Examples

Here are some demonstrating how to use DocETL's pandas integration for various tasks.

Note that caching is enabled, but intermediate outputs are not persisted like in DocETL's YAML interface.

Example 1: Analyzing Customer Reviews

Extract structured insights from customer reviews:

```
import pandas as pd
from docetl import SemanticAccessor

# Load customer reviews
df = pd.DataFrame({
    "review": [
        "Great laptop, fast processor but battery life could be better",
        "The camera quality is amazing, especially in low light",
        "Keyboard feels cheap and the screen is too dim"
    ]
})

# Configure semantic accessor
df.semantic.set_config(default_model="gpt-4o-mini")

# Extract structured insights
result = df.semantic.map(
    prompt="""Analyze this product review and extract:
    1. Mentioned features
    2. Sentiment per feature
    3. Overall sentiment

    Review: {{input.review}}""",
    output={
        "schema": {
            "features": "list[str]",
            "feature_sentiments": "dict[str, str]",
            "overall_sentiment": "str"
        }
    }
)

# Filter for negative reviews
negative_reviews = result.semantic.filter(
    prompt="Is this review predominantly negative? Consider the overall
sentiment and feature sentiments.\n{{input}}"
)
```

Example 2: Deduplicating Customer Records

Identify and merge duplicate customer records using fuzzy matching:

```
# Customer records from two sources
df1 = pd.DataFrame({
    "name": ["John Smith", "Mary Johnson"],
    "email": ["john@email.com", "mary.j@email.com"],
    "address": ["123 Main St", "456 Oak Ave"]
})

df2 = pd.DataFrame({
    "name": ["John A Smith", "Mary Johnson"],
    "email": ["john@email.com", "mary.johnson@email.com"],
    "address": ["123 Main Street", "456 Oak Avenue"]
})

# Merge records with fuzzy matching
merged = df1.semantic.merge(
    df2,
    comparison_prompt="""Compare these customer records and determine if they
represent the same person.
    Consider name variations, email patterns, and address formatting.

    Record 1:
    Name: {{input1.name}}
    Email: {{input1.email}}
    Address: {{input1.address}}

    Record 2:
    Name: {{input2.name}}
    Email: {{input2.email}}
    Address: {{input2.address}}""",
    fuzzy=True, # This will automatically invoke optimization
)
```

Example 3: Topic Analysis of News Articles

Group and summarize news articles by topic:

```
# News articles
df = pd.DataFrame({
    "title": ["Apple's New iPhone Launch", "Tech Giants Face Regulation", "AI
Advances in Healthcare"],
    "content": ["Apple announced...", "Lawmakers propose...", "Researchers
develop..."]
})

# First, use a semantic map to extract the topic from each article
df = df.semantic.map(
    prompt="Extract the topic from this article: {{input.content}}",
    output={"schema": {"topic": "str"}}
)
```

```
# Group similar articles and generate summaries
summaries = df.semantic.agg(
    # First, group similar articles
    fuzzy=True,
    reduce_keys=["topic"],
    comparison_prompt="""Are these articles about the same topic or closely
related topics?

    Article 1:
    Title: {{input1.title}}
    Content: {{input1.content}}

    Article 2:
    Title: {{input2.title}}
    Content: {{input2.content}}""",

    # Then, generate a summary for each group
    reduce_prompt="""Summarize these related articles into a comprehensive
overview:

    Articles:
    {{inputs}}""",

    output={
        "schema": {
            "summary": "str",
            "key_points": "list[str]"
        }
    }
)

# Summaries will be a df with the following columns:
# - topic: str (because this was the reduce_keys)
# - summary: str
# - key_points: list[str]
```

Example 4: Multi-step Analysis Pipeline

Combine multiple operations for complex analysis:

```
# Social media posts
posts = pd.DataFrame({
    "text": ["Just tried the new iPhone 15!", "Having issues with iOS 17",
"Android is better"],
    "timestamp": ["2024-01-01", "2024-01-02", "2024-01-03"]
})

# 1. Extract structured information
analyzed = posts.semantic.map(
    prompt="""Analyze this social media post and extract:
    1. Product mentioned
    2. Sentiment
    3. Issues/Praise points
```

```
    Post: {{input.text}}""",
    output={
        "schema": {
            "product": "str",
            "sentiment": "str",
            "points": "list[str]"
        }
    }
)

# 2. Filter relevant posts
relevant = analyzed.semantic.filter(
    prompt="Is this post about Apple products? {{input}}"
)


# 3. Group by issue and summarize
summaries = relevant.semantic.agg(
    fuzzy=True,
    reduce_keys=["product"],
    comparison_prompt="Do these posts discuss the same product?",
    reduce_prompt="Summarize the feedback about this product",
    output={
        "schema": {
            "summary": "str",
            "frequency": "int",
            "severity": "str"
        }
    }
)

# Summaries will be a df with the following columns:
# - product: str (because this was the reduce_keys)
# - summary: str
# - frequency: int
# - severity: str

# Track total cost
print(f"Total analysis cost: ${summaries.semantic.total_cost}")
```

Example 5: Error Handling and Validation

Implement robust error handling and validation:

```
# Product descriptions
df = pd.DataFrame({
    "description": ["High-performance laptop...", "Wireless earbuds...",
"Invalid/"]
})

try:
    result = df.semantic.map(
        prompt="Extract product specifications from: {{input.description}}.
```

```
There should be at least one feature.",
        output={
            "schema": {
                "category": "str",
                "features": "list[str]",
                "price_range": "enum[budget, mid-range, premium, luxury]"
            }
        },
        # Validation rules
        validate=[
            "len(output['features']) >= 1",
        ],
        # Retry configuration
        num_retries_on_validate_failure=2,
    )

    # Check operation history
    for op in result.semantic.history:
        print(f"Operation: {op.op_type}")
        print(f"Modified columns: {op.output_columns}")

except Exception as e:
    print(f"Error during processing: {e}")
```

Example 6: PDF Analysis

DocETL supports native PDF handling with Claude and Gemini, in map and filter operations. Suppose you have a column in your pandas dataframe with PDF paths (1 path per row), and you want the LLM to do some analysis for each PDF. You can do this by setting the `pdf_url_key` parameter in the map or filter operation.

```
df = pd.DataFrame({
    "PdfPath": ["https://docetlcloudbank.blob.core.windows.net/ntsb-
reports/Report_N617GC.pdf",
"https://docetlcloudbank.blob.core.windows.net/ntsb-
reports/Report_CEN25LA075.pdf"]
})

result_df = df.semantic.map(
    prompt="Summarize the air crash report and determine any contributing
factors",
    output={
        "schema": {"summary": "str", "contributing_factors": "list[str]"}
    },
    pdf_url_key="PdfPath", # This is the column with the PDF paths
)

print(result_df.head()) # The result will have the same number of rows as the
input dataframe, with the summary and contributing factors added
```

Example 7: Synthetic Data Generation

DocETL supports generating multiple outputs for each input using the `n` parameter in the map operation. This is useful for synthetic data generation, A/B testing content variations, or creating multiple alternatives for each input.

```
# Starter concepts for content generation
df = pd.DataFrame({
    "product": ["Smart Watch", "Wireless Earbuds", "Home Security Camera"],
    "target_audience": ["Fitness Enthusiasts", "Music Lovers", "Homeowners"]
})

# Generate 5 marketing headlines for each product-audience combination
variations = df.semantic.map(
    prompt="""Generate a compelling marketing headline for the following
product and target audience:

    Product: {{input.product}}
    Target Audience: {{input.target_audience}}

    The headline should:
    - Be attention-grabbing and memorable
    - Speak directly to the target audience's needs or desires
    - Highlight the key benefit of the product
    - Be between 5-10 words
    """,
    output={"schema": {"headline": "str"}, "n": 5}  # Generate 5 variations
for each input row
)

print(f"Original dataframe rows: {len(df)}")
print(f"Generated variations: {len(variations)}")  # Should be 5x the original
count

# Variations dataframe will contain:
# - All original columns (product, target_audience)
# - The new headline column
# - 5 rows for each original row (15 total for this example)

# You can also combine this with other operations
# Filter to only keep the best headlines
best_headlines = variations.semantic.filter(
    prompt="""Is this headline exceptional and likely to drive high
engagement?

    Product: {{input.product}}
    Target Audience: {{input.target_audience}}
    Headline: {{input.headline}}

    Consider catchiness, emotional appeal, and clarity.
    """
)
```

This example will generate 15 total variations (3 products × 5 variations each). You can adjust the `n` parameter to generate more or fewer variations as needed.

Example 8: Document Processing with Split and Gather

Process long documents by splitting them into chunks and adding contextual information:

```
# Long documents that need to be processed in chunks
df = pd.DataFrame({
    "document_id": ["doc1", "doc2"],
    "title": ["Technical Manual", "Research Paper"],
    "content": [
        "Chapter 1: Introduction\n\nThis manual provides comprehensive
guidance for system installation and configuration. The installation process
involves several critical steps that must be followed in order.\n\nChapter 2:
Prerequisites\n\nBefore beginning installation, ensure all system requirements
are met. This includes hardware specifications, software dependencies, and
network connectivity.\n\nChapter 3: Installation\n\nThe installation wizard
guides users through the setup process. Select appropriate configuration
options based on your deployment environment.",
        "Abstract\n\nThis research investigates the impact of machine learning
on data processing efficiency. Our methodology involved testing multiple
algorithms across diverse datasets.\n\nIntroduction\n\nMachine learning has
revolutionized data processing across industries. Previous studies have shown
significant improvements in processing speed and
accuracy.\n\nMethodology\n\nWe conducted experiments using three different ML
algorithms: neural networks, decision trees, and support vector machines. Each
algorithm was tested on datasets ranging from 1,000 to 1,000,000 records."
    ]
})

# Step 1: Split documents into manageable chunks
chunks = df.semantic.split(
    split_key="content",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 1}
)

print(f"Original documents: {len(df)}")
print(f"Generated chunks: {len(chunks)}")

# Step 2: Add contextual information to each chunk
enhanced_chunks = chunks.semantic.gather(
    content_key="content_chunk",
    doc_id_key="semantic_split_0_id",
    order_key="semantic_split_0_chunk_num",
    peripheral_chunks={
        "previous": {"head": {"count": 1}},  # Include 1 previous chunk
        "next": {"head": {"count": 1}}       # Include 1 next chunk
    }
)

# Step 3: Extract structured information from each chunk with context
analyzed_chunks = enhanced_chunks.semantic.map(
    prompt="""Analyze this document chunk with its surrounding context and
extract:
    1. Main topic or section
```

```
    2. Key concepts discussed
    3. Action items or requirements (if any)

    Document chunk with context:
    {{input.content_chunk_rendered}}""",
    output={
        "schema": {
            "section_topic": "str",
            "key_concepts": "list[str]",
            "action_items": "list[str]"
        }
    }
)

# Step 4: Aggregate insights by document
document_summaries = analyzed_chunks.semantic.agg(
    reduce_keys=["document_id"],
    reduce_prompt="""Create a comprehensive summary of this document based on
all its chunks:

    Document chunks:
    {% for chunk in inputs %}
    Section: {{chunk.section_topic}}
    Key concepts: {{chunk.key_concepts | join(', ')}}
    Action items: {{chunk.action_items | join(', ')}}
    ---
    {% endfor %}""",
    output={
        "schema": {
            "document_summary": "str",
            "all_key_concepts": "list[str]",
            "all_action_items": "list[str]"
        }
    }
)

print(f"Final document summaries: {len(document_summaries)}")
print(f"Total processing cost: ${document_summaries.semantic.total_cost}")
```

Example 9: Data Structure Processing with Unnest

Handle complex nested data structures commonly found in JSON APIs or survey responses:

```
# Survey data with nested responses
survey_df = pd.DataFrame({
    "respondent_id": [1, 2, 3],
    "demographics": [
        {"age": 25, "location": "NYC", "education": "Bachelor's"},
        {"age": 34, "location": "SF", "education": "Master's"},
        {"age": 28, "location": "Chicago", "education": "PhD"}
    ],
    "interests": [
        ["technology", "sports", "music"],
```

```
        ["science", "reading"],
        ["art", "travel", "cooking", "photography"]
    ],
    "ratings": [
        {"product_quality": 4, "customer_service": 5, "value": 3},
        {"product_quality": 5, "customer_service": 4, "value": 4},
        {"product_quality": 3, "customer_service": 3, "value": 5}
    ]
})

# Step 1: Unnest demographics into separate columns
with_demographics = survey_df.semantic.unnest(
    unnest_key="demographics",
    expand_fields=["age", "location", "education"]
)

# Step 2: Unnest interests (each interest becomes a separate row)
individual_interests = with_demographics.semantic.unnest(
    unnest_key="interests"
)

# Step 3: For ratings, expand all fields
with_ratings = individual_interests.semantic.unnest(
    unnest_key="ratings",
    expand_fields=["product_quality", "customer_service", "value"]
)

print(f"Original survey responses: {len(survey_df)}")
print(f"Individual interest entries: {len(individual_interests)}")
print(f"Final flattened dataset: {len(with_ratings)}")

# Now you can analyze individual interests by demographics
interest_analysis = with_ratings.semantic.map(
    prompt="""Analyze this person's interest in the context of their
demographics:

    Person: {{input.age}} years old, {{input.education}}, from
{{input.location}}
    Interest: {{input.interests}}
    Product ratings: Quality={{input.product_quality}}, Service=
{{input.customer_service}}, Value={{input.value}}

    Provide insights about how this interest might relate to their
demographics and satisfaction.""",
    output={
        "schema": {
            "demographic_insight": "str",
            "interest_category": "str",
            "satisfaction_correlation": "str"
        }
    }
)

# Aggregate insights by interest category
category_insights = interest_analysis.semantic.agg(
    reduce_keys=["interest_category"],
    reduce_prompt="""Summarize insights about people interested in
```

```
{{inputs[0].interest_category}}:

    {% for person in inputs %}
    - {{person.age}} year old {{person.education}} from {{person.location}}:
{{person.demographic_insight}}
    {% endfor %}""",
    output={
        "schema": {
            "category_summary": "str",
            "typical_demographics": "str",
            "satisfaction_patterns": "str"
        }
    }
)
```

Example 10: Combined Document Workflow

A comprehensive example combining multiple operations for end-to-end document processing:

```
# Research papers with metadata
papers_df = pd.DataFrame({
    "paper_id": ["P001", "P002", "P003"],
    "title": [
        "Deep Learning for Natural Language Processing",
        "Quantum Computing Applications in Cryptography",
        "Sustainable Energy Storage Solutions"
    ],
    "abstract": ["This paper explores...", "We investigate...", "This study
examines..."],
    "full_text": [
        "Introduction\n\nDeep learning has revolutionized
NLP...\n\nMethodology\n\nWe employed transformer
architectures...\n\nResults\n\nOur experiments show significant
improvements...\n\nConclusion\n\nThe findings demonstrate...",
        "Introduction\n\nQuantum computing
promises...\n\nBackground\n\nClassical cryptography relies...\n\nQuantum
Algorithms\n\nShor's algorithm can factor...\n\nConclusion\n\nQuantum
computing will require...",
        "Introduction\n\nRenewable energy adoption...\n\nCurrent
Challenges\n\nEnergy storage remains...\n\nProposed Solutions\n\nWe propose
novel battery...\n\nResults\n\nOur testing shows..."
    ],
    "authors": [
        ["Dr. Smith", "Prof. Johnson", "Dr. Lee"],
        ["Prof. Chen", "Dr. Wilson"],
        ["Dr. Brown", "Prof. Davis", "Dr. Taylor", "Prof. Anderson"]
    ]
})

# Step 1: Unnest authors for author-level analysis
author_papers = papers_df.semantic.unnest(unnest_key="authors")

# Step 2: Split full text into sections
```

```
paper_sections = papers_df.semantic.split(
    split_key="full_text",
    method="delimiter",
    method_kwargs={"delimiter": "\n\n", "num_splits_to_group": 1}
)

# Step 3: Add context to each section
contextual_sections = paper_sections.semantic.gather(
    content_key="full_text_chunk",
    doc_id_key="semantic_split_0_id",
    order_key="semantic_split_0_chunk_num",
    peripheral_chunks={
        "previous": {"head": {"count": 1}},
        "next": {"head": {"count": 1}}
    }
)

# Step 4: Extract insights from each section
section_insights = contextual_sections.semantic.map(
    prompt="""Analyze this paper section in context:

Paper: {{input.title}}
Section content: {{input.full_text_chunk_rendered}}

Extract:
1. Section type (Introduction, Methodology, Results, etc.)
2. Key findings or claims
3. Technical concepts mentioned
4. Research gaps or future work mentioned""",
    output={
        "schema": {
            "section_type": "str",
            "key_findings": "list[str]",
            "technical_concepts": "list[str]",
            "future_work": "list[str]"
        }
    }
)

# Step 5: Aggregate insights by paper
paper_summaries = section_insights.semantic.agg(
    reduce_keys=["paper_id"],
    reduce_prompt="""Create a comprehensive analysis of this research paper:

Paper: {{inputs[0].title}}

Sections analyzed:
{% for section in inputs %}
{{section.section_type}}: {{section.key_findings | join(', ')}}
Technical concepts: {{section.technical_concepts | join(', ')}}
{% endfor %}

Provide a structured summary.""",
    output={
        "schema": {
            "comprehensive_summary": "str",
            "main_contributions": "list[str]",
```

```
            "methodology_type": "str",
            "research_field": "str"
        }
    }
)

# Step 6: Cross-paper analysis
field_analysis = paper_summaries.semantic.agg(
    reduce_keys=["research_field"],
    fuzzy=True,  # Group similar research fields
    reduce_prompt="""Analyze research trends in this field based on these
papers:

{% for paper in inputs %}
Paper: {{paper.title}}
Summary: {{paper.comprehensive_summary}}
Contributions: {{paper.main_contributions | join(', ')}}
---
{% endfor %}""",
    output={
        "schema": {
            "field_trends": "str",
            "common_methodologies": "list[str]",
            "emerging_themes": "list[str]"
        }
    }
)

print(f"Papers processed: {len(papers_df)}")
print(f"Authors identified: {len(author_papers)}")
print(f"Sections analyzed: {len(section_insights)}")
print(f"Research fields identified: {len(field_analysis)}")
print(f"Total processing cost: ${field_analysis.semantic.total_cost}")
```