Text emotion detection.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

RAM Disk

Share

[ ] Show code

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 404.7/404.7 kB 7.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.1/9.1 MB 58.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 90.5 MB/s eta 0:00:00
added 22 packages in 3s
3 packages are looking for funding
  run `npm fund` for details
✅ All dependencies installed successfully.
```

[1] ✓ 10m

```
# ================================================================================
# EMOTION DETECTION FROM TEXTUAL COMMENTS - COMPLETE SYSTEM
# ================================================================================
# Project: Automated Detection of Different Emotions from Textual Comments
# Architecture: Transformer-based Deep Learning Model
# Deployment: Google Colab with Streamlit Frontend
# ================================================================================

# ================================================================================
# CELL 1: ENVIRONMENT SETUP & INSTALLATIONS
# ================================================================================

# Run this cell first to install all required packages
"""
!pip install -q transformers==4.36.0 datasets==2.16.0 torch==2.1.0 accelerate==0.25.0
!pip install -q scikit-learn==1.3.2 pandas==2.1.4 numpy==1.26.2 matplotlib==3.8.2 seaborn==0.13.0
!pip install -q optuna==3.5.0
!pip install -q streamlit==1.29.0 pyngrok==7.0.5
!pip install -q blinker==1.7.0
"""

# ================================================================================
# CELL 2: IMPORT LIBRARIES
# ================================================================================

import os
import json
import re
import warnings
from datetime import datetime
from typing import Dict, List, Tuple, Any, Optional

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import torch
from torch.utils.data import Dataset, DataLoader
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer,
    pipeline
)
from datasets import load_dataset, Dataset as HFDataset
from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    accuracy_score,
    precision_recall_fscore_support,
    confusion_matrix,
    classification_report
)

warnings.filterwarnings('ignore')

print("✓ All libraries imported successfully!")

# ================================================================================
# CELL 3: SYSTEM CONFIGURATION
# ================================================================================

class SystemConfig:
    """Central configuration for the entire system"""

    # Model Configuration
    MODEL_NAME = "distilbert-base-uncased"
    MAX_LENGTH = 128
    BATCH_SIZE = 16
    LEARNING_RATE = 2e-5
    NUM_EPOCHS = 3
    WARMUP_STEPS = 500
    WEIGHT_DECAY = 0.01

    # Emotion Labels
    EMOTIONS = ['sadness', 'joy', 'love', 'anger', 'fear', 'surprise']
    NUM_LABELS = len(EMOTIONS)

    # Paths
    MODEL_DIR = "./emotion_model"
    LOGS_DIR = "./logs"
    RESULTS_DIR = "./results"

    # Data Configuration
    TEST_SIZE = 0.2
    VAL_SIZE = 0.1
    RANDOM_SEED = 42

    # Device Configuration
    DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

    @classmethod
    def setup_directories(cls):
        """Create necessary directories"""
        for dir_path in [cls.MODEL_DIR, cls.LOGS_DIR, cls.RESULTS_DIR]:
```

```
            os.makedirs(dir_path, exist_ok=True)

config = SystemConfig()
config.setup_directories()

print(f"System Configuration Loaded")
print(f"Device: {config.DEVICE}")
print(f"Model: {config.MODEL_NAME}")
print(f"Emotions: {config.EMOTIONS}")

# ============================================================================
# CELL 4: DATA PREPROCESSING
# ============================================================================

class DataPreprocessor:
    """Handles all data preprocessing tasks"""

    def __init__(self):
        self.slang_dict = {
            'nxt': 'next', '2mrw': 'tomorrow', 'r u': 'are you',
            'gud': 'good', 'u': 'you', 'ur': 'your', 'thx': 'thanks',
            'plz': 'please', 'w8': 'wait', 'gr8': 'great',
            'b4': 'before', '2day': 'today', 'btw': 'by the way',
            'idk': "i don't know", 'omg': 'oh my god',
            'lol': 'laughing out loud', 'brb': 'be right back',
        }

    def clean_text(self, text: str) -> str:
        """Clean and normalize text"""
        if not isinstance(text, str):
            return ""

        text = text.lower()
        text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)
        text = re.sub(r'@\w+', '', text)
        text = re.sub(r'#', '', text)

        for slang, full in self.slang_dict.items():
            text = re.sub(r'\b' + slang + r'\b', full, text)

        text = ' '.join(text.split())
        return text.strip()

class DataManager:
    """Manages data loading, preparation and splitting"""

    def __init__(self, preprocessor: DataPreprocessor):
        self.preprocessor = preprocessor
        self.train_data = None
        self.val_data = None
        self.test_data = None

    def load_emotion_dataset(self):
        """Load emotion dataset from Hugging Face"""
        print("Loading emotion dataset...")
        dataset = load_dataset("emotion")

        train_df = pd.DataFrame(dataset['train'])
        test_df = pd.DataFrame(dataset['test'])
        val_df = pd.DataFrame(dataset['validation'])

        df = pd.concat([train_df, val_df, test_df], ignore_index=True)
        print(f"Dataset loaded: {len(df)} samples")
        return df

    def prepare_data(self) -> Dict:
        """Prepare and split data"""
        print("\n" + "="*70)
        print("STEP 1: DATA PREPARATION")
        print("="*70)

        df = self.load_emotion_dataset()

        print("Cleaning text data...")
        df['text'] = df['text'].apply(self.preprocessor.clean_text)
        df = df[df['text'].str.len() > 0].reset_index(drop=True)

        print("Splitting dataset...")
        train_val_df, test_df = train_test_split(
            df, test_size=config.TEST_SIZE,
            random_state=config.RANDOM_SEED,
            stratify=df['label']
        )

        train_df, val_df = train_test_split(
            train_val_df, test_size=config.VAL_SIZE/(1-config.TEST_SIZE),
            random_state=config.RANDOM_SEED,
            stratify=train_val_df['label']
        )

        self.train_data = train_df.reset_index(drop=True)
        self.val_data = val_df.reset_index(drop=True)
        self.test_data = test_df.reset_index(drop=True)

        print(f"\nData Split Summary:")
        print(f"  Training samples: {len(train_df)}")
        print(f"  Validation samples: {len(val_df)}")
        print(f"  Test samples: {len(test_df)}")

        print("\nEmotion Distribution (Training Set):")
        for i, emotion in enumerate(config.EMOTIONS):
            count = (train_df['label'] == i).sum()
            print(f"  {emotion}: {count} ({count/len(train_df)*100:.1f}%)")

        return {
            'train': self.train_data,
            'val': self.val_data,
            'test': self.test_data
        }

# ============================================================================
# CELL 5: MODEL ARCHITECTURE
# ============================================================================

class EmotionDataset(Dataset):
    """Custom Dataset for emotion classification"""

    def __init__(self, texts: List[str], labels: List[int], tokenizer, max_length: int):
```

```
            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,
                add_special_tokens=True,
                max_length=self.max_length,
                padding='max_length',
                truncation=True,
                return_attention_mask=True,
                return_tensors='pt'
            )

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

    class ModelBuilder:
        """Builds and configures the emotion detection model"""

        def __init__(self):
            self.tokenizer = None
            self.model = None

        def build_model(self):
            """Build transformer model"""
            print("\n" + "="*70)
            print("STEP 2: MODEL DEVELOPMENT")
            print("="*70)

            print(f"Loading tokenizer and model: {config.MODEL_NAME}")

            self.tokenizer = AutoTokenizer.from_pretrained(config.MODEL_NAME)
            self.model = AutoModelForSequenceClassification.from_pretrained(
                config.MODEL_NAME,
                num_labels=config.NUM_LABELS,
                problem_type="single_label_classification"
            )

            self.model.to(config.DEVICE)

            print(f"Model loaded successfully on {config.DEVICE}")
            print(f"Total parameters: {sum(p.numel() for p in self.model.parameters()):,}")
            print(f"Trainable parameters: {sum(p.numel() for p in self.model.parameters() if p.requires_grad):,}")

            return self.model, self.tokenizer

    # ============================================================================
    # CELL 6: TRAINING MODULE
    # ============================================================================

    class ModelTrainer:
        """Handles model training and validation"""

        def __init__(self, model, tokenizer):
            self.model = model
            self.tokenizer = tokenizer
            self.trainer = None

        def compute_metrics(self, eval_pred):
            """Compute evaluation metrics"""
            predictions, labels = eval_pred
            predictions = np.argmax(predictions, axis=1)

            precision, recall, f1, _ = precision_recall_fscore_support(
                labels, predictions, average='weighted', zero_division=0
            )
            acc = accuracy_score(labels, predictions)

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

        def train_model(self, train_data, val_data):
            """Train the model"""
            print("\n" + "="*70)
            print("STEP 3: TRAINING AND VALIDATION")
            print("="*70)

            print("Preparing datasets...")
            train_dataset = EmotionDataset(
                train_data['text'].tolist(),
                train_data['label'].tolist(),
                self.tokenizer,
                config.MAX_LENGTH
            )

            val_dataset = EmotionDataset(
                val_data['text'].tolist(),
                val_data['label'].tolist(),
                self.tokenizer,
                config.MAX_LENGTH
            )

            training_args = TrainingArguments(
                output_dir=config.MODEL_DIR,
                num_train_epochs=config.NUM_EPOCHS,
                per_device_train_batch_size=config.BATCH_SIZE,
                per_device_eval_batch_size=config.BATCH_SIZE,
                warmup_steps=config.WARMUP_STEPS,
                weight_decay=config.WEIGHT_DECAY,
                learning_rate=config.LEARNING_RATE,
                logging_dir=config.LOGS_DIR,
                logging_steps=100,
```

```
            eval_strategy="epoch",
            save_strategy="epoch",
            load_best_model_at_end=True,
            metric_for_best_model="f1",
            save_total_limit=2,
            report_to="none"
        )

        self.trainer = Trainer(
            model=self.model,
            args=training_args,
            train_dataset=train_dataset,
            eval_dataset=val_dataset,
            compute_metrics=self.compute_metrics
        )

        print("\nStarting training...")
        print(f"Total training samples: {len(train_dataset)}")
        print(f"Total validation samples: {len(val_dataset)}")
        print(f"Batch size: {config.BATCH_SIZE}")
        print(f"Number of epochs: {config.NUM_EPOCHS}")

        train_result = self.trainer.train()

        print("\nSaving model...")
        self.trainer.save_model(config.MODEL_DIR)
        self.tokenizer.save_pretrained(config.MODEL_DIR)

        print("Training completed successfully!")
        return train_result

# ============================================================================
# CELL 7: EVALUATION MODULE
# ============================================================================

class ModelEvaluator:
    """Comprehensive model evaluation"""

    def __init__(self, model, tokenizer):
        self.model = model
        self.tokenizer = tokenizer
        self.pipeline = None
        self.results = {}

    def create_pipeline(self):
        """Create inference pipeline"""
        self.pipeline = pipeline(
            "text-classification",
            model=self.model,
            tokenizer=self.tokenizer,
            device=0 if config.DEVICE == "cuda" else -1
        )

    def evaluate_model(self, test_data):
        """Comprehensive model evaluation"""
        print("\n" + "="*70)
        print("STEP 4: MODEL TESTING AND EVALUATION")
        print("="*70)

        if self.pipeline is None:
            self.create_pipeline()

        print("Running predictions on test set...")
        texts = test_data['text'].tolist()
        true_labels = test_data['label'].tolist()

        predictions = []
        confidences = []

        for text in texts:
            result = self.pipeline(text, top_k=1)[0]
            pred_label = int(result['label'].split('_')[1])
            predictions.append(pred_label)
            confidences.append(result['score'])

        accuracy = accuracy_score(true_labels, predictions)
        precision, recall, f1, _ = precision_recall_fscore_support(
            true_labels, predictions, average='weighted', zero_division=0
        )

        print(f"\nOverall Performance Metrics:")
        print(f"  Accuracy: {accuracy:.4f}")
        print(f"  Precision: {precision:.4f}")
        print(f"  Recall: {recall:.4f}")
        print(f"  F1-Score: {f1:.4f}")
        print(f"  Average Confidence: {np.mean(confidences):.4f}")

        print("\nPer-Emotion Performance:")
        report = classification_report(
            true_labels, predictions,
            target_names=config.EMOTIONS,
            digits=4,
            zero_division=0
        )
        print(report)

        cm = confusion_matrix(true_labels, predictions)
        self._plot_confusion_matrix(cm)

        self.results = {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1': f1,
            'confusion_matrix': cm.tolist(),
            'predictions': predictions,
            'confidences': confidences
        }

        return self.results

    def _plot_confusion_matrix(self, cm):
        """Plot confusion matrix"""
        plt.figure(figsize=(10, 8))
        sns.heatmap(
            cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=config.EMOTIONS,
            yticklabels=config.EMOTIONS
        )
```

```
        )
        plt.title('Confusion Matrix - Emotion Detection', fontsize=14, fontweight='bold')
        plt.ylabel('True Emotion', fontsize=12)
        plt.xlabel('Predicted Emotion', fontsize=12)
        plt.tight_layout()

        plot_path = os.path.join(config.RESULTS_DIR, 'confusion_matrix.png')
        plt.savefig(plot_path, dpi=300, bbox_inches='tight')
        print(f"\nConfusion matrix saved to: {plot_path}")
        plt.show()

    def test_scenarios(self):
        """Test model on various scenarios"""
        print("\n" + "="*70)
        print("TESTING SAMPLE SCENARIOS")
        print("="*70)

        test_cases = [
            ("I'm not happy at all", "Negation"),
            ("Yeah, that's just great... not!", "Sarcasm"),
            ("r u gud? thx 4 asking", "Slang/Abbreviations"),
            ("I love this but I'm also scared", "Mixed emotions"),
            ("This is the best worst day ever", "Contradictory"),
            ("Can't wait to not do this again", "Double negation"),
        ]

        print("\nSample Test Results:")
        print("-" * 70)

        for text, scenario in test_cases:
            result = self.pipeline(text, top_k=1)[0]
            emotion_idx = int(result['label'].split('_')[1])
            emotion = config.EMOTIONS[emotion_idx]
            confidence = result['score']

            print(f"\nScenario: {scenario}")
            print(f"Input: '{text}'")
            print(f"Predicted: {emotion} (Confidence: {confidence:.4f})")

# ============================================================================
# CELL 8: MAIN EXECUTION PIPELINE
# ============================================================================

def main():
    """Main execution pipeline"""

    print("\n" + "="*70)
    print("EMOTION DETECTION SYSTEM - COMPLETE PIPELINE")
    print("="*70)
    print(f"Start Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

    # Step 1: Data Preparation
    preprocessor = DataPreprocessor()
    data_manager = DataManager(preprocessor)
    data_splits = data_manager.prepare_data()

    # Step 2: Model Development
    model_builder = ModelBuilder()
    model, tokenizer = model_builder.build_model()

    # Step 3: Training & Validation
    trainer = ModelTrainer(model, tokenizer)
    train_result = trainer.train_model(
        data_splits['train'],
        data_splits['val']
    )

    # Step 4: Model Testing & Evaluation
    evaluator = ModelEvaluator(model, tokenizer)
    test_results = evaluator.evaluate_model(data_splits['test'])
    evaluator.test_scenarios()

    # Step 5: Save results
    results_summary = {
        'timestamp': datetime.now().isoformat(),
        'model': config.MODEL_NAME,
        'test_accuracy': float(test_results['accuracy']),
        'test_f1': float(test_results['f1']),
        'test_precision': float(test_results['precision']),
        'test_recall': float(test_results['recall']),
        'emotions': config.EMOTIONS
    }

    results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
    with open(results_path, 'w') as f:
        json.dump(results_summary, f, indent=2)

    print(f"\n{'='*70}")
    print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
    print(f"{'='*70}")
    print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"\nResults saved to: {config.RESULTS_DIR}")
    print(f"Model saved to: {config.MODEL_DIR}")

    return test_results
    evaluator.test_scenarios()

    # Step 5: Save results
    results_summary = {
        'timestamp': datetime.now().isoformat(),
        'model': config.MODEL_NAME,
        'test_accuracy': float(test_results['accuracy']),
        'test_f1': float(test_results['f1']),
        'test_precision': float(test_results['precision']),
        'test_recall': float(test_results['recall']),
        'emotions': config.EMOTIONS
    }

    results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
    with open(results_path, 'w') as f:
        json.dump(results_summary, f, indent=2)

    print(f"\n{'='*70}")
    print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
    print(f"{'='*70}")
    print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"\nResults saved to: {config.RESULTS_DIR}")
    print(f"Model saved to: {config.MODEL_DIR}")
```

```python
return test_results
evaluator.test_scenarios()

# Step 5: Save results
results_summary = {
    'timestamp': datetime.now().isoformat(),
    'model': config.MODEL_NAME,
    'test_accuracy': float(test_results['accuracy']),
    'test_f1': float(test_results['f1']),
    'test_precision': float(test_results['precision']),
    'test_recall': float(test_results['recall']),
    'emotions': config.EMOTIONS
}

results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
with open(results_path, 'w') as f:
    json.dump(results_summary, f, indent=2)

print(f"\n{'='*70}")
print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
print(f"{'='*70}")
print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"\nResults saved to: {config.RESULTS_DIR}")
print(f"Model saved to: {config.MODEL_DIR}")

return test_results
evaluator.test_scenarios()

# Step 5: Save results
results_summary = {
    'timestamp': datetime.now().isoformat(),
    'model': config.MODEL_NAME,
    'test_accuracy': float(test_results['accuracy']),
    'test_f1': float(test_results['f1']),
    'test_precision': float(test_results['precision']),
    'test_recall': float(test_results['recall']),
    'emotions': config.EMOTIONS
}

results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
with open(results_path, 'w') as f:
    json.dump(results_summary, f, indent=2)

print(f"\n{'='*70}")
print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
print(f"{'='*70}")
print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"\nResults saved to: {config.RESULTS_DIR}")
print(f"Model saved to: {config.MODEL_DIR}")

return test_results
evaluator.test_scenarios()

# Step 5: Save results
results_summary = {
    'timestamp': datetime.now().isoformat(),
    'model': config.MODEL_NAME,
    'test_accuracy': float(test_results['accuracy']),
    'test_f1': float(test_results['f1']),
    'test_precision': float(test_results['precision']),
    'test_recall': float(test_results['recall']),
    'emotions': config.EMOTIONS
}

results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
with open(results_path, 'w') as f:
    json.dump(results_summary, f, indent=2)

print(f"\n{'='*70}")
print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
print(f"{'='*70}")
print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"\nResults saved to: {config.RESULTS_DIR}")
print(f"Model saved to: {config.MODEL_DIR}")

return test_results
evaluator.test_scenarios()

# Step 5: Save results
results_summary = {
    'timestamp': datetime.now().isoformat(),
    'model': config.MODEL_NAME,
    'test_accuracy': float(test_results['accuracy']),
    'test_f1': float(test_results['f1']),
    'test_precision': float(test_results['precision']),
    'test_recall': float(test_results['recall']),
    'emotions': config.EMOTIONS
}

results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
with open(results_path, 'w') as f:
    json.dump(results_summary, f, indent=2)

print(f"\n{'='*70}")
print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
print(f"{'='*70}")
print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"\nResults saved to: {config.RESULTS_DIR}")
print(f"Model saved to: {config.MODEL_DIR}")

return test_results
evaluator.test_scenarios()

# Step 5: Save results
results_summary = {
    'timestamp': datetime.now().isoformat(),
    'model': config.MODEL_NAME,
    'test_accuracy': float(test_results['accuracy']),
    'test_f1': float(test_results['f1']),
    'test_precision': float(test_results['precision']),
    'test_recall': float(test_results['recall']),
    'emotions': config.EMOTIONS
}

results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
with open(results_path, 'w') as f:
    json.dump(results_summary, f, indent=2)
```

```
    print(f"\n{'='*70}")
    print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
    print(f"{'='*70}")
    print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"\nResults saved to: {config.RESULTS_DIR}")
    print(f"Model saved to: {config.MODEL_DIR}")

    return test_results
    evaluator.test_scenarios()

    # Step 5: Save results
    evaluator.test_scenarios()

    # Step 5: Save results
    results_summary = {
        'timestamp': datetime.now().isoformat(),
        'model': config.MODEL_NAME,
        'test_accuracy': float(test_results['accuracy']),
        'test_f1': float(test_results['f1']),
        'test_precision': float(test_results['precision']),
        'test_recall': float(test_results['recall']),
        'emotions': config.EMOTIONS
    }

    results_path = os.path.join(config.RESULTS_DIR, 'final_results.json')
    with open(results_path, 'w') as f:
        json.dump(results_summary, f, indent=2)

    print(f"\n{'='*70}")
    print("PIPELINE EXECUTION COMPLETED SUCCESSFULLY!")
    print(f"{'='*70}")
    print(f"End Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"\nResults saved to: {config.RESULTS_DIR}")
    print(f"Model saved to: {config.MODEL_DIR}")

    return test_results
```