Medical Data Pipeline

A comprehensive, industry-standard pipeline for processing medical queries, generating ICD codes, and training machine learning models for medical query classification.

Table of Contents

Overview

This pipeline processes medical specialty queries through multiple stages:

1. **Stage 2**: Query generation and classification
2. **Stage 3**: ICD code generation and verification
3. **Stage 4**: Specialty verification
4. **Stage 5**: Triplet dataset creation (preserved as-is)
5. **Hyperparameter Tuning**: Model optimization

Key Features

- **Modular Design**: Each stage is independently executable
- **Robust Error Handling**: Comprehensive retry mechanisms and error collection
- **Configurable**: JSON-based configuration management
- **Distributed Processing**: Support for chunk-based parallel processing
- **Azure Integration**: Full Azure OpenAI and ML Workspace support
- **Comprehensive Logging**: Structured logging with multiple levels
- **Industry Standards**: Following software engineering best practices

Architecture

```
medical-pipeline/
├── config.py                 # Configuration management
├── logging_utils.py          # Standardized logging
├── error_handling.py         # Error handling utilities
├── azure_utils.py            # Azure integration
├── base_processor.py         # Base classes for processors
├── stage2_query_generator.py     # Query generation
├── stage2_query_classifier.py    # Query classification
├── stage3_icd_processor.py       # ICD generation & verification
├── stage4_specialty_verifier.py  # Specialty verification
├── data_pipeline_stage5_step1.py # Triplet creation (preserved)
├── hyperparameter_tuner.py       # Model optimization
├── pipeline_orchestrator.py      # Main orchestrator
└── config.json               # Configuration file
```

Installation

Prerequisites

- Python 3.8+
- Azure ML Workspace access
- Required Python packages:

```bash
pip install pandas numpy torch transformers
pip install langchain langchain-openai
pip install sentence-transformers
pip install azureml-core azure-identity
pip install optuna faiss-cpu
pip install tqdm pathlib
```

Setup

1. Clone the repository and navigate to the project directory
2. Create a default configuration file:

```bash
python pipeline_orchestrator.py --create-config
```

3. Update `config.json` with your specific paths and parameters
4. Ensure Azure ML workspace configuration is available

Configuration

The pipeline uses a JSON configuration file (`config.json`) with the following sections:

Model Configuration

```json
{
  "model": {
    "gpt_4o_version": "2024-05-01-preview",
    "gpt_4o_deployment": "gpt-4o",
    "gpt_41_version": "2024-12-01-preview",
    "gpt_41_deployment": "gpt-4.1",
    "max_tokens": 4000,
    "temperature": 0.9,
    "seed": 1337
  }
}
```

Path Configuration

```json
{
  "paths": {
    "base_datasets_path": "../../../datasets",
    "augmented_datasets_path": "../../../datasets/datasets_augmented",
    "embeddings_path": "../../../datasets/embeddings",
    "model_path": "../../../../model/NovaSearch_stella_en_1.5B_v5/",
    "icd_reference_file": "../../../../dataset/icd10.csv"
  }
}
```

Processing Configuration

```json
```

```
{
  "processing": {
    "batch_size": 8,
    "num_chunks": 4,
    "target_queries_per_specialty": 250,
    "similarity_threshold": 0.9,
    "top_k_positives": 10,
    "num_hard_negatives": 50,
    "retry_attempts": 3
  }
}
```

Usage

Pipeline Orchestrator (Recommended)

The main entry point for running the pipeline:

```bash
# Run complete pipeline
python pipeline_orchestrator.py --stages stage2 stage3 stage4

# Run specific stage
python pipeline_orchestrator.py --stages stage2 --stage2-generation-model gpt-4o

# Run with chunk processing
python pipeline_orchestrator.py --stages stage3 --chunk-index 0

# Validate configuration only
python pipeline_orchestrator.py --validate-only
```

Individual Stage Execution

Each stage can also be run independently:

```bash
```

```
# Stage 2: Query Generation
python stage2_query_generator.py --model-name gpt-4o --load-retry

# Stage 2: Query Classification
python stage2_query_classifier.py --model-name gpt-4.1 --batch-mode --batch-path /path/to/data

# Stage 3: ICD Processing
python stage3_icd_processor.py --mode all --chunk-index 0

# Stage 4: Specialty Verification
python stage4_specialty_verifier.py --mode all --chunk-index 0

# Hyperparameter Tuning
python hyperparameter_tuner.py --train-path data/train.csv --eval-path data/eval.csv --n-trials 20
```

Pipeline Stages

Stage 2: Query Processing

Query Generation

- Generates medical queries for specialties using GPT models
- Supports retry mechanisms for failed generations
- Configurable number of queries per specialty

Query Classification

- Classifies queries as diagnostic, procedural, or exclude
- Filters to keep only diagnostic queries
- Supports batch and single-file processing modes

Stage 3: ICD Processing

Unified Processing Modes:

- `generate`: Generate ICD codes using GPT-4o and GPT-4.1
- `combine`: Combine results from both models
- `verify`: Verify and filter irrelevant codes
- `all`: Run complete pipeline (default)

Key Features:

- Dual model processing for better coverage
- Code validation against ICD reference
- Automatic deduplication and cleaning

Stage 4: Specialty Verification

Processing Modes:

- `combine`: Combine filtered datasets
- `verify`: Verify specialty assignments
- `all`: Complete verification pipeline

Features:

- Validates query-specialty relevance
- Filters out non-relevant assignments
- Supports chunk-based distributed processing

Hyperparameter Tuning

Optimization Features:

- Optuna-based hyperparameter search
- Custom Recall@K evaluation
- Stratified sampling for efficiency
- Configurable search spaces

Parameters Optimized:

- Batch size
- Learning rate
- Number of epochs
- Contrastive loss margin

Examples

Complete Pipeline Execution

```bash
```

```
# Run full pipeline with custom configuration
python pipeline_orchestrator.py \
  --config my_config.json \
  --stages stage2 stage3 stage4 \
  --log-level INFO
```

Distributed Processing

bash

```
# Process chunk 0 of 4 total chunks
python pipeline_orchestrator.py \
  --stages stage3 \
  --chunk-index 0 \
  --stage3-mode all

# Process chunk 1 of 4 total chunks
python pipeline_orchestrator.py \
  --stages stage3 \
  --chunk-index 1 \
  --stage3-mode all
```

Hyperparameter Tuning

bash

```
# Run hyperparameter optimization
python pipeline_orchestrator.py \
  --stages hyperparameter \
  --hp-train-path datasets/train.csv \
  --hp-eval-path datasets/eval.csv \
  --hp-n-trials 50 \
  --hp-sample-fraction 0.01
```

Stage 3 Execution Modes

The reformatted Stage 3 addresses your requirement for a single script execution:

bash

```
# Run all steps in sequence (replaces 3 separate runs)
python stage3_icd_processor.py --mode all --combine-results --create-splits

# Or run individual steps
python stage3_icd_processor.py --mode generate  # Generate ICD codes
python stage3_icd_processor.py --mode combine   # Combine results
python stage3_icd_processor.py --mode verify    # Verify codes
```

Logging and Monitoring

Log Files

- Pipeline logs are saved to `logs/` directory
- Each component has separate log files with timestamps
- Error reports are automatically generated on failures

Log Levels

- `DEBUG`: Detailed debugging information
- `INFO`: General information about pipeline progress
- `WARNING`: Warning messages for non-critical issues
- `ERROR`: Error messages for failures

Monitoring Progress

```bash
# Monitor logs in real-time
tail -f logs/pipeline_orchestrator_*.log

# Check specific stage logs
tail -f logs/stage3_icd_processor_*.log
```

Error Handling

Automatic Recovery

- **Retry Mechanisms**: Automatic retry for API failures
- **Resume Capability**: Skip already processed items
- **Error Collection**: Comprehensive error tracking

- **Graceful Shutdown**: Clean shutdown on interruption

Retry Lists

Failed items are automatically saved to retry lists:

```bash
# Rerun failed items
python stage2_query_generator.py --load-retry --dataset-type pickle
```

Error Reports

Detailed error reports are saved as JSON files:

- `error_report_TIMESTAMP.json`
- `retry_dict_*.json`
- `pipeline_results_TIMESTAMP.json`

Configuration Management

Environment Variables

Critical settings can be overridden via environment variables:

```bash
export AZURE_OPENAI_API_KEY="your-api-key"
export AZURE_OPENAI_ENDPOINT="your-endpoint"
```

Config Validation

```bash
# Validate configuration before running
python pipeline_orchestrator.py --validate-only
```

Performance Optimization

Distributed Processing

- Use `--chunk-index` for parallel processing
- Default: 4 chunks (configurable in config.json)

- Each chunk can run on separate compute instances

Memory Management

- Automatic memory monitoring and logging
- Configurable batch sizes
- Progress tracking with resource usage

GPU Usage

```bash
# Enable GPU for embedding generation (Stage 5)
export CUDA_VISIBLE_DEVICES=0,1,2,3
```

Troubleshooting

Common Issues

1. **Azure Authentication Failures**

   ```bash
   az login
   az account set --subscription "your-subscription-id"
   ```

2. **Missing Dependencies**

   ```bash
   pip install -r requirements.txt
   ```

3. **Configuration Errors**

   ```bash
   python pipeline_orchestrator.py --validate-only
   ```

4. **Memory Issues**
   - Reduce batch_size in configuration
   - Use chunk-based processing
   - Monitor memory usage in logs

Debug Mode

```bash
# Enable debug logging
python pipeline_orchestrator.py --log-level DEBUG --stages stage2
```

Log Analysis

```bash
# Check for errors in logs
grep -i "error" logs/*.log

# Check retry statistics
grep -i "retry" logs/*.log
```

File Structure

Input Files Required

- ICD reference file (CSV format)
- Medical specialty datasets (JSON format)
- Azure ML workspace configuration

Output Files Generated

- Processed query datasets
- ICD code mappings
- Verification results
- Error reports and retry lists
- Hyperparameter optimization results

Contributing

Code Standards

- Follow PEP 8 style guidelines
- Add comprehensive docstrings
- Include type hints
- Write unit tests for new features

Adding New Stages

1. Inherit from `BaseProcessor`
2. Implement required methods
3. Add configuration options
4. Update pipeline orchestrator
5. Add comprehensive logging

Testing

```bash
# Run configuration validation
python -m pytest tests/test_config.py

# Test individual components
python -m pytest tests/test_stage2.py
```

License

This project is licensed under the MIT License - see the LICENSE file for details.

Support

For issues and questions:

1. Check the troubleshooting section
2. Review log files for error details
3. Validate configuration settings
4. Create an issue with detailed error logs