
ceml

André Artelt

Oct 04, 2023

USER GUIDE

1	Counterfactuals for Explaining Machine Learning models - CEML	1
1.1	Installation	1
1.2	Tutorial	2
1.3	Advanced	12
1.4	Theoretical background on the computation of counterfactual explanations	17
1.5	FAQ	17
1.6	ceml	18
1.7	ceml.backend.jax	79
1.8	ceml.backend.torch	83
1.9	ceml.backend.tensorflow	85
2	Indices and tables	89
	Python Module Index	91
	Index	93

COUNTERFACTUALS FOR EXPLAINING MACHINE LEARNING MODELS - CEML

CEML is a Python toolbox for computing counterfactuals. Counterfactuals can be used to explain the predictions of machine learning models.

It supports many common machine learning frameworks:

- scikit-learn (1.3.1)
- PyTorch (2.0.1)
- Keras & Tensorflow (2.13.1)

Furthermore, CEML is easy to use and can be extended very easily. See the following user guide for more information on how to use and extend ceml.

1.1 Installation

Note: Python 3.8 is required!

1.1.1 PyPi

```
pip install ceml
```

Note: The package hosted on PyPI uses the cpu only. If you want to use the gpu, you have to install CEML manually - see next section.

1.1.2 Git

Download or clone the repository:

```
git clone https://github.com/andreArtelt/ceml.git
cd ceml
```

Install all requirements (listed in `requirements.txt`):

```
pip install -r requirements.txt
```

Note: If you want to use a gpu/tpu, you have to install the gpu version of jax, tensorflow and PyTorch manually.

Do not use `pip install -r requirements.txt`

Install the toolbox itself:

```
pip install
```

1.2 Tutorial

CEML was designed with usability in mind.

In the subsequent paragraphs, we demonstrate how to use CEML with models from different machine learning libraries.

1.2.1 scikit-learn

Classification

Computing a counterfactual of a sklearn classifier is done by using the `ceml.sklearn.models.generate_counterfactual()` function.

We must specify the model we want to use, the input whose prediction we want to explain and the requested target prediction (prediction of the counterfactual). In addition we can restrict the features that can be used for computing a counterfactual, specify a regularization of the counterfactual and specifying the optimization algorithm used for computing a counterfactual.

A complete example of a classification task is given below:

```
1 #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  from sklearn.datasets import load_iris
4  from sklearn.model_selection import train_test_split
5  from sklearn.metrics import accuracy_score
6  from sklearn.tree import DecisionTreeClassifier
7
8  from ceml.sklearn import generate_counterfactual
9
10
11 if __name__ == "__main__":
12     # Load data
```

(continues on next page)

(continued from previous page)

```

13 X, y = load_iris(return_X_y=True)
14 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↳state=4242)
15
16 # Whitelist of features - list of features we can change/use when computing a_
↳counterfactual
17 features_whitelist = None # We can use all features
18
19 # Create and fit model
20 model = DecisionTreeClassifier(max_depth=3)
21 model.fit(X_train, y_train)
22
23 # Select data point for explaining its prediction
24 x = X_test[1,:]
25 print("Prediction on x: {0}".format(model.predict([x])))
26
27 # Compute counterfactual
28 print("\nCompute counterfactual ....")
29 print(generate_counterfactual(model, x, y_target=0, features_whitelist=features_
↳whitelist))

```

Regression

The interface for computing a counterfactual of a regression model is exactly the same.

But because it might be very difficult or even impossible (e.g. knn or decision tree) to achieve a requested prediction exactly, we can specify a tolerance range in which the prediction is accepted.

We can do so by defining a function that takes a prediction as an input and returns *True* if the prediction is accepted (it is in the range of tolerated predictions) and *False* otherwise. For instance, if our target value is 25.0 but we are also happy if it deviates not more than 0.5, we could come up with the following function:

```

1 done = lambda z: np.abs(z - 25.0) <= 0.5

```

This function can be passed as a value of the optional argument *done* to the `ceml.sklearn.models.generate_counterfactual()` function.

A complete example of a regression task is given below:

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 import numpy as np
4 from sklearn.datasets import load_boston
5 from sklearn.model_selection import train_test_split
6 from sklearn.linear_model import Ridge
7
8 from ceml.sklearn import generate_counterfactual
9
10
11 if __name__ == "__main__":
12     # Load data
13     X, y = load_boston(return_X_y=True)
14     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↳state=4242)

```

(continues on next page)

```

15
16     # Whitelist of features - list of features we can change/use when computing a
↳counterfactual
17     features_whitelist = [0, 1, 2, 3, 4]    # Use the first five features only
18
19     # Create and fit model
20     model = Ridge()
21     model.fit(X_train, y_train)
22
23     # Select data point for explaining its prediction
24     x = X_test[1,:]
25     print("Prediction on x: {}".format(model.predict([x])))
26
27     # Compute counterfactual
28     print("\nCompute counterfactual ....")
29     y_target = 25.0
30     done = lambda z: np.abs(y_target - z) <= 0.5    # Since we might not be able to
↳achieve `y_target` exactly, we tell ceml that we are happy if we do not deviate more
↳than 0.5 from it.
31     print(generate_counterfactual(model, x, y_target=y_target, features_
↳whitelist=features_whitelist, C=1.0, regularization="l2", optimizer="bfgs", done=done))

```

Pipeline

Often our machine learning pipeline contains more than one model. E.g. we first scale the input and/or reduce the dimensionality before classifying it.

The interface for computing a counterfactual when using a pipeline is identical to the one when using a single model only. We can simply pass a `sklearn.pipeline.Pipeline` instance as the value of the parameter `model` to the function `ceml.sklearn.models.generate_counterfactual()`.

Take a look at the `ceml.sklearn.pipeline.PipelineCounterfactual` class to see which preprocessings are supported.

A complete example of a classification pipeline with the standard scaler `sklearn.preprocessing.StandardScaler` and logistic regression `sklearn.linear_model.LogisticRegression` is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  from sklearn.datasets import load_iris
4  from sklearn.preprocessing import StandardScaler
5  from sklearn.pipeline import make_pipeline
6  from sklearn.model_selection import train_test_split
7  from sklearn.metrics import accuracy_score
8  from sklearn.linear_model import LogisticRegression
9
10 from ceml.sklearn import generate_counterfactual
11
12
13 if __name__ == "__main__":
14     # Load data
15     X, y = load_iris(return_X_y=True)

```

(continues on next page)

(continued from previous page)

```

16 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↳state=4242)
17
18 # Whitelist of features - list of features we can change/use when computing a_
↳counterfactual
19 features_whitelist = [1, 3] # Use the second and fourth feature only
20
21 # Create and fit the pipeline
22 scaler = StandardScaler()
23 model = LogisticRegression(solver='lbfgs', multi_class='multinomial') # Note that_
↳ceml requires: multi_class='multinomial'
24
25 model = make_pipeline(scaler, model)
26 model.fit(X_train, y_train)
27
28 # Select data point for explaining its prediction
29 x = X_test[1,:]
30 print("Prediction on x: {0}".format(model.predict([x])))
31
32 # Compute counterfactual
33 print("\nCompute counterfactual ....")
34 print(generate_counterfactual(model, x, y_target=0, features_whitelist=features_
↳whitelist))

```

Change optimization parameters

Sometimes it might become necessary to change to default parameters of the optimization methods - e.g. changing the solver, the maximum number of iterations, etc. This can be done by passing the optional argument `optimizer_args` to the `ceml.sklearn.models.generate_counterfactual()` function. The value of `optimizer_args` must be a dictionary where some parameters like verbosity, solver, maximum number of iterations, tolerance thresholds, etc. can be changed - note that not all parameters are used by every optimization algorithm (e.g. “epsilon”, “solver” and “solver_verbosity” are only used if `optimizer="mp"`).

A short code snippet demonstrating how to change some optimization parameters is given below:

```

1 import cvxpy as cp
2 from ceml.sklearn import generate_counterfactual
3 #.....
4
5 #model = .....
6 #x_orig = .....
7 #y_target = .....
8
9 # Change optimization parameters
10 #opt = ....
11 opt_args = {"epsilon": 10.e-4, "solver": cp.SCS, "solver_verbosity": False, "max_iter":_
↳200}
12
13 # Compute counterfactual explanations
14 x_cf, y_cf, delta = generate_counterfactual(model, x_orig, y_target, features_
↳whitelist=None, C=0.1, regularization="l1", optimizer=opt, optimizer_args=opt_args,_
↳return_as_dict=False)

```

(continues on next page)

15

1.2.2 Plausible counterfactuals

In *Convex Density Constraints for Computing Plausible Counterfactual Explanations* (Artelt et al. 2020) a general framework for computing plausible counterfactuals was proposed. CEML currently implements these methods for softmax regression and decision tree classifiers.

In order to compute plausible counterfactual explanations, some preparations are required:

Use the `ceml.sklearn.plausibility.prepare_computation_of_plausible_counterfactuals()` function for creating a dictionary that can be passed to functions for generating counterfactuals. You have to provide class dependent fitted Gaussian Mixture Models (GMMs) and the training data itself. In addition, you can also provide an affine preprocessing and a requested density/plausibility threshold (if you do not specify any, a suitable threshold will be selected automatically).

A complete example for computing a plausible counterfactual of a digit classifier (logistic regression) is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import numpy as np
4  import random
5  random.seed(424242)
6  import matplotlib.pyplot as plt
7  from sklearn.linear_model import LogisticRegression
8  from sklearn.mixture import GaussianMixture
9  from sklearn.model_selection import GridSearchCV, train_test_split
10 from sklearn.decomposition import PCA
11 from sklearn.datasets import load_digits
12 from sklearn.metrics import accuracy_score
13 from sklearn.utils import shuffle
14
15 from ceml.sklearn.softmaxregression import softmaxregression_generate_counterfactual
16 from ceml.sklearn.plausibility import prepare_computation_of_plausible_counterfactuals
17
18
19 if __name__ == "__main__":
20     # Load data set
21     X, y = load_digits(return_X_y=True);pca_dim=40
22
23     X, y = shuffle(X, y, random_state=42)
24     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=4242)
25
26     # Choose target labels
27     y_test_target = []
28     labels = np.unique(y)
29     for i in range(X_test.shape[0]):
30         y_test_target.append(random.choice(list(filter(lambda l: l != y_test[i],
↪labels))))
31     y_test_target = np.array(y_test_target)
32

```

(continues on next page)

(continued from previous page)

```

33  # Reduce dimensionality
34  X_train_orig = np.copy(X_train)
35  X_test_orig = np.copy(X_test)
36  projection_matrix = None
37  projection_mean_sub = None
38
39  pca = PCA(n_components=pca_dim)
40  pca.fit(X_train)
41
42  projection_matrix = pca.components_ # Projection matrix
43  projection_mean_sub = pca.mean_
44
45  X_train = np.dot(X_train - projection_mean_sub, projection_matrix.T)
46  X_test = np.dot(X_test - projection_mean_sub, projection_matrix.T)
47
48  # Fit classifier
49  model = LogisticRegression(multi_class="multinomial", solver="lbfgs", random_
↪state=42)
50  model.fit(X_train, y_train)
51
52  # Compute accuracy on test set
53  print("Accuracy: {0}".format(accuracy_score(y_test, model.predict(X_test))))
54
55  # For each class, fit density estimators
56  density_estimators = {}
57  kernel_density_estimators = {}
58  labels = np.unique(y)
59  for label in labels:
60      # Get all samples with the 'correct' label
61      idx = y_train == label
62      X_ = X_train[idx, :]
63
64      # Optimize hyperparameters
65      cv = GridSearchCV(estimator=GaussianMixture(covariance_type='full'), param_grid={
↪'n_components': range(2, 10)}, n_jobs=-1, cv=5)
66      cv.fit(X_)
67      n_components = cv.best_params_["n_components"]
68
69      # Build density estimators
70      de = GaussianMixture(n_components=n_components, covariance_type='full', random_
↪state=42)
71      de.fit(X_)
72
73      density_estimators[label] = de
74
75  # Build dictionary for ceml
76  plausibility_stuff = prepare_computation_of_plausible_counterfactuals(X_train_orig,
↪y_train, density_estimators, projection_mean_sub, projection_matrix)
77
78  # Compute and plot counterfactual with vs. without density constraints
79  i = 0
80

```

(continues on next page)

```

81 x_orig = X_test[i,:]
82 x_orig_orig = X_test_orig[i,:]
83 y_orig = y_test[i]
84 y_target = y_test_target[i]
85 print("Original label: {0}".format(y_orig))
86 print("Target label: {0}".format(y_target))
87
88 if(model.predict([x_orig]) == y_target): # Model already predicts target label!
89     raise ValueError("Requested prediction already satisfied")
90
91 # Compute plausible counterfactual
92 x_cf_plausible = softmaxregression_generate_counterfactual(model, x_orig_orig, y_
↪target, plausibility=plausibility_stuff)
93 x_cf_plausible_projected = np.dot(x_cf_plausible - projection_mean_sub, projection_
↪matrix.T)
94 print("Predicted label of plausible counterfactual: {0}".format(model.predict([x_cf_
↪plausible_projected])))
95
96 # Compute closest counterfactual
97 plausibility_stuff["use_density_constraints"] = False
98 x_cf = softmaxregression_generate_counterfactual(model, x_orig_orig, y_target,
↪plausibility=plausibility_stuff)
99 x_cf_projected = np.dot(x_cf - projection_mean_sub, projection_matrix.T)
100 print("Predicted label of closest counterfactual: {0}".format(model.predict([x_cf_
↪projected])))
101
102 # Plot results
103 fig, axes = plt.subplots(3, 1)
104 axes[0].imshow(x_orig_orig.reshape(8, 8)) # Original sample
105 axes[1].imshow(x_cf.reshape(8, 8)) # Closest counterfactual
106 axes[2].imshow(x_cf_plausible.reshape(8, 8)) # Plausible counterfactual
107 plt.show()

```

1.2.3 PyTorch

Computing a counterfactual of a PyTorch model is done by using the `cemi.torch.counterfactual.generate_counterfactual()` function.

We must provide the PyTorch model within a class that is derived from `torch.nn.Module` and `cemi.model.model.ModelWithLoss`. In this class, we must overwrite the `predict` function and the `get_loss` function which returns a loss that we want to use - a couple of differentiable loss functions are implemented in `cemi.backend.torch.costfunctions`.

Besides the model, we must specify the input whose prediction we want to explain and the desired target prediction (prediction of the counterfactual). In addition we can restrict the features that can be used for computing a counterfactual, specify a regularization of the counterfactual and specifying the optimization algorithm used for computing a counterfactual.

A complete example of a softmax regression model using the negative-log-likelihood is given below:

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 import torch

```

(continues on next page)

(continued from previous page)

```

4 torch.manual_seed(424242)
5 import numpy as np
6 from sklearn.datasets import load_iris
7 from sklearn.model_selection import train_test_split
8 from sklearn.metrics import accuracy_score
9
10 from cemi.torch import generate_counterfactual
11 from cemi.backend.torch.costfunctions import NegLogLikelihoodCost
12 from cemi.model import ModelWithLoss
13
14
15 # Neural network - Softmax regression
16 class Model(torch.nn.Module, ModelWithLoss):
17     def __init__(self, input_size, num_classes):
18         super(Model, self).__init__()
19
20         self.linear = torch.nn.Linear(input_size, num_classes)
21         self.softmax = torch.nn.Softmax(dim=0)
22
23     def forward(self, x):
24         return self.linear(x) # NOTE: Softmax is build into CrossEntropyLoss
25
26     def predict_proba(self, x):
27         return self.softmax(self.forward(x))
28
29     def predict(self, x, dim=1):
30         return torch.argmax(self.forward(x), dim=dim)
31
32     def get_loss(self, y_target, pred=None):
33         return NegLogLikelihoodCost(input_to_output=self.predict_proba, y_target=y_
↪target)
34
35
36 if __name__ == "__main__":
37     # Load data
38     X, y = load_iris(return_X_y=True)
39     X = X.astype(np.dtype(np.float32))
40
41     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=1)
42
43     # numpy -> torch tensor
44     x = torch.from_numpy(X_train)
45     labels = torch.from_numpy(y_train)
46
47     x_test = torch.from_numpy(X_test)
48     y_test = torch.from_numpy(y_test)
49
50     # Create and fit model
51     model = Model(4, 3)
52
53     learning_rate = 0.001

```

(continues on next page)

```

54 momentum = 0.9
55 criterion = torch.nn.CrossEntropyLoss()
56 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=momentum)
57
58 num_epochs = 800
59 for epoch in range(num_epochs):
60     optimizer.zero_grad()
61     outputs = model(x)
62     loss = criterion(outputs, labels)
63     loss.backward()
64     optimizer.step()
65
66 # Evaluation
67 y_pred = model.predict(x_test).detach().numpy()
68 print("Accuracy: {0}".format(accuracy_score(y_test, y_pred)))
69
70 # Select a data point whose prediction has to be explained
71 x_orig = X_test[1,:]
72 print("Prediction on x: {0}".format(model.predict(torch.from_numpy(np.array([x_
↪orig])))))
73
74 # Whitelist of features we can use/change when computing the counterfactual
75 features_whitelist = [0, 2] # Use the first and third feature only
76
77 # Compute counterfactual
78 print("\nCompute counterfactual ....")
79 print(generate_counterfactual(model, x_orig, y_target=0, features_whitelist=features_
↪whitelist, regularization="l1", C=0.1, optimizer="nelder-mead"))

```

1.2.4 Tensorflow & Keras

Since keras is a higher-level interface for tensorflow and nowadays part of tensorflow, we do not need to distinguish between keras and tensorflow models when using cemi.

Computing a counterfactual of a tensorflow/keras model is done by using the `cemi.tfkeras.counterfactual.generate_counterfactual()` function.

Note: We have to run in *eager execution mode* when computing a counterfactual! Since tensorflow 2, eager execution is enabled by default.

We must provide the tensorflow/keras model within a class that is derived from the `cemi.model.model.ModelWithLoss` class. In this class, we must overwrite the `predict` function and `get_loss` function which returns a loss that we want to use - a couple of differentiable loss functions are implemented in `cemi.backend.tensorflow.costfunctions`.

Besides the model, we must specify the input whose prediction we want to explain and the desired target prediction (prediction of the counterfactual). In addition we can restrict the features that can be used for computing a counterfactual, specify a regularization of the counterfactual and specifying the optimization algorithm used for computing a counterfactual.

A complete example of a softmax regression model using the negative-log-likelihood is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import tensorflow as tf
4  import numpy as np
5  from sklearn.datasets import load_iris
6  from sklearn.model_selection import train_test_split
7  from sklearn.metrics import accuracy_score
8
9  from ceml.tfkeras import generate_counterfactual
10 from ceml.backend.tensorflow.costfunctions import NegLogLikelihoodCost
11 from ceml.model import ModelWithLoss
12
13
14 # Neural network - Softmax regression
15 class Model(ModelWithLoss):
16     def __init__(self, input_size, num_classes):
17         super(Model, self).__init__()
18
19         self.model = tf.keras.models.Sequential([
20             tf.keras.layers.Dense(num_classes, activation='softmax', input_shape=(input_
↪size,))
21         ])
22
23     def fit(self, x_train, y_train, num_epochs=800):
24         self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
↪metrics=['accuracy'])
25
26         self.model.fit(x_train, y_train, epochs=num_epochs, verbose=False)
27
28     def predict(self, x):
29         return np.argmax(self.model(x), axis=1)
30
31     def predict_proba(self, x):
32         return self.model(x)
33
34     def __call__(self, x):
35         return self.predict(x)
36
37     def get_loss(self, y_target, pred=None):
38         return NegLogLikelihoodCost(input_to_output=self.model.predict_proba, y_target=y_
↪target)
39
40
41 if __name__ == "__main__":
42     tf.random.set_seed(42)    # Fix random seed
43
44     # Load data
45     X, y = load_iris(return_X_y=True)
46
47     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=1)
48
49     # Create and fit model

```

(continues on next page)

```

50 model = Model(4, 3)
51 model.fit(X_train, y_train)
52
53 # Evaluation
54 y_pred = model.predict(X_test)
55 print("Accuracy: {0}".format(accuracy_score(y_test, y_pred)))
56
57 # Select a data point whose prediction has to be explained
58 x_orig = X_test[1,:]
59 print("Prediction on x: {0}".format(model.predict(np.array([x_orig])))
60
61 # Whitelist of features we can use/change when computing the counterfactual
62 features_whitelist = None
63
64 # Compute counterfactual
65 optimizer = tf.compat.v1.train.GradientDescentOptimizer(learning_rate=1.0) # Init_
↪ optimization algorithm
66 optimizer_args = {"max_iter": 1000}
67
68 print("\nCompute counterfactual ....")
69 print(generate_counterfactual(model, x_orig, y_target=0, features_whitelist=features_
↪ whitelist, regularization="l1", C=0.01, optimizer=optimizer, optimizer_args=optimizer_
↪ args))

```

1.3 Advanced

CEML can be easily extended and all major components can be customized to fit the users needs.

Below is a (non-exhaustive) list of some (common) use cases:

1.3.1 Custom regularization

Instead of using one of the predefined regularizations, we can pass a custom regularization to `ceml.sklearn.models.generate_counterfactual()`.

All regularization implementations must be classes derived from `ceml.costfunctions.costfunctions.CostFunction`. In case of scikit-learn, if we want to use a gradient based optimization algorithm, we must derive from `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax` - note that `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax` is already derived from `ceml.costfunctions.costfunctions.CostFunction`.

Note: For tensorflow/keras or PyTorch models the base classes are `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf` and `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`.

The computation of the regularization itself must be implemented in the `score_impl` function.

A complete example of a re-implementation of the l2-regularization is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import jax.numpy as npx
4  from sklearn.datasets import load_iris
5  from sklearn.model_selection import train_test_split
6  from sklearn.metrics import accuracy_score
7  from sklearn.naive_bayes import GaussianNB
8
9  from ceml.sklearn import generate_counterfactual
10 from ceml.backend.jax.costfunctions import CostFunctionDifferentiableJax
11
12
13 # Custom implementation of the l2-regularization. Note that this regularization is
14 ↪differentiable
15 class MyRegularization(CostFunctionDifferentiableJax):
16     def __init__(self, x_orig):
17         self.x_orig = x_orig
18
19         super(MyRegularization, self).__init__()
20
21     def score_impl(self, x):
22         return npx.sum(npx.square(x - self.x_orig)) # Note: This expression must be
23 ↪written in jax and it must be differentiable!
24
25 if __name__ == "__main__":
26     # Load data
27     X, y = load_iris(return_X_y=True)
28     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
29 ↪state=4242)
30
31     # Whitelist of features - list of features we can change/use when computing a
32 ↪counterfactual
33     features_whitelist = None # All features can be used.
34
35     # Create and fit the model
36     model = GaussianNB() # Note that ceml requires: multi_class='multinomial'
37     model.fit(X_train, y_train)
38
39     # Select data point for explaining its prediction
40     x = X_test[1,:]
41     print("Prediction on x: {0}".format(model.predict([x])))
42
43     # Create custom regularization function
44     regularization = MyRegularization(x)
45
46     # Compute counterfactual
47     print("\nCompute counterfactual ....")
48     print(generate_counterfactual(model, x, y_target=0, features_whitelist=features_
49 ↪whitelist, regularization=regularization, optimizer="bfgs"))

```

1.3.2 Custom loss function

In order to use a custom loss function we have to do three things:

1. Implement the loss function. This is the same as implementing a custom regularization - a regularization is a loss function that works on the input rather than on the output.
2. Derive a child class from the model class and overwrite the `get_loss` function to use our custom loss function.
3. Derive a child class from the counterfactual class of the model and overwrite the `rebuild_model` function to use our model from the previous step.

A complete example of using a custom loss for a linear regression model is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import numpy as np
4  import jax.numpy as npx
5  from sklearn.datasets import load_boston
6  from sklearn.model_selection import train_test_split
7  from sklearn.metrics import accuracy_score
8  from sklearn.linear_model import Ridge
9
10
11 from ceml.sklearn import generate_counterfactual
12 from ceml.sklearn import LinearRegression, LinearRegressionCounterfactual
13 from ceml.backend.jax.costfunctions import CostFunctionDifferentiableJax
14
15
16 # Custom implementation of the l2-regularization. Note that this regularization is
17 ↳differentiable.
18 class MyLoss(CostFunctionDifferentiableJax):
19     def __init__(self, input_to_output, y_target):
20         self.y_target = y_target
21
22         super(MyLoss, self).__init__(input_to_output)
23
24     def score_impl(self, y):
25         return npx.abs(y - y_target)**4
26
27 # Derive a new class from ceml.sklearn.linearregression.LinearRegression and overwrite
28 ↳the get_loss method to use our custom loss MyLoss
29 class LinearRegressionWithMyLoss(LinearRegression):
30     def __init__(self, model):
31         super(LinearRegressionWithMyLoss, self).__init__(model)
32
33     def get_loss(self, y_target, pred=None):
34         if pred is None:
35             return MyLoss(self.predict, y_target)
36         else:
37             return MyLoss(pred, y_target)
38
39 # Derive a new class from ceml.sklearn.linearregression.LinearRegressionCounterfactual
40 ↳that uses our new linear regression wrapper LinearRegressionWithMyLoss for computing
41 ↳counterfactuals

```

(continues on next page)

(continued from previous page)

```

38 class MyLinearRegressionCounterfactual(LinearRegressionCounterfactual):
39     def __init__(self, model):
40         super(MyLinearRegressionCounterfactual, self).__init__(model)
41
42     def rebuild_model(self, model):
43         return LinearRegressionWithMyLoss(model)
44
45
46 if __name__ == "__main__":
47     # Load data
48     X, y = load_boston(return_X_y=True)
49     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↳state=4242)
50
51     # Whitelist of features - list of features we can change/use when computing a_
↳counterfactual
52     features_whitelist = None # All features can be used.
53
54     # Create and fit model
55     model = Ridge()
56     model.fit(X_train, y_train)
57
58     # Select data point for explaining its prediction
59     x = X_test[1,:]
60     print("Prediction on x: {}".format(model.predict([x])))
61
62     # Compute counterfactual
63     print("\nCompute counterfactual ....")
64     y_target = 25.0
65     done = lambda z: np.abs(y_target - z) <= 0.5 # Since we might not be able to_
↳achieve `y_target` exactly, we tell ceml that we are happy if we do not deviate more_
↳than 0.5 from it.
66
67     cf = MyLinearRegressionCounterfactual(model) # Since we are using our own loss_
↳function, we can no longer use standard method generate_counterfactual
68     print(cf.compute_counterfactual(x, y_target=y_target, features_whitelist=features_
↳whitelist, regularization="l2", C=1.0, optimizer="bfgs", done=done))

```

1.3.3 Add a custom optimizer

We can use a custom optimization method by:

1. Derive a new class from `ceml.optim.optimizer.Optimizer` and implement the custom optimization method.
2. Create a new instance of this class and pass it as the argument for the `optimizer` parameter to the function `ceml.sklearn.models.generate_counterfactual()` (or any other function that computes a counterfactual).

A complete example of using a custom optimization method for computing counterfactuals from a logistic regression model is given below:

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  import numpy as np
4  from sklearn.datasets import load_iris
5  from sklearn.model_selection import train_test_split
6  from sklearn.linear_model import LogisticRegression
7  from scipy.optimize import minimize
8
9  from cemi.sklearn import generate_counterfactual
10 from cemi.optim import Optimizer
11
12
13 # Custom optimization method that simply calls the BFGS optimizer from scipy
14 class MyOptimizer(Optimizer):
15     def __init__(self):
16         self.f = None
17         self.f_grad = None
18         self.x0 = None
19         self.tol = None
20         self.max_iter = None
21
22         super(MyOptimizer, self).__init__()
23
24     def init(self, f, f_grad, x0, tol=None, max_iter=None):
25         self.f = f
26         self.f_grad = f_grad
27         self.x0 = x0
28         self.tol = tol
29         self.max_iter = max_iter
30
31     def is_grad_based(self):
32         return True
33
34     def __call__(self):
35         optimum = minimize(fun=self.f, x0=self.x0, jac=self.f_grad, tol=self.tol,
↳ options={'maxiter': self.max_iter}, method="BFGS")
36         return np.array(optimum["x"])
37
38
39 if __name__ == "__main__":
40     # Load data
41     X, y = load_iris(return_X_y=True)
42
43     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↳ state=4242)
44
45     # Create and fit model
46     model = LogisticRegression(solver='lbfgs', multi_class='multinomial')
47     model.fit(X_train, y_train)
48
49     # Select data point for explaining its prediction
50     x = X_test[1,:]
51     print("Prediction on x: {0}".format(model.predict([x])))

```

(continues on next page)

(continued from previous page)

```

52 # Compute counterfactual by using our custom optimizer 'MyOptimizer'
53 print("\nCompute counterfactual ....")
54 print(generate_counterfactual(model, x, y_target=0, optimizer=MyOptimizer(),
55 ↪ features_whitelist=None, regularization="l1", C=0.5))

```

1.4 Theoretical background on the computation of counterfactual explanations

1.4.1 References

Details on the computation of counterfactual explanations can be found in the following papers:

- On the computation of counterfactual explanations – A survey
- Efficient computation of counterfactual explanations of LVQ models
- Convex Density Constraints for Computing Plausible Counterfactual Explanations

1.5 FAQ

How can I install CEML?

```
pip install ceml
```

See *Installation* for more information.

Under which license is ceml released? MIT license - See [License](#).

How can I cite CEML? You can cite CEML by using the following BibTeX entry:

```

@misc{ceml,
  author = {André Artelt},
  title = {CEML: Counterfactuals for Explaining Machine Learning models - A
↪ Python toolbox},
  year = {2019 - 2023},
  publisher = {GitHub},
  journal = {GitHub repository},
  howpublished = {\url{https://www.github.com/andreArtelt/ceml}}
}

```

Please consider citing CEML if it helps you in your research.

Is citing CEML mandatory? No, of course not. But it is best practice to cite everything you have used (literature as well as software).

How can I submit a bug report? Go to [Github](#) and create a new issue.

Please provide a detailed description of the bug and how to reproduce it.

How can I contribute? Go to [Github](#) and create a new pull request.

Note: Make sure that your code fits into ceml. Your code should follow the code style and design/architecture of ceml.

Can you implement a particular method/algorithm? Maybe.

Go to [Github](#) and create a new issue. I will take a look at it and let you know whether/when I will/can implement this.

Where can I learn more about the computation of counterfactual explanations? Please take a look at the section *Theoretical background on the computation of counterfactual explanations*.

1.6 ceml

1.6.1 ceml.sklearn

`ceml.sklearn.counterfactual`

`class ceml.sklearn.counterfactual.SklearnCounterfactual(model, **kws)`

Bases: `ceml.model.counterfactual.Counterfactual`, `abc.ABC`

Base class for computing a counterfactual of a *sklearn* model.

The *SklearnCounterfactual* class can compute counterfactuals of *sklearn* models.

Parameters `model` (*object*) – The *sklearn* model that is used for computing the counterfactual.

model

An instance of a *sklearn* model.

Type `object`

mymodel

Rebuild model.

Type instance of `ceml.model.ModelWithLoss`

Note: The class *SklearnCounterfactual* can not be instantiated because it contains an abstract method.

`compute_counterfactual(x, y_target, features_whitelist=None, regularization='l1', C=1.0, optimizer='auto', optimizer_args=None, return_as_dict=True, done=None)`

Computes a counterfactual of a given input *x*.

Parameters

- **x** (*numpy.ndarray*) – The data point *x* whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested prediction of the counterfactual.
- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is `None`, all features can be used.

The default is `None`.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*. Supported values:

– `l1`: Penalizes the absolute deviation.

– `l2`: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if the cost function is differentiable) or `None` if no regularization is requested.

If *regularization* is `None`, no regularization is used.

The default is "l1".

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

If no regularization is used (*regularization=None*), *C* is ignored.

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

Use "auto" if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

Some models (see paper) support the use of mathematical programs for computing counterfactuals. In this case, you can use the option "mp" - please read the documentation of the corresponding model for further information.

The default is "auto".

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – A callable that returns `True` if a counterfactual with a given output/prediction is accepted and `False` otherwise.

If *done* is `None`, the output/prediction of the counterfactual must match *y_target* exactly.

The default is `None`.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is `False`

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

abstract rebuild_model(*model*)

Rebuilds a *sklearn* model.

Converts a *sklearn* model into a class:*ceml.model.ModelWithLoss* instance so that we have a model specific cost function and can compute the derivative with respect to the input.

Parameters *model* – The *sklearn* model that is used for computing the counterfactual.

Returns The wrapped *model*

Return type `ceml.model.ModelWithLoss`

ceml.sklearn.plausibility

`ceml.sklearn.plausibility.prepare_computation_of_plausible_counterfactuals`(*X*, *y*, *gmms*, *projection_mean_sub*=None, *projection_matrix*=None, *density_thresholds*=None)

Computes all steps that are independent of a concrete sample when computing a plausible counterfactual explanations. Because the computation of a plausible counterfactual requires quite an amount of computation that does not depend on the concret sample we want to explain, it make sense to pre compute as much as possible (reduce redundant computations).

Parameters

- **X** (*numpy.ndarray*) – Data points.
- **y** (*numpy.ndarray*) – Labels of data points *X*. Assumed to be [0, 1, 2, ...].
- **gmms** (*list(int)*) – List of class dependent Gaussian Mixture Models (GMMs).
- **projection_mean_sub** (*numpy.ndarray*, optional) – The negative bias of the affine preprocessing.
The default is None.
- **projection_matrix** (*numpy.ndarray*, optional) – The projection matrix of the affine preprocessing.
The default is None.
- **density_threshold** (*float*, optional) – Density threshold at which we consider a counterfactual to be plausible.

If no density threshold is specified (*density_threshold* is set to None), the median density of the samples *X* is chosen as a threshold.

The default is None.

Returns All necessary (pre computable) stuff needed for the computation of plausible counterfactuals.

Return type *dict*

ceml.sklearn.decisiontree

class `ceml.sklearn.decisiontree.DecisionTreeCounterfactual(model, **kws)`

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.sklearn.decisiontree.PlausibleCounterfactualOfDecisionTree`

Class for computing a counterfactual of a decision tree model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

compute_all_counterfactuals(*x*, *y_target*, *features_whitelist=None*, *regularization='l1'*)

Computes all counterfactuals of a given input *x*.

Parameters

- **model** (a `sklearn.tree.DecisionTreeClassifier` or `sklearn.tree.DecisionTreeRegressor` instance.) – The decision tree model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction is supposed to be explained.
- **y_target** (`int` or `float` or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (`str` or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is None, no regularization is used.

The default is “l1”.

Returns List of all counterfactuals.

Return type `list(np.array)`

Raises

- **TypeError** – If an invalid argument is passed to the function.
- **ValueError** – If no counterfactual exists.

compute_counterfactual(*x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=None*, *optimizer=None*, *return_as_dict=True*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.tree.DecisionTreeClassifier` or `sklearn.tree.DecisionTreeRegressor` instance.) – The decision tree model that is used for computing the counterfactual.

- **x** (*numpy.ndarray*) – The input x whose prediction is supposed to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (*str* or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x .

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*None*) – Not used - is always None.

The only reason for including this parameter is to match the signature of other [cemi](#).
[sklearn.counterfactual](#).[SklearnCounterfactual](#) children.

- **optimizer** (*None*) – Not used - is always None.

The only reason for including this parameter is to match the signature of other [cemi](#).
[sklearn.counterfactual](#).[SklearnCounterfactual](#) children.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

rebuild_model(*model*)

Rebuild a `sklearn.linear_model.LogisticRegression` model.

Does nothing.

Parameters *model* (instance of `sklearn.tree.DecisionTreeClassifier` or `sklearn.tree.DecisionTreeRegressor`) – The *sklearn* decision tree model.

Returns

Return type None

Note: In contrast to many other `SklearnCounterfactual` instances, we do not rebuild the model because we do not need/can compute gradients in a decision tree. We compute the set of counterfactuals without using a “common” optimization algorithms like Nelder-Mead.

```
ceml.sklearn.decisiontree.decisiontree_generate_counterfactual(model, x, y_target,
                                                              features_whitelist=None,
                                                              regularization='l1',
                                                              return_as_dict=True, done=None,
                                                              plausibility=None)
```

Computes a counterfactual of a given input x .

Parameters

- **model** (a `sklearn.tree.DecisionTreeClassifier` or `sklearn.tree.DecisionTreeRegressor` instance.) – The decision tree model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input x whose prediction has to be explained.
- **y_target** (`int` or `float` or a callable that returns `True` if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **feature_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `feature_whitelist` is `None`, all features can be used.

The default is `None`.

- **regularization** (`str` or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x . Supported values:
 - `l1`: Penalizes the absolute deviation.
 - `l2`: Penalizes the squared deviation.

You can use your own custom penalty function by setting `regularization` to a callable that can be called on a potential counterfactual and returns a scalar.

If `regularization` is `None`, no regularization is used.

The default is `“l1”`.

- **return_as_dict** (`boolean`, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (`callable`, optional) – Not used.

- **plausibility** (`dict`, optional.) – If set to a valid dictionary (see `ceml.sklearn.plausibility.prepare_computation_of_plausible_counterfactuals()`), a plausible counterfactual (as proposed in Artelt et al. 2020) is computed. Note that in this case, all other parameters are ignored.

If `plausibility` is `None`, the closest counterfactual is computed.

The default is `None`.

Returns

A dictionary where the counterfactual is stored in ‘`x_cf`’, its prediction in ‘`y_cf`’ and the changes to the original input in ‘`delta`’.

(*x_cf*, *y_cf*, *delta*): triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

cemi.sklearn.knn

class `cemi.sklearn.knn.KNN(model, dist='l2', **kws)`

Bases: `cemi.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.neighbors.KNeighborsClassifier` and `sklearn.neighbors.KNeighborsRegressor` classes.

The *KNN* class rebuilds a *sklearn* knn model.

Parameters

- **model** (instance of `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor`) – The knn model.
- **dist** (*str* or callable, optional) – Computes the distance between a prototype and a data point.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

You can use your own custom distance function by setting *dist* to a callable that can be called on a data point and returns a scalar.

The default is “l2”.

Note: *dist* must not be None.

x

The training data set.

Type *numpy.array*

y

The ground truth of the training data set.

Type *numpy.array*

dist

The distance function.

Type *callable*

Raises TypeError – If *model* is not an instance of `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor`

get_loss(*y_target*, *pred*=None)

Creates and returns a loss function.

Builds a cost function where we penalize the minimum distance to the nearest prototype which is consistent with the target *y_target*.

Parameters

- **y_target** (*int*) – The target class.

- **pred** (*callable*, optional) – A callable that maps an input to an input. E.g. using the `ceml.optim.input_wrapper.InputWrapper` class.

If *pred* is None, no transformation is applied to the input before passing it into the loss function.

The default is None.

Returns Initialized cost function. Target label is *y_target*.

Return type `ceml.backend.jax.costfunctions.TopKMinOfListDistCost`

predict(*x*)

Note: This function is a placeholder only.

This function does not predict anything and just returns the given input.

class `ceml.sklearn.knn.KnnCounterfactual`(*model*, *dist*='l2', ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`

Class for computing a counterfactual of a knn model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuilds a `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor` model.

Converts a `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor` instance into a `ceml.sklearn.knn.KNN` instance.

Parameters *model* (instance of `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor`) – The *sklearn* knn model.

Returns The wrapped knn model.

Return type `ceml.sklearn.knn.KNN`

`ceml.sklearn.knn.knn_generate_counterfactual`(*model*, *x*, *y_target*, *features_whitelist*=None, *dist*='l2', *regularization*='l1', *C*=1.0, *optimizer*='nelder-mead', *optimizer_args*=None, *return_as_dict*=True, *done*=None)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.neighbors.KNeighborsClassifier` or `sklearn.neighbors.KNeighborsRegressor` instance.) – The knn model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **dist** (*str* or callable, optional) – Computes the distance between a prototype and a data point.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

You can use your own custom distance function by setting *dist* to a callable that can be called on a data point and returns a scalar.

The default is “11”.

Note: *dist* must not be None.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*. Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optimizer.optimizer.desc_to_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is None, the output/prediction of the counterfactual must match *y_target* exactly.

The default is None.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

ceml.sklearn.linearregression

class ceml.sklearn.linearregression.**LinearRegression**(*model*, ***kws*)

Bases: *ceml.model.model.ModelWithLoss*

Class for rebuilding/wrapping the sklearn.linear_model.base.LinearModel class

The *LinearRegression* class rebuilds a softmax regression model from a given weight vector and intercept.

Parameters **model** (instance of sklearn.linear_model.base.LinearModel) – The linear regression model (e.g. sklearn.linear_model.LinearRegression or sklearn.linear_model.Ridge).

w

The weight vector (a matrix if we have a multi-dimensional output).

Type *numpy.ndarray*

b

The intercept/bias (a vector if we have a multi-dimensional output).

Type *numpy.ndarray*

dim

Dimensionality of the input data.

Type *int*

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Build a squared-error cost function where the target is *y_target*.

Parameters

- **y_target** (*float*) – The target value.
- **pred** (*callable*, optional) – A callable that maps an input to the output (regression).

If *pred* is None, the class method *predict* is used for mapping the input to the output (regression)

The default is None.

Returns Initialized squared-error cost function. Target is *y_target*.

Return type *ceml.backend.jax.costfunctions.SquaredError*

predict(*x*)

Predict the output of a given input.

Computes the regression on a given input *x*.

Parameters *x* (*numpy.ndarray*) – The input *x* whose output is going to be predicted.

Returns An array containing the predicted output.

Return type *jax.numpy.array*

class `ceml.sklearn.linearregression.LinearRegressionCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.ConvexQuadraticProgram`

Class for computing a counterfactual of a linear regression model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuild a `sklearn.linear_model.base.LinearModel` model.

Converts a `sklearn.linear_model.base.LinearModel` into a `ceml.sklearn.linearregression.LinearRegression`.

Parameters *model* (instance of `sklearn.linear_model.base.LinearModel`) – The *sklearn* linear regression model (e.g. `sklearn.linear_model.LinearRegression` or `sklearn.linear_model.Ridge`).

Returns The wrapped linear regression model.

Return type `ceml.sklearn.linearregression.LinearRegression`

`ceml.sklearn.linearregression.linearregression_generate_counterfactual`(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='mp'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.linear_model.base.LinearModel` instance.) – The linear regression model (e.g. `sklearn.linear_model.LinearRegression` or `sklearn.linear_model.Ridge`) that is used for computing the counterfactual.
- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*float*) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is *None*, all features can be used.

The default is *None*.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or `None` if no regularization is requested.

If *regularization* is `None`, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

Linear regression supports the use of mathematical programs for computing counterfactuals - set *optimizer* to “mp” for using a convex quadratic program for computing the counterfactual. Note that in this case the hyperparameter *C* is ignored.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “mp”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – A callable that returns `True` if a counterfactual with a given output/prediction is accepted and `False` otherwise.

If *done* is `None`, the output/prediction of the counterfactual must match *y_target* exactly.

The default is `None`.

Note: It might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is `False`

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

cemi.sklearn.lvq

```
class cemi.sklearn.lvq.CQPHelper(mymodel, x_orig, y_target, indices_other_prototypes,
                                features_whitelist=None, regularization='l1', optimizer_args=None,
                                **kws)
```

Bases: `cemi.optim.cvx.ConvexQuadraticProgram`

```
class cemi.sklearn.lvq.LVQ(model, dist='l2', **kws)
```

Bases: `cemi.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` and `sklearn_lvq.LmrslvqModel` classes.

The `LVQ` class rebuilds a *sklearn-lvq* lvq model.

Parameters

- **model** (instance of `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel`) – The lvq model.
- **dist** (*str* or callable, optional) – Computes the distance between a prototype and a data point.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

You can use your own custom distance function by setting *dist* to a callable that can be called on a data point and returns a scalar.

The default is “l2”.

Note: *dist* must not be None.

prototypes

The prototypes.

Type *numpy.array*

labels

The labels of the prototypes.

Type *numpy.array*

dist

The distance function.

Type *callable*

model

The original *sklearn-lvq* model.

Type *object*

model_class

The class of the *sklearn-lvq* model.

Type *class*

dim

Dimensionality of the input data.

Type *int*

Raises `TypeError` – If *model* is not an instance of `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel`

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Builds a cost function where we penalize the minimum distance to the nearest prototype which is consistent with the target *y_target*.

Parameters

- **y_target** (*int*) – The target class.
- **pred** (*callable*, optional) – A callable that maps an input to an input. E.g. using the `ceml.optim.input_wrapper.InputWrapper` class.

If *pred* is `None`, no transformation is applied to the input before putting it into the loss function.

The default is `None`.

Returns Initialized cost function. Target label is *y_target*.

Return type `ceml.backend.jax.costfunctions.MinOfListDistCost`

predict(*x*)

Note: This function is a placeholder only.

This function does not predict anything and just returns the given input.

class `ceml.sklearn_lvq.LvqCounterfactual`(*model*, *dist='l2'*, *cqphelper=<class 'ceml.sklearn_lvq.CQPHelper'>*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.DCQP`

Class for computing a counterfactual of a lvq model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuilds a `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel` model.

Converts a `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel` instance into a `ceml.sklearn_lvq.LVQ` instance.

Parameters *model* (instance of `sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel`) – The *sklearn-lvq* lvq model.

Returns The wrapped lvq model.

Return type `ceml.sklearn_lvq.LVQ`

solve(*x_orig*, *y_target*, *regularization*, *features_whitelist*, *return_as_dict*, *optimizer_args*)

Approximately solves the DCQP by using the penalty convex-concave procedure.

Parameters \mathbf{x}_0 (*numpy.ndarray*) – The initial data point for the penalty convex-concave procedure - this could be anything, however a “good” initial solution might lead to a better result.

`ceml.sklearn.lvq.lvq_generate_counterfactual(model, x, y_target, features_whitelist=None, dist='l2', regularization='l1', C=1.0, optimizer='auto', optimizer_args=None, return_as_dict=True, done=None)`

Computes a counterfactual of a given input x .

Parameters

- **model** (a `sklearn.neighbors.sklearn_lvq.GlvqModel`, `sklearn_lvq.GmlvqModel`, `sklearn_lvq.LgmlvqModel`, `sklearn_lvq.RslvqModel`, `sklearn_lvq.MrslvqModel` or `sklearn_lvq.LmrslvqModel` instance.) – The lvq model that is used for computing the counterfactual.

Note: Only lvq models from sklearn-lvq are supported.

- **x** (*numpy.ndarray*) – The input x whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **dist** (*str* or callable, optional) – Computes the distance between a prototype and a data point.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

You can use your own custom distance function by setting *dist* to a callable that can be called on a data point and returns a scalar.

The default is “l1”.

Note: *dist* must not be None.

- **regularization** (*str* or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x . Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “l1”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If C is a list, all values in C are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of C are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

Use “auto” if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

The default is “auto”.

Learning vector quantization supports the use of mathematical programs for computing counterfactuals - set `optimizer` to “mp” for using a convex quadratic program (G(M)LVQ) or a DCQP (otherwise) for computing the counterfactual. Note that in this case the hyperparameter C is ignored. Because the DCQP is a non-convex problem, we are not guaranteed to find the best solution (it might even happen that we do not find a solution at all) - we use the penalty convex-concave procedure for approximately solving the DCQP.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – Not used.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if `return_as_dict` is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.models

```
ceml.sklearn.models.generate_counterfactual(model, x, y_target, features_whitelist=None, dist='l2',
                                             regularization='l1', C=1.0, optimizer='auto',
                                             optimizer_args=None, return_as_dict=True, done=None)
```

Computes a counterfactual of a given input x .

Parameters

- **model** (*object*) – The sklearn model that is used for computing the counterfactual.
- **x** (*numpy.ndarray*) – The input x whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `features_whitelist` is None, all features can be used.

The default is None.

- **dist** (*str* or callable, optional) – Computes the distance between a prototype and a data point.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

You can use your own custom distance function by setting *dist* to a callable that can be called on a data point and returns a scalar.

The default is “11”.

Note: *dist* must not be None.

Note: Only needed if *model* is a LVQ or KNN model!

- **regularization** (*str* or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optimizer.optimizer.desc_to_optim()` for details.

Use “auto” if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “auto”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If *True*, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If *False*, the results are returned as a triple.

The default is *True*.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is *None*, the output/prediction of the counterfactual must match *y_target* exactly.

The default is *None*.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is *False*

Return type *dict* or *triple*

Raises **ValueError** – If *model* contains an unsupported model.

ceml.sklearn.naivebayes

class ceml.sklearn.naivebayes.GaussianNB(*model*, ***kws*)

Bases: *ceml.model.model.ModelWithLoss*

Class for rebuilding/wrapping the sklearn.naive_bayes.GaussianNB class

The *GaussianNB* class rebuilds a gaussian naive bayes model from a given set of parameters (priors, means and variances).

Parameters **model** (instance of sklearn.naive_bayes.GaussianNB) – The gaussian naive bayes model.

class_priors

Class dependent priors.

Type *numpy.ndarray*

means

Class and feature dependent means.

Type *numpy.array*

variances

Class and feature dependent variances.

Type *numpy.ndarray*

dim

Dimensionality of the input data.

Type *int*

is_binary

True if *model* is a binary classifier, False otherwise.

Type *boolean*

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Build a negative-log-likelihood cost function where the target is *y_target*.

Parameters

- **y_target** (*int*) – The target class.
- **pred** (*callable*, optional) – A callable that maps an input to the output (class probabilities).
If *pred* is *None*, the class method *predict* is used for mapping the input to the output (class probabilities)
The default is *None*.

Returns Initialized negative-log-likelihood cost function. Target label is *y_target*.

Return type `ceml.backend.jax.costfunctions.NegLogLikelihoodCost`

predict(*x*)

Predict the output of a given input.

Computes the class probabilities for a given input *x*.

Parameters **x** (*numpy.ndarray*) – The input *x* that is going to be classified.

Returns An array containing the class probabilities.

Return type *jax.numpy.array*

class `ceml.sklearn.naivebayes.GaussianNbCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.SDP`, `ceml.optim.cvx.DCQP`

Class for computing a counterfactual of a gaussian naive bayes model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuild a `sklearn.naive_bayes.GaussianNB` model.

Converts a `sklearn.naive_bayes.GaussianNB` into a `ceml.sklearn.naivebayes.GaussianNB`.

Parameters **model** (instance of `sklearn.naive_bayes.GaussianNB`) – The *sklearn* gaussian naive bayes model.

Returns The wrapped gaussian naive bayes model.

Return type `ceml.sklearn.naivebayes.GaussianNB`

solve(*x_orig*, *y_target*, *regularization*, *features_whitelist*, *return_as_dict*, *optimizer_args*)

Approximately solves the DCQP by using the penalty convex-concave procedure.

Parameters **x0** (*numpy.ndarray*) – The initial data point for the penalty convex-concave procedure - this could be anything, however a “good” initial solution might lead to a better result.

`ceml.sklearn.naivebayes.gaussiannb_generate_counterfactual`(*model*, *x*, *y_target*,
features_whitelist=None,
regularization='l1', *C=1.0*,
optimizer='auto',
optimizer_args=None,
return_as_dict=True, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.naive_bayes.GaussianNB` instance.) – The gaussian naive bayes model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input x whose prediction has to be explained.
- **y_target** (`int` or `float` or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `features_whitelist` is None, all features can be used.

The default is None.

- **regularization** (`str` or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x . Supported values:
 - 11: Penalizes the absolute deviation.
 - 12: Penalizes the squared deviation.

`regularization` can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If `regularization` is None, no regularization is used.

The default is “11”.

- **C** (`float` or `list(float)`, optional) – The regularization strength. If C is a list, all values in C are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of C are tried.

C is ignored if no regularization is used (`regularization=None`).

The default is 1.0

- **optimizer** (`str` or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

Use “auto” if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

The default is “auto”.

Gaussian naive Bayes supports the use of mathematical programs for computing counterfactuals - set `optimizer` to “mp” for using a semi-definite program (binary classifier) or a DCQP (otherwise) for computing the counterfactual. Note that in this case the hyperparameter C is ignored. Because the DCQP is a non-convex problem, we are not guaranteed to find the best solution (it might even happen that we do not find a solution at all) - we use the penalty convex-concave procedure for approximately solving the DCQP.

- **optimizer_args** (`dict`, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – Not used.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.Lda

class ceml.sklearn.Lda(*model*, ***kws*)

Bases: *ceml.model.model.ModelWithLoss*

Class for rebuilding/wrapping the sklearn.discriminant_analysis.LinearDiscriminantAnalysis class.

The *Lda* class rebuilds a lda model from a given parameters.

Parameters *model* (instance of sklearn.discriminant_analysis.LinearDiscriminantAnalysis) – The lda model.

class_priors

Class dependend priors.

Type *numpy.ndarray*

means

Class dependend means.

Type *numpy.ndarray*

sigma_inv

Inverted covariance matrix.

Type *numpy.ndarray*

dim

Dimensionality of the input data.

Type *int*

Raises TypeError – If *model* is not an instance of sklearn.discriminant_analysis.LinearDiscriminantAnalysis

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Build a negative-log-likelihood cost function where the target is *y_target*.

Parameters

- **y_target** (*int*) – The target class.

- **pred** (*callable*, optional) – A callable that maps an input to the output (class probabilities).
If *pred* is None, the class method *predict* is used for mapping the input to the output (class probabilities)
The default is None.

Returns Initialized negative-log-likelihood cost function. Target label is *y_target*.

Return type `ceml.backend.jax.costfunctions.NegLogLikelihoodCost`

predict(*x*)

Predict the output of a given input.

Computes the class probabilities for a given input *x*.

Parameters **x** (*numpy.ndarray*) – The input *x* that is going to be classified.

Returns An array containing the class probabilities.

Return type *jax.numpy.array*

class `ceml.sklearn.lda.LdaCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.ConvexQuadraticProgram`, `ceml.optim.cvx.PlausibleCounterfactualOfHyperplaneClassifier`

Class for computing a counterfactual of a lda model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuild a `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` model.

Converts a `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` into a `ceml.sklearn.lda.Lda`.

Parameters **model** (instance of `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`) – The *sklearn* lda model - note that *store_covariance* must be set to True.

Returns The wrapped qda model.

Return type `ceml.sklearn.lda.Lda`

`ceml.sklearn.lda.lda_generate_counterfactual`(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='mp'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*, *plausibility=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` instance.) – The lda model that is used for computing the counterfactual.
- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.
If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*. Supported values:
 - 11: Penalizes the absolute deviation.
 - 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

Linear discriminant analysis supports the use of mathematical programs for computing counterfactuals - set *optimizer* to “mp” for using a convex quadratic program for computing the counterfactual. Note that in this case the hyperparameter *C* is ignored.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “mp”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – Not used.
- **plausibility** (*dict*, optional.) – If set to a valid dictionary (see `ceml.sklearn.plausibility.prepare_computation_of_plausible_counterfactuals()`), a plausible counterfactual (as proposed in Artelt et al. 2020) is computed. Note that in this case, all other parameters are ignored.

If *plausibility* is None, the closest counterfactual is computed.

The default is None.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(*x_cf*, *y_cf*, *delta*) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

cemi.sklearn.qda

class `cemi.sklearn.qda.Qda(model, **kws)`

Bases: `cemi.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` class.

The `Qda` class rebuilds a lda model from a given parameters.

Parameters `model` (instance of `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`) – The qda model.

class_priors

Class dependend priors.

Type `numpy.ndarray`

means

Class dependend means.

Type `numpy.ndarray`

sigma_inv

Class dependend inverted covariance matrices.

Type `numpy.ndarray`

dim

Dimensionality of the input data.

Type `int`

is_binary

True if *model* is a binary classifier, False otherwise.

Type `boolean`

Raises TypeError – If *model* is not an instance of `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Build a negative-log-likelihood cost function where the target is *y_target*.

Parameters

- **y_target** (`int`) – The target class.
- **pred** (`callable`, optional) – A callable that maps an input to the output (class probabilities).
If *pred* is None, the class method `predict` is used for mapping the input to the output (class probabilities)
The default is None.

Returns Initialized negative-log-likelihood cost function. Target label is *y_target*.

Return type `ceml.backend.jax.costfunctions.NegLogLikelihoodCost`

predict(*x*)

Predict the output of a given input.

Computes the class probabilities for a given input *x*.

Parameters **x** (*numpy.ndarray*) – The input *x* that is going to be classified.

Returns An array containing the class probabilities.

Return type *jax.numpy.array*

class `ceml.sklearn.qda.QdaCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.SDP`, `ceml.optim.cvx.DCQP`

Class for computing a counterfactual of a qda model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuild a `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` model.

Converts a `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` into a `ceml.sklearn.qda.Qda`.

Parameters **model** (instance of `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis`) – The *sklearn* qda model - note that *store_covariance* must be set to True.

Returns The wrapped qda model.

Return type `ceml.sklearn.qda.Qda`

solve(*x_orig*, *y_target*, *regularization*, *features_whitelist*, *return_as_dict*, *optimizer_args*)

Approximately solves the DCQP by using the penalty convex-concave procedure.

Parameters **x0** (*numpy.ndarray*) – The initial data point for the penalty convex-concave procedure - this could be anything, however a “good” initial solution might lead to a better result.

`ceml.sklearn.qda.qda_generate_counterfactual`(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='auto'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis` instance.) – The qda model that is used for computing the counterfactual.
- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.
If *features_whitelist* is None, all features can be used.
The default is None.
- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*. Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or `None` if no regularization is requested.

If *regularization* is `None`, no regularization is used.

The default is "11".

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is "nelder-mead".

Quadratic discriminant analysis supports the use of mathematical programs for computing counterfactuals - set *optimizer* to "mp" for using a semi-definite program (binary classifier) or a DCQP (otherwise) for computing the counterfactual. Note that in this case the hyperparameter *C* is ignored. Because the DCQP is a non-convex problem, we are not guaranteed to find the best solution (it might even happen that we do not find a solution at all) - we use the penalty convex-concave procedure for approximately solving the DCQP.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – Not used.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is `False`

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.pipeline

class `ceml.sklearn.pipeline.PipelineCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`

Class for computing a counterfactual of a softmax regression model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

build_loss(*regularization*, *x_orig*, *y_target*, *pred*, *grad_mask*, *C*, *input_wrapper*)

Build a loss function.

Overwrites the `build_loss` method from base class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

Parameters

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`) – Regularizer of the counterfactual. Penalty for deviating from the original input x .

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if your cost function is differentiable) or `None` if no regularization is requested.

If *regularization* is `None`, no regularization is used.

- **x_orig** (`numpy.array`) – The original input whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested output.
- **pred** (*callable*) – A callable that maps an input to the output.
If *pred* is `None`, the class method `predict` is used for mapping the input to the output.
- **grad_mask** (`numpy.array`) – Gradient mask determining which dimensions can be used.
- **C** (*float* or *list(float)*) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.
C is ignored if no regularization is used (*regularization=None*).
- **input_wrapper** (*callable*) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

Returns Initialized cost function. Target is set to *y_target*.

Return type `ceml.costfunctions.costfunctions.CostFunction`

compute_counterfactual(*x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='auto'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input x .

Parameters

- **x** (`numpy.ndarray`) – The data point x whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested prediction of the counterfactual.

- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is *None*, all features can be used.

The default is *None*.

- **regularization** (*str* or *ceml.costfunctions.costfunctions.CostFunction*, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*. Supported values:

– 11: Penalizes the absolute deviation.

– 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of *ceml.costfunctions.costfunctions.CostFunction* (or *ceml.costfunctions.costfunctions.DifferentiableCostFunction* if the cost function is differentiable) or *None* if no regularization is requested.

If *regularization* is *None*, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

If no regularization is used (*regularization=None*), *C* is ignored.

The default is 1.0

- **optimizer** (*str* or instance of *ceml.optim.optimizer.Optimizer*, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See *ceml.optim.optimizer.prepare_optim()* for details.

Use “auto” if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

As an alternative, we can use any (custom) optimizer that is derived from the *ceml.optim.optimizer.Optimizer* class.

Some models (see paper) support the use of mathematical programs for computing counterfactuals. In this case, you can use the option “mp” - please read the documentation of the corresponding model for further information.

The default is “auto”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is *None*.

- **return_as_dict** (*boolean*, optional) – If *True*, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If *False*, the results are returned as a triple.

The default is *True*.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is *None*, the output/prediction of the counterfactual must match *y_target* exactly.

The default is *None*.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

rebuild_model(*model*)

Rebuild a `sklearn.pipeline.Pipeline` model.

Converts a `sklearn.pipeline.Pipeline` into a `ceml.sklearn.pipeline.PipelineModel`.

Parameters *model* (instance of `sklearn.pipeline.Pipeline`) – The *sklearn* pipeline model.

Returns The wrapped pipeline model.

Return type `ceml.sklearn.pipeline.Pipeline`

class `ceml.sklearn.pipeline.PipelineModel`(*models*, ***kws*)

Bases: `ceml.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.pipeline.Pipeline` class

The `PipelineModel` class rebuilds a pipeline model from a given list of *sklearn* models.

Parameters *models* (*list(object)*) – Ordered list of all *sklearn* models in the pipeline.

models

Ordered list of all *sklearn* models in the pipeline.

Type *list(objects)*

get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Builds a cost function where the target is *y_target*.

Parameters

- **y_target** (*int* or *float*) – The requested output.
- **pred** (*callable*, optional) – A callable that maps an input to the output.
If *pred* is None, the class method *predict* is used for mapping the input to the output.
The default is None.

Returns Initialized cost function. Target is set to *y_target*.

Return type `ceml.costfunctions.costfunctions.CostFunction`

predict(*x*)

Predicts the output of a given input.

Computes the prediction of a given input *x*.

Parameters *x* (*numpy.ndarray*) – The input *x*.

Returns Output of the pipeline (might be scalar or smth. higher-dimensional).

Return type *numpy.array*

`ceml.sklearn.pipeline.pipeline_generate_counterfactual` (*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='nelder-mead'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.pipeline.Pipeline` instance.) – The model pipeline that is used for computing the counterfactual.
- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “l1”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

Use “auto” if you do not know what optimizer to use - a suitable optimizer is chosen automatically.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

Some models (see paper) support the use of mathematical programs for computing counterfactuals. In this case, you can use the option “mp” - please read the documentation of the corresponding model for further information.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is None, the output/prediction of the counterfactual must match *y_target* exactly.

The default is None.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

cemi.sklearn.randomforest

```
class cemi.sklearn.randomforest.EnsembleVotingCost(models, y_target, input_wrapper=None,
                                                  epsilon=0, **kws)
```

Bases: *cemi.costfunctions.costfunctions.CostFunction*

Loss function of an ensemble of models.

The loss is the negative fraction of models that predict the correct output.

Parameters

- **models** (*list(object)*) – List of models
- **y_target** (*int, float* or a callable that returns True if a given prediction is accepted.) – The requested prediction.
- **input_wrapper** (*callable*, optional) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

The default is None.

score_impl(*x*)

Implementation of the loss function.

class `ceml.sklearn.randomforest.RandomForest(model, **kws)`

Bases: `ceml.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor` class.

Parameters `model` (instance of `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor`) – The random forest model.

Raises `TypeError` – If `model` is not an instance of `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor`

get_loss(`y_target`, `input_wrapper=None`)

Creates and returns a loss function.

Parameters

- **y_target** (*int, float* or a callable that returns True if a given prediction is accepted.) – The requested prediction.
- **input_wrapper** (*callable*) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

Returns Initialized loss function. The target output is `y_target`.

Return type `ceml.sklearn.randomforest.EnsembleVotingCost`

predict(`x`)

Predict the output of a given input.

Computes the class label of a given input `x`.

Parameters `x` (*numpy.ndarray*) – The input `x` that is going to be classified.

Returns Prediction.

Return type *int* or *float*

class `ceml.sklearn.randomforest.RandomForestCounterfactual(model, **kws)`

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`

Class for computing a counterfactual of a random forest model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

build_loss(`regularization`, `x_orig`, `y_target`, `pred`, `grad_mask`, `C`, `input_wrapper`)

Build the (non-differentiable) cost function: Regularization + Loss

compute_counterfactual(`x`, `y_target`, `features_whitelist=None`, `regularization='l1'`, `C=1.0`, `optimizer='nelder-mead'`, `optimizer_args=None`, `return_as_dict=True`, `done=None`)

Computes a counterfactual of a given input `x`.

Parameters

- **x** (*numpy.ndarray*) – The input `x` whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested prediction of the counterfactual.
- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `feature_whitelist` is None, all features can be used.

The default is None.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x . Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if the cost function is differentiable) or `None` if no regularization is requested.

If *regularization* is `None`, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

If no regularization is used (*regularization=None*), *C* is ignored.

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

Note: The cost function of a random forest model is not differentiable - we can not use a gradient-based optimization algorithm.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – A callable that returns `True` if a counterfactual with a given output/prediction is accepted and `False` otherwise.

If *done* is `None`, the output/prediction of the counterfactual must match *y_target* exactly.

The default is `None`.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

rebuild_model(*model*)

Rebuilds a `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor` model.

Converts a `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor` instance into a `ceml.sklearn.randomforest.RandomForest` instance.

Parameters *model* (instance of `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor`) – The *sklearn* random forest model.

Returns The wrapped random forest model.

Return type `ceml.sklearn.randomforest.RandomForest`

`ceml.sklearn.randomforest.randomforest_generate_counterfactual`(*model*, *x*, *y_target*,
features_whitelist=None,
regularization='l1', *C=1.0*,
optimizer='nelder-mead',
optimizer_args=None,
return_as_dict=True,
done=None)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.ensemble.RandomForestClassifier` or `sklearn.ensemble.RandomForestRegressor` instance.) – The random forest model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int` or `float` or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (`str` or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is `None`, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

Note: The cost function of a random forest model is not differentiable - we can not use a gradient-based optimization algorithm.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – Not used.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is `False`

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.isolationforest

class `ceml.sklearn.isolationforest.IsolationForest(model, **kws)`

Bases: `ceml.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.ensemble.IsolationForest` class.

Parameters **model** (instance of `sklearn.ensemble.IsolationForest`) – The isolation forest model.

Raises TypeError – If *model* is not an instance of `sklearn.ensemble.IsolationForest`

get_loss(*y_target*, *input_wrapper=None*)

Creates and returns a loss function.

Parameters

- **y_target** (*int*) – The target class - either +1 or -1
- **input_wrapper** (*callable*) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

Returns Initialized loss function. Target label is *y_target*.

Return type `ceml.sklearn.isolationforest.IsolationForestCost`

predict(*x*)

Predict the output of a given input.

Computes the class label of a given input *x*.

Parameters **x** (*numpy.ndarray*) – The input *x* that is going to be classified.

Returns Prediction.

Return type *int*

```
class ceml.sklearn.isolationforest.IsolationForestCost(models, y_target, input_wrapper=None,
                                                    epsilon=0, **kwds)
```

Bases: `ceml.costfunctions.costfunctions.CostFunction`

Loss function of an isolation forest.

The loss is the negative averaged length of the decision paths.

Parameters

- **models** (*list(object)*) – List of decision trees.
- **y_target** (*int*) – The requested prediction - either -1 or +1.
- **input_wrapper** (*callable*, optional) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

The default is None.

score_impl(*x*)

Implementation of the loss function.

```
class ceml.sklearn.isolationforest.IsolationForestCounterfactual(model, **kwds)
```

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`

Class for computing a counterfactual of an isolation forest model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

```
compute_counterfactual(x, y_target, features_whitelist=None, regularization='l1', C=1.0,
                      optimizer='nelder-mead', optimizer_args=None, return_as_dict=True,
                      done=None)
```

Computes a counterfactual of a given input *x*.

Parameters

- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested prediction of the counterfactual.
- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is None, all features can be used.

The default is None.

- **regularization** (*str* or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input x . Supported values:
 - 11: Penalizes the absolute deviation.
 - 12: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if the cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If C is a list, all values in C are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of C are tried.

If no regularization is used (*regularization=None*), C is ignored.

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optimizer.optimizer.desc_to_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

Note: The cost function of an isolation forest model is not differentiable - we can not use a gradient-based optimization algorithm.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is None, the output/prediction of the counterfactual must match *y_target* exactly.

The default is None.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

rebuild_model(*model*)

Rebuilds a `sklearn.ensemble.IsolationForest` model.

Converts a `sklearn.ensemble.IsolationForest` into a `ceml.sklearn.isolationforest.IsolationForest`.

Parameters *model* (instance of `sklearn.ensemble.IsolationForest`) – The *sklearn* isolation forest model.

Returns The wrapped isolation forest model.

Return type `ceml.sklearn.isolationforest.IsolationForest`

`ceml.sklearn.isolationforest.isolationforest_generate_counterfactual`(*model*, *x*, *y_target*,
features_whitelist=None,
regularization='l1',
C=1.0,
optimizer='nelder-mead',
optimizer_args=None,
return_as_dict=True)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.ensemble.IsolationForest` instance.) – The isolation forest model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int`) – The requested prediction of the counterfactual - either -1 or +1.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **regularization** (`str` or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

– l1: Penalizes the absolute deviation.

– l2: Penalizes the squared deviation.

regularization can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or None if no regularization is requested.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optimizer.optimizer.desc_to_optim()` for details.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “nelder-mead”.

Note: The cost function of an isolation forest model is not differentiable - we can not use a gradient-based optimization algorithm.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.softmaxregression

class `ceml.sklearn.softmaxregression.SoftmaxCounterfactual`(*model*, ***kws*)

Bases: `ceml.sklearn.counterfactual.SklearnCounterfactual`, `ceml.optim.cvx.MathematicalProgram`, `ceml.optim.cvx.ConvexQuadraticProgram`, `ceml.optim.cvx.PlausibleCounterfactualOfHyperplaneClassifier`

Class for computing a counterfactual of a softmax regression model.

See parent class `ceml.sklearn.counterfactual.SklearnCounterfactual`.

rebuild_model(*model*)

Rebuilds a `sklearn.linear_model.LogisticRegression` model.

Converts a `sklearn.linear_model.LogisticRegression` into a `ceml.sklearn.softmaxregression.SoftmaxRegression`.

Parameters *model* (instance of `sklearn.linear_model.LogisticRegression`) – The *sklearn* softmax regression model.

Returns The wrapped softmax regression model.

Return type `caml.sklearn.softmaxregression.SoftmaxRegression`

class `caml.sklearn.softmaxregression.SoftmaxRegression(model, **kws)`

Bases: `caml.model.model.ModelWithLoss`

Class for rebuilding/wrapping the `sklearn.linear_model.LogisticRegression` class.

The `SoftmaxRegression` class rebuilds a softmax regression model from a given weight vector and intercept.

Parameters `model` (instance of `sklearn.linear_model.LogisticRegression`) – The softmax regression model.

w

The weight vector (a matrix if we have more than two classes).

Type `numpy.ndarray`

b

The intercept/bias (a vector if we have more than two classes).

Type `numpy.ndarray`

dim

Dimensionality of the input data.

Type `int`

is_multiclass

True if `model` is a binary classifier, False otherwise.

Type `boolean`

Raises `TypeError` – If `model` is not an instance of `sklearn.linear_model.LogisticRegression`

get_loss(`y_target`, `pred=None`)

Creates and returns a loss function.

Builds a negative-log-likelihood cost function where the target is `y_target`.

Parameters

- **y_target** (`int`) – The target class.
- **pred** (`callable`, optional) – A callable that maps an input to the output (class probabilities).
If `pred` is None, the class method `predict` is used for mapping the input to the output (class probabilities)
The default is None.

Returns Initialized negative-log-likelihood cost function. Target label is `y_target`.

Return type `caml.backend.jax.costfunctions.NegLogLikelihoodCost`

predict(`x`)

Predict the output of a given input.

Computes the class probabilities for a given input `x`.

Parameters `x` (`numpy.ndarray`) – The input `x` that is going to be classified.

Returns An array containing the class probabilities.

Return type `jax.numpy.array`

`ceml.sklearn.softmaxregression.softmaxregression_generate_counterfactual`(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='mp'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*, *plausibility=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.linear_model.LogisticRegression` instance.) – The softmax regression model that is used for computing the counterfactual.

Note: `model.multi_class` must be set to *multinomial*.

- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int` or `float` or a callable that returns `True` if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `features_whitelist` is `None`, all features can be used.

The default is `None`.

- **regularization** (`str` or `ceml.costfunctions.costfunctions.CostFunction`, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

`regularization` can be a description of the regularization, an instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.CostFunctionDifferentiable` if your cost function is differentiable) or `None` if no regularization is requested.

If `regularization` is `None`, no regularization is used.

The default is “11”.

- **C** (`float` or `list(float)`, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (`regularization=None`).

The default is 1.0

- **optimizer** (`str` or instance of `ceml.optim.optimizer.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.prepare_optim()` for details.

Softmax regression supports the use of mathematical programs for computing counterfactuals - set *optimizer* to “mp” for using a convex quadratic program for computing the counterfactual. Note that in this case the hyperparameter *C* is ignored.

As an alternative, we can use any (custom) optimizer that is derived from the `ceml.optim.optimizer.Optimizer` class.

The default is “mp”.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – Not used.
- **plausibility** (*dict*, optional.) – If set to a valid dictionary (see `ceml.sklearn.plausibility.prepare_computation_of_plausible_counterfactuals()`), a plausible counterfactual (as proposed in Artelt et al. 2020) is computed. Note that in this case, all other parameters are ignored.

If *plausibility* is None, the closest counterfactual is computed.

The default is None.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type *dict* or *triple*

Raises Exception – If no counterfactual was found.

ceml.sklearn.utils

`ceml.sklearn.utils.build_regularization_loss(regularization, x, input_wrapper=None)`

Build a regularization loss.

Parameters

- **regularization** (*str*, `ceml.costfunctions.costfunctions.CostFunction` or None) – Description of the regularization, instance of `ceml.costfunctions.costfunctions.CostFunction` (or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if your cost function is differentiable) or None if no regularization is requested.

See `ceml.sklearn.utils.desc_to_regcost()` for a list of supported descriptions.

If no regularization is requested, an instance of `ceml.backend.jax.costfunctions.costfunctions.DummyCost` is returned. This cost function always outputs zero, no matter what the input is.

- **x** (*numpy.array*) – The original input from which we do not want to deviate much.

- **input_wrapper** (*callable*, optional) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

If *input_wrapper* is `None`, the input is passed without any modifications.

The default is `None`.

Returns An instance of `ceml.costfunctions.costfunctions.CostFunction` or the user defined, callable, regularization.

Return type *callable*

Raises **TypeError** – If *regularization* has an invalid type.

`ceml.sklearn.utils.desc_to_dist(desc)`

Converts a description of a distance metric into a *jax.numpy* function.

Supported descriptions:

- l1: l1-norm
- l2: l2-norm

Parameters **desc** (*str*) – Description of the distance metric.

Returns The distance function implemented as a *jax.numpy* function.

Return type *callable*

Raises **ValueError** – If *desc* contains an invalid description.

`ceml.sklearn.utils.desc_to_regcost(desc, x, input_wrapper)`

Converts a description of a regularization into a *jax.numpy* function.

Supported descriptions:

- l1: l1-regularization
- l2: l2-regularization

Parameters

- **desc** (*str*) – Description of the distance metric.
- **x** (*numpy.array*) – The original input from which we do not want to deviate much.
- **input_wrapper** (*callable*) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

Returns The regularization function implemented as a *jax.numpy* function.

Return type *callable*

Raises **ValueError** – If *desc* contains an invalid description.

1.6.2 ceml.tfkeras

ceml.tfkeras.counterfactual

class `ceml.tfkeras.counterfactual.TfCounterfactual`(*model*, ***kws*)

Bases: `ceml.model.counterfactual.Counterfactual`

Class for computing a counterfactual of a tensorflow model.

See parent class `ceml.model.counterfactual.Counterfactual`.

Parameters *model* (instance of `ceml.model.model.ModelWithLoss`) – The tensorflow model that is used for computing counterfactuals. The model has to be wrapped inside a class that is derived from the `ceml.model.model.ModelWithLoss` class.

Raises

- **TypeError** – If *model* is not an instance of `ceml.model.model.ModelWithLoss`.
- **Exception** – If eager execution is not enabled.

compute_counterfactual(*x*, *y_target*, *features_whitelist=None*, *regularization=None*, *C=1.0*, *optimizer='nelder-mead'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **x** (*numpy.ndarray*) – The input *x* whose prediction has to be explained.
- **y_target** (*int* or *float* or a callable that returns True if a given prediction is accepted.) – The requested prediction of the counterfactual.
- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is None, all features can be used.

The default is None.

- **regularization** (*str* or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

– 11: Penalizes the absolute deviation.

– 12: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is None, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or instance of `tf.train.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.desc_to_optim()` for details.

As an alternative, any optimizer from tensorflow can be used - *optimizer* must be an instance of `tf.train.Optimizer`.

The default is “nelder-mead”.

- **optimizer_args** (*dict*, optional) – Dictionary containing additional parameters for the optimization algorithm.

Supported parameters (keys):

- `tol`: Tolerance for termination
- `max_iter`: Maximum number of iterations

If *optimizer_args* is `None` or if some parameters are missing, default values are used.

The default is `None`.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – A callable that returns `True` if a counterfactual with a given output/prediction is accepted and `False` otherwise.

If *done* is `None`, the output/prediction of the counterfactual must match *y_target* exactly.

The default is `None`.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is `False`

Return type dict or triple

Raises Exception – If no counterfactual was found.

```
ceml.tfkeras.counterfactual.generate_counterfactual(model, x, y_target, features_whitelist=None,
                                                  regularization=None, C=1.0,
                                                  optimizer='nelder-mead', optimizer_args=None,
                                                  return_as_dict=True, done=None)
```

Computes a counterfactual of a given input *x*.

Parameters

- **model** (instance of `ceml.model.model.ModelWithLoss`) – The tensorflow model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int` or `float` or a callable that returns `True` if a given prediction is accepted.) – The requested prediction of the counterfactual.

- **feature_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is *None*, all features can be used.

The default is *None*.

- **regularization** (*str* or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is *None*, no regularization is used.

The default is “11”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

If no regularization is used (*regularization=None*), *C* is ignored.

The default is 1.0

- **optimizer** (*str* or instance of `tf.train.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.desc_to_optim()` for details.

As an alternative, any optimizer from tensorflow can be used - *optimizer* must be an an instance of `tf.train.Optimizer`.

The default is “nelder-mead”.

- **optimizer_args** (*dict*, optional) – Dictionary containing additional parameters for the optimization algorithm.

Supported parameters (keys):

- *tol*: Tolerance for termination
- *max_iter*: Maximum number of iterations

If *optimizer_args* is *None* or if some parameters are missing, default values are used.

The default is *None*.

- **return_as_dict** (*boolean*, optional) – If *True*, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If *False*, the results are returned as a triple.

The default is *True*.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is *None*, the output/prediction of the counterfactual must match *y_target* exactly.

The default is *None*.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type dict or triple

1.6.3 ceml.torch

ceml.torch.counterfactual

class `ceml.torch.counterfactual.TorchCounterfactual`(*model*, *device=torch.device*, ***kwargs*)

Bases: `ceml.model.counterfactual.Counterfactual`

Class for computing a counterfactual of a PyTorch model.

See parent class `ceml.model.counterfactual.Counterfactual`.

Parameters

- **model** (instance of `torch.nn.Module` and `ceml.model.model.ModelWithLoss`) – The PyTorch model that is used for computing counterfactuals. The model has to be wrapped inside a class that is derived from the classes `torch.nn.Module` and `ceml.model.model.ModelWithLoss`.
- **device** (`torch.device`) – Specifies the hardware device (e.g. cpu or gpu) we are working on.
The default is `torch.device("cpu")`.

Raises `TypeError` – If model is not an instance of `torch.nn.Module` and `ceml.model.model.ModelWithLoss`.

compute_counterfactual(*x*, *y_target*, *features_whitelist=None*, *regularization=None*, *C=1.0*, *optimizer='nelder-mead'*, *optimizer_args=None*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int` or `float`) – The requested prediction of the counterfactual.
- **feature_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.
If *feature_whitelist* is None, all features can be used.
The default is None.
- **regularization** (`str` or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.
Supported values:

- l1: Penalizes the absolute deviation.
- l2: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is `None`, no regularization is used.

The default is “l1”.

- **C** (*float* or *list(float)*, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

C is ignored if no regularization is used (*regularization=None*).

The default is 1.0

- **optimizer** (*str* or class that is derived from `torch.optim.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.desc_to_optim()` for details.

As an alternative, any optimizer from PyTorch can be used - *optimizer* must be class that is derived from `torch.optim.Optimizer`.

The default is “nelder-mead”.

- **optimizer_args** (*dict*, optional) – Dictionary containing additional parameters for the optimization algorithm.

Supported parameters (keys):

- *args*: Arguments of the optimization algorithm (e.g. learning rate, momentum, ...)
- *lr_scheduler*: Learning rate scheduler (see `torch.optim.lr_scheduler`)
- *lr_scheduler_args*: Arguments of the learning rate scheduler
- *tol*: Tolerance for termination
- *max_iter*: Maximum number of iterations

If *optimizer_args* is `None` or if some parameters are missing, default values are used.

The default is `None`.

Note: The parameters *tol* and *max_iter* are passed to all optimization algorithms. Whereas the other parameters are only passed to PyTorch optimizers.

- **return_as_dict** (*boolean*, optional) – If `True`, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If `False`, the results are returned as a triple.

The default is `True`.

- **done** (*callable*, optional) – A callable that returns `True` if a counterfactual with a given output/prediction is accepted and `False` otherwise.

If *done* is `None`, the output/prediction of the counterfactual must match *y_target* exactly.

The default is `None`.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in 'x_cf', its prediction in 'y_cf' and the changes to the original input in 'delta'.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type dict or triple

Raises Exception – If no counterfactual was found.

```
ceml.torch.counterfactual.generate_counterfactual(model, x, y_target, device=torch.device,
                                                features_whitelist=None, regularization=None,
                                                C=1.0, optimizer='nelder-mead',
                                                optimizer_args=None, return_as_dict=True,
                                                done=None)
```

Computes a counterfactual of a given input *x*.

Parameters

- **model** (instance of `torch.nn.Module` and `ceml.model.model.ModelWithLoss`) – The PyTorch model that is used for computing the counterfactual.
- **x** (`numpy.ndarray`) – The input *x* whose prediction has to be explained.
- **y_target** (`int` or `float`) – The requested prediction of the counterfactual.
- **device** (`torch.device`) – Specifies the hardware device (e.g. `cpu` or `gpu`) we are working on.

The default is `torch.device("cpu")`.

- **feature_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is `None`, all features can be used.

The default is `None`.

- **regularization** (`str` or callable, optional) – Regularizer of the counterfactual. Penalty for deviating from the original input *x*.

Supported values:

- 11: Penalizes the absolute deviation.
- 12: Penalizes the squared deviation.

You can use your own custom penalty function by setting *regularization* to a callable that can be called on a potential counterfactual and returns a scalar.

If *regularization* is `None`, no regularization is used.

The default is "11".

- **C** (`float` or `list(float)`, optional) – The regularization strength. If *C* is a list, all values in *C* are tried and as soon as a counterfactual is found, this counterfactual is returned and no other values of *C* are tried.

If no regularization is used (*regularization=None*), *C* is ignored.

The default is 1.0

- **optimizer** (*str* or class that is derived from `torch.optim.Optimizer`, optional) – Name/Identifier of the optimizer that is used for computing the counterfactual. See `ceml.optim.optimizer.desc_to_optim()` for details.

As an alternative, any optimizer from PyTorch can be used - *optimizer* must be class that is derived from `torch.optim.Optimizer`.

The default is “nelder-mead”.

- **optimizer_args** (*dict*, optional) – Dictionary containing additional parameters for the optimization algorithm.

Supported parameters (keys):

- `args`: Arguments of the optimization algorithm (e.g. learning rate, momentum, ...)
- `lr_scheduler`: Learning rate scheduler (see `torch.optim.lr_scheduler`)
- `lr_scheduler_args`: Arguments of the learning rate scheduler
- `tol`: Tolerance for termination
- `max_iter`: Maximum number of iterations

If *optimizer_args* is None or if some parameters are missing, default values are used.

The default is None.

Note: The parameters *tol* and *max_iter* are passed to all optimization algorithms. Whereas the other parameters are only passed to PyTorch optimizers.

- **return_as_dict** (*boolean*, optional) – If True, returns the counterfactual, its prediction and the needed changes to the input as dictionary. If False, the results are returned as a triple.

The default is True.

- **done** (*callable*, optional) – A callable that returns *True* if a counterfactual with a given output/prediction is accepted and *False* otherwise.

If *done* is None, the output/prediction of the counterfactual must match *y_target* exactly.

The default is None.

Note: In case of a regression it might not always be possible to achieve a given output/prediction exactly.

Returns

A dictionary where the counterfactual is stored in ‘x_cf’, its prediction in ‘y_cf’ and the changes to the original input in ‘delta’.

(x_cf, y_cf, delta) : triple if *return_as_dict* is False

Return type dict or triple

ceml.torch.utils

`ceml.torch.utils.build_regularization_loss(regularization, x, input_wrapper=None)`

Builds a regularization loss.

Parameters

- **desc** (*str*, *callable* or *None*) – Description of the regularization, a callable regularization (not mandatory but we recommend to put your custom regularization into a class and make it a child of `ceml.costfunctions.costfunctions.CostFunction` or `ceml.costfunctions.costfunctions.DifferentiableCostFunction` if your cost function is differentiable) or *None* if no regularization is desired.

See `ceml.torch.utils.desc_to_regcost()` for a list of supported descriptions.

If no regularization is requested, an instance of `ceml.backend.torch.costfunctions.costfunctions.DummyCost` is returned. This cost function always outputs zero, no matter what the input is.

- **x** (*numpy.array*) – The original input from which we do not want to deviate much.
- **input_wrapper** (*callable*, optional) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

If *input_wrapper* is *None*, input is passed without any modifications.

The default is *None*.

Returns An instance of `ceml.costfunctions.costfunctions.CostFunction` or the user defined, callable, regularization.

Return type *callable*

Raises **TypeError** – If *regularization* has an invalid type.

`ceml.torch.utils.desc_to_dist(desc)`

Converts a description of a distance metric into a *torch* function.

Supported descriptions:

- 11: 11-norm
- 12: 12-norm

Parameters **desc** (*str*) – Description of the distance metric.

Returns The distance function implemented as a *torch* function.

Return type *callable*

Raises **ValueError** – If *desc* contains an invalid description.

`ceml.torch.utils.desc_to_regcost(desc, x, input_wrapper)`

Converts a description of a regularization into a *torch* function.

Supported descriptions:

- 11: 11-regularization
- 12: 12-regularization

Parameters

- **desc** (*str*) – Description of the distance metric.

- **x** (*numpy.array*) – The original input from which we do not want to deviate much.
- **input_wrapper** (*callable*) – Converts the input (e.g. if we want to exclude some features/dimensions, we might have to include these missing features before applying any function to it).

Is ignored!

Returns The regularization function implemented as a *torch* function.

Return type *callable*

Raises **ValueError** – If *desc* contains an invalid description.

1.6.4 caml.costfunctions

caml.costfunctions.costfunctions

class `caml.costfunctions.costfunctions.CostFunction`(*input_to_output=None, **kws*)

Bases: `abc.ABC`

Base class of a cost function.

Note: The class `CostFunction` can not be instantiated because it contains an abstract method.

abstract `score_impl(x)`

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input *x*.

Note: All derived classes must implement this method.

class `caml.costfunctions.costfunctions.CostFunctionDifferentiable`(*input_to_output=None, **kws*)

Bases: `caml.costfunctions.costfunctions.CostFunction`

Base class of a differentiable cost function.

Note: The class `CostFunctionDifferentiable` can not be instantiated because it contains an abstract method.

abstract `grad(mask=None)`

Computes the gradient.

Abstract method for computing the gradient of the cost function.

Returns A function that computes the gradient for a given input.

Return type *callable*

Note: All derived classes must implement this method.

class `ceml.costfunctions.costfunctions.RegularizedCost`(*penalize_input*, *penalize_output*, *C=1.0*,
***kws*)

Bases: `ceml.costfunctions.costfunctions.CostFunction`

Regularized cost function.

The `RegularizedCost` class implements a regularized cost function. The cost function is the sum of a regularization term (weighted by the regularization strength *C*) and a term that penalizes wrong predictions.

Parameters

- **input_to_output** (*callable*) – Function for computing the output from the input. The output is then put into the *penalize_output* function.
- **penalize_input** (`ceml.costfunctions.costfunctions.CostFunction`) – Regularization of the input.
- **penalize_output** (`ceml.costfunctions.costfunctions.CostFunction`) – Loss function for the output/prediction.
- **C** (*float*) – Regularization strength.

score_impl(*x*)

Applying the cost function to a given input.

Computes the cost function fo a given input *x*.

Parameters **x** (*numpy.array*) – Value of the unknown variable.

Returns The loss/cost.

Return type *float*

1.6.5 ceml.model

`ceml.model.counterfactual`

class `ceml.model.counterfactual.Counterfactual`(***kws*)

Bases: `abc.ABC`

Base class for computing a counterfactual.

Note: The class `Counterfactual` can not be instantiated because it contains an abstract method.

abstract `compute_counterfactual`()

Compute a counterfactual.

Abstract method for computing a counterfactual.

Note: All derived classes must implement this method.

ceml.model.model

class `ceml.model.model.Model(**kws)`

Bases: `abc.ABC`

Base class of a model.

Note: The class `Model` can not be instantiated because it contains an abstract method.

abstract predict(*x*)

Predict the output of a given input.

Abstract method for computing a prediction.

Note: All derived classes must implement this method.

class `ceml.model.model.ModelWithLoss(**kws)`

Bases: `ceml.model.model.Model`

Base class of a model that comes with its own loss function.

Note: The class `ModelWithLoss` can not be instantiated because it contains an abstract method.

abstract get_loss(*y_target*, *pred=None*)

Creates and returns a loss function.

Builds a cost function where the target is *y_target*.

Returns The cost function.

Return type `ceml.costfunctions.costfunction.Cost`

Note: All derived classes must implement this method.

1.6.6 ceml.optim

ceml.optim.input_wrapper

class `ceml.optim.input_wrapper.InputWrapper(features_whitelist, x_orig, **kws)`

Bases: `object`

Class for wrapping an input.

The `InputWrapper` class wraps an inputs to hide some of its dimensions/features to subsequent methods.

Parameters

- **features_whitelist** (*list(int)*) – A non-empty list of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *feature_whitelist* is `None`, all features can be used.

- **x_orig** (*numpy.array*) – The original input that is going to be wrapped - this is the input whose prediction has to be explained.

Raises ValueError – If *features_whitelist* is an empty list.

complete(*x*)

Completing a given input.

Adds the fixed/hidden dimensions from the original input to the given input.

Parameters **x** (*array_like*:) – The input to be completed.

Returns The completed input.

Return type *numpy.array*

extract_from(*x*)

Extracts the whitelisted dimensions from a given input.

Parameters **x** (*array_like*:) – The input to be processed.

Returns The extracted input - only whitelisted features/dimensions are kept.

Return type *numpy.array*

ceml.optimizer.optimizer

class `ceml.optim.optimizer.BFGS`(***kws*)

Bases: `ceml.optim.optimizer.Optimizer`

BFGS optimization algorithm.

Note: The BFGS optimization algorithm is a Quasi-Newton method.

init(*f*, *f_grad*, *x0*, *tol=None*, *max_iter=None*)

Initializes all parameters.

Parameters

- **f** (*callable*) – The objective that is minimized.
- **f_grad** (*callable*) – The gradient of the objective.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **tol** (*float*, optional) – Tolerance for termination.
tol=None is equivalent to *tol=0*.
The default is None.
- **max_iter** (*int*, optional) – Maximum number of iterations.
If *max_iter* is None, the default value of the particular optimization algorithm is used.
Default is None.

class `ceml.optim.optimizer.ConjugateGradients`(***kws*)

Bases: `ceml.optim.optimizer.Optimizer`

Conjugate gradients optimization algorithm.

init(*f*, *f_grad*, *x0*, *tol=None*, *max_iter=None*)

Initializes all parameters.

Parameters

- **f** (*callable*) – The objective that is minimized.

- **f_grad** (*callable*) – The gradient of the objective.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **tol** (*float*, optional) – Tolerance for termination.
tol=None is equivalent to *tol=0*.
The default is None.
- **max_iter** (*int*, optional) – Maximum number of iterations.
If *max_iter* is None, the default value of the particular optimization algorithm is used.
Default is None.

class `ceml.optim.optimizer.NelderMead(**kws)`

Bases: `ceml.optim.optimizer.Optimizer`

Nelder-Mead optimization algorithm.

Note: The Nelder-Mead algorithm is a gradient-free optimization algorithm.

init(*f*, *x0*, *tol=None*, *max_iter=None*)

Initializes all parameters.

Parameters

- **f** (*callable*) – The objective that is minimized.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **tol** (*float*, optional) – Tolerance for termination.
tol=None is equivalent to *tol=0*.
The default is None.
- **max_iter** (*int*, optional) – Maximum number of iterations.
If *max_iter* is None, the default value of the particular optimization algorithm is used.
Default is None.

class `ceml.optim.optimizer.Optimizer(**kws)`

Bases: `abc.ABC`

Abstract base class of an optimizer.

All optimizers must be derived from the `Optimizer` class.

Note: Any class derived from `Optimizer` has to implement the abstract methods `init`, `__call__` and `is_grad_based`.

class `ceml.optim.optimizer.Powell(**kws)`

Bases: `ceml.optim.optimizer.Optimizer`

Powell optimization algorithm.

Note: The Powell algorithm is a gradient-free optimization algorithm.

init(*f*, *x0*, *tol=None*, *max_iter=None*)

Initializes all parameters.

Parameters

- **f** (*callable*) – The objective that is minimized.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **tol** (*float*, optional) – Tolerance for termination.

tol=None is equivalent to *tol=0*.

The default is None.

- **max_iter** (*int*, optional) – Maximum number of iterations.

If *max_iter* is None, the default value of the particular optimization algorithm is used.

Default is None.

`ceml.optim.optimizer.is_optimizer_grad_based`(*optim*)

Determines whether a specific optimization algorithm (specified by a description in *desc*) needs a gradient.

Supported descriptions:

- `nelder-mead`: *Gradient-free* Nelder-Mead optimizer (also called Downhill-Simplex)
- `powell`: *Gradient-free* Powell optimizer
- `bfgs`: BFGS optimizer
- `cg`: Conjugate gradients optimizer

Parameters **optim** (*str* or instance of `ceml.optim.optimizer.Optimizer`) – Description of the optimization algorithm or an instance of `ceml.optim.optimizer.Optimizer`.

Returns True if the optimization algorithm needs a gradient, False otherwise.

Return type *bool*

Raises

- **ValueError** – If *optim* contains an invalid description.
- **TypeError** – If *optim* is neither a string nor an instance of `ceml.optim.optimizer.Optimizer`.

`ceml.optim.optimizer.prepare_optimizer`(*optim*, *f*, *x0*, *f_grad=None*, *optimizer_args=None*)

Creates and initializes an optimization algorithm (instance of `ceml.optim.optimizer.Optimizer`) specified by a description of the algorithm.

Supported descriptions:

- `nelder-mead`: *Gradient-free* Nelder-Mead optimizer (also called downhill simplex method)
- `powell`: *Gradient-free* Powell optimizer
- `bfgs`: BFGS optimizer
- `cg`: Conjugate gradients optimizer

Parameters

- **optim** (*str* or instance of `ceml.optim.optimizer.Optimizer`) – Description of the optimization algorithm or an instance of `ceml.optim.optimizer.Optimizer`.

- **f** (instance of `ceml.costfunctions.costfunctions.CostFunction` or *callable*) – The objective that has to be minimized.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **f_grad** (*callable*, optional) – The gradient of the objective.

If *f_grad* is None, no gradient is used. Note that some optimization algorithms require a gradient!

The default is None.

- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

Returns An instance of `ceml.optim.optimizer.Optimizer`

Return type *callable*

Raises

- **ValueError** – If *optim* contains an invalid description or if no gradient is specified but and *optim* describes a gradient based optimization algorithm.
- **TypeError** – If *optim* is neither a string nor an instance of `ceml.optim.optimizer.Optimizer`.

`ceml.optimizer.ga`

```
class ceml.optim.ga.EvolutionaryOptimizer(population_size=100, select_by_fitness=0.5,
                                          mutation_prob=0.1, mutation_scaling=4.0, **kwds)
```

Bases: `ceml.optim.optimizer.Optimizer`

Evolutionary/Genetic optimization algorithm.

Note: This genetic algorithm is a gradient-free optimization algorithm.

This implementation encodes an individual as a *numpy.array* - if you want to use a different representation, you have to derive a new class from this class and reimplement all relevant methods.

Parameters

- **population_size** (*int*) – The size of the population
The default is 100
- **select_by_fitness** (*float*) – The fraction of individuals that is selected according to their fitness.
The default is 0.5
- **mutation_prob** (*float*) – The probability that an offspring is mutated.
The default is 0.1
- **mutation_scaling** (*float*) – Standard deviation of the normal distribution for mutating features.
The default is 4.0

compute_fitness(*x*)

Computes the fitness of a given individual *x*.

Parameters **x** (*numpy.array*) – The representation of the individual.

crossover(*x0*, *x1*)

Produces an offspring from the individuals *x0* and *x1*.

Note: This method implements **single-point crossover**. If you want to use a different crossover strategy, you have to derive a new class from this one and reimplement the method *crossover*

Parameters

- **x0** (*numpy.array*) – The representation of first individual.
- **x1** (*numpy.array*) – The representation of second individual.

Returns The representation of offspring created from *x0* and *x1*.

Return type *numpy.array*

init(*f*, *x0*, *tol=None*, *max_iter=None*)

Initializes all remaining parameters.

Parameters

- **f** (*callable*) – The objective that is minimized.
- **x0** (*numpy.array*) – The initial value of the unknown variable.
- **tol** (*float*, optional) – Tolerance for termination.
tol=None is equivalent to *tol=0*.
The default is 0.
- **max_iter** (*int*, optional) – Maximum number of iterations.
If *max_iter* is None, the default value of the particular optimization algorithm is used.
Default is None.

mutate(*x*)

Mutates a given individual *x*.

Parameters **x** (*numpy.array*) – The representation of the individual.

Returns The representation of the mutated individual *x*.

Return type *numpy.array*

select_candidates(*fitness*)

Selects a the most fittest individuals from the current population for producing offsprings.

Parameters **fitness** (*list(float)*) – Fitness of the individuals.

Returns The selected individuals.

Return type *list(numpy.array)*

validate(*x*)

Validates a given individual *x*.

This methods checks whether a given individual is valid (in the sense that the feature characteristics are valid) and if not it makes it valid by changing some of its features.

Note: This implementation is equivalent to the identity function. The input is returned without any changes - we do not restrict the input space! If you want to make some restrictions on the input space, you have to derive a new class from this one and reimplement the method *validate*.

Parameters \mathbf{x} (*numpy.array*) – The representation of the individual x .

Returns The representation of the validated individual.

Return type *numpy.array*

ceml.optimizer.cvx

class ceml.optim.cvx.ConvexQuadraticProgram(**kws)

Bases: abc.ABC, *ceml.optim.cvx.SupportAffinePreprocessing*

Base class for a convex quadratic program - for computing counterfactuals.

epsilon

“Small” non-negative number for relaxing strict inequalities.

Type *float*

build_solve_opt(*x_orig*, *y*, *features_whitelist=None*, *mad=None*, *optimizer_args=None*)

Builds and solves the convex quadratic optimization problem.

Parameters

- **$\mathbf{x_orig}$** (*numpy.ndarray*) – The original data point.
- **\mathbf{y}** (*int* or *float*) – The requested prediction of the counterfactual - e.g. a class label.
- **$\mathbf{features_whitelist}$** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If *features_whitelist* is None, all features can be used.

The default is None.

- **\mathbf{mad}** (*numpy.ndarray*, optional) – Weights for the weighted Manhattan distance.

If *mad* is None, the Euclidean distance is used.

The default is None.

- **$\mathbf{optimizer_args}$** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is None.

Returns

The solution of the optimization problem.

If no solution exists, *None* is returned.

Return type *numpy.ndarray*

class ceml.optim.cvx.DCQP(**kws)

Bases: *ceml.optim.cvx.SupportAffinePreprocessing*

Class for a difference-of-convex-quadratic program (DCQP) - for computing counterfactuals.

$$\min_{\vec{x} \in \mathbb{R}^d} \vec{x}^\top Q_0 \vec{x} + \vec{q}^\top \vec{x} + c - \vec{x}^\top Q_1 \vec{x} \quad \text{s.t.} \quad \vec{x}^\top A_0 \vec{x} + \vec{x}^\top \vec{b}_i + r_i - \vec{x}^\top A_1 \vec{x} \leq 0 \quad \forall i$$

pccp

Implementation of the penalty convex-concave procedure for approximately solving the DCQP.

Type instance of `ceml.optim.cvx.PenaltyConvexConcaveProcedure`

epsilon

“Small” non-negative number for relaxing strict inequalities.

Type `float`

build_program(`model, x_orig, y_target, Q0, Q1, q, c, A0_i, A1_i, b_i, r_i, features_whitelist=None, mad=None, optimizer_args=None`)

Builds the DCQP.

Parameters

- **model** (*object*) – The model that is used for computing the counterfactual - must provide a method `predict`.
- **x** (`numpy.ndarray`) – The data point x whose prediction has to be explained.
- **y_target** (*int* or *float*) – The requested prediction of the counterfactual - e.g. a class label.
- **Q0** (`numpy.ndarray`) – The matrix Q_0 of the DCQP.
- **Q1** (`numpy.ndarray`) – The matrix Q_1 of the DCQP.
- **q** (`numpy.ndarray`) – The vector q of the DCQP.
- **c** (*float*) – The constant c of the DCQP.
- **A0_i** (*list(numpy.ndarray)*) – List of matrices $A0_i$ of the DCQP.
- **A1_i** (*list(numpy.ndarray)*) – List of matrices $A1_i$ of the DCQP.
- **b_i** (*list(numpy.ndarray)*) – List of vectors b_i of the DCQP.
- **r_i** (*list(float)*) – List of constants r_i of the DCQP.
- **features_whitelist** (*list(int)*, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.
If `features_whitelist` is `None`, all features can be used.
The default is `None`.
- **mad** (`numpy.ndarray`, optional) – Weights for the weighted Manhattan distance.
If `mad` is `None`, the Euclidean distance is used.
The default is `None`.
- **optimizer_args** (*dict*, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.
The default is `None`.

solve(*x0*)

Approximately solves the DCQP by using the penalty convex-concave procedure.

Parameters **x0** (`numpy.ndarray`) – The initial data point for the penalty convex-concave procedure - this could be anything, however a “good” initial solution might lead to a better result.

class `ceml.optim.cvx.MathematicalProgram`(***kws*)

Bases: `object`

Base class for a mathematical program.

```
class ceml.optim.cvx.PenaltyConvexConcaveProcedure(model, Q0, Q1, q, c, A0_i, A1_i, b_i, r_i,
                                                features_whitelist=None, mad=None,
                                                optimizer_args=None, **kws)
```

Bases: `ceml.optim.cvx.SupportAffinePreprocessing`

Implementation of the penalty convex-concave procedure for approximately solving a DCQP.

```
class ceml.optim.cvx.SDP(**kws)
```

Bases: `abc.ABC`

Base class for a semi-definite program (SDP) - for computing counterfactuals.

epsilon

“Small” non-negative number for relaxing strict inequalities.

Type `float`

```
build_solve_opt(x_orig, y, features_whitelist=None, optimizer_args=None)
```

Builds and solves the SDP.

Parameters

- **x_orig** (`numpy.ndarray`) – The original data point.
- **y** (`int` or `float`) – The requested prediction of the counterfactual - e.g. a class label.
- **features_whitelist** (`list(int)`, optional) – List of feature indices (dimensions of the input space) that can be used when computing the counterfactual.

If `features_whitelist` is `None`, all features can be used.

The default is `None`.

- **optimizer_args** (`dict`, optional) – Dictionary for overriding the default hyperparameters of the optimization algorithm.

The default is `None`.

Returns

The solution of the optimization problem.

If no solution exists, `None` is returned.

Return type `numpy.ndarray`

```
class ceml.optim.cvx.SupportAffinePreprocessing(**kws)
```

Bases: `object`

Base class for a mathematical programs that support an affine preprocessing.

1.7 ceml.backend.jax

1.7.1 ceml.backend.jax.costfunctions

```
class ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax(input_to_output=None,
                                                                                **kws)
```

Bases: `ceml.costfunctions.costfunctions.CostFunctionDifferentiable`

Base class of differentiable cost functions implemented in jax.

grad(*mask=None*)

Computes the gradient with respect to the input.

Parameters **mask** (*numpy.array*, optional) – A mask that is multiplied elementwise to the gradient - can be used to mask some features/dimensions.

If *mask* is None, the gradient is not masked.

The default is None.

Returns The gradient.

Return type *callable*

class `ceml.backend.jax.costfunctions.costfunctions.DummyCost(**kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Dummy cost function - always returns zero.

score_impl(*x*)

Computes the loss - always returns zero.

class `ceml.backend.jax.costfunctions.costfunctions.L1Cost(x_orig, input_to_output=None, **kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

L1 cost function.

score_impl(*x*)

Computes the loss - l1 norm.

class `ceml.backend.jax.costfunctions.costfunctions.L2Cost(x_orig, input_to_output=None, **kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

L2 cost function.

score_impl(*x*)

Computes the loss - l2 norm.

class `ceml.backend.jax.costfunctions.costfunctions.LMadCost(x_orig, mad, input_to_output=None, **kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Manhattan distance weighted feature-wise with the inverse median absolute deviation (MAD).

score_impl(*x*)

Computes the loss.

class `ceml.backend.jax.costfunctions.costfunctions.MinOfListDistCost(dist, samples, input_to_output=None, **kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Minimum distance to a list of data points.

score_impl(*x*)

Computes the loss.

class `ceml.backend.jax.costfunctions.costfunctions.MinOfListDistExCost(omegas, samples, input_to_output=None, **kws)`

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Minimum distance to a list of data points.

In contrast to *MinOfListDistCost*, *MinOfListDistExCost* uses a user defined metric matrix (distortion of the Euclidean distance).

score_impl(*x*)
Computes the loss.

class `ceml.backend.jax.costfunctions.costfunctions.NegLogLikelihoodCost`(*input_to_output*,
y_target, ***kws*)

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Negative-log-likelihood cost function.

score_impl(*y*)
Computes the loss - negative-log-likelihood.

class `ceml.backend.jax.costfunctions.costfunctions.RegularizedCost`(*penalize_input*,
penalize_output, *C=1.0*,
***kws*)

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Regularized cost function.

score_impl(*x*)
Computes the loss.

class `ceml.backend.jax.costfunctions.costfunctions.SquaredError`(*input_to_output*, *y_target*,
***kws*)

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Squared error cost function.

score_impl(*y*)
Computes the loss - squared error.

class `ceml.backend.jax.costfunctions.costfunctions.TopKMinOfListDistCost`(*dist*, *samples*, *k*, *input_to_output=None*,
***kws*)

Bases: `ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax`

Computes the sum of the distances to the *k* closest samples.

score_impl(*x*)
Computes the loss.

1.7.2 ceml.backend.jax.preprocessing

class `ceml.backend.jax.preprocessing.AffinePreprocessing`(*A*, *b*, ***kws*)

Bases: `object`

Wrapper for an affine mapping (preprocessing)

class `ceml.backend.jax.preprocessing.MinMaxScaler`(*min_*, *scale*, ***kws*)

Bases: `ceml.model.model.Model`, `ceml.backend.jax.preprocessing.affine_preprocessing.AffinePreprocessing`

Wrapper for the min max scaler.

predict(*x*)
Computes the forward pass.

class `ceml.backend.jax.preprocessing.Model`(***kws*)

Bases: `abc.ABC`

Base class of a model.

Note: The class `Model` can not be instantiated because it contains an abstract method.

abstract `predict`(*x*)

Predict the output of a given input.

Abstract method for computing a prediction.

Note: All derived classes must implement this method.

class `ceml.backend.jax.preprocessing.Normalizer`(***kws*)

Bases: `ceml.model.model.Model`

Wrapper for the normalizer.

predict(*x*)

Computes the forward pass.

class `ceml.backend.jax.preprocessing.PCA`(*w*, ***kws*)

Bases: `ceml.model.model.Model`, `ceml.backend.jax.preprocessing.affine_preprocessing.AffinePreprocessing`

Wrapper for PCA - Principle component analysis.

predict(*x*)

Computes the forward pass.

class `ceml.backend.jax.preprocessing.PolynomialFeatures`(*powers*, ***kws*)

Bases: `ceml.model.model.Model`

Wrapper for polynomial feature transformation.

predict(*x*)

Computes the forward pass.

class `ceml.backend.jax.preprocessing.StandardScaler`(*mu*, *sigma*, ***kws*)

Bases: `ceml.model.model.Model`, `ceml.backend.jax.preprocessing.affine_preprocessing.AffinePreprocessing`

Wrapper for the standard scaler.

predict(*x*)

Computes the forward pass.

`ceml.backend.jax.preprocessing.reduce`(*function*, *sequence*[, *initial*]) → value

Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. If *initial* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

1.8 ceml.backend.torch

1.8.1 ceml.backend.torch.costfunctions

class `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch(**kws)`

Bases: `ceml.costfunctions.costfunctions.CostFunctionDifferentiable`

Base class of differentiable cost functions implemented in PyTorch.

grad()

Warning: Do not use this method!

Call `backward()` of the output tensor. After that, the gradient of each variable `myvar` - that is supposed to have gradient - can be accessed as `myvar.grad`

Raises `NotImplementedError` –

class `ceml.backend.torch.costfunctions.costfunctions.DummyCost(**kws)`

Bases: `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`

Dummy cost function - always returns zero.

score_impl(x)

Computes the loss - always return zero.

class `ceml.backend.torch.costfunctions.costfunctions.L1Cost(x_orig, **kws)`

Bases: `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`

L1 cost function.

score_impl(x)

Computes the loss - l1 norm.

class `ceml.backend.torch.costfunctions.costfunctions.L2Cost(x_orig, **kws)`

Bases: `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`

L2 cost function.

score_impl(x)

Computes the loss - l2 norm.

class `ceml.backend.torch.costfunctions.costfunctions.LMadCost(x_orig, mad, **kws)`

Bases: `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`

Manhattan distance weighted feature-wise with the inverse median absolute deviation (MAD).

score_impl(x)

Computes the loss.

class `ceml.backend.torch.costfunctions.costfunctions.MinOfListCost(dist, samples, **kws)`

Bases: `ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch`

Minimum distance to a list of data points.

score_impl(x)

Computes the loss.

```

class ceml.backend.torch.costfunctions.costfunctions.NegLogLikelihoodCost(y_target, **kws)
    Bases: ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch
    Negative-log-likelihood cost function.

    score_impl(y)
        Computes the loss - negative-log-likelihood.

class ceml.backend.torch.costfunctions.costfunctions.RegularizedCost(penalize_input,
                                                                    penalize_output, C=1.0,
                                                                    **kws)
    Bases: ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch
    Regularized cost function.

    score_impl(x)
        Computes the loss.

class ceml.backend.torch.costfunctions.costfunctions.SquaredError(y_target, **kws)
    Bases: ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch
    Squared error cost function.

    score_impl(y)
        Computes the loss - squared error.

```

1.8.2 ceml.backend.torch.optimizer

```

class ceml.backend.torch.optimizer.optimizer.TorchOptimizer(**kws)
    Bases: ceml.optim.optimizer.Optimizer
    Wrapper for a PyTorch optimization algorithm.

    The TorchOptimizer provides an interface for wrapping an arbitrary PyTorch optimization algorithm (see torch.optim) and minimizing a given loss function.

    init(model, loss, x, optim, optim_args, lr_scheduler=None, lr_scheduler_args=None, tol=None, max_iter=1,
          grad_mask=None, device=torch.device)
        Initializes all parameters.

```

Parameters

- **model** (instance of `torch.nn.Module`) – The model that is to be used.
- **loss** (instance of `ceml.backend.torch.costfunctions.RegularizedCost`) – The loss that has to be minimized.
- **x** (*numpy.ndarray*) – The starting value of *x* - usually this is the original input whose prediction has to be explained..
- **optim** (instance of `torch.optim.Optimizer`) – Optimizer for minimizing the loss.
- **optim_args** (*dict*) – Arguments of the optimization algorithm (e.g. learning rate, momentum, ...)
- **lr_scheduler** (Learning rate scheduler (see `torch.optim.lr_scheduler`)) – Learning rate scheduler (see `torch.optim.lr_scheduler`).
The default is None.
- **lr_scheduler_args** (*dict*, optional) – Arguments of the learning rate scheduler.
The default is None.

- **tol** (*float*, optional) – Tolerance for termination.
The default is 0.0
- **max_iter** (*int*, optional) – Maximum number of iterations.
The default is 1.
- **grad_mask** (*numpy.array*, optional) – Mask that is multiplied element wise on top of the gradient - can be used to hold some dimensions constant.
If *grad_mask* is *None*, no gradient mask is used.
The default is *None*.
- **device** (*torch.device*) – Specifies the hardware device (e.g. *cpu* or *gpu*) we are working on.
The default is *torch.device("cpu")*.

Raises **TypeError** – If the type of *loss* or *model* is not correct.

1.9 ceml.backend.tensorflow

1.9.1 ceml.backend.tensorflow.costfunctions

class `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf(**kws)`

Bases: `ceml.costfunctions.costfunctions.CostFunctionDifferentiable`

Base class of differentiable cost functions implemented in tensorflow.

grad()

Warning: Do not use this method!
Use 'tf.GradientTape' for computing the gradient.

Raises **NotImplementedError** –

class `ceml.backend.tensorflow.costfunctions.costfunctions.DummyCost(**kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

Dummy cost function - always returns zero.

score_impl(x)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input *x*.

Note: All derived classes must implement this method.

class `ceml.backend.tensorflow.costfunctions.costfunctions.L1Cost(x_orig, **kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

L1 cost function.

score_impl(x)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input x .

Note: All derived classes must implement this method.

class `ceml.backend.tensorflow.costfunctions.costfunctions.L2Cost(x_orig, **kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

L2 cost function.

score_impl(x)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input x .

Note: All derived classes must implement this method.

class `ceml.backend.tensorflow.costfunctions.costfunctions.LMadCost(x_orig, mad, **kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

Manhattan distance weighted feature-wise with the inverse median absolute deviation (MAD).

score_impl(x)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input x .

Note: All derived classes must implement this method.

class `ceml.backend.tensorflow.costfunctions.costfunctions.NegLogLikelihoodCost(y_target, **kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

Negative-log-likelihood cost function.

score_impl(y)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input x .

Note: All derived classes must implement this method.

class `ceml.backend.tensorflow.costfunctions.costfunctions.RegularizedCost(penalize_input, penalize_output, C=1.0, **kws)`

Bases: `ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

Regularized cost function.

score_impl(x)

Applying the cost function to a given input.

Abstract method for computing applying the cost function to a given input x .

Note: All derived classes must implement this method.

class `caml.backend.tensorflow.costfunctions.costfunctions.SquaredError`(*y_target*, ***kws*)
 Bases: `caml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf`

Squared error cost function.

score_impl(*y*)

Computes the loss - squared error.

1.9.2 `caml.backend.tensorflow.optimizer`

class `caml.backend.tensorflow.optimizer.optimizer.TfOptimizer`(***kws*)

Bases: `caml.optim.optimizer.Optimizer`

Wrapper for a tensorflow optimization algorithm.

The `TfOptimizer` provides an interface for wrapping an arbitrary tensorflow optimization algorithm (see `tf.train.Optimizer`) and minimizing a given loss function.

init(*model*, *loss*, *x*, *optim*, *tol=None*, *max_iter=1*, *grad_mask=None*)

Initializes all parameters.

Parameters

- **model** (*callable* or instance of `tf.keras.Model`) – The model that is to be used.
- **loss** (instance of `caml.backend.tensorflow.costfunctions.RegularizedCost`) – The loss that has to be minimized.
- **x** (`numpy.ndarray`) – The starting value of *x* - usually this is the original input whose prediction has to be explained..
- **optim** (instance of `tf.train.Optimizer`) – Optimizer for minimizing the loss.
- **tol** (*float*, optional) – Tolerance for termination.
The default is 0.0
- **max_iter** (*int*, optional) – Maximum number of iterations.
The default is 1.
- **grad_mask** (`numpy.array`, optional) – Mask that is multiplied element wise on top of the gradient - can be used to hold some dimensions constant.
If *grad_mask* is `None`, no gradient mask is used.
The default is `None`.

Raises **TypeError** – If the type of *loss* or *model* is not correct.

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

- `ceml.backend.jax.costfunctions.costfunctions`,
79
- `ceml.backend.jax.preprocessing`, 81
- `ceml.backend.tensorflow.costfunctions.costfunctions`,
85
- `ceml.backend.tensorflow.optimizer.optimizer`,
87
- `ceml.backend.torch.costfunctions.costfunctions`,
83
- `ceml.backend.torch.optimizer.optimizer`, 84
- `ceml.costfunctions.costfunctions`, 69
- `ceml.model.counterfactual`, 70
- `ceml.model.model`, 71
- `ceml.optim.cvx`, 77
- `ceml.optim.ga`, 75
- `ceml.optim.input_wrapper`, 71
- `ceml.optim.optimizer`, 72
- `ceml.sklearn.counterfactual`, 18
- `ceml.sklearn.decisiontree`, 21
- `ceml.sklearn.isolationforest`, 52
- `ceml.sklearn.knn`, 24
- `ceml.sklearn.lda`, 38
- `ceml.sklearn.linearregression`, 27
- `ceml.sklearn.lvq`, 30
- `ceml.sklearn.models`, 33
- `ceml.sklearn.naivebayes`, 35
- `ceml.sklearn.pipeline`, 44
- `ceml.sklearn.plausibility`, 20
- `ceml.sklearn.qda`, 41
- `ceml.sklearn.randomforest`, 48
- `ceml.sklearn.softmaxregression`, 56
- `ceml.sklearn.utils`, 59
- `ceml.tfkeras.counterfactual`, 61
- `ceml.torch.counterfactual`, 64
- `ceml.torch.utils`, 68

A

`AffinePreprocessing` (class in `ceml.backend.jax.preprocessing`), 81

B

`b` (`ceml.sklearn.linearregression.LinearRegression` attribute), 27

`b` (`ceml.sklearn.softmaxregression.SoftmaxRegression` attribute), 57

`BFGS` (class in `ceml.optim.optimizer`), 72

`build_loss()` (`ceml.sklearn.pipeline.PipelineCounterfactual` method), 44

`build_loss()` (`ceml.sklearn.randomforest.RandomForestCounterfactual` method), 49

`build_program()` (`ceml.optim.cvx.DCQP` method), 78

`build_regularization_loss()` (in module `ceml.sklearn.utils`), 59

`build_regularization_loss()` (in module `ceml.torch.utils`), 68

`build_solve_opt()` (`ceml.optim.cvx.ConvexQuadraticProgram` method), 77

`build_solve_opt()` (`ceml.optim.cvx.SDP` method), 79

C

`ceml.backend.jax.costfunctions.costfunctions` module, 79

`ceml.backend.jax.preprocessing` module, 81

`ceml.backend.tensorflow.costfunctions.costfunctions` module, 85

`ceml.backend.tensorflow.optimizer.optimizer` module, 87

`ceml.backend.torch.costfunctions.costfunctions` module, 83

`ceml.backend.torch.optimizer.optimizer` module, 84

`ceml.costfunctions.costfunctions` module, 69

`ceml.model.counterfactual` module, 70

`ceml.model.model` module, 71

`ceml.optim.cvx` module, 77

`ceml.optim.ga` module, 75

`ceml.optim.input_wrapper` module, 71

`ceml.optim.optimizer` module, 72

`ceml.sklearn.counterfactual` module, 18

`ceml.sklearn.decisiontree` module, 21

`ceml.sklearn.isolationforest` module, 52

`ceml.sklearn.knn` module, 24

`ceml.sklearn.lda` module, 38

`ceml.sklearn.linearregression` module, 27

`ceml.sklearn.lvq` module, 30

`ceml.sklearn.models` module, 33

`ceml.sklearn.naivebayes` module, 35

`ceml.sklearn.pipeline` module, 44

`ceml.sklearn.plausibility` module, 20

`ceml.sklearn.qda` module, 41

`ceml.sklearn.randomforest` module, 48

`ceml.sklearn.softmaxregression` module, 56

`ceml.sklearn.utils` module, 59

`ceml.tfkeras.counterfactual` module, 61

`ceml.torch.counterfactual` module, 64

- ceml.torch.utils
 module, 68
- class_priors (ceml.sklearn.lda.Lda attribute), 38
- class_priors (ceml.sklearn.naivebayes.GaussianNB attribute), 35
- class_priors (ceml.sklearn.qda.Qda attribute), 41
- complete() (ceml.optim.input_wrapper.InputWrapper method), 72
- compute_all_counterfactuals()
 (ceml.sklearn.decisiontree.DecisionTreeCounterfactual method), 21
- compute_counterfactual()
 (ceml.model.counterfactual.Counterfactual method), 70
- compute_counterfactual()
 (ceml.sklearn.counterfactual.SklearnCounterfactual method), 18
- compute_counterfactual()
 (ceml.sklearn.decisiontree.DecisionTreeCounterfactual method), 21
- compute_counterfactual()
 (ceml.sklearn.isolationforest.IsolationForestCounterfactual method), 53
- compute_counterfactual()
 (ceml.sklearn.pipeline.PipelineCounterfactual method), 44
- compute_counterfactual()
 (ceml.sklearn.randomforest.RandomForestCounterfactual method), 49
- compute_counterfactual()
 (ceml.tfkeras.counterfactual.TfCounterfactual method), 61
- compute_counterfactual()
 (ceml.torch.counterfactual.TorchCounterfactual method), 64
- compute_fitness() (ceml.optim.ga.EvolutionaryOptimizer method), 75
- ConjugateGradients (class in ceml.optim.optimizer), 72
- ConvexQuadraticProgram (class in ceml.optim.cvx), 77
- CostFunction (class in ceml.costfunctions.costfunctions), 69
- CostFunctionDifferentiable (class in ceml.costfunctions.costfunctions), 69
- CostFunctionDifferentiableJax (class in ceml.backend.jax.costfunctions.costfunctions), 79
- CostFunctionDifferentiableTf (class in ceml.backend.tensorflow.costfunctions.costfunctions), 85
- CostFunctionDifferentiableTorch (class in ceml.backend.torch.costfunctions.costfunctions), 83
- Counterfactual (class in ceml.model.counterfactual), 70
- CQPHelper (class in ceml.sklearn.lvq), 30
- crossover() (ceml.optim.ga.EvolutionaryOptimizer method), 76
- ## D
- DCQP (class in ceml.optim.cvx), 77
- decisiontree_generate_counterfactual() (in module ceml.sklearn.decisiontree), 23
- DecisionTreeCounterfactual (class in ceml.sklearn.decisiontree), 21
- desc_to_dist() (in module ceml.sklearn.utils), 60
- desc_to_dist() (in module ceml.torch.utils), 68
- desc_to_regcost() (in module ceml.sklearn.utils), 60
- desc_to_regcost() (in module ceml.torch.utils), 68
- dim (ceml.sklearn.lda.Lda attribute), 38
- dim (ceml.sklearn.linearregression.LinearRegression attribute), 27
- dim (ceml.sklearn.lvq.LVQ attribute), 30
- dim (ceml.sklearn.naivebayes.GaussianNB attribute), 35
- dim (ceml.sklearn.qda.Qda attribute), 41
- dim (ceml.sklearn.softmaxregression.SoftmaxRegression attribute), 57
- dist (ceml.sklearn.knn.KNN attribute), 24
- dist (ceml.sklearn.lvq.LVQ attribute), 30
- DummyCost (class in ceml.backend.jax.costfunctions.costfunctions), 80
- DummyCost (class in ceml.backend.tensorflow.costfunctions.costfunctions), 85
- DummyCost (class in ceml.backend.torch.costfunctions.costfunctions), 83
- ## E
- EnsembleVotingCost (class in ceml.sklearn.randomforest), 48
- epsilon (ceml.optim.cvx.ConvexQuadraticProgram attribute), 77
- epsilon (ceml.optim.cvx.DCQP attribute), 78
- epsilon (ceml.optim.cvx.SDP attribute), 79
- EvolutionaryOptimizer (class in ceml.optim.ga), 75
- extract_from() (ceml.optim.input_wrapper.InputWrapper method), 72
- ## G
- GaussianNB (class in ceml.sklearn.naivebayes), 35
- gausiannb_generate_counterfactual() (in module ceml.sklearn.naivebayes), 36
- GaussianNbCounterfactual (class in ceml.sklearn.naivebayes), 36
- generate_counterfactual() (in module ceml.sklearn.models), 33
- generate_counterfactual() (in module ceml.tfkeras.counterfactual), 62

- generate_counterfactual() (in module *ceml.torch.counterfactual*), 66
- get_loss() (*ceml.model.model.ModelWithLoss* method), 71
- get_loss() (*ceml.sklearn.isolationforest.IsolationForest* method), 52
- get_loss() (*ceml.sklearn.knn.KNN* method), 24
- get_loss() (*ceml.sklearn.lda.Lda* method), 38
- get_loss() (*ceml.sklearn.linearregression.LinearRegression* method), 27
- get_loss() (*ceml.sklearn.lvq.LVQ* method), 31
- get_loss() (*ceml.sklearn.naivebayes.GaussianNB* method), 35
- get_loss() (*ceml.sklearn.pipeline.PipelineModel* method), 46
- get_loss() (*ceml.sklearn.qda.Qda* method), 41
- get_loss() (*ceml.sklearn.randomforest.RandomForest* method), 49
- get_loss() (*ceml.sklearn.softmaxregression.SoftmaxRegression* method), 57
- grad() (*ceml.backend.jax.costfunctions.costfunctions.CostFunctionDifferentiableJax* method), 79
- grad() (*ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf* method), 85
- grad() (*ceml.backend.torch.costfunctions.costfunctions.CostFunctionDifferentiableTorch* method), 83
- grad() (*ceml.costfunctions.costfunctions.CostFunctionDifferentiable* method), 69
- I**
- init() (*ceml.backend.tensorflow.optimizer.optimizer.TfOptimizer* method), 87
- init() (*ceml.backend.torch.optimizer.optimizer.TorchOptimizer* method), 84
- init() (*ceml.optim.ga.EvolutionaryOptimizer* method), 76
- init() (*ceml.optim.optimizer.BFGS* method), 72
- init() (*ceml.optim.optimizer.ConjugateGradients* method), 72
- init() (*ceml.optim.optimizer.NelderMead* method), 73
- init() (*ceml.optim.optimizer.Powell* method), 73
- InputWrapper (class in *ceml.optim.input_wrapper*), 71
- is_binary (*ceml.sklearn.naivebayes.GaussianNB* attribute), 35
- is_binary (*ceml.sklearn.qda.Qda* attribute), 41
- is_multiclass (*ceml.sklearn.softmaxregression.SoftmaxRegression* attribute), 57
- is_optimizer_grad_based() (in module *ceml.optim.optimizer*), 74
- IsolationForest (class in *ceml.sklearn.isolationforest*), 52
- isolationforest_generate_counterfactual() (in module *ceml.sklearn.isolationforest*), 55
- IsolationForestCost (class in *ceml.sklearn.isolationforest*), 53
- IsolationForestCounterfactual (class in *ceml.sklearn.isolationforest*), 53
- K**
- KNN (class in *ceml.sklearn.knn*), 24
- knn_generate_counterfactual() (in module *ceml.sklearn.knn*), 25
- KnnCounterfactual (class in *ceml.sklearn.knn*), 25
- L**
- L1Cost (class in *ceml.backend.jax.costfunctions.costfunctions*), 80
- L1Cost (class in *ceml.backend.tensorflow.costfunctions.costfunctions*), 85
- L1Cost (class in *ceml.backend.torch.costfunctions.costfunctions*), 83
- L2Cost (class in *ceml.backend.jax.costfunctions.costfunctions*), 80
- L2Cost (class in *ceml.backend.tensorflow.costfunctions.costfunctions*), 86
- L2Cost (class in *ceml.backend.torch.costfunctions.costfunctions*), 83
- Labels (*ceml.sklearn.kv.LVQ* attribute), 30
- Lda (class in *ceml.sklearn.lda*), 38
- lda_generate_counterfactual() (in module *ceml.sklearn.lda*), 39
- LdaCounterfactual (class in *ceml.sklearn.lda*), 39
- LinearRegression (class in *ceml.sklearn.linearregression*), 27
- linearregression_generate_counterfactual() (in module *ceml.sklearn.linearregression*), 28
- LinearRegressionCounterfactual (class in *ceml.sklearn.linearregression*), 28
- LMadCost (class in *ceml.backend.jax.costfunctions.costfunctions*), 80
- LMadCost (class in *ceml.backend.tensorflow.costfunctions.costfunctions*), 86
- LMadCost (class in *ceml.backend.torch.costfunctions.costfunctions*), 83
- LVQ (class in *ceml.sklearn.lvq*), 30
- lvq_generate_counterfactual() (in module *ceml.sklearn.lvq*), 32
- LvqCounterfactual (class in *ceml.sklearn.lvq*), 31
- M**
- MathematicalProgram (class in *ceml.optim.cvx*), 78
- means (*ceml.sklearn.lda.Lda* attribute), 38
- means (*ceml.sklearn.naivebayes.GaussianNB* attribute), 35
- means (*ceml.sklearn.qda.Qda* attribute), 41
- MinMaxScaler (class in *ceml.backend.jax.preprocessing*), 81

MinOfListCost (class in `ceml.torch.utils`, 68
 `ceml.backend.torch.costfunctions.costfunctions`), 83
 MinOfListDistCost (class in `ceml.backend.jax.costfunctions.costfunctions`), 80
 MinOfListDistExCost (class in `ceml.backend.jax.costfunctions.costfunctions`), 80
 model (`ceml.sklearn.counterfactual.SklearnCounterfactual` attribute), 18
 model (`ceml.sklearn.lvq.LVQ` attribute), 30
 Model (class in `ceml.backend.jax.preprocessing`), 81
 Model (class in `ceml.model.model`), 71
 model_class (`ceml.sklearn.lvq.LVQ` attribute), 30
 models (`ceml.sklearn.pipeline.PipelineModel` attribute), 46
 ModelWithLoss (class in `ceml.model.model`), 71
 module
 `ceml.backend.jax.costfunctions.costfunctions`, 79
 `ceml.backend.jax.preprocessing`, 81
 `ceml.backend.tensorflow.costfunctions.costfunctions`, 85
 `ceml.backend.tensorflow.optimizer.optimizer`, 87
 `ceml.backend.torch.costfunctions.costfunctions`, 83
 `ceml.backend.torch.optimizer.optimizer`, 84
 `ceml.costfunctions.costfunctions`, 69
 `ceml.model.counterfactual`, 70
 `ceml.model.model`, 71
 `ceml.optim.cvx`, 77
 `ceml.optim.ga`, 75
 `ceml.optim.input_wrapper`, 71
 `ceml.optim.optimizer`, 72
 `ceml.sklearn.counterfactual`, 18
 `ceml.sklearn.decisiontree`, 21
 `ceml.sklearn.isolationforest`, 52
 `ceml.sklearn.knn`, 24
 `ceml.sklearn.lda`, 38
 `ceml.sklearn.linearregression`, 27
 `ceml.sklearn.lvq`, 30
 `ceml.sklearn.models`, 33
 `ceml.sklearn.naivebayes`, 35
 `ceml.sklearn.pipeline`, 44
 `ceml.sklearn.plausibility`, 20
 `ceml.sklearn.qda`, 41
 `ceml.sklearn.randomforest`, 48
 `ceml.sklearn.softmaxregression`, 56
 `ceml.sklearn.utils`, 59
 `ceml.tfkeras.counterfactual`, 61
 `ceml.torch.counterfactual`, 64
 mutate() (`ceml.optim.ga.EvolutionaryOptimizer` method), 76
 mymodel (`ceml.sklearn.counterfactual.SklearnCounterfactual` attribute), 18
N
 NegLogLikelihoodCost (class in `ceml.backend.jax.costfunctions.costfunctions`), 81
 NegLogLikelihoodCost (class in `ceml.backend.tensorflow.costfunctions.costfunctions`), 86
 NegLogLikelihoodCost (class in `ceml.backend.torch.costfunctions.costfunctions`), 83
 NelderMead (class in `ceml.optim.optimizer`), 73
 Normalizer (class in `ceml.backend.jax.preprocessing`), 82
O
 Optimizer (class in `ceml.optim.optimizer`), 73
P
 PCA (class in `ceml.backend.jax.preprocessing`), 82
 pccp (`ceml.optim.cvx.DCQP` attribute), 78
 PenaltyConvexConcaveProcedure (class in `ceml.optim.cvx`), 79
 pipeline_generate_counterfactual() (in module `ceml.sklearn.pipeline`), 47
 PipelineCounterfactual (class in `ceml.sklearn.pipeline`), 44
 PipelineModel (class in `ceml.sklearn.pipeline`), 46
 PolynomialFeatures (class in `ceml.backend.jax.preprocessing`), 82
 Powell (class in `ceml.optim.optimizer`), 73
 predict() (`ceml.backend.jax.preprocessing.MinMaxScaler` method), 81
 predict() (`ceml.backend.jax.preprocessing.Model` method), 82
 predict() (`ceml.backend.jax.preprocessing.Normalizer` method), 82
 predict() (`ceml.backend.jax.preprocessing.PCA` method), 82
 predict() (`ceml.backend.jax.preprocessing.PolynomialFeatures` method), 82
 predict() (`ceml.backend.jax.preprocessing.StandardScaler` method), 82
 predict() (`ceml.model.model.Model` method), 71
 predict() (`ceml.sklearn.isolationforest.IsolationForest` method), 53
 predict() (`ceml.sklearn.knn.KNN` method), 25
 predict() (`ceml.sklearn.lda.Lda` method), 39

- `predict()` (*ceml.sklearn.linearregression.LinearRegression* method), 27
- `predict()` (*ceml.sklearn.lvq.LVQ* method), 31
- `predict()` (*ceml.sklearn.naivebayes.GaussianNB* method), 36
- `predict()` (*ceml.sklearn.pipeline.PipelineModel* method), 46
- `predict()` (*ceml.sklearn.qda.Qda* method), 42
- `predict()` (*ceml.sklearn.randomforest.RandomForest* method), 49
- `predict()` (*ceml.sklearn.softmaxregression.SoftmaxRegression* method), 57
- `prepare_computation_of_plausible_counterfactuals()` (in module *ceml.sklearn.plausibility*), 20
- `prepare_optim()` (in module *ceml.optim.optimizer*), 74
- prototypes (*ceml.sklearn.lvq.LVQ* attribute), 30
- ## Q
- Qda* (class in *ceml.sklearn.qda*), 41
- `qda_generate_counterfactual()` (in module *ceml.sklearn.qda*), 42
- QdaCounterfactual* (class in *ceml.sklearn.qda*), 42
- ## R
- RandomForest* (class in *ceml.sklearn.randomforest*), 48
- `randomforest_generate_counterfactual()` (in module *ceml.sklearn.randomforest*), 51
- RandomForestCounterfactual* (class in *ceml.sklearn.randomforest*), 49
- `rebuild_model()` (*ceml.sklearn.counterfactual.SklearnCounterfactual* method), 20
- `rebuild_model()` (*ceml.sklearn.decisiontree.DecisionTreeCounterfactual* method), 22
- `rebuild_model()` (*ceml.sklearn.isolationforest.IsolationForestCounterfactual* method), 55
- `rebuild_model()` (*ceml.sklearn.knn.KnnCounterfactual* method), 25
- `rebuild_model()` (*ceml.sklearn.lda.LdaCounterfactual* method), 39
- `rebuild_model()` (*ceml.sklearn.linearregression.LinearRegressionCounterfactual* method), 28
- `rebuild_model()` (*ceml.sklearn.lvq.LvqCounterfactual* method), 31
- `rebuild_model()` (*ceml.sklearn.naivebayes.GaussianNbCounterfactual* method), 36
- `rebuild_model()` (*ceml.sklearn.pipeline.PipelineCounterfactual* method), 46
- `rebuild_model()` (*ceml.sklearn.qda.QdaCounterfactual* method), 42
- `rebuild_model()` (*ceml.sklearn.randomforest.RandomForestCounterfactual* method), 51
- `rebuild_model()` (*ceml.sklearn.softmaxregression.SoftmaxCounterfactual* method), 56
- `reduce()` (in module *ceml.backend.jax.preprocessing*), 82
- RegularizedCost* (class in *ceml.backend.jax.costfunctions.costfunctions*), 81
- RegularizedCost* (class in *ceml.backend.tensorflow.costfunctions.costfunctions*), 86
- RegularizedCost* (class in *ceml.backend.torch.costfunctions.costfunctions*), 84
- RegularizedCost* (class in *ceml.costfunctions.costfunctions*), 69
- ## S
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.DummyCost* method), 80
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.L1Cost* method), 80
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.L2Cost* method), 80
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.LMadCost* method), 80
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.MinOfListDi* method), 80
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.MinOfListDi* method), 81
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.NegLogLikel* method), 81
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.RegularizedC* method), 81
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.SquaredError* method), 81
- `score_impl()` (*ceml.backend.jax.costfunctions.costfunctions.TopKMinOfL* method), 81
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.Dumm* method), 85
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.L1Co* method), 85
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.L2Co* method), 86
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.LMad* method), 86
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.NegL* method), 86
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.Regul* method), 86
- `score_impl()` (*ceml.backend.tensorflow.costfunctions.costfunctions.Squar* method), 87
- `score_impl()` (*ceml.backend.torch.costfunctions.costfunctions.DummyCo* method), 83
- `score_impl()` (*ceml.backend.torch.costfunctions.costfunctions.L1Co* method), 83

T
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.L2Cost method), 83
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.LMadCost method), 83
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.MinOfListCost method), 83
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.NegLogLikelihoodCost method), 84
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.RegularizedCost method), 84
 score_impl() (ceml.backend.torch.costfunctions.costfunctions.SquaredError method), 84
 score_impl() (ceml.costfunctions.costfunctions.CostFunction method), 69
 score_impl() (ceml.costfunctions.costfunctions.RegularizedCost method), 70
V
 score_impl() (ceml.sklearn.isolationforest.IsolationForestCost method), 53
 score_impl() (ceml.sklearn.randomforest.EnsembleVotingCost method), 48
 SDP (class in ceml.optim.cvx), 79
 select_candidates() (ceml.optim.ga.EvolutionaryOptimizer method), 76
 sigma_inv (ceml.sklearn.lda.Lda attribute), 38
 sigma_inv (ceml.sklearn.qda.Qda attribute), 41
 SklearnCounterfactual (class in ceml.sklearn.counterfactual), 18
 SoftmaxCounterfactual (class in ceml.sklearn.softmaxregression), 56
 SoftmaxRegression (class in ceml.sklearn.softmaxregression), 57
 softmaxregression_generate_counterfactual() (in module ceml.sklearn.softmaxregression), 57
 solve() (ceml.optim.cvx.DCQP method), 78
 solve() (ceml.sklearn.lvq.LvqCounterfactual method), 31
 solve() (ceml.sklearn.naivebayes.GaussianNbCounterfactual method), 36
 solve() (ceml.sklearn.qda.QdaCounterfactual method), 42
 SquaredError (class in ceml.backend.jax.costfunctions.costfunctions), 81
 SquaredError (class in ceml.backend.tensorflow.costfunctions.costfunctions), 87
 SquaredError (class in ceml.backend.torch.costfunctions.costfunctions), 84
 StandardScaler (class in ceml.backend.jax.preprocessing), 82
 SupportAffinePreprocessing (class in ceml.optim.cvx), 79
TfCounterfactual (class in ceml.tfkeras.counterfactual), 61
TfOptimizer (class in ceml.backend.tensorflow.optimizer.optimizer), 87
TopKMinOfListDistCost (class in ceml.backend.jax.costfunctions.costfunctions), 81
TorchCounterfactual (class in ceml.backend.torch.counterfactual), 64
TorchOptimizer (class in ceml.backend.torch.optimizer.optimizer), 84
U
 update() (ceml.optim.ga.EvolutionaryOptimizer method), 76
V
 variances (ceml.sklearn.naivebayes.GaussianNB attribute), 35
W
 w (ceml.sklearn.linearregression.LinearRegression attribute), 27
 w (ceml.sklearn.softmaxregression.SoftmaxRegression attribute), 57
X
 X (ceml.sklearn.knn.KNN attribute), 24
Y
 y (ceml.sklearn.knn.KNN attribute), 24