Advanced Python Data Science Optimization Techniques

Optimizing Data Analysis with Pandas and NumPy

Python's data science ecosystem, particularly Pandas and NumPy, offers powerful capabilities that can be significantly enhanced through advanced optimization techniques. These techniques can transform standard data analysis workflows into high-performance, memory-efficient operations capable of handling massive datasets.

Advanced Data Selection Strategies

Sophisticated data selection techniques can dramatically improve performance and code readability:

```
 import pandas as pd
import numpy as np

# MultiIndex for hierarchical data organization
def optimize_with_multiindex(df):
    # Create a hierarchical index for complex data organization
    df_indexed = df.set_index(['Year', 'Month', 'Category'])

    # Efficient slicing with MultiIndex
    subset = df_indexed.loc[(2023, 'January', slice(None)), :]

    # Cross-sectional selection
    cross_section = df_indexed.xs(('Electronics'), level='Category')

    return subset, cross_section

# Boolean indexing with NumPy for vectorized filtering
def advanced_boolean_filtering(arr):
    # Complex conditional filtering
    mask1 = (arr > 0) & (arr < 100)
    mask2 = (arr % 2 == 0) | (arr % 7 == 0)

    # Combined masks for sophisticated filtering
    combined_mask = mask1 & mask2
    filtered_data = arr[combined_mask]

    return filtered_data
```

Performance Optimization Techniques

When working with large datasets, performance optimization becomes critical:

```python
 # Memory optimization with categorical data
def optimize_memory_usage(df):
    # Before optimization
    initial_memory = df.memory_usage(deep=True).sum() / 1e6  # MB

    # Convert object columns with few unique values to categorical
    for col in df.select_dtypes(include=['object']).columns:
        if df[col].nunique() / len(df) < 0.5:  # If less than 50% unique values
            df[col] = df[col].astype('category')

    # Convert numeric columns to smaller dtypes where possible
    for col in df.select_dtypes(include=['int64']).columns:
        if df[col].min() >= 0:
            if df[col].max() < 255:
                df[col] = df[col].astype('uint8')
            elif df[col].max() < 65535:
                df[col] = df[col].astype('uint16')

    # After optimization
    optimized_memory = df.memory_usage(deep=True).sum() / 1e6  # MB

    return df, {
        'initial_memory_mb': initial_memory,
        'optimized_memory_mb': optimized_memory,
        'reduction_percentage': (1 - optimized_memory/initial_memory) * 100
    }

# Processing large datasets with chunking
def process_large_dataset(filepath, chunk_size=100000):
    results = []

    # Process file in manageable chunks to avoid memory overload
    for chunk in pd.read_csv(filepath, chunksize=chunk_size):
        # Process each chunk
        processed = chunk.groupby('key').agg({
            'value1': 'sum',
            'value2': 'mean',
            'value3': lambda x: np.percentile(x, 95)
        })
        results.append(processed)

    # Combine results from all chunks
    combined_results = pd.concat(results)
    final_results = combined_results.groupby(level=0).sum()

    return final_results
```

Advanced Time Series Analysis

Time series data requires specialized techniques for optimal analysis:

```
# High-performance time series operations
def advanced_time_series_analysis(df):
    # Ensure datetime index
    df.index = pd.to_datetime(df.index)

    # Efficient resampling for different time frequencies
    daily = df.resample('D').mean()
    weekly = df.resample('W').mean()
    monthly = df.resample('M').mean()

    # Advanced rolling operations with custom window types
    # Exponentially weighted moving average gives more weight to recent observations
    ewma = df.ewm(span=30, min_periods=10).mean()

    # Time-based rolling with variable window size
    expanding_std = df.expanding(min_periods=10).std()

    # Custom rolling function
    def custom_percentile(x):
        return np.percentile(x, q=90)

    custom_rolling = df.rolling(window=20).apply(custom_percentile)

    return {
        'resampled': {'daily': daily, 'weekly': weekly, 'monthly': monthly},
        'rolling': {'ewma': ewma, 'expanding_std': expanding_std, 'custom':
custom_rolling}
    }
```

Vectorized Operations for Maximum Performance

Vectorization is key to achieving optimal performance with NumPy and Pandas:

```
# Vectorized operations instead of loops
def vectorized_calculations(df):
    # Avoid loops with vectorized operations

    # Bad practice (slow)
    def slow_calculation(df):
        result = []
        for i in range(len(df)):
            result.append(df.iloc[i, 0] * df.iloc[i, 1] + df.iloc[i, 2])
        return result

    # Good practice (fast vectorized operation)
```

```
    def fast_calculation(df):
        return df.iloc[:, 0] * df.iloc[:, 1] + df.iloc[:, 2]

    # Advanced vectorized calculations
    def complex_vectorized_operation(df):
        # Element-wise operations across multiple columns
        result1 = np.sqrt(np.square(df['x']) + np.square(df['y']))

        # Conditional vectorized operations
        result2 = np.where(df['value'] > 0,
                           np.log1p(df['value']),
                           -np.log1p(np.abs(df['value'])))

        # Vectorized statistical functions
        result3 = (df - df.mean()) / df.std()  # Z-score normalization

        return pd.DataFrame({
            'euclidean_distance': result1,
            'signed_log': result2,
            'normalized': result3
        })

    return complex_vectorized_operation(df)
```

Advanced Aggregation and Grouping

Complex aggregation operations can extract deeper insights from data:

```
 # Sophisticated aggregation techniques
def advanced_aggregation(df):
    # Multiple aggregations per column
    agg_result = df.groupby('category').agg({
        'numeric1': ['min', 'max', 'mean', 'std'],
        'numeric2': ['median', 'sum', lambda x: x.quantile(0.95)],
        'date_column': ['first', 'last', lambda x: (x.max() - x.min()).days]
    })

    # Named aggregations for better readability
    named_agg = df.groupby('category').agg(
        min_value=('numeric1', 'min'),
        max_value=('numeric1', 'max'),
        avg_value=('numeric1', 'mean'),
        range_days=('date_column', lambda x: (x.max() - x.min()).days)
    )

    # Grouped filter operations
    filtered_groups = df.groupby('category').filter(lambda x: len(x) > 10 and
x['numeric1'].mean() > 100)

    # Transform operations that preserve the original dataframe shape
```

```python
    df_transformed = df.copy()
    df_transformed['normalized'] = df.groupby('category')['numeric1'].transform(
        lambda x: (x - x.mean()) / x.std()
    )

    return {
        'multi_agg': agg_result,
        'named_agg': named_agg,
        'filtered': filtered_groups,
        'transformed': df_transformed
    }
```

Parallel Processing for Data Science

Leveraging parallel processing can dramatically speed up data operations:

```python
import multiprocessing
from joblib import Parallel, delayed

# Parallel data processing
def parallel_data_processing(df, n_jobs=-1):
    # Split dataframe into chunks for parallel processing
    chunks = np.array_split(df, multiprocessing.cpu_count() if n_jobs == -1 else
n_jobs)

    # Define processing function
    def process_chunk(chunk):
        # Complex operations on chunk
        result = chunk.groupby('key').apply(lambda x:
            pd.Series({
                'mean_val': x['value'].mean(),
                'max_val': x['value'].max(),
                'count': len(x),
                'complex_calc': np.sum(np.sqrt(x['value']) * np.log1p(x['value']))
            })
        )
        return result

    # Process chunks in parallel
    results = Parallel(n_jobs=n_jobs)(
        delayed(process_chunk)(chunk) for chunk in chunks
    )

    # Combine results
    combined_results = pd.concat(results)

    # Aggregate results by key
    final_results = combined_results.groupby(level=0).mean()
```

```
    return final_results
```

Advanced Data Visualization Integration

Integrating visualization with data processing creates powerful analytical workflows:

```
 import matplotlib.pyplot as plt
import seaborn as sns

# Advanced visualization with data processing
def integrated_visualization_workflow(df):
    # Prepare data with advanced processing
    processed_data = df.groupby(['category', pd.Grouper(key='date', freq='M')]).agg({
        'value': ['mean', 'std', 'count'],
        'growth': 'sum'
    })

    # Flatten multi-level columns
    processed_data.columns = ['_'.join(col).strip() for col in
processed_data.columns.values]
    processed_data = processed_data.reset_index()

    # Create visualization
    fig, axes = plt.subplots(2, 2, figsize=(14, 10))

    # Time series by category
    for category, data in processed_data.groupby('category'):
        axes[0, 0].plot(data['date'], data['value_mean'], label=category)
    axes[0, 0].set_title('Monthly Average by Category')
    axes[0, 0].legend()

    # Correlation heatmap
    numeric_cols = [col for col in processed_data.columns if
processed_data[col].dtype in ['float64', 'int64']]
    sns.heatmap(processed_data[numeric_cols].corr(), annot=True,
cmap='coolwarm', ax=axes[0, 1])
    axes[0, 1].set_title('Correlation Matrix')

    # Distribution plot
    sns.histplot(data=processed_data, x='value_mean', hue='category', kde=True,
ax=axes[1, 0])
    axes[1, 0].set_title('Distribution of Monthly Averages')

    # Scatter plot with size representing count
    sns.scatterplot(
        data=processed_data,
        x='value_mean',
        y='value_std',
        size='value_count',
```

```
        hue='category',
        ax=axes[1, 1]
    )
    axes[1, 1].set_title('Variability vs. Average (size=count)')

    plt.tight_layout()
    return fig, processed_data
```

Sources

- [Deep Dive into Pandas and NumPy: Advanced Data Analysis Techniques]()
- [Advanced Python Techniques for Efficient Data Analysis]()
- [Mastering Code Optimization with Numpy and Pandas for Large Datasets]()