Data Quality: MiniProject

Khalid Belhajjame

Data Preparation Operations

Data Processing Operation	**Category**
Instance Selection	Horizontal Data Reduction
Drop Rows	
Undersampling	
Feature Selection	Vertical Data Reduction
Drop Columns	
Imputation	Data Transformation
Value transformation	
Binarization	
Normalization	
Discretization	
Instance Generation	Horizontal Data Augmentation
Oversampling	
Space Transformation	Vertical Data Augmentation
String Indexer	
One-Hot Encoder	
Join	Data Fusion
Append	

Objective

- Define a way to capture the provenance of the following operations using tensors in an efficient manner on large dataset
  - Fitler operation (horizontal reduction)
  - Oversampling (horizontal augmentation)
  - Join (data fusion)
  - Union (data fusion)
- Two methods for capturing the provenance:
  1. Given an operation, the input dataset(s) and the output dataset, derive the tensor capturing the provenance
  2. Modify the operation so as to make the capture of the provenancemore efficient
  - We will opt for the second (by usoing the decorator design pattern)
- We will be using sparse binary tensors

Data Transformation

Data Transformation Operations in this category neither alter the schema of the dataset nor the number of records. Instead, they modify specific attribute values by applying a transformation function.

```
output_df = input_df.copy()
```

```
output_df['Age'] = input_df['Age'].fillna(input_df['Age'].mean())
```

	Name	Age	Salary
0	Alice	25.0	50000.0
1	Bob	NaN	60000.0
2	Charlie	30.0	NaN
3	David	35.0	80000.0

	Name	Age	Salary
0	Alice	25.0	50000.0
1	Bob	30.0	60000.0
2	Charlie	30.0	NaN
3	David	35.0	80000.0

Vertical Data Reduction

Vertical Reduction There are two operations that fall in this category, namely Feature Selection and Drop Columns. Both of this operations remove some of the attributes characterizing the data records in the input dataset D^{in} and produce a next dataset D^{out} that reflects the dataset obtained as a result.

```
df_reduced = df[['Name', 'Salary']]
```

```
Original DataFrame:
      Name  Age  Salary
0    Alice   24   70000
1      Bob   17   40000
2  Charlie   35  120000
3    David   45  110000
4      Eve   19   50000

Reduced DataFrame with selected features ('Name' and 'Salary'):
      Name  Salary
0    Alice   70000
1      Bob   40000
2  Charlie  120000
3    David  110000
4      Eve   50000
```

Horizontal Data Reduction

Horizontal Reduction Given a dataset D^{in}, an operation that performs horizontal reduction produces a new dataset D^{out}, where the data records in D^{out} are subsets of those in D^{in}: $D^{out} \subseteq D^{in}$. Data manipulations that fall into this category include the following operations: filtering, instance selection, row deletion, and undersampling.

```
df_reduced = df[df['Age'] >= 25]
```

```
Original DataFrame:
      Name  Age  Salary
0    Alice   24   70000
1      Bob   17   40000
2  Charlie   35  120000
3    David   45  110000
4      Eve   19   50000

Reduced DataFrame with rows where Age >= 25:
      Name  Age  Salary
2  Charlie   35  120000
3    David   45  110000
```

Vertical Data Augmentation

Vertical Data Augmentation Operations in this category include Space Transformation, String Indexer, and One-Hot Encoder. Given an input dataset D^{in}, applying vertical data augmentation produces a dataset D^{out} with a different schema from D^{in}. However, D^{in} and D^{out} have the same number of records, with the i^{th} record in D^{out} corresponding to the i^{th} record in D^{in}.

```
one_hot_encoded = pd.get_dummies(df['City'], prefix='City')
df_with_encoding = pd.concat([df, one_hot_encoded], axis=1)
```

	Name	City
0	Alice	Paris
1	Bob	London
2	Charlie	Paris
3	David	Berlin

	Name	City	City_Berlin	City_London	City_Paris
0	Alice	Paris	0	0	1
1	Bob	London	0	1	0
2	Charlie	Paris	0	0	1
3	David	Berlin	[illegible]	0	0

Horizontal Data Augmentation

Horizontal Data Augmentation Operations that fall into this category are Instance Generation and Oversampling.

```
# Apply SMOTE
X_resampled, y_resampled = SMOTE().fit_resample(X, y)

# Create the Output DataFrame
output_df = pd.concat([pd.DataFrame(X_resampled, columns=X.columns),
                       pd.Series(y_resampled, name='Category')], axis=1)
```

	Age	Income	Category
0	25	40000	0
1	30	50000	0
2	35	60000	0
3	40	70000	1
4	45	80000	1

	Age	Income	Category
0	25.0	40000.0	0
1	30.0	50000.0	0
2	35.0	60000.0	0
3	40.0	70000.0	1
4	45.0	80000.0	1
5	42.5	75000.0	1
6	43.8	78000.0	1
7	41.3	71000.0	1

Join

Join The join of the datasets D^l and D^r, implemented using the Merge operation in the Pandas library, and denoted by $D^l \bowtie_C^t D^r$, produces a dataset D^j as a result of joining D^l and D^r on a boolean condition C, where t represents the join type (inner, left outer, right outer, or full outer).

Table 2.2: Dataset D^l

	ID	Birthdate	Gender	Postcode
1	10	1996-07-12	F	90210
2	20	1994-03-08	M	$\perp$
3	30	$\perp$	F	12345
4	40	1987-11-23	M	67890

Table 2.3: D^r Dataset

	ID	Name
1	20	Alice
2	40	Bob

Table 2.4: D^j dataset obtained by the following join $D^l \bowtie_{\text{inner}} D^r$

	ID	Birthdate	Gender	Postcode	Name
1	20	1994-03-08	M	$\perp$	Alice
2	40	1987-11-23	M	67890	Bob

Append

Append The append operation, implemented using *Concat* in the Pandas library, appends the records of a dataset D^l at the end of the D^r, denoted by $D^l \uplus D^r$. The two datsets do not need to have the same schema, and as such the results are extended with null for the mismatching attributes.

Table 2.2: Dataset D^l

	ID	Birthdate	Gender	Postcode
1	10	1996-07-12	F	90210
2	20	1994-03-08	M	$\perp$
3	30	$\perp$	F	12345
4	40	1987-11-23	M	67890

Table 2.3: D^r Dataset

	ID	Name
1	20	Alice
2	40	Bob

Table 2.5: D^a dataset obtained by the following append $D^l \uplus D^r$

	ID	Birthdate	Gender	Postcode	Name
1	10	1996-07-12	F	90210	$\perp$
2	20	1994-03-08	M	$\perp$	$\perp$
3	30	$\perp$	F	12345	$\perp$
4	40	1987-11-23	M	67890	$\perp$
5	20	$\perp$	$\perp$	$\perp$	Alice
6	40	$\perp$	$\perp$	$\perp$	Bob

Objective of the project

Task 1: To develop a python class (which we could name SITNProv) that can be used to infer the provenance of each of the operations just presented.

Given the input data frame(s), output data frame and the kind of the operation (vertical reduction, horizontal reduction, etc.), construct a tensor that informs on the provenance of the data records of the output data frames and how they depends on the input data frames.

We will be using **binary sparse tensors**.

We will be using tensors, specifically binary sparse tensors, to capture the provenance

Join The join of the datasets D^l and D^r, implemented using the Merge operation in the Pandas library, and denoted by $D^l \bowtie_C^t D^r$, produces a dataset D^j as a result of joining D^l and D^r on a boolean condition C, where t represents the join type (inner, left outer, right outer, or full outer).

Table 2.2: Dataset D^l

	ID	Birthdate	Gender	Postcode
1	10	1996-07-12	F	90210
2	20	1994-03-08	M	$\perp$
3	30	$\perp$	F	12345
4	40	1987-11-23	M	67890

Table 2.3: D^r Dataset

	ID	Name
1	20	Alice
2	40	Bob

$$T = \left(\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \right)$$

Table 2.4: D^j dataset obtained by the following join $D^l \bowtie_{\text{inner}} D^r$

	ID	Birthdate	Gender	Postcode	Name
1	20	1994-03-08	M	$\perp$	Alice
2	40	1987-11-23	M	67890	Bob

The SITNProv Class

```
class SITNProv:
def __init__(self, func):
    self.func = func
  def __call__(self, *args, **kwargs):
    method_name = self.func.__name__
    if method_name == "filter":
      return self.decorate_filter(*args, **kwargs)
    elif method_name == "merge":
      return self.decorate_merge(*args, **kwargs)
    else:
      raise NotImplementedError(f"Decoration for '{method_name}' is not implemented.")
  def decorate_filter(self, df, condition):
    """Decorates the filter operation."""
    # Add logic to track input-output relationships and create a provenance tensor
    pass

  def decorate_merge(self, df1, df2, **kwargs):
    """Decorates the merge operation."""
    # Add logic to track input-output relationships and create a provenance tensor
    pass
```

Usage of the SITNProv Class

This is an example, you can develop the class differently

```
# Create an instance of the TensorProv decorator for pandas operations
filter_with_prov = SITNProv(pd.DataFrame.query)  # Assuming filtering is done via `query`
merge_with_prov = SITNProv(pd.merge)

# Example DataFrames
df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8]})
df1 = pd.DataFrame({'key': [1, 2], 'value': ['A', 'B']})
df2 = pd.DataFrame({'key': [2, 3], 'value': ['C', 'D']})

# Filter example
condition = 'A > 2'
filtered_df, filter_tensor = filter_with_prov(df, condition)
print("Filtered DataFrame:")
print(filtered_df)
print("Filter Tensor:")
print(filter_tensor)

# Merge example
merged_df, merge_tensor = merge_with_prov(df1, df2, on='key', how='inner')
print("\nMerged DataFrame:")
print(merged_df)
print("Merge Tensor:")
print(merge_tensor)
```

Objective of the project

Task 2: Using the tensors that captre the provenance, develop operations on tensors that can be used to for querying provenance information. That is connect given output records with the corresponding input records, and vice versa.

1. You can start by performing this operation for a single operarion (single task)
2. Go on to show how this can be performed to succeedings operations with a data pipeline

Logistics

- You will work in teams of 2
  - You can also work on your own if you prfere
- Develop the SITNProv class
- Develop methods for querying the provenance captred by the tensor
- You can start by using examles of simple operations in Pythoon
- For more complete use cases for both the capture and querying of provenance, you can use the processing pipeline provided within the zip project:
  - Compass data pipeline
  - German data pipeline
  - Census data pipeline
- Assess the performance in term of storage space required for storing the tensors, as well as the processing time required for provenance queries