Privacy Proofs for OpenDP: Impute Constant Transformation

Grace Tian

Summer 2021

Contents

1 Algorithm Implementation

1.1 Code in Rust

The current OpenDP library contains the `make_impute_constant` function implementing the impute constant function. This is defined in lines 62-75 of the file `impute.rs` in the Git repository (https://github.com/opendp/opendp/blob/21-impute/rust/opendp/src/trans/impute.rs#L62-L75).

```
58 /// A [`Transformation`] that imputes elementwise with a constant value.
59 /// Maps a Vec<Option<T>> -> Vec<T> if input domain is AllDomain<Option<T>>,
60 ///     or Vec<T> -> Vec<T> if input domain is NullableDomain<AllDomain<T>>
61 /// Type argument DA is "Domain of the Atom"; the domain type inside VectorDomain.
62 pub fn make_impute_constant<DA, M>(
63     constant: DA::NonNull
64 ) -> Fallible<Transformation<VectorDomain<DA>, VectorDomain<AllDomain<DA::NonNull>>, M, M>>
65     where DA: ImputableDomain,
66           DA::NonNull: 'static + Clone,
67           DA::Carrier: 'static,
68           M: DatasetMetric {
69     if DA::is_null(&constant) { return fallible!(MakeTransformation, "Constant may not be null.") }
70 
71     make_row_by_row(
72         DA::new(),
73         AllDomain::new(),
74         move |v| DA::impute_constant(v, &constant).clone())
75 }
```

1.2 Pseudo Code in Python

Preconditions

To ensure the correctness of the output, we require the following preconditions:

- **User-specified types:**
  - Variable `constant` must be of type `DA::NonNull`
  - Type `DA` must have traits `ImputableDomain`.
  - `DA::NonNull` must have traits `Clone`

Postconditions

- Either a valid `Transformation` is returned or an error is returned.
- (grace) Should I also say that `impute_constant` must satisfy the preconditions of make row by row? (See pseudo code 19.) That is, `impute_constant` must be a pure function?

```
1  def make_impute_constant(constant : DA::NonNull):
2      # instead of VectorDomain(DA), we add ::new() to get a new instance of
       DA. This is bcause DA has the ImputableDomain trait.
3      input_domain = VectorDomain(DA::new())
4      output_domain = VectorDomain(AllDomain(DA::NonNull))
5      input_metric = SymmetricDistance()
6      output_metric = SymmetricDistance()
7
8      assert(not constant.is_null); # not DA::is_null(constant)
9
10     def Relation(d_in: u32, d_out: u32) -> bool:
11         return d_out >= d_in*1
12
13     def function(data: Vec[DA::Carrier]) -> Vec[DA::NonNull]:
14         def impute_constant(x: DA) -> DA::NonNull:
15             return constant if x.is_null else x
16         return list(map(impute_constant, data))
17
18     # can we comment out input_metric, output_metric, and
       stability_relation, and function if it's not being called anymore?
19     return make_row_by_row(input_domain, output_domain, impute_constant);
```

1.3 Proof

Lemma 1.1 (`DA::NonNull` contains `null`). *A `var` of type `DA::NonNull` can be of type `null`.*

Proof. Let the domain of atom variable `DA` be `InherentNullDomain<AllDomain<f64>>`. Recall that `InherentNullDomain` exists for types that can represent null inherently in the carrier type. Then the type

`DA::NonNull == InherentNullDomain<AllDomain<f64>>::NonNull == f64`.

The latter holds because in the `InherentNullDomain` implementation in the rust code https://github.com/opendp/opendp/blob/main/rust/opendp/src/trans/impute.rs#L48-L56, the `type NonNull = Self::Carrier`. The `Carrier` of `VectorDomain<AllDomain<T>>` has type `T`, so in this case the `::Carrier` is type `f64`.

Therefore `var` is also of type `f64`. `f64` can contain `null` values, so we are done. ☐

Lemma 1.2 (Precondition for row transform)**.** *`make_impute_constant` satisfies the preconditions of `make_row_by_row`. That is, the atom function `impute_constant` in pseudo code line 14 is a pure function.*

Proof. We reference the examples in Pure function Wikipedia page for the definition of a pure function, and later as a check list to verify the properties of a pure function hold. To verify whether `impute_constant` is a pure function, it must satisfy the following properties:

1. The function return values are identical for identical arguments.

2. The function application has no side effects.

The first property is satisfied because there is no static variable defined within `make_impute_constant`, no mutable reference argument, no return value of an input stream, all of which could potentially cause different function outputs for inputs. Even though the function returns a non-local variable `constant`, `constant` is never changed within the code of `make_impute_function`, so the return value stays the same for same input to `impute_constant`, all of which could produce side effects.

The second property is satisfied because the local static variables and non-local variables are unchanged, and there are no mutable reference arguments or input/output streams called within the `impute_constant`. □

Theorem 1.3. *For every setting of the input parameters `constant` to `make_impute_constant` such that the given preconditions hold, the transformation returned by `make_impute_constant` has the following properties:*

1. (Appropriate output domain). *If vector v is in the `input_domain`, then `function(v)` is in the `output_domain`.*

2. (Domain-Metric Compatibility). *The domain `input_domain` matches one of the possible domains listed in the definition of `input_metric`, and likewise `output_domain` matches one of the possible domains listed in the definition of `output_metric`.*

3. (Stability Guarantee). *For every pair of elements v, w in `input_domain` and for every pair (`d_in`, `d_out`), where `d_in` is of the associated type for `input_metric` and `d_out` is the associated type for `output_metric`, if v, w are d_{in}-close under `input_metric` and `Relation`(`d_in`, `d_out`) = `True`, then `function`(v), `function`(w) are d_{out}-close under `output_metric`.*

Proof. 1. **(Appropriate output domain).**

In the case of `make_impute_constant`, this corresponds to showing that for every vector v of elements of type `DA::Carrier`, `function`(v) is a vector of elements of type `DA::NonNull`. We can also say that `function`(v) is a vector of elements that does not contain any `NonNull` values.

The `function`(v) has type `Vec[DA::NonNull]` follows from the assumption that element v is in `input_domain` and from the type signature of `function` in line 13 of the pseudocode (Section 1.2). The type signature takes in an element of type `Vec[DA::Carrier]` and returns an element of type `Vec[DA::NonNull]`. If the Rust code compiles correctly, then the type correctness follows from the definition of the type signature enforced by Rust. Otherwise, the code will halt at compile time.

The type signature is not a sufficient check, since by Lemma 1.1, the function's output type can represent a value (e.g. `float nan`) that is not a member in the `output_domain`. (grace) How do I clarify that the output domain `VectorDomain(AllDomain(DA::NonNull))` should actually contain NO NULLS? The code seems too loose because in Lemma 1.1 we just showed that `DA::NonNull` can contain null values. To ensure that function returns the appropriate output domain at run time, we must ensure that at run time the `function`(v) is not null. To do so, we must check whether the `constant` is not null.

Since the `constant` has type `DA::NonNull` and `DA::NonNull` may itself be `f64` by Lemma 1.1, without an extra check `constant` can be null. We must check whether `constant` is null because otherwise if `constant` is null, the user can nonsensically impute `null` entries in a vector with `null` value.

We check whether the constant is null in pseudo code line 8, before it gets called in `function`. If `constant` is `null`, then the check raises a construction-time error, so the function is never run. Thus we are done.

(grace) I didn't use the fact that v is of type DA::Carrier; am I doing something wrong?

2. **(Domain-metric compatibility).** The Symmetric distance is both the `input_metric` and `output_metric`. Symmetric distance is compatible with `VectorDomain(T)` for any generic type `T`, as stated in "List of definitions used in the pseudocode". The theorem holds because for `make_impute_constant`, the input domain is `VectorDomain(DA)` and the output domain is `VectorDomain(AllDomain(DA::NonNull))`.

3. **(Stability guarantee).** We know that $d_{in} \leq d_{out}$ because `Relation`(d_{in}, d_{out}) `= True`. Since the vectors v, w are d_{in}-close, then $d_{Sym}(v, w) \leq d_{in}$.

   Recall that `make_impute_constant` satisfies the pure function precondition of `make_row_by_row` in Lemma 1.2 and `make_impute_constant` is a row transform (pseudocode line 19). Then we can apply the stability guarantee of `make_row_by_row`, which gives us

$$d_{Sym}(\texttt{function}(v), \texttt{function}(w)) \leq d_{Sym}(v, w).$$

   Therefore the transformation is d_{out}-close: $d_{sym}(\texttt{function}(v), \texttt{function}(w)) = d_{sym}(v, w) \leq d_{in} \leq d_{out}$

$\square$