This file provides a very high-level explanation of:

- the algorithm that generates counterfactual explanations
- the algorithm that performs outcome optimization
- the algorithm that evaluates a sample's plausibility

Algorithm for counterfactual explanations

For each feature that has been made actionable, a handler is attached to it. Handlers generate meaningful random values for this feature in a given radius (they also define a distance metric) around the reference point. Handlers know the train set and generate values that respect the distribution of the features in the train set.

The algorithm is then:

1. Define two small concentric hyperspheres centered on the reference point, one with a minimum radius, the other with a maximum radius.
2. Generate some points between these two hyperspheres using the handlers.
3. Increase the radii of the hyperspheres and repeat step 2 until some counterfactual explanations are found (i.e. points for which the prediction differs from the reference's).
4. Once some counterfactual explanations are found for a given min radius and max radius, generate more valid counterfactual explanations using the handlers between these two radii.
5. At this point of the algorithm, some counterfactual explanations were found. The next phase is trying to improve them by resetting as many features as possible to the original while keeping the counterfactual nature in order to bring them closer to the reference.
6. Once counterfactual explanations have been improved, if too many were found, they are selected by performing a KMeans clustering with `k=number_of_desired_counterfactual_explanations` and returning one counterfactual explanation from each of the clusters to ensure diversity.

Counterfactual explanations don't usually belong to the train dataset, as they were generated randomly around the reference point.

This algorithm is a trade-off between:

- closeness to the reference
- plausibility of the generated points
- diversity of results

Algorithm for outcome optimization

Principle

For each feature that has been made actionable, a handler is attached to it. Handlers generate meaningful random values for this feature. Handlers know the train set and generate values that respect the distribution of the feature in the train set.

Handlers are used by a genetic algorithm to generate new candidate populations.

A "*lineage*" mechanism is introduced to search several distinct local optima, and to ensure that the final optimized population is diverse:

- Each sample belongs to a certain lineage.
- When a new sample is generated by a perturbation of another sample, both samples belong to the same lineage.
- When a new sample is generated ex-nihilo, it starts a new lineage.

The lineage is taken into account during the "*selection*" step of the genetic algorithm:

- If a new optimum has been found for a given lineage, the next generation will contain some of this lineage's samples, even if there are better samples in other lineages.
- If a lineage failed to improve, it might have reached a local optimum, so all corresponding samples are removed from the population.

When a lineage goes extinct, its best representative is probably a local optimum, so it's saved in a cache.

After a given number of iterations, the cache contains the values that must be returned by the algorithm. If it contains too many values, a KMeans clustering is performed with `k=number_of_desired_counterfactual_explanations` and each cluster's best representative is returned to ensure diversity.

Pseudocode

```
population = init_population_with_random_samples_from_dataset()
pantheon = []

for 0..n_iterations
   population = concat(
      population,
      generate_values_with_perturbations_of_existing_population(population),
      generate_random_uniform_values()
   )
   population, best_samples_from_newly_extinct_lineages = select(population)
   pantheon = concat(pantheon, best_samples_from_newly_extinct_lineages)

clusters = kmeans_clustering(pantheon)
return get_best_individual_from_each_cluster(clusters)
```

Plausibility

Using a `RandomTreesEmbedding`, computing correlations between some points and the points of the train set gives an indication of how plausible the points are.

Since this “correlation score” is not very interpretable, it's transformed into a value between 0% and 100% that's easier to understand using the following method:

1. The `RandomTreesEmbedding` is used on the points of the train set to find their "correlation score". It gives baselines.
2. The quantiles of these values are computed.
3. The "correlation scores" of the counterfactual explanations are compared to the quantiles to find how plausible they are compared to the points of the train set.

e.g. If a counterfactual explanation has a plausibility of 25%, it means that 25% of the points from the train set are more likely to be outliers (have a worse “correlation score”).