Master 2 Big Data and Artificial Intelligence, Paris-Dauphine University

Data Quality: MiniProject

Abdelhedi Youssef, Ettriki Aicha, Habassi Khalil, and Migniha Rodrigue

January 8, 2025

1 Introduction

This project aims to develop a Python class, tentatively named `tensprov`, to efficiently capture and infer the provenance of data transformations in large datasets. The core idea is to use binary sparse tensors to represent and track the dependencies between input and output datasets for various data operations.

The project focuses on designing a framework that captures the provenance of key data processing operations, such as filtering, oversampling, joining, and unions. To ensure efficiency and scalability, multiple alternatives will be explored and implemented to derive and optimize provenance capture.

The deliverables include the development of the `tensprov` class, performance evaluation in terms of processing time, and the implementation of at least two alternative approaches for each operation type. The overarching goal is to establish an effective and computationally efficient way to trace data dependencies, providing valuable insights into the provenance of processed data.

2 Class & Methods

Class Definition: `TensProv`

The `TensProv` class is designed to manage the provenance of data transformations by capturing how input data frames are transformed into output data frames through various operations. The primary goal of this class is to track the relationship between data transformations and their sources using tensors, specifically sparse binary tensors.

The code snippet for the class initialization is as follows:

```
class TensProv:
    def __init__(self):
        self.tensor = None
        self.performance = {}
```

The class initialization involves two key components:

- `self.tensor`: This attribute is initialized as `None` and will later store the tensor representing the provenance of the data. The tensor is a sparse binary matrix where each element indicates the relationship between input and output records.
- `self.performance`: This is an empty dictionary that will store performance metrics, such as processing time, for different operations performed within the class. It allows for the assessment of the efficiency of the tensor construction and operations.

The class aims to handle different types of data transformations, including horizontal and vertical reductions, joins, unions, and oversampling, by constructing the provenance tensor according to the type of operation. The initialization sets up the necessary attributes for these operations and the tracking of their performance.

2.1 Data Filtering Methods

This section describes three methods for filtering data in a DataFrame and constructing a provenance tensor. These methods are implemented within the function `filter_data`, which applies a horizontal filter on the input DataFrame and builds a provenance tensor to map the filtered data back to the original.

2.1.1 Method : Standard Filtering

The standard method filters rows based on a condition function and constructs the provenance tensor. The steps are as follows:

1. Apply the `condition_func` function to each row of the input DataFrame (`data_in`) to create a boolean mask.
2. Use the boolean mask to filter the rows of `data_in`, resulting in the filtered DataFrame (`data_out`).

3. Depending on the `method` argument, invoke one of the two specialized tensor construction methods: `direct` or `alt`.

The main function implementation is as follows:

```
def filter_data(self, data_in,
    condition_func, method='direct'):

    start_time = time.time()

    # (1) Generate data_out
    bool_mask = data_in.apply(
        condition_func, axis=1)
    data_out = data_in[bool_mask].copy()

    # (2) Build the tensor
    if method == 'direct':
        self._filter_data_direct(data_in
            , data_out)
    elif method == 'alt':
        self._filter_data_alt(data_in,
            data_out)
    else:
        raise ValueError(f"Unknown
            filter method: {method}")

    self.performance[f"filter_data({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method provides the overall structure for filtering and tensor construction, with flexibility for choosing the specific implementation method.

2.1.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method explicitly maps the indices of `data_out` to their corresponding positions in `data_in`. The steps are as follows:

1. Create dictionaries to map indices of `data_in` and `data_out` to their positions.

2. Iterate over the indices of `data_out`, using the dictionaries to map rows and columns.

3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _filter_data_direct(self, data_in,
    data_out):

    rows = data_out.index
    col_idx_map = {idx: i for i, idx in
        enumerate(data_in.index)}
    row_idx_map = {idx: i for i, idx in
        enumerate(data_out.index)}

    row_list, col_list, data_list = [],
        [], []
    for out_idx in data_out.index:
        row_list.append(row_idx_map[
            out_idx])
        col_list.append(col_idx_map[
            out_idx])
        data_list.append(1)

    shape = (len(data_out), len(data_in)
        )
    self.tensor = csr_matrix((data_list,
         (row_list, col_list)), shape=
        shape)
```

This method is flexible but involves explicit dictionary creation and iteration, which may increase computational cost for large datasets.

2.1.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method simplifies tensor construction by using the `get_indexer_for` function. The steps are as follows:

1. Use the `get_indexer_for` function to obtain the positions of `data_out` indices in `data_in`.

2. Create arrays representing the rows, columns, and values for the provenance tensor.

3. Construct the tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _filter_data_alt(self, data_in,
    data_out):

    row_list = np.arange(len(data_out))
    # Get the position of each index of
        data_out in data_in
    col_array = data_in.index.
        get_indexer_for(data_out.index)
    data_array = np.ones(len(data_out))

    shape = (len(data_out), len(data_in)
        )
    self.tensor = csr_matrix((data_array
        , (row_list, col_array)), shape=
        shape)
```

This method avoids explicit dictionary creation and loops, making it more concise and efficient for large datasets.

2.1.4 Comparison of Methods

The two methods differ as follows:

1. **Standard Filtering:** This method provides a general structure for filtering and delegates tensor construction to the selected implementation (`direct` or `alt`).
2. **Direct Mapping:** Uses explicit dictionaries to map indices, offering flexibility but with potentially higher computational cost.
3. **Alternative Mapping:** Relies on `get_indexer_for`, simplifying the code and improving efficiency for large datasets.

Each method constructs the same provenance tensor as a sparse matrix, and the choice depends on the dataset size and computational constraints.

2.2 Column Dropping Methods

This section describes three methods for vertical reduction (dropping columns) in a DataFrame and constructing the provenance tensor. These methods are implemented in the function `drop_columns`, which removes specified columns from the input DataFrame and builds a provenance tensor.

2.2.1 Method : Standard Column Dropping

The standard method drops specified columns and constructs the provenance tensor. The steps are as follows:

1. Drop the specified columns from the input DataFrame (`data_in`) to generate the output DataFrame (`data_out`).
2. Depending on the `method` argument, invoke one of the two specialized tensor construction methods: `direct` or `alt`.

The implementation is as follows:

```
def drop_columns(self, data_in,
    columns_to_drop, method='direct'):

    start_time = time.time()

    # (1) Generate data_out
    data_out = data_in.drop(columns=
        columns_to_drop).copy()

    # (2) Build the tensor
    if method == 'direct':
        self._drop_columns_direct(
            data_in, data_out)
    elif method == 'alt':
        self._drop_columns_alt(data_in,
            data_out)
    else:
        raise ValueError(f"Unknown
            drop_columns method: {method
            }")

    self.performance[f"drop_columns({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method provides the general structure for column dropping and delegates tensor construction to the chosen implementation.

2.2.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method explicitly maps the retained columns in `data_out` to their positions in `data_in`. The steps are as follows:

1. Create dictionaries to map the columns of `data_in` and `data_out` to their positions.
2. Iterate over the retained columns in `data_out`, using the dictionaries to map rows and columns.
3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _drop_columns_direct(self, data_in,
    data_out):

    out_cols = data_out.columns
    in_cols = data_in.columns

    row_map = {col: i for i, col in
        enumerate(out_cols)}
    col_map = {col: i for i, col in
        enumerate(in_cols)}

    row_list, col_list, data_list = [],
        [], []
    for col in out_cols:
        if col in col_map:
            row_list.append(row_map[col
                ])
            col_list.append(col_map[col
                ])
            data_list.append(1)

    shape = (len(data_out.columns), len(
        data_in.columns))
    self.tensor = csr_matrix((data_list,
         (row_list, col_list)), shape=
        shape)
```

This method is straightforward but involves explicit mapping, which can be computationally expensive for large datasets.

2.2.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method simplifies tensor construction by iterating over the retained columns in `data_out`. The steps are as follows:

1. Create dictionaries to map the columns of `data_in` and `data_out`.
2. Iterate over the retained columns in `data_out`, assigning a value of 1 in the provenance tensor for each match.
3. Construct the tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _drop_columns_alt(self, data_in,
    data_out):

    in_cols = list(data_in.columns)
    out_cols = list(data_out.columns)

    row_list, col_list, data_list = [],
        [], []
    row_map = {col: i for i, col in
        enumerate(out_cols)}
    col_map = {col: i for i, col in
        enumerate(in_cols)}

    for col in out_cols:
        row_list.append(row_map[col])
        col_list.append(col_map[col])
        data_list.append(1)

    shape = (len(out_cols), len(in_cols)
        )
    self.tensor = csr_matrix((data_list,
         (row_list, col_list)), shape=
        shape)
```

This method is slightly more concise and avoids unnecessary condition checks.

2.2.4 Comparison of Methods

The two methods differ as follows:

1. **Standard Column Dropping:** Provides the general structure for column dropping and delegates tensor construction to the selected implementation.
2. **Direct Mapping:** Uses explicit dictionaries for mapping, offering flexibility but with potentially higher computational cost.
3. **Alternative Mapping:** Relies on simpler iteration and mapping, making it concise and efficient for large datasets.

Each method constructs the same provenance tensor as a sparse matrix, and the choice depends on the dataset size and computational constraints.

2.3 Oversampling Methods

This section describes three methods for horizontal augmentation (duplicating rows) in a DataFrame and constructing the provenance tensor. These methods are implemented in the function `oversample_data`, which duplicates the rows of the input DataFrame and builds a provenance tensor.

2.3.1 Method : Standard Oversampling

The standard method duplicates rows of the input DataFrame (`data_in`) a specified number of times (`multiplier`) and constructs the provenance tensor. The steps are as follows:

1. Generate the output DataFrame (`data_out`) by concatenating `data_in` `multiplier` times.

2. Depending on the `method` argument, invoke one of the two specialized tensor construction methods: `direct` or `alt`.

The implementation is as follows:

```
def oversample_data(self, data_in,
    multiplier=2, method='direct'):

    start_time = time.time()

    # (1) Generate data_out
    data_out_list = []
    for _ in range(multiplier):
        data_out_list.append(data_in.
            copy())
    data_out = pd.concat(data_out_list,
        ignore_index=True)

    # (2) Build the tensor
    if method == 'direct':
        self._oversample_direct(data_in,
             data_out, multiplier)
    elif method == 'alt':
        self._oversample_alt(data_in,
            data_out, multiplier)
    else:
        raise ValueError(f"Unknown
            oversample method: {method}"
            )

    self.performance[f"oversample_data({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method provides the general structure for oversampling and delegates tensor construction to the chosen implementation.

2.3.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method maps each row in `data_out` to the corresponding row in `data_in`, cycling through the rows of `data_in`. The steps are as follows:

1. Calculate the shape of the provenance tensor as (`len(data_out)`, `len(data_in)`).

2. Iterate over the rows of `data_out`, determining the corresponding row in `data_in` using the modulo operation.

3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _oversample_direct(self, data_in,
    data_out, multiplier):

    shape = (len(data_out), len(data_in)
        )
    rows, cols, data_vals = [], [], []

    for out_i in range(len(data_out)):
        in_i = out_i % len(data_in)
        rows.append(out_i)
        cols.append(in_i)
        data_vals.append(1)

    self.tensor = csr_matrix((data_vals,
         (rows, cols)), shape=shape)
```

This method is computationally efficient and straightforward for large datasets.

2.3.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method iterates over the rows of `data_in` for each duplication step, explicitly mapping the rows. The steps are as follows:

1. Calculate the shape of the provenance tensor as (`len(data_out)`, `len(data_in)`).

2. Iterate over each duplication step and each row of `data_in`, mapping them explicitly to rows in `data_out`.

3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _oversample_alt(self, data_in,
    data_out, multiplier):

    shape = (len(data_out), len(data_in)
        )
    rows, cols, data_vals = [], [], []

    num_in = len(data_in)
    # For each duplication step k
    for k in range(multiplier):
        # For each row i in data_in
        for i in range(num_in):
            out_i = k * num_in + i
            rows.append(out_i)
            cols.append(i)
            data_vals.append(1)

    self.tensor = csr_matrix((data_vals,
         (rows, cols)), shape=shape)
```

This method is slightly more explicit and provides better readability for certain use cases.

2.3.4 Comparison of Methods

The two methods differ as follows:

1. **Standard Oversampling:** Provides the general structure for oversampling and delegates tensor construction to the selected implementation.
2. **Direct Mapping:** Uses modulo operations for efficient mapping, making it computationally efficient for large datasets.
3. **Alternative Mapping:** Iterates explicitly over duplication steps, providing better readability and flexibility for specific applications.

Each method constructs the same provenance tensor as a sparse matrix, and the choice depends on the dataset size and computational constraints.

2.4 One-Hot Encoding Methods

This section describes two methods for performing one-hot encoding on a categorical column of a DataFrame, followed by the construction of a provenance tensor. The function `one_hot_encode` handles vertical augmentation (encoding) and tensor construction.

2.4.1 Method : Standard One-Hot Encoding

The standard one-hot encoding method generates a new DataFrame by creating binary columns for each category in the specified column. The steps are as follows:

1. Create dummy columns for the specified categorical column using `pd.get_dummies`, and concatenate these columns with the original DataFrame (after removing the original categorical column).
2. Depending on the `method` argument, invoke either the `direct` or `alt` method to construct the provenance tensor.

The implementation is as follows:

```
def one_hot_encode(self, data_in, column
    , method='direct'):

    start_time = time.time()

    # (1) Generate data_out
    dummies = pd.get_dummies(data_in[
        column], prefix=column, dtype=
        int)
    data_out = pd.concat([data_in.drop(
        columns=[column]), dummies],
        axis=1)

    # (2) Build the tensor
    if method == 'direct':
        self._one_hot_direct(data_in,
            data_out, column)
    elif method == 'alt':
        self._one_hot_alt(data_in,
            data_out, column)
    else:
        raise ValueError(f"Unknown
            one_hot_encode method: {
            method}")

    self.performance[f"one_hot_encode({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method provides the basic structure for one-hot encoding and delegates tensor construction to the selected method.

2.4.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method performs one-hot encoding by explicitly iterating over the rows of `data_in` and constructing the tensor by adding entries for both the unchanged columns and the newly created one-hot columns. The steps are as follows:

1. For each row in `data_in`, add entries for the unchanged columns in the output tensor.
2. For the one-hot encoded column, find the corresponding column in `data_out` based on the category value and add it to the tensor.
3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _one_hot_direct(self, data_in,
    data_out, column):

    row_count = len(data_in)
    col_count_out = len(data_out.columns
        )

    rows, cols, data_vals = [], [], []

    for i in range(row_count):
        # Original columns (except the
            dropped one)
        for c in data_in.columns:
            if c != column:
                out_col_index = data_out
                    .columns.get_loc(c)
                rows.append(i)
                cols.append(
                    out_col_index)
                data_vals.append(1)
        # One-hot column corresponding
            to the value
        new_col_name = f"{column}_{
            data_in.iloc[i][column]}"
        if new_col_name in data_out.
            columns:
            out_col_index = data_out.
                columns.get_loc(
                new_col_name)
            rows.append(i)
            cols.append(out_col_index)
            data_vals.append(1)

    shape = (row_count, col_count_out)
    self.tensor = csr_matrix((data_vals,
        (rows, cols)), shape=shape)
```

This method is straightforward and efficient for creating a one-hot encoded tensor with direct mapping.

2.4.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method is an alternative approach to one-hot encoding, which iterates over the rows of `data_in` in a more explicit manner. The steps are as follows:

1. For each row in `data_in`, add entries for the unchanged columns in the output tensor.
2. For the one-hot encoded column, find the corresponding column in `data_out` based on the category value and add it to the tensor.
3. Construct the provenance tensor as a sparse matrix (`csr_matrix`).

The implementation is as follows:

```
def _one_hot_alt(self, data_in, data_out
    , column):

    row_count = len(data_in)
    col_count_out = len(data_out.columns
        )

    row_list, col_list, data_list = [],
        [], []

    # Same principles, just a different
        style
    for i, row in data_in.iterrows():
        # Unchanged columns
        for c in data_in.columns:
            if c != column:
                out_col_index = data_out
                    .columns.get_loc(c)
                row_list.append(i)
                col_list.append(
                    out_col_index)
                data_list.append(1)

        # One-hot columns
        val = row[column]
        colname = f"{column}_{val}"
        if colname in data_out.columns:
            out_col_index = data_out.
                columns.get_loc(colname)
            row_list.append(i)
            col_list.append(
                out_col_index)
            data_list.append(1)

    shape = (row_count, col_count_out)
    self.tensor = csr_matrix((data_list,
         (row_list, col_list)), shape=
        shape)
```

This method is more explicit and provides better readability for cases where the logic needs to be adjusted.

2.4.4 Comparison of Methods

The two methods differ in the way they iterate over the data and construct the provenance tensor:

1. **Direct Mapping:** Uses a more direct approach by iterating over rows and columns, adding entries for both unchanged and one-hot encoded columns.
2. **Alternative Mapping:** Offers a more explicit approach with better readability and flexibility, but essentially performs the same operations as the direct method.

Both methods result in the same one-hot encoded tensor, but the choice of method depends on readability and performance requirements.

2.5 Joining DataFrames and Building Provenance Tensor

This section describes how to join two DataFrames on a specified column and build a provenance tensor. The function `join_data` allows you to merge two DataFrames and construct the provenance tensor based on the matching rows. Two methods for tensor construction are provided: `direct` and `alt`.

2.5.1 Method : Standard DataFrame Join

The `join_data` method joins two DataFrames (`left_df` and `right_df`) on a specified column (`on`) and constructs the provenance tensor based on the chosen method. The process is as follows:

1. Merge the two DataFrames using `pd.merge`, specifying the join type (`how`) and the column on which to join (`on`).
2. Depending on the chosen `method`, invoke either the `direct` or `alt` method to build the provenance tensor.

The implementation is as follows:

```
def join_data(self, left_df, right_df,
    on, how='inner', method='direct'):

    start_time = time.time()

    # (1) Merge the DataFrames
    data_out = pd.merge(left_df,
        right_df, on=on, how=how)

    # (2) Build the tensor
    if method == 'direct':
        self._join_data_direct(left_df,
            right_df, data_out, on)
    elif method == 'alt':
        self._join_data_alt(left_df,
            right_df, data_out, on)
    else:
        raise ValueError(f"Unknown
            join_data method: {method}")

    self.performance[f"join_data({method
        })"] = time.time() - start_time
    return data_out, self.tensor
```

This method serves as the general structure for joining DataFrames and delegating the construction of the provenance tensor to the specified method.

2.5.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method builds the provenance tensor by matching rows between the two DataFrames based on the join key (`on`). The steps are as follows:

1. Calculate the total number of rows in both `left_df` and `right_df`.
2. For each row in `data_out`, find the corresponding rows in `left_df` and `right_df` that match the join key.
3. Build the provenance tensor by mapping these matched rows to their corresponding positions in the `data_out` DataFrame.

The implementation is as follows:

```
def _join_data_direct(self, left_df,
    right_df, data_out, on):

    total_in_rows = len(left_df) + len(
        right_df)
    shape = (len(data_out),
        total_in_rows)
    rows, cols, data_vals = [], [], []

    left_map = {idx: i for i, idx in
        enumerate(left_df.index)}
    right_map = {idx: i + len(left_df)
        for i, idx in enumerate(right_df
        .index)}

    for out_i, row in data_out.iterrows
        ():
        key_val = row[on]
        left_matches = left_df.index[
            left_df[on] == key_val]
        right_matches = right_df.index[
            right_df[on] == key_val]

        for lm in left_matches:
            rows.append(out_i)
            cols.append(left_map[lm])
            data_vals.append(1)
        for rm in right_matches:
            rows.append(out_i)
            cols.append(right_map[rm])
            data_vals.append(1)

    self.tensor = csr_matrix((data_vals,
         (rows, cols)), shape=shape)
```

This method performs a direct matching of rows based on the join key and builds the provenance tensor by explicitly mapping matched rows to their corresponding positions.

2.5.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method uses hashing to map the join key and improve the efficiency of finding matching rows. The steps are as follows:

1. Add a new column containing the hash of the join key (`on`) for both `left_df`, `right_df`, and `data_out`.
2. For each row in `data_out`, find the corresponding rows in `left_df` and `right_df` based on the hash of the join key.

3. Build the provenance tensor by mapping the matched rows using their hash values.

4. Clean up by removing the hash column from the DataFrames.

The implementation is as follows:

```
def _join_data_alt(self, left_df,
    right_df, data_out, on):

    import hashlib

    total_in_rows = len(left_df) + len(
        right_df)
    shape = (len(data_out),
        total_in_rows)
    rows, cols, data_vals = [], [], []

    # Add hash column
    left_df["_hash_key"] = left_df[on].
        apply(lambda x: hashlib.md5(str(
        x).encode()).hexdigest())
    right_df["_hash_key"] = right_df[on
        ].apply(lambda x: hashlib.md5(
        str(x).encode()).hexdigest())
    data_out["_hash_key"] = data_out[on
        ].apply(lambda x: hashlib.md5(
        str(x).encode()).hexdigest())

    left_map = {idx: i for i, idx in
        enumerate(left_df.index)}
    right_map = {idx: i + len(left_df)
        for i, idx in enumerate(right_df
        .index)}

    for out_i, row in data_out.iterrows
        ():
        out_hash = row["_hash_key"]
        left_matches = left_df.index[
            left_df["_hash_key"] ==
            out_hash]
        right_matches = right_df.index[
            right_df["_hash_key"] ==
            out_hash]

        for lm in left_matches:
            rows.append(out_i)
            cols.append(left_map[lm])
            data_vals.append(1)
        for rm in right_matches:
            rows.append(out_i)
            cols.append(right_map[rm])
            data_vals.append(1)

    # Cleanup
    del left_df["_hash_key"]
    del right_df["_hash_key"]
    del data_out["_hash_key"]

    self.tensor = csr_matrix((data_vals,
         (rows, cols)), shape=shape)
```

This method improves the efficiency of matching rows by using hashing, which is particularly useful when dealing with large datasets.

2.5.4 Comparison of Methods

The two methods differ as follows:

1. **Direct Mapping:** This method explicitly matches rows based on the join key and builds the provenance tensor by mapping matched rows to their corresponding positions.

2. **Alternative Mapping:** This method improves efficiency by hashing the join key, making it faster to find matching rows in large datasets.

Both methods construct the same provenance tensor, but the choice between them depends on the size of the dataset and the computational constraints.

2.6 Appending Data and Constructing Provenance Tensor

This section describes how to vertically concatenate two DataFrames (`df1` and `df2`) and construct a provenance tensor using two different methods: `direct` and `alt`. The `append_data` method is used for concatenation and tensor construction.

2.6.1 Method : Data Concatenation and Provenance Tensor Construction

The general process involves vertically concatenating two DataFrames and constructing the provenance tensor based on the specified method.

1. Vertically concatenate `df1` and `df2` to form `data_out`.

2. Depending on the `method` argument, invoke one of the two tensor construction methods: `direct` or `alt`.

The implementation of the `append_data` method is as follows:

```
def append_data(self, df1, df2, method='
    direct'):

    start_time = time.time()

    # (1) Generate data_out
    data_out = pd.concat([df1, df2]).
        reset_index(drop=True)

    # (2) Build the tensor
    if method == 'direct':
        self._append_direct(df1, df2,
            data_out)
    elif method == 'alt':
        self._append_alt(df1, df2,
            data_out)
    else:
        raise ValueError(f"Unknown
            append_data method: {method}
            ")

    self.performance[f"append_data({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method handles the data concatenation and delegates the task of constructing the provenance tensor to the appropriate helper function based on the specified method.

2.6.2 Method 1: Direct Mapping (Method 'direct')

The `direct` method constructs the provenance tensor by mapping each row in `data_out` to its corresponding index in `df1` or `df2`, based on its position.

1. The total number of rows in the concatenated DataFrame is computed as the sum of the rows in `df1` and `df2`.

2. A mapping is created for the indices of `df1` and `df2` to keep track of the provenance.

3. The provenance tensor is built by iterating over `data_out` and assigning each row in `data_out` to its corresponding index in `df1` or `df2`.

The implementation is as follows:

```
def _append_direct(self, df1, df2,
    data_out):

    total_in_rows = len(df1) + len(df2)
    shape = (len(data_out),
        total_in_rows)
    rows, cols, data_vals = [], [], []

    df1_map = {idx: i for i, idx in
        enumerate(df1.index)}
    df2_map = {idx: i + len(df1) for i,
        idx in enumerate(df2.index)}

    for out_i in range(len(data_out)):
        if out_i < len(df1):
            original_idx = df1.index[
                out_i]
            rows.append(out_i)
            cols.append(df1_map[
                original_idx])
            data_vals.append(1)
        else:
            out_i_in_df2 = out_i - len(
                df1)
            original_idx = df2.index[
                out_i_in_df2]
            rows.append(out_i)
            cols.append(df2_map[
                original_idx])
            data_vals.append(1)

    self.tensor = csr_matrix((data_vals,
        (rows, cols)), shape=shape)
```

This method provides a simple and efficient way to map rows in the concatenated DataFrame to their respective sources, `df1` and `df2`.

2.6.3 Method 2: Alternative Mapping (Method 'alt')

The `alt` method constructs the provenance tensor in a slightly different way, iterating over `df1` and `df2` explicitly and tracking the row locations.

1. Similar to the `direct` method, the total number of rows is calculated.

2. A list of row and column indices is created by iterating over both `df1` and `df2`.

3. The provenance tensor is built by iterating over the rows of both DataFrames and appending the indices to the list.

The implementation is as follows:

```
def _append_alt(self, df1, df2, data_out
    ):
    total_in_rows = len(df1) + len(df2)
    shape = (len(data_out),
        total_in_rows)
    row_list, col_list, data_list = [],
        [], []

    # Iterate over df1
    curr_out = 0
    for idx in df1.index:
        row_list.append(curr_out)
        col_list.append(df1.index.
            get_loc(idx))
        data_list.append(1)
        curr_out += 1

    # Iterate over df2
    for idx in df2.index:
        row_list.append(curr_out)
        col_list.append(len(df1) + df2.
            index.get_loc(idx))
        data_list.append(1)
        curr_out += 1

    self.tensor = csr_matrix((data_list,
         (row_list, col_list)), shape=
        shape)
```

This method provides a more explicit way of constructing the tensor, and is useful when you need more control over the iteration process.

2.6.4 Comparison of Methods

The two methods differ in their approach to constructing the provenance tensor:

1. **Direct Method:** Efficiently maps rows from `data_out` to `df1` and `df2` using a modulo-based mapping, which is computationally efficient for large datasets.

2. **Alternative Method:** Explicitly iterates over the rows of both DataFrames, providing more control and flexibility but potentially less efficient for large datasets.

Both methods result in the same provenance tensor, and the choice between them depends on the specific needs of the task.

2.7 Data Transformation Methods

This section describes two methods for transforming data in a DataFrame by filling missing values (NaN) in a specified column. The function `transform_data` is used to perform the transformation and also constructs a provenance tensor.

2.7.1 Method : Standard Transformation

The standard method performs a data transformation by filling missing values in the specified column with the mean of that column in the input DataFrame (`data_in`). The steps are as follows:

1. Generate the output DataFrame (`data_out`) by copying the input DataFrame (`data_in`).

2. Depending on the `method` argument, invoke one of the two specialized data transformation methods: `direct` or `alt`.

3. Construct the provenance tensor, which is a diagonal matrix where each row in `data_out` comes directly from the corresponding row in `data_in`.

The implementation is as follows:

```
def transform_data(self, data_in, column
    , method='direct'):
    import time
    start_time = time.time()

    # (1) Generate data_out (copy of
        data_in)
    data_out = data_in.copy()

    # Depending on the method, apply
        different fillna strategies
    if method == 'direct':
        self._transform_data_direct(
            data_in, data_out, column)
    elif method == 'alt':
        self._transform_data_alt(data_in
            , data_out, column)
    else:
        raise ValueError(f"Unknown
            transform_data method: {
            method}")

    # (2) Build the provenance tensor
    # Since the schema and number of
        rows do not change,
    # each row out_i directly comes from
         the same row in_i.
    # => Create a diagonal matrix (1 on
        the diagonal).
    num_rows = len(data_in)
    shape = (num_rows, num_rows)
    rows = np.arange(num_rows)
    cols = np.arange(num_rows)
    data_vals = np.ones(num_rows)  # 1
        on each diagonal
    self.tensor = csr_matrix((data_vals,
         (rows, cols)), shape=shape)

    self.performance[f"transform_data({
        method})"] = time.time() -
        start_time
    return data_out, self.tensor
```

This method allows for flexibility in filling missing values and provides a simple provenance tensor that indicates the origin of each row.

2.7.2 Method 1: Direct Transformation (Method 'direct')

The `direct` method fills missing values (NaN) in the specified column with the mean of that column in the input DataFrame. The steps are as follows:

1. Calculate the mean value of the specified column in `data_in`.
2. Replace the missing values in `data_out` with the calculated mean value.

The implementation is as follows:

```
def _transform_data_direct(self, data_in
    , data_out, column):
    """
    Direct method:
    Replaces NaN in 'column' with the
        mean of 'column' in data_in.
    """
    mean_val = data_in[column].mean()  #
        Mean of the column
    data_out[column] = data_out[column].
        fillna(mean_val)
```

This method is typically used when the mean of the column is a reasonable substitute for missing values.

2.7.3 Method 2: Alternative Transformation (Method 'alt')

The `alt` method fills missing values (NaN) in the specified column with the median of that column in the input DataFrame. This method can be extended to use other strategies, but here it uses the median as an alternative to the mean. The steps are as follows:

1. Calculate the median value of the specified column in `data_in`.
2. Replace the missing values in `data_out` with the calculated median value.

The implementation is as follows:

```
def _transform_data_alt(self, data_in,
    data_out, column):
    """
    Alternative method:
    Replaces NaN in 'column' with the
        median of 'column' in data_in.
    (Or another criterion, just to
        illustrate an alternative).
    """
    median_val = data_in[column].median
        ()
    data_out[column] = data_out[column].
        fillna(median_val)
```

This method is useful when the median is a more robust measure than the mean, especially in the presence of outliers.

2.7.4 Comparison of Methods

The two methods differ as follows:

1. **Standard Transformation:** Provides the basic structure for transforming data and building the provenance tensor, with the choice of transformation method left to the user.
2. **Direct Transformation:** Fills missing values with the mean of the specified column in the input DataFrame.
3. **Alternative Transformation:** Fills missing values with the median of the specified column, providing a more robust alternative to the mean.

Each method constructs the provenance tensor as a diagonal matrix, where each row in the output DataFrame comes from the same row in the input DataFrame, and the diagonal represents this direct mapping.

2.8 Performance Evaluation

This method returns the performance metrics of the operations that have been executed so far. It accesses a dictionary that tracks the execution times of various operations.

1. `evaluate_performance` provides a simple way to retrieve the performance data for the operations performed by the class. It does not take any arguments.
2. It returns the `performance` dictionary, which contains the timing information for each operation.

The implementation is as follows:

```
def evaluate_performance(self):
    """Returns the performance of the
        operations performed."""
    return self.performance
```

This method is useful for performance monitoring, especially when working with large datasets or complex transformations, as it allows you to track the time taken for each operation.

2.9 Converting Sparse Tensor to Dense Format

This method converts the sparse tensor to a dense format using NumPy arrays. It returns the dense representation of the tensor if it exists, or `None` if no tensor is present.

1. `get_tensor_dense` checks if the tensor is not `None`. If it is not, it converts the sparse matrix to a dense NumPy array using the `toarray()` method.
2. If the tensor is `None`, it simply returns `None`.

The implementation is as follows:

```python
def get_tensor_dense(self):
    """Returns the sparse binary tensor
        in dense form (numpy array)."""
    if self.tensor is not None:
        return self.tensor.toarray()
    else:
        return None
```

This method is useful for cases where you need to perform operations on the tensor that require a dense representation, such as certain mathematical operations or when working with libraries that do not support sparse matrices. However, dense matrices consume more memory, so it should be used cautiously, especially with large tensors.

3 Examples and Performances

The following examples in this section will provide detailed descriptions of the methods previously discussed. Each example aims to demonstrate the practical application of the techniques covered earlier, showcasing how they can be used to manipulate and transform data in various ways. These methods, which include operations such as filtering, column removal, one-hot encoding, and data joining, are fundamental in data preprocessing and are widely used in data science workflows.

```
=== Différences entre DataFrame original et filtré ===

Lignes filtrées (Direct) :
        ID     city  val  cuisine
4        5   madrid  742  italian
8        9   berlin  845  italian
11      12   berlin  821  italian
14      15    tokyo  998  italian
15      16    paris  519  italian
...    ...      ...  ...      ...
9981  9982  atlanta  749  italian
9983  9984  atlanta  151  italian
9984  9985  atlanta  621  italian
9996  9997   berlin  338  italian
9998  9999    tokyo  858  italian

[1993 rows x 4 columns]

Lignes non filtrées (Direct) :
         ID    city  val   cuisine
0         1   tokyo  771  japanese
1         2  madrid  199    french
2         3  berlin  720    french
3         4  madrid  493  japanese
...
9998   9999   tokyo  858   italian
9999  10000  berlin  903  japanese

[10000 rows x 4 columns]
```

Figure 1: Difference between the original dataframe and the filtered one. The filtered dataframe excludes rows that do not meet the specified condition, reducing the dataset size.

```
=== Différences entre DataFrame original et après Drop ===

Colonnes avant suppression :
Index(['ID', 'city', 'val', 'cuisine'], dtype='object')

Colonnes après suppression (Direct) :
Index(['ID', 'val', 'cuisine'], dtype='object')

Colonnes après suppression (Alt) :
Index(['ID', 'val', 'cuisine'], dtype='object')
```

Figure 2: Difference between the columns of the original dataframe and after the drop. This shows how certain columns have been removed from the dataset based on the specified list of columns to drop.

DataFrame avant One-Hot Encoding (Direct):

	ID	city	val	cuisine
0	1	tokyo	771	japanese
1	2	madrid	199	french
2	3	berlin	720	french
3	4	madrid	493	japanese
4	5	madrid	742	italian

Figure 3: Dataframe before the One-Hot Encoding. This represents the original dataframe where categorical variables have not yet been converted to numerical values.

DataFrame après One-Hot Encoding (Direct):

	ID	city	val	cuisine_american	cuisine_french	cuisine_german	cuisine_italian	cuisine_japanese
0	1	tokyo	771	0	0	0	0	1
1	2	madrid	199	0	1	0	0	0
2	3	berlin	720	0	1	0	0	0
3	4	madrid	493	0	0	0	0	1
4	5	madrid	742	0	0	0	1	0

Figure 4: Dataframe after the One-Hot Encoding. The categorical columns are transformed into binary columns representing each category with a 1 or 0.

```
=== DataFrames avant et après Join ===

DataFrame 1 (df) original:
   ID    city  val   cuisine
0   1   tokyo  771  japanese
1   2  madrid  199    french
2   3  berlin  720    french
3   4  madrid  493  japanese
4   5  madrid  742   italian

DataFrame 2 (df2) original:
     ID  other_val   cuisine
0  9000       2101   italian
1  9001       3099    french
2  9002       1513    french
3  9003       2832  japanese
4  9004       8137  japanese
```

Figure 5: The two dataframes before the join. This shows the original dataframes before any merging or joining operations, illustrating their independent structures.

DataFrame après Join (Direct):

	ID	city	val	cuisine_x	other_val	cuisine_y
0	9000	madrid	666	french	2101	italian
1	9001	berlin	922	italian	3099	french
2	9002	paris	618	japanese	1513	french
3	9003	berlin	797	american	2832	japanese
4	9004	paris	151	french	8137	japanese

Figure 6: The two dataframes after the join. This represents the dataframes after they have been joined based on a common column, combining the relevant data from both sources.

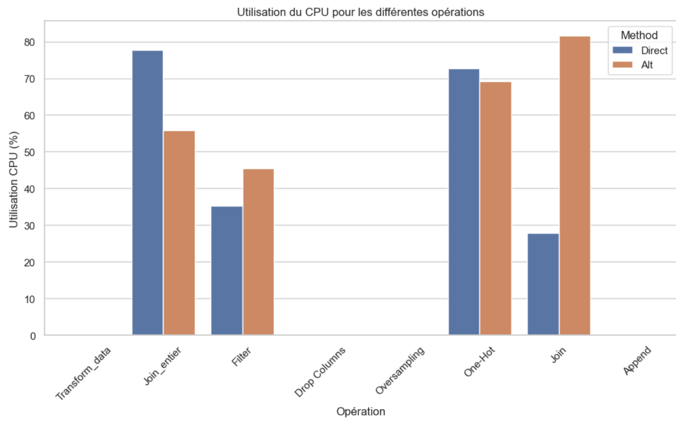

Figure 7: We notice that the join entier operation uses much more CPU with the direct method than with the ALT method. However, the ALT method uses significantly more CPU in the join and filter operations compared to the direct method.

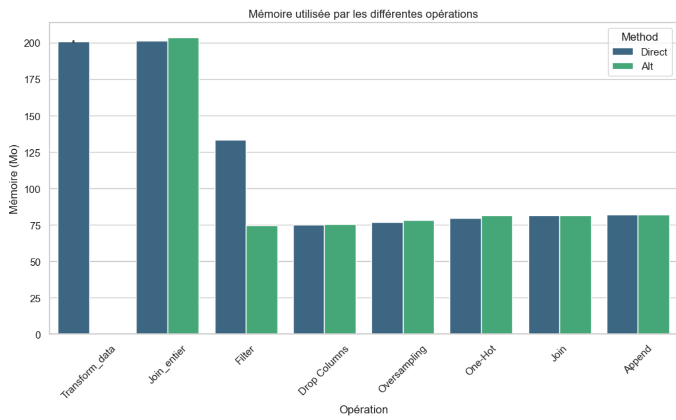

Figure 8: The only difference in memory usage is observed in the transform data and filter operations, where the direct method consumes significantly more memory.

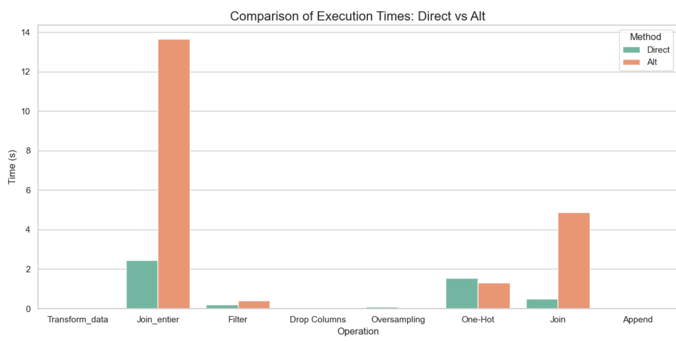

Figure 9: The alt method takes significantly more execution time in the join operations.