Estimating Confidence Intervals

Our trained generative causal models and specific sample set might not accurately represent the ground truth and there are typically numerical approximations in our algorithms. When we answer a causal query without computing its confidence intervals, we essentially obtain point estimates. However, these are not very useful when assessing the confidence in our results. Confidence intervals assist us in quantifying the uncertainty in our results introduced by modeling inaccuracies and numerical approximations.

We therefore recommend to use confidence intervals as a default for all causal queries. This avoids a false sense of confidence in a specific value that might be due to a skewed data set or models that are not fitted optimally.

To compute confidence intervals, we use a method called bootstrapping and it's implemented in the function `dowhy.gcm.confidence_intervals()`.

How to use it

Let's say we have a function that computes Pi using the Monte Carlo method with 1000 trials:

```
>>> import numpy as np
>>>
>>> def compute_pi_monte_carlo():
>>>     trials = 1000
>>>     return 4*(np.random.default_rng().uniform(-1, 1, (trials,))**2+
>>>               np.random.default_rng().uniform(-1, 1, (trials,))**2 <= 1).sum()
```

Executing this function will give us a result:

```
>>> compute_pi_monte_carlo()
3.188
```

Skip to main content

This looks like a reasonable number, but we can't tell how reliable that result is, given that it was computed using a stochastic method. Now, let's use `dowhy.gcm.confidence_intervals()` to compute the confidence interval:

```
>>> from dowhy import gcm
>>>
>>> median, intervals = gcm.confidence_intervals(compute_pi_monte_carlo,
>>>                                               num_bootstrap_resamples=1000)
>>> median, intervals
(array([3.136]), array([[3.0518, 3.224 ]]))
```

The first item in the result is the median of all results calculated. The second is the interval in which we are 95% confident that computed values will fall into. By increasing the number of `trials` we can improve the reliability of `compute_pi_monte_carlo` and hence, narrow the confidence interval. E.g. with `trials = 10_000` we get an interval of [3.1132 , 3.16602], with `trials = 100_000` we get [3.13256 , 3.150016].

Computing confidence intervals for causal queries

With this understanding, we can now apply `confidence_intervals` to one of our causal queries, e.g. `distribution_change`. Let's generate some data first:

```
>>> import pandas as pd
>>> from scipy.stats import halfnorm
>>>
>>> X = halfnorm.rvs(size=1000, loc=0.5, scale=0.2)
>>> Y = halfnorm.rvs(size=1000, loc=1.0, scale=0.2)
>>> Z = np.maximum(X, Y) + np.random.normal(loc=0, scale=1, size=1000)
>>> data_old = pd.DataFrame(data=dict(X=X, Y=Y, Z=Z))
>>>
>>> X = halfnorm.rvs(size=1000, loc=0.5, scale=0.2)
>>> Y = halfnorm.rvs(size=1000, loc=1.0, scale=0.2)
>>> Z = X + Y + np.random.normal(loc=0, scale=1, size=1000)
>>> data_new = pd.DataFrame(data=dict(X=X, Y=Y, Z=Z))
```

Here, the causal mechanism for Z changes between the old and new data. As always, next, we'll model cause-effect relationships as an SCM and assign causal mechanisms:

Skip to main content

```
>>> gcm.auto.assign_causal_mechanisms(causal_model, data_old)
```

And now, instead of calling `distribution_change` directly we call it using `confidence_intervals`. However, since `confidence_intervals` takes a function that takes no parameters, we must first define a function without parameters:

```
>>> def f():
>>>     return gcm.distribution_change(causal_model, data_old, data_new, 'Z')
```

And now:

```
>>> gcm.confidence_intervals(f)
({'X': 0.0005804787010800414, 'Y': 0.0030143299612704804, 'Z': 0.1724206687905231}
{'X': array([-0.00427554,  0.00891714]),
'Y': array([-0.00605089,  0.01782684]),
'Z': array([0.15441771, 0.19062329])})
```

The result again contains a first item, which is the median values of the contribution scores. The second item contains the intervals of the contribution scores for each variable.

To avoid defining a new function, we can also streamline that call by using a lambda:

```
>>> gcm.confidence_intervals(lambda: gcm.distribution_change(causal_model,
>>>                                                          data_old, data_new,
>>>                                                          target_node='Z'))
```

Conveniently bootstrapping graph training on random subsets of training data

Many of the causal queries in the GCM package require a trained causal graph as a first argument. To compute confidence intervals for these methods, we need to explicitly re-train our causal graph multiple times with different random subsets of data and also run our causal query with each newly trained graph. To do this conveniently, the GCM package provides a function `fit_and_compute`. Assuming that we have `data` and a causal graph:

```
>>> Z = np.random.normal(loc=0, scale=1, size=1000)
```

Skip to main content

```
>>> data = pd.DataFrame(dict(X=X, Y=Y, Z=Z))
>>>
>>> causal_model = gcm.StructuralCausalModel(nx.DiGraph([('Z', 'Y'), ('Z', 'X'), (
>>> gcm.auto.assign_causal_mechanisms(causal_model, data)
```

we can now use `fit_and_compute` as follows:

```
>>> strength_median, strength_intervals = gcm.confidence_intervals(
>>>     gcm.fit_and_compute(gcm.arrow_strength,
>>>                         causal_model,
>>>                         bootstrap_training_data=data,
>>>                         target_node='Y'))
>>> strength_median, strength_intervals
({('X', 'Y'): 45.90886398636573, ('Z', 'Y'): 15.47129383737619},
{('X', 'Y'): array([42.88319632, 50.43890079]), ('Z', 'Y'): array([13.44202416, 17
```

Runtime cost versus confidence

In certain scenarios it is prohibitely expensive to re-train causal graphs multiple times. E.g. when using AutoGluon as prediction models for the additive noise model, a single `fit` execution can quickly go into hours. Bootstrapping with 20 resamples will then quickly go into days depending on how much we can parallelize.

For that reason, sometimes the tradeoff is to only bootstrap on the causal query, not on the training. To make this analogous to the approach we used above, there is the function **`bootstrap_sampling()`**. This function assumes that the causal graph is already trained. Then it can be used as follows:

```
>>> gcm.fit(causal_model, data)
>>>
>>> strength_median, strength_intervals = gcm.confidence_intervals(
>>>     gcm.bootstrap_sampling(gcm.arrow_strength,
>>>                            causal_model,
>>>                            target_node='Y'))
>>>
>>> strength_median, strength_intervals
({('X', 'Y'): 46.07299374572871, ('Z', 'Y'): 15.358850280972195},
{('X', 'Y'): array([44.95914495, 47.63918151]), ('Z', 'Y'): array([15.04323069, 15
```

`dowhy.gcm.bootstrap_sampling()` accomplishes the same thing as the lambda we've seen earlier. However, using `bootstrap_sampling` is more expressive. We therefore recommend its

Skip to main content

Understanding the need for confidence intervals

As explained in the beginnging, the models and specific sample set might not accurately represent the ground truth and there are typically numerical approximations in our algorithms. The error comes from three sources:

- Variance of the "optimal" parameters for a model, i.e. running `fit` twice with the same/slightly different data can yield two different models (becomes even worse when there is a stochastic element in the fit process of the prediction models as well). For instance, training an AutoGluon model twice on even exactly the same data would return two different models with slightly different performance.
- Approximations in our algorithms. For instance, when estimating distribution change with 6+ upstream nodes, we will only approximate the Shapley values (the approximation has a stochastic factor). Therefore, running distribution change twice with exactly the same input and generated samples will return two different results.
- Even if we do not approximate the Shapley values, the algorithms are typically based on some specific samples from the generative models, i.e. variance can also come from the variance between two sets of drawn samples. Therefore, even if the algorithm itself is deterministic (e.g. evaluate all possible subsets for the Shapley values), we would use randomly generated samples which yields different results when calling an algorithm twice.

Previous
Customizing Causal Mechanism Assignment

Next
Performing Causal Tasks