Handout 2: Implementing Structured Output Systems

Machine Learning for Smarter Innovation

1 Handout 2: Implementing Structured Output Systems

1.1 Step-by-Step Implementation Guide

1.1.1 Prerequisites

- Basic Python knowledge
- OpenAI or Anthropic API key
- Understanding of JSON format
- Read Handout 1 first

1.1.2 Part 1: Understanding Function Calling

Function calling is OpenAI's way of getting structured outputs. Instead of asking the model to "return JSON", you define the exact structure you want.

How It Works

1. You define a function schema (what fields, what types)
2. AI decides if it should call the function
3. AI generates the arguments in JSON format
4. You receive structured, validated data

Basic Example

```
import openai

# Define your function schema
functions = [
    {
        "name": "extract_review_data",
        "description": "Extract structured data from a restaurant review",
        "parameters": {
            "type": "object",
            "properties": {
                "rating": {
                    "type": "integer",
                    "description": "Overall rating from 1-5",
                    "minimum": 1,
                    "maximum": 5
                },
                "price_level": {
                    "type": "string",
                    "enum": ["cheap", "moderate", "expensive"],
                    "description": "Price category"
```

```
                },
                "food_quality": {
                    "type": "integer",
                    "minimum": 1,
                    "maximum": 5
                }
            },
            "required": ["rating", "price_level"]
        }
    }
]

# Call the API
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[
        {"role": "user", "content": "Review: The food was amazing! 5 stars. A
    bit pricey at $40 per person but worth it."}
    ],
    functions=functions,
    function_call={"name": "extract_review_data"}  # Force function call
)

# Extract the result
function_call = response.choices[0].message.function_call
arguments = json.loads(function_call.arguments)

print(arguments)
# Output: {"rating": 5, "price_level": "expensive", "food_quality": 5}
```

1.1.3 Part 2: Pydantic for Validation

Pydantic gives you type-safe Python with automatic validation.

Installation

```
pip install pydantic
```

Define Your Schema

```
from pydantic import BaseModel, Field, validator
from typing import List, Optional

class RestaurantReview(BaseModel):
    rating: int = Field(..., ge=1, le=5, description="Overall rating")
    food_quality: int = Field(..., ge=1, le=5)
    service_quality: int = Field(..., ge=1, le=5)
    price_level: str = Field(..., regex="^(cheap|moderate|expensive)$")
    avg_price_per_person: Optional[float] = Field(None, gt=0)
    themes: List[str] = Field(default_factory=list, max_items=5)
    recommended_for: List[str] = []

    @validator('price_level')
    def validate_price_level(cls, v):
        allowed = ['cheap', 'moderate', 'expensive']
        if v not in allowed:
            raise ValueError(f'Must be one of {allowed}')
        return v

    @validator('recommended_for')
    def validate_recommendations(cls, v):
        allowed = ['date', 'family', 'business', 'friends', 'solo']
```

```
        for item in v:
            if item not in allowed:
                raise ValueError(f'{item} not in allowed values')
        return v

# Use it
try:
    review = RestaurantReview(
        rating=5,
        food_quality=5,
        service_quality=4,
        price_level="moderate",
        themes=["italian", "romantic", "authentic"]
    )
    print(review.dict())  # Convert to dictionary
    print(review.json())  # Convert to JSON string
except ValidationError as e:
    print(f"Validation failed: {e}")
```

1.1.4 Part 3: Complete Implementation

Here's a production-ready implementation:

```
import openai
import json
import time
from pydantic import BaseModel, Field, ValidationError
from typing import Optional, List

# 1. Define Pydantic model
class ReviewData(BaseModel):
    rating: int = Field(..., ge=1, le=5)
    food_quality: int = Field(..., ge=1, le=5)
    service_quality: int = Field(..., ge=1, le=5)
    price_level: str
    confidence: Optional[float] = None

# 2. Define OpenAI function
functions = [{
    "name": "extract_review",
    "description": "Extract structured data from review",
    "parameters": ReviewData.schema()
}]

# 3. Extraction function with retry logic
def extract_review_data(review_text: str, max_retries: int = 3) -> Optional[
    ReviewData]:
    """Extract structured data from review with retry logic"""

    for attempt in range(max_retries):
        try:
            # Call OpenAI
            response = openai.ChatCompletion.create(
                model="gpt-4",
                temperature=0.1,  # Low temperature for consistency
                messages=[
                    {"role": "system", "content": "You are a data extraction
    expert. Extract information accurately."},
                    {"role": "user", "content": f"Extract data from this
    review:\n\n{review_text}"}
                ],
```

```
                functions=functions,
                function_call={"name": "extract_review"}
            )

            # Parse result
            function_call = response.choices[0].message.function_call
            data = json.loads(function_call.arguments)

            # Validate with Pydantic
            review_data = ReviewData(**data)

            return review_data

        except ValidationError as e:
            print(f"Validation error on attempt {attempt + 1}: {e}")
            if attempt < max_retries - 1:
                time.sleep(2 ** attempt)  # Exponential backoff
                continue
            else:
                return None

        except Exception as e:
            print(f"API error on attempt {attempt + 1}: {e}")
            if attempt < max_retries - 1:
                time.sleep(2 ** attempt)
                continue
            else:
                return None

    return None

# 4. Use it
review_text = """
The food was absolutely amazing! Best Italian I've had in years.
Service was friendly and attentive. A bit pricey at $45 per person
but definitely worth it for a special occasion.
"""

result = extract_review_data(review_text)

if result:
    print("Success!")
    print(result.json(indent=2))
else:
    print("Extraction failed after all retries")
```

1.1.5 Part 4: Error Handling Patterns

Pattern 1: Graceful Degradation

```
def extract_with_fallback(text: str) -> dict:
    """Try AI extraction, fall back to rule-based"""

    # Try AI first
    result = extract_with_ai(text)
    if result and validate(result):
        return result

    # Fallback to rule-based
    print("AI failed, using rule-based extraction")
    return rule_based_extraction(text)

def rule_based_extraction(text: str) -> dict:
```

```
    """Simple keyword-based extraction as fallback"""
    import re

    # Extract rating with regex
    rating_match = re.search(r'(\d+)\s*stars?|\b(one|two|three|four|five)\b',
    text.lower())

    # Extract price indicators
    price_match = re.search(r'\$(\d+)', text)

    return {
        "rating": extract_rating(rating_match) if rating_match else 3,
        "price_level": estimate_price(price_match) if price_match else "
    moderate",
        "confidence": 0.5  # Low confidence for rule-based
    }
```

Pattern 2: Validation Layers

```
def multi_stage_validation(data: dict) -> bool:
    """Three-stage validation"""

    # Stage 1: Schema validation
    try:
        review = ReviewData(**data)
    except ValidationError:
        return False

    # Stage 2: Business rules
    if review.rating == 5 and review.service_quality < 3:
        # Suspicious: 5-star overall but poor service?
        return False

    if review.price_level == "cheap" and review.avg_price_per_person > 30:
        # Inconsistent: cheap but $30+?
        return False

    # Stage 3: Confidence check
    if hasattr(review, 'confidence') and review.confidence < 0.7:
        # Low confidence, needs human review
        return False

    return True
```

1.1.6 Part 5: Testing Strategy

Unit Tests

```
import pytest

def test_schema_validation():
    """Test Pydantic schema catches invalid data"""

    # Valid data
    valid = ReviewData(
        rating=5,
        food_quality=5,
        service_quality=4,
        price_level="moderate"
    )
    assert valid.rating == 5
```

```
    # Invalid rating
    with pytest.raises(ValidationError):
        ReviewData(
            rating=6,  # Too high!
            food_quality=5,
            service_quality=4,
            price_level="moderate"
        )

    # Invalid price level
    with pytest.raises(ValidationError):
        ReviewData(
            rating=5,
            food_quality=5,
            service_quality=4,
            price_level="super_expensive"  # Not in enum!
        )
```

Integration Tests

```
def test_full_pipeline():
    """Test complete extraction pipeline"""

    test_review = """
    Amazing restaurant! Food quality is top-notch, 5 stars.
    Service was excellent. Price is moderate around $25 per person.
    """

    result = extract_review_data(test_review)

    assert result is not None
    assert result.rating == 5
    assert result.food_quality == 5
    assert result.price_level == "moderate"
```

1.1.7 Part 6: Production Deployment

Environment Setup

```
import os
from dotenv import load_dotenv

# Load API keys
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# Configuration
CONFIG = {
    "model": "gpt-4",
    "temperature": 0.1,
    "max_tokens": 500,
    "timeout": 30,
    "max_retries": 3
}
```

Logging

```
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

```python
def extract_with_logging(text: str) -> Optional[ReviewData]:
    """Extract with comprehensive logging"""

    logger.info(f"Starting extraction for text of length {len(text)}")

    try:
        result = extract_review_data(text)

        if result:
            logger.info("Extraction successful")
            logger.debug(f"Extracted data: {result.json()}")
        else:
            logger.warning("Extraction failed after all retries")

        return result

    except Exception as e:
        logger.error(f"Unexpected error: {e}", exc_info=True)
        return None
```

Monitoring

```python
from datetime import datetime

class MetricsCollector:
    def __init__(self):
        self.successes = 0
        self.failures = 0
        self.total_time = 0

    def record_success(self, duration: float):
        self.successes += 1
        self.total_time += duration

    def record_failure(self):
        self.failures += 1

    def get_metrics(self) -> dict:
        total = self.successes + self.failures
        return {
            "success_rate": self.successes / total if total > 0 else 0,
            "failure_rate": self.failures / total if total > 0 else 0,
            "avg_duration": self.total_time / self.successes if self.successes
    > 0 else 0,
            "total_processed": total
        }

# Use it
metrics = MetricsCollector()

def extract_with_metrics(text: str) -> Optional[ReviewData]:
    start_time = time.time()

    result = extract_review_data(text)

    duration = time.time() - start_time

    if result:
        metrics.record_success(duration)
    else:
        metrics.record_failure()

    return result
```

1.1.8 Part 7: Common Issues and Solutions

Issue 1: Inconsistent Outputs **Problem:** Same input gives different outputs

Solution: - Set temperature to 0 - Use deterministic model - Add more examples to prompt - Use function calling instead of raw JSON

Issue 2: Slow Performance **Problem:** Takes 5+ seconds per request

Solution:

```
# Use caching
from functools import lru_cache

@lru_cache(maxsize=1000)
def extract_cached(text: str) -> dict:
    """Cache results for identical inputs"""
    return extract_review_data(text)

# Use smaller model for simple tasks
CONFIG = {
    "model": "gpt-3.5-turbo",  # Faster than gpt-4
    "temperature": 0.1
}
```

Issue 3: High API Costs **Problem:** Spending $500/month on API calls

Solution: - Cache aggressively - Use smaller models when possible - Batch requests - Reduce prompt length - Remove unnecessary examples after tuning

1.1.9 Part 8: Next Steps

1. **Build the workshop project** - Restaurant review extraction
2. **Add monitoring** - Track success rates in production
3. **Optimize costs** - Implement caching and batching
4. **Read Handout 3** - Advanced techniques for 99%+ reliability

1.1.10 Key Takeaways

- Function calling + Pydantic = 95%+ reliability
- Always implement retry logic with exponential backoff
- Test with real data, not just happy paths
- Monitor everything in production
- Have fallback strategies for failures
- Start simple, add complexity gradually

Next: Handout 3 for production-grade advanced techniques