IntelliFlow ADK Contributions Documentation

Overview

This document provides a comprehensive overview of the contributions made by the IntelliFlow project to the Google Agent Development Kit (ADK) Python repository. These contributions enhance the data analysis capabilities of the ADK, making it more powerful and versatile for real-world data analysis tasks.

Table of Contents

Introduction

The Google Agent Development Kit (ADK) provides a powerful framework for building agent-based applications. However, its data analysis capabilities were limited, particularly in the area of data preprocessing. The IntelliFlow project has addressed this gap by implementing comprehensive data preprocessing tools and enhancing the Data Analysis Agent template.

These contributions enable the ADK to handle real-world data analysis tasks more effectively, as real-world data often requires cleaning and preprocessing before meaningful analysis can be performed. By contributing these enhancements back to the ADK Python repository, the IntelliFlow project is not only improving its own capabilities but also providing valuable tools for the broader ADK community.

Data Preprocessing Tools

The core of our contribution is the implementation of comprehensive data preprocessing tools for the Data Analysis Agent template. These tools enable the agent to handle common data quality issues and prepare data for analysis.

Missing Value Handling

Missing values are a common issue in real-world datasets. Our implementation provides multiple strategies for handling missing values:

- **Drop**: Remove rows or columns with missing values
- **Fill with Mean/Median/Mode**: Replace missing values with statistical measures
- **Fill with Constant**: Replace missing values with a specified constant
- **Forward/Backward Fill**: Propagate the last valid observation forward/backward
- **Interpolation**: Estimate missing values based on surrounding values

Example usage:

```
request = DataPreprocessRequest(
    data_id="my_data",
    operation="clean_missing_values",
    parameters={
        "strategy": "fill_mean",
        "columns": ["numeric_col1", "numeric_col2"]
    }
)
response = await preprocessing_tools.preprocess_data(request)
```

Outlier Detection and Treatment

Outliers can significantly impact analysis results. Our implementation provides methods for identifying and handling outliers:

- **Z-Score Method**: Identify values that deviate significantly from the mean
- **IQR Method**: Identify values outside the interquartile range
- **Percentile Method**: Identify values outside specified percentiles

For each method, multiple treatment options are available: - **Remove**: Remove rows with outliers - **Cap**: Cap outliers at a threshold value - **Replace with Null**: Replace outliers with null values

Example usage:

```python
request = DataPreprocessRequest(
    data_id="my_data",
    operation="handle_outliers",
    parameters={
        "method": "zscore",
        "columns": ["numeric_col"],
        "threshold": 3,
        "treatment": "cap"
    }
)
response = await preprocessing_tools.preprocess_data(request)
```

Feature Engineering

Feature engineering is crucial for creating informative features for analysis. Our implementation provides tools for creating new features from existing ones:

- **Polynomial Features**: Create polynomial and interaction terms
- **Interaction Features**: Create pairwise interaction terms
- **Binning**: Convert continuous variables into categorical bins
- **Date Features**: Extract components from date/time columns
- **Text Features**: Extract features from text columns

Example usage:

```python
request = DataPreprocessRequest(
    data_id="my_data",
    operation="engineer_features",
    parameters={
        "operation": "date_features",
        "column": "date_col",
        "features": ["year", "month", "day", "dayofweek"]
    }
)
response = await preprocessing_tools.preprocess_data(request)
```

Categorical Encoding

Categorical variables need to be encoded for use in many analysis methods. Our implementation provides multiple encoding methods:

- **One-Hot Encoding**: Create binary columns for each category
- **Label Encoding**: Assign a unique integer to each category
- **Ordinal Encoding**: Assign integers based on a specified order
- **Target Encoding**: Replace categories with the mean of the target variable

Example usage:

```
 request = DataPreprocessRequest(
     data_id="my_data",
     operation="encode_categorical",
     parameters={
         "method": "one_hot",
         "columns": ["categorical_col"],
         "drop_first": True
     }
)
response = await preprocessing_tools.preprocess_data(request)
```

Data Normalization

Normalization is important for many analysis methods. Our implementation provides multiple normalization methods:

- **Min-Max Scaling**: Scale values to a specified range
- **Z-Score Standardization**: Scale values to have mean 0 and standard deviation 1
- **Robust Scaling**: Scale values based on the median and interquartile range
- **Log Transformation**: Apply a logarithmic transformation

Example usage:

```
 request = DataPreprocessRequest(
     data_id="my_data",
     operation="normalize_data",
     parameters={
         "method": "minmax",
         "columns": ["numeric_col"],
         "feature_range": [0, 1]
     }
)
response = await preprocessing_tools.preprocess_data(request)
```

Duplicate Removal

Duplicate records can bias analysis results. Our implementation provides methods for identifying and removing duplicate records:

Example usage:

```
 request = DataPreprocessRequest(
     data_id="my_data",
     operation="remove_duplicates",
     parameters={
         "subset": ["col1", "col2"],
         "keep": "first"
     }
 )
 response = await preprocessing_tools.preprocess_data(request)
```

Data Type Conversion

Converting data types is often necessary for proper analysis. Our implementation provides tools for converting column data types:

Example usage:

```
request = DataPreprocessRequest(
    data_id="my_data",
    operation="convert_types",
    parameters={
        "type_mappings": {
            "col1": "float32",
            "col2": "category",
            "col3": "datetime"
        }
    }
)
response = await preprocessing_tools.preprocess_data(request)
```

Enhanced Data Analysis Agent

In addition to the data preprocessing tools, we've enhanced the Data Analysis Agent template to provide a more comprehensive data analysis workflow:

Updated Agent Interface

We've added preprocessing operations as a configurable parameter in the agent constructor:

```
agent = DataAnalysisAgent(
    name="data_analyst",
    model="gemini-1.5-pro",
    data_sources=["data.csv", "data.json", "data.xlsx"],
    analysis_types=["summary", "correlation", "distribution",
"outliers", "time_series"],
    visualization_types=["line", "bar", "scatter", "histogram",
"boxplot", "heatmap", "pie"],
    preprocessing_operations=[
        "clean_missing_values", "handle_outliers",
"engineer_features",
        "encode_categorical", "normalize_data",
"remove_duplicates", "convert_types"
    ]
)
```

Improved Default Instruction

We've enhanced the default instruction to include preprocessing guidance and a recommended data analysis workflow:

```
instruction = (
    "You are a specialized data analysis assistant. "
    "Your goal is to help users analyze data, extract insights,
"
    "and create visualizations. "
    "Follow these guidelines:\n\n"


"1. Understand the user's data analysis needs clearly before
proceeding.\n"
    "2. Use the provided tools to load, preprocess, transform,
analyze, and visualize data.\n"
    "3. Always consider data quality issues and apply
appropriate preprocessing steps.\n"
    "4. Explain your analysis approach and findings in clear,
concise language.\n"
    "5. When presenting results, include both the raw data and
your interpretation.\n"

"6. For visualizations, explain what the visualization shows and
why it's useful.\n"
    "7. If you encounter limitations or need more information,
ask the user.\n\n"
)
```

Integrated Preprocessing Tools

We've seamlessly integrated preprocessing tools with existing data loading, transformation, analysis, and visualization capabilities:

```
async def preprocess_data(
    self, request: DataPreprocessRequest
) -> DataPreprocessResponse:
    """Preprocess data.

    Args:
        request: The request containing the data ID and
preprocessing operation.

    Returns:
        A response indicating whether the data was preprocessed
successfully.
    """
    return await
self._preprocessing_tools.preprocess_data(request)
```

Implementation Details

The implementation of the data preprocessing tools follows a modular, object-oriented design:

DataPreprocessRequest and DataPreprocessResponse

These classes define the interface for preprocessing operations:

```
class DataPreprocessRequest(BaseModel):
    """Request to preprocess data."""

    data_id: str = Field(...,
description="The identifier of the data to preprocess.")
    operation: str = Field(..., description="The preprocessing
operation to perform.")
    parameters: Dict[str, Any] = Field(
        default_factory=dict, description="Parameters for the
preprocessing operation."
    )


class DataPreprocessResponse(BaseModel):
    """Response from preprocessing data."""

    success: bool = Field(...,
description="Whether the data was preprocessed successfully.")
    message: str = Field(..., description="A message describing
the result.")
    data_id: Optional[str] = Field(
        None, description="An identifier for the preprocessed
data."
    )
    statistics: Dict[str, Any] = Field(
        default_factory=dict, description="Statistics about the
preprocessing operation."
    )
```

DataPreprocessingTools

This class implements the preprocessing operations:

```
class DataPreprocessingTools:
    """Tools for preprocessing data."""

    def __init__(self, data_store: Dict[str, pd.DataFrame]):
        """Initialize the data preprocessing tools.

        Args:
            data_store: A dictionary mapping data IDs to
DataFrames.
        """
        self._data_store = data_store

    async def preprocess_data(
        self, request: DataPreprocessRequest
    ) -> DataPreprocessResponse:
        """Preprocess data.

        Args:
            request: The request containing the data ID and
preprocessing operation.

        Returns:
            A response indicating whether the data was
preprocessed successfully.
        """
        # Implementation details...
```

Integration with DataAnalysisToolset

The preprocessing tools are integrated with the DataAnalysisToolset:

```
class DataAnalysisToolset(BaseTool):
    """A toolset for data analysis tasks."""

    name: str = "data_analysis_toolset"
    description: str = "A set of tools for data analysis tasks."

    def __init__(
        self,
        data_sources: Optional[List[str]] = None,
        analysis_types: Optional[List[str]] = None,
        visualization_types: Optional[List[str]] = None,
        memory_service: Optional[BaseMemoryService] = None,
    ):
        """Initialize the DataAnalysisToolset."""
        super().__init__()
        self.data_sources = data_sources or []
        self.analysis_types = analysis_types or []
        self.visualization_types = visualization_types or []
        self.memory_service = memory_service
        self._data_store: Dict[str, pd.DataFrame] = {}
        self._temp_dir = tempfile.mkdtemp()
        self._preprocessing_tools =
DataPreprocessingTools(self._data_store)
```

Testing and Validation

We've implemented comprehensive tests to ensure the reliability and correctness of our contributions:

Unit Tests

We've created unit tests for all preprocessing operations, covering both normal usage and edge cases:

```
 @pytest.mark.asyncio
async def test_handle_missing_values(preprocessing_tools):
    """Test handling missing values."""
    # Test drop strategy
    request = DataPreprocessRequest(
        data_id="test_data",
        operation="clean_missing_values",
        parameters={"strategy": "drop"}
    )
    response = await
preprocessing_tools.preprocess_data(request)

    assert response.success
    assert response.data_id is not None
    assert
preprocessing_tools._data_store[response.data_id].shape[0] == 4
# 2 rows with NaN dropped
```

Integration Tests

We've verified the integration of preprocessing tools with the existing Data Analysis Agent framework:

```
@pytest.mark.asyncio
async def test_data_analysis_agent():
    """Test the Data Analysis Agent with preprocessing
capabilities."""
    agent = DataAnalysisAgent(
        name="test_agent",
        model="gemini-1.5-pro",
        preprocessing_operations=[
            "clean_missing_values", "handle_outliers",
"engineer_features",
            "encode_categorical", "normalize_data",
"remove_duplicates", "convert_types"
        ]
    )

    # Test preprocessing capabilities
    response = await agent.generate_content(
        "Clean missing values in the data using mean
imputation."
    )

    assert "successfully" in response.text.lower()
```

Example Validation

We've ensured that the example works end-to-end with realistic data:

```
async def main():
    """Run the example."""
    # Create a sample dataset
    df = pd.DataFrame({
        "numeric_col": [1, 2, None, 4, 5, 100],  # Contains
missing value and outlier
        "categorical_col": ["A", "B", "A", None, "C", "B"],  #
Contains missing value
        "date_col": pd.date_range(start="2023-01-01", periods=6,
freq="D"),
    })

    # Save the dataset to a temporary file
    temp_dir = tempfile.mkdtemp()
    data_path = os.path.join(temp_dir, "sample_data.csv")
    df.to_csv(data_path, index=False)

    # Create a data analysis agent
    agent = DataAnalysisAgent(
        name="data_analyst",
        model="gemini-1.5-pro",
        data_sources=[data_path],
        analysis_types=["summary", "correlation",
"distribution", "outliers", "time_series"],
        visualization_types=["line", "bar", "scatter",
"histogram", "boxplot", "heatmap", "pie"],
        preprocessing_operations=[
            "clean_missing_values", "handle_outliers",
"engineer_features",
            "encode_categorical", "normalize_data",
"remove_duplicates", "convert_types"
        ]
    )

    # Example 1: Load data and get a summary
    print("\n=== Example 1: Load data and get a summary ===")
    response = await agent.generate_content(
        f"Load data from {data_path} and provide a summary of
the data."
    )
    print(response.text)
```

Integration with IntelliFlow

The enhanced ADK data analysis capabilities have been integrated into the IntelliFlow project:

Project Structure

```
IntelliFlow/
├── common/
│   ├── enhanced_adk/
│   │   ├── data_analysis/
│   │   │   ├── __init__.py
│   │   │   ├── data_preprocessing_tools.py
│   │   │   ├── data_analysis_agent.py
│   │   │   └── README.md
│   │   └── __init__.py
├── examples/
│   ├── __init__.py
│   └── data_analysis_example.py
```

Usage in IntelliFlow

The enhanced ADK data analysis capabilities are used in the IntelliFlow project for:

- **Data Ingestion**: Preprocessing raw data before analysis
- **Analysis**: Performing statistical analysis and modeling

- **Visualization**: Creating charts and graphs
- **Insight Generation**: Extracting insights from data

Example usage in IntelliFlow:

```python
from IntelliFlow.common.enhanced_adk.data_analysis import
DataAnalysisAgent

# Create a data analysis agent
agent = DataAnalysisAgent(
    name="data_analyst",
    model="gemini-1.5-pro",
    data_sources=["data.csv", "data.json", "data.xlsx"],
    analysis_types=["summary", "correlation", "distribution",
"outliers", "time_series"],
    visualization_types=["line", "bar", "scatter", "histogram",
"boxplot", "heatmap", "pie"],
    preprocessing_operations=[
        "clean_missing_values", "handle_outliers",
"engineer_features",
        "encode_categorical", "normalize_data",
"remove_duplicates", "convert_types"
    ]
)

# Use the agent for data analysis
response = await agent.generate_content(
    "Load data from data.csv, clean missing values, handle
outliers, and provide a summary of the data."
)
```

Future Work

We plan to continue contributing to the ADK Python repository with additional enhancements:

1. **Advanced Visualization Capabilities**: Implement interactive visualizations and additional chart types.
2. **Enhanced Memory System**: Improve the memory service with better search capabilities and persistent storage.
3. **Data Caching Mechanism**: Implement a caching system for improved performance with large datasets.
4. **Domain-Specific Analysis Tools**: Create specialized tools for financial, scientific, and business data analysis.

Conclusion

The contributions made by the IntelliFlow project to the ADK Python repository significantly enhance its data analysis capabilities, making it more suitable for real-world data analysis tasks. The preprocessing tools address a critical gap in the existing implementation, as real-world data often requires cleaning and preprocessing before meaningful analysis can be performed.

By contributing these enhancements back to the ADK Python repository, the IntelliFlow project is not only improving its own capabilities but also providing valuable tools for the broader ADK community. This aligns with our commitment to open-source collaboration and knowledge sharing.

We encourage the ADK community to use these enhanced data analysis capabilities and provide feedback for further improvements.