataFrame column.
    """
    df = df.copy()
    df[f'{text_column}_clean'] = df[text_column].apply(
        lambda x: preprocess_text(x, **kwargs)
    )

    # Report preprocessing stats
    original_lens = df[text_column].str.len()
    clean_lens = df[f'{text_column}_clean'].str.len()

    print(f"Preprocessing complete:")
    print(f"  Original avg length: {original_lens.mean():.1f} chars")
    print(f"  Cleaned avg length: {clean_lens.mean():.1f} chars")
    print(f"  Empty after cleaning: {(clean_lens == 0).sum()}")
```

```
    return df

# Example usage
sample_df = pd.DataFrame({
    'text': [
        "@user Check out this product! https://example.com #amazing",
        "Worst experience ever!!!              ",
        "It's okay, nothing special but works fine.",
    ]
})
processed_df = preprocess_dataframe(sample_df, 'text')
print(processed_df[['text', 'text_clean']])
```

1.4.3 Handling Long Texts

```
def analyze_long_text(text, classifier, max_length=512, overlap=100):
    """
    Handle texts longer than BERT's max length using sliding window.
    """
    tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
    tokens = tokenizer.tokenize(text)

    if len(tokens) <= max_length - 2:  # Account for [CLS] and [SEP]
        return classifier(text)[0]

    # Split into overlapping chunks
    chunk_size = max_length - 2 - overlap
    chunks = []
    chunk_texts = []

    for i in range(0, len(tokens), chunk_size):
        chunk = tokens[i:i + max_length - 2]
        chunk_text = tokenizer.convert_tokens_to_string(chunk)
        chunk_texts.append(chunk_text)

    # Analyze each chunk
    results = classifier(chunk_texts)

    # Aggregate scores (weighted by confidence)
    total_weight = 0
    weighted_scores = {'positive': 0, 'negative': 0, 'neutral': 0}

    for result in results:
        weight = result['score']
        label = result['label'].lower()
        if label in weighted_scores:
            weighted_scores[label] += weight
            total_weight += weight

    # Normalize
    for label in weighted_scores:
        weighted_scores[label] /= total_weight

    # Return dominant sentiment
    dominant = max(weighted_scores, key=weighted_scores.get)
    return {
        'label': dominant,
        'score': weighted_scores[dominant],
        'all_scores': weighted_scores,
        'n_chunks': len(chunk_texts)
    }
```

1.5 4. Fine-tuning BERT for Custom Domains

1.5.1 Creating a Custom Dataset

```
class SentimentDataset(Dataset):
    """
    PyTorch Dataset for sentiment analysis fine-tuning.
    """

    def __init__(self, texts, labels, tokenizer, max_length=128):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

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

    def __getitem__(self, idx):
        text = str(self.texts[idx])
        label = self.labels[idx]

        encoding = self.tokenizer(
            text,
            truncation=True,
            padding='max_length',
            max_length=self.max_length,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'labels': torch.tensor(label, dtype=torch.long)
        }

def create_datasets(df, text_col, label_col, tokenizer, test_size=0.2):
    """
    Create train and validation datasets from DataFrame.
    """
    X_train, X_val, y_train, y_val = train_test_split(
        df[text_col].tolist(),
        df[label_col].tolist(),
        test_size=test_size,
        random_state=42,
        stratify=df[label_col]
    )

    train_dataset = SentimentDataset(X_train, y_train, tokenizer)
    val_dataset = SentimentDataset(X_val, y_val, tokenizer)

    print(f"Training samples: {len(train_dataset)}")
    print(f"Validation samples: {len(val_dataset)}")
    print(f"Label distribution (train): {pd.Series(y_train).value_counts().
    to_dict()}")

    return train_dataset, val_dataset
```

1.5.2 Fine-tuning Process

```
def fine_tune_sentiment_model(train_dataset, val_dataset, model_name="bert-
    base-uncased",
                              num_labels=3, epochs=3, batch_size=16):
    """
    Fine-tune a BERT model for sentiment analysis.
    """
    # Load model
    model = AutoModelForSequenceClassification.from_pretrained(
        model_name,
        num_labels=num_labels
    )

    # Training arguments
    training_args = TrainingArguments(
        output_dir='./sentiment_model',
        num_train_epochs=epochs,
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size * 2,
        warmup_steps=500,
        weight_decay=0.01,
        logging_dir='./logs',
        logging_steps=100,
        eval_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=True,
        metric_for_best_model="eval_loss",
    )

    # Custom metrics computation
    def compute_metrics(eval_pred):
        predictions, labels = eval_pred
        predictions = np.argmax(predictions, axis=1)

        accuracy = accuracy_score(labels, predictions)
        precision, recall, f1, _ = precision_recall_fscore_support(
            labels, predictions, average='weighted'
        )

        return {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1': f1
        }

    # Create trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        compute_metrics=compute_metrics
    )

    # Train
    print("Starting fine-tuning...")
    trainer.train()

    # Evaluate
    eval_results = trainer.evaluate()
    print(f"\nFinal evaluation results:")
```

```
    for key, value in eval_results.items():
        print(f"  {key}: {value:.4f}")

    return trainer.model, trainer

# Example usage (with prepared data):
# model, trainer = fine_tune_sentiment_model(train_dataset, val_dataset)
```

1.6 5. Model Evaluation and Validation

1.6.1 Comprehensive Evaluation

```
def evaluate_sentiment_model(model, tokenizer, test_texts, test_labels,
                             label_names=['negative', 'neutral', 'positive']):
    """
    Comprehensive evaluation of sentiment model.
    """
    predictions = []

    model.eval()
    for text in test_texts:
        inputs = tokenizer(
            text,
            return_tensors="pt",
            truncation=True,
            padding=True,
            max_length=512
        )

        with torch.no_grad():
            outputs = model(**inputs)
            pred = torch.argmax(outputs.logits, dim=-1)
            predictions.append(pred.item())

    # Calculate metrics
    print("Classification Report:")
    print(classification_report(test_labels, predictions, target_names=
    label_names))

    # Confusion matrix
    cm = confusion_matrix(test_labels, predictions)
    print("\nConfusion Matrix:")
    print(pd.DataFrame(cm, index=label_names, columns=label_names))

    # Per-class accuracy
    print("\nPer-class accuracy:")
    for i, label in enumerate(label_names):
        mask = np.array(test_labels) == i
        if mask.sum() > 0:
            class_acc = (np.array(predictions)[mask] == i).mean()
            print(f"  {label}: {class_acc:.3f}")

    return predictions, cm
```

1.6.2 Cross-Validation for Robust Evaluation

```
def cross_validate_sentiment(texts, labels, n_splits=5):
    """
    Cross-validation for sentiment analysis.
    Uses pre-trained model for efficiency (fine-tuning each fold is slow).
    """
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)
    classifier = pipeline("sentiment-analysis", device=device)

    all_scores = {'accuracy': [], 'precision': [], 'recall': [], 'f1': []}

    for fold, (train_idx, val_idx) in enumerate(skf.split(texts, labels)):
        print(f"\nFold {fold + 1}/{n_splits}")

        X_val = [texts[i] for i in val_idx]
        y_val = [labels[i] for i in val_idx]

        # Predict
        predictions = []
        for text in X_val:
            result = classifier(text)[0]
            # Map to label indices (adjust based on model output)
            if 'positive' in result['label'].lower():
                pred = 2
            elif 'negative' in result['label'].lower():
                pred = 0
            else:
                pred = 1
            predictions.append(pred)

        # Calculate metrics
        acc = accuracy_score(y_val, predictions)
        prec, rec, f1, _ = precision_recall_fscore_support(y_val, predictions,
    average='weighted')

        all_scores['accuracy'].append(acc)
        all_scores['precision'].append(prec)
        all_scores['recall'].append(rec)
        all_scores['f1'].append(f1)

        print(f"  Accuracy: {acc:.3f}, F1: {f1:.3f}")

    # Summary
    print("\n" + "="*50)
    print("Cross-Validation Summary:")
    for metric, scores in all_scores.items():
        print(f"  {metric}: {np.mean(scores):.3f} (+/- {np.std(scores):.3f})")

    return all_scores
```

1.7 6. Production Deployment

1.7.1 Batch Processing Pipeline

```
class SentimentPipeline:
    """
    Production-ready sentiment analysis pipeline.
    """
```

```
    def __init__(self, model_name='cardiffnlp/twitter-roberta-base-sentiment-
latest',
                 confidence_threshold=0.7):
        self.classifier = pipeline("sentiment-analysis", model=model_name,
device=device)
        self.confidence_threshold = confidence_threshold

    def analyze_batch(self, texts, batch_size=32):
        """
        Process texts in batches for efficiency.
        """
        results = []
        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]
            batch_results = self.classifier(batch)
            results.extend(batch_results)
        return results

    def analyze_with_confidence(self, texts):
        """
        Separate high-confidence from low-confidence predictions.
        """
        results = self.analyze_batch(texts)

        high_confidence = []
        needs_review = []

        for text, result in zip(texts, results):
            item = {
                'text': text,
                'sentiment': result['label'],
                'confidence': result['score']
            }

            if result['score'] >= self.confidence_threshold:
                high_confidence.append(item)
            else:
                needs_review.append(item)

        return high_confidence, needs_review

    def aggregate_sentiment(self, texts):
        """
        Aggregate sentiment across multiple texts.
        """
        results = self.analyze_batch(texts)

        sentiment_counts = {'POSITIVE': 0, 'NEGATIVE': 0, 'NEUTRAL': 0}
        total_confidence = {'POSITIVE': 0, 'NEGATIVE': 0, 'NEUTRAL': 0}

        for result in results:
            label = result['label'].upper()
            if label in sentiment_counts:
                sentiment_counts[label] += 1
                total_confidence[label] += result['score']

        total = len(results)
        summary = {
            'total_texts': total,
            'sentiment_distribution': {k: v/total for k, v in sentiment_counts
.items()},
            'average_confidence': {k: v/max(sentiment_counts[k], 1) for k, v
in total_confidence.items()},
```

```
            'dominant_sentiment': max(sentiment_counts, key=sentiment_counts.
    get)
        }

        return summary

# Usage
pipeline_obj = SentimentPipeline()
# high_conf, low_conf = pipeline_obj.analyze_with_confidence(texts)
# summary = pipeline_obj.aggregate_sentiment(texts)
```

1.8 Common Hyperparameters

Parameter	Typical Value	Description
max_length	512	Maximum token sequence length
batch_size	32	Training batch size (reduce if OOM)
learning_rate	2e-5 to 5e-5	Fine-tuning learning rate
epochs	3-4	Number of training epochs

() *	() *	() *	
0.2821	0.3333	Typical	Description
	0.3846	Value	
() *	() *	() *	
0.2821	0.3333	1000	Learning rate warmup steps
() *	() *	() *	
0.2821	0.3333	0.01	L2 regularization
() *	() *	() *	
0.2821	0.3333	0.9	Minimum confidence threshold for auto-classification

---

1.9 Practice Projects

1. **Product Review Analyzer**: Build a system that analyzes e-commerce reviews, extracts sentiment, identifies common complaints, and generates a summary report with recommendations.
2. **Social Media Monitor**: Create a pipeline that monitors brand mentions on social media, classifies sentiment in real-time, and alerts on sudden negative sentiment spikes.
3. **Customer Feedback Classifier**: Fine-tune a model on customer support tickets, classify by sentiment and urgency, and route to appropriate teams.
4. **Comparative Review Analysis**: Analyze product reviews for multiple competitors, generate comparative sentiment analysis, and identify competitive advantages/disadvantages.

1.10 Troubleshooting

Issue	Cause	Solution
Out of memory (OOM)	Batch size too large	Reduce batch_size, use gradient accumulation
Slow inference	CPU processing	Use GPU (device=0) or batch processing
Poor domain performance	Model not domain-specific	Fine-tune on domain-specific data or use domain model
Inconsistent predictions	Model uncertainty	Use confidence thresholding, ensemble models

[illegible]	[illegible]	Solution
Long text handling	BERT max 512 tokens	Use sliding window or summarization
Multilingual text	English-only model	Use multilingual model variant
Sarcasm misclassification	Sarcasm is hard for all models	Add sarcasm examples to training data
Neutral over-prediction	Class imbalance	Balance training data, adjust thresholds

---

1.11 Next Steps

- Read the advanced handout for aspect-based sentiment and emotion detection
- Experiment with different pre-trained models for your domain
- Build labeled datasets for fine-tuning
- Implement monitoring for model drift in production

- Explore multi-task learning combining sentiment with other NLP tasks

---

Sentiment analysis reveals customer voice at scale. The goal is not perfect classification but actionable insights. Combine model predictions with human review for high-stakes decisions.