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.

```
def make_impute_constant(constant : DA::NonNull):
    # instead of VectorDomain(DA), we add ::new() to get a new instance of DA. This is bcause DA has the ImputableDomain trait. Discuss among interns.
    input_domain = VectorDomain(DA::new())
    output_domain = VectorDomain(AllDomain(DA::NonNull))
    input_metric = SymmetricDistance()
    output_metric = SymmetricDistance()

    # check constant for nullity first
    assert(not constant.is_null); # not DA::is_null(constant)
    def Relation(d_in: u32, d_out: u32) -> bool:
        return d_out >= d_in*1

    def function(data: Vec[DA::Carrier]) -> Vec[DA::NonNull]:
        def impute_constant(x: DA) -> DA::NonNull:
            return constant if x.is_null else x
        return list(map(impute_constant, data))

    return Transformation(input_domain, output_domain, function, input_metric, output_metric, stability_relation=Relation)
    # return make_row_by_row(input_domain, output_domain, impute_constant);
```

(grace) Will need to change pseudocode so that it returns the result of a make row by row transformation (which the code does) instead of a Transformation directly.

1.3 Proof

(grace) Should this go in the proofs list? (grace) I'm also confused why we have implementation of DA::NonNull but it doesn't do what the name says. Seems pretty counter intuitive.

Lemma 1.1 (`DA::NonNull` contains `null`)**.** *`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. □

Theorem 1.2. *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).** (grace) Things to confirm: (1) is the type signature correct (input and output type)? The input and output domain correct? (2) What is the reason the type signature is not sufficient? Is it because the output type is more general than the output domain? Or is it theoretically because constant can be null value even if it has the type DA::NonNull? (3) so, is it correct that a variable with type DA::NonNull can contain null values?

In the case of `make_impute_constant`, this corresponds to showing that for every vector v of elements of type `DA`, `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)` 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), which takes in an element of type `Vec(DA)` 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 raises an exception for incorrect input type.

Lemma 1.1 tells us the type signature is not sufficient check because (1) the output type of `VectorDomain(DA::NonNull)` is more general than the output domain of `VectorDomain(DA::NonNull)`. (grace) ?? The output type matches the output domain I wrote, so something's wrong. (2) `constant` can be `null` even if it is of type `DA::NonNull`. For (2), if the constant is null, then `impute_constant` function (line **??**) could potentially return `null` values since it replaces a `null` input with `constant`.

(1) and (2) are resolved because in the Rust code and pseudo code we check whether `constant` is null in pseudocode line 9.

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

   The function transformation just replaces the `null` element in vectors v and w with `constant`. Since the null element is also counted toward the symmetric distance of the transformation, the symmetric distance of `function`(v) and `function`(w) stays the same. Therefore the transformation is d_{out} close: $d_{sym}(\texttt{function}(v), \texttt{function}(w)) = d_{sym}(v, w) \leq d_{in} \leq d_{out}$

$\square$