ceml

André Artelt

Oct 13, 2020

USER GUIDE

CHAPTER

ONE

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 learing models.

It supports many common machine learning frameworks:

- scikit-learn (0.23.2)
- PyTorch (1.6.0)
- Keras & Tensorflow (2.2.1)

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 3.6 or higher 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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

from ceml.sklearn import generate_counterfactual


if __name__ == "__main__":
    # Load data
    X, y = load_iris(True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=4242)

    # Whitelist of features - list of features we can change/use when computing a␣
↪counterfactual
    features_whitelist = None   # We can use all features

    # Create and fit model
```

(continues on next page)

(continued from previous page)

```
    model = DecisionTreeClassifier(max_depth=3)
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    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 so by defining a function that takes a prediction as an input and returns *True* if the predictions 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:

```
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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import Ridge

from ceml.sklearn import generate_counterfactual


if __name__ == "__main__":
    # Load data
    X, y = load_boston(True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=4242)

    # Whitelist of features - list of features we can change/use when computing a␣
↪counterfactual
    features_whitelist = [0, 1, 2, 3, 4]    # Use the first five features only

    # Create and fit model
    model = Ridge()
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
```

(continues on next page)

(continued from previous page)

```
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    y_target = 25.0
    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.
    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 `skelarn.preprocessing.StandardScaler` and logistic regression `sklearn.linear_model.LogisticRegression` is given below:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

from ceml.sklearn import generate_counterfactual


if __name__ == "__main__":
    # Load data
    X, y = load_iris(True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4242)

    # Whitelist of features - list of features we can change/use when computing a counterfactual
    features_whitelist = [1, 3]   # Use the second and fourth feature only

    # Create and fit the pipeline
    scaler = StandardScaler()
    model = LogisticRegression(solver='lbfgs', multi_class='multinomial')    # Note that ceml requires: multi_class='multinomial'

```

(continues on next page)

(continued from previous page)

```
    model = make_pipeline(scaler, model)
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    print(generate_counterfactual(model, x, y_target=0, features_whitelist=features_
↪whitelist))
```

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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
import random
random.seed(424242)
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.mixture import GaussianMixture
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn.utils import shuffle

from ceml.sklearn.softmaxregression import softmaxregression_generate_counterfactual
from ceml.sklearn.plausibility import prepare_computation_of_plausible_counterfactuals


if __name__ == "__main__":
    # Load data set
    X, y = load_digits(return_X_y=True);pca_dim=40

    X, y = shuffle(X, y, random_state=42)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=4242)

    # Choose target labels
```

(continues on next page)

(continued from previous page)

```
    y_test_target = []
    labels = np.unique(y)
    for i in range(X_test.shape[0]):
        y_test_target.append(random.choice(list(filter(lambda l: l != y_test[i], labels))))
    y_test_target = np.array(y_test_target)

    # Reduce dimensionality
    X_train_orig = np.copy(X_train)
    X_test_orig = np.copy(X_test)
    projection_matrix = None
    projection_mean_sub = None

    pca = PCA(n_components=pca_dim)
    pca.fit(X_train)

    projection_matrix = pca.components_ # Projection matrix
    projection_mean_sub = pca.mean_

    X_train = np.dot(X_train - projection_mean_sub, projection_matrix.T)
    X_test = np.dot(X_test - projection_mean_sub, projection_matrix.T)

    # Fit classifier
    model = LogisticRegression(multi_class="multinomial", solver="lbfgs", random_state=42)
    model.fit(X_train, y_train)

    # Compute accuracy on test set
    print("Accuracy: {0}".format(accuracy_score(y_test, model.predict(X_test))))

    # For each class, fit density estimators
    density_estimators = {}
    kernel_density_estimators = {}
    labels = np.unique(y)
    for label in labels:
        # Get all samples with the 'correct' label
        idx = y_train == label
        X_ = X_train[idx, :]

        # Optimize hyperparameters
        cv = GridSearchCV(estimator=GaussianMixture(covariance_type='full'), param_grid={'n_components': range(2, 10)}, n_jobs=-1, cv=5)
        cv.fit(X_)
        n_components = cv.best_params_["n_components"]

        # Build density estimators
        de = GaussianMixture(n_components=n_components, covariance_type='full', random_state=42)
        de.fit(X_)

        density_estimators[label] = de

    # Build dictionary for ceml
    plausibility_stuff = prepare_computation_of_plausible_counterfactuals(X_train_orig, y_train, density_estimators, projection_mean_sub, projection_matrix)

    # Compute and plot counterfactual with vs. without density constraints
```

(continues on next page)

(continued from previous page)

```
    i = 0

    x_orig = X_test[i,:]
    x_orig_orig = X_test_orig[i,:]
    y_orig = y_test[i]
    y_target = y_test_target[i]
    print("Original label: {0}".format(y_orig))
    print("Target label: {0}".format(y_target))

    if(model.predict([x_orig]) == y_target):  # Model already predicts target label!
        raise ValueError("Requested prediction already satisfied")

    # Compute plausible counterfactual
    x_cf_plausible = softmaxregression_generate_counterfactual(model, x_orig_orig, y_target, plausibility=plausibility_stuff)
    x_cf_plausible_projected = np.dot(x_cf_plausible - projection_mean_sub, projection_matrix.T)
    print("Predictec label of plausible countrefactual: {0}".format(model.predict([x_cf_plausible_projected])))

    # Compute closest counterfactual
    plausibility_stuff["use_density_constraints"] = False
    x_cf = softmaxregression_generate_counterfactual(model, x_orig_orig, y_target, plausibility=plausibility_stuff)
    x_cf_projected = np.dot(x_cf - projection_mean_sub, projection_matrix.T)
    print("Predicted label of closest counterfactual: {0}".format(model.predict([x_cf_projected])))

    # Plot results
    fig, axes = plt.subplots(3, 1)
    axes[0].imshow(x_orig_orig.reshape(8, 8))     # Original sample
    axes[1].imshow(x_cf.reshape(8, 8))            # Closest counterfactual
    axes[2].imshow(x_cf_plausible.reshape(8, 8)) # Plausible counterfactual
    plt.show()
```

1.2.3 PyTorch

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

We must provide the PyTorch model within a class that is derived from `torch.nn.Module` and *ceml.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 `ceml.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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import torch
```

(continues on next page)

(continued from previous page)

```
torch.manual_seed(424242)
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

from ceml.torch import generate_counterfactual
from ceml.backend.torch.costfunctions import NegLogLikelihoodCost
from ceml.model import ModelWithLoss


# Neural network - Softmax regression
class Model(torch.nn.Module, ModelWithLoss):
    def __init__(self, input_size, num_classes):
        super(Model, self).__init__()

        self.linear = torch.nn.Linear(input_size, num_classes)
        self.softmax = torch.nn.Softmax(dim=0)

    def forward(self, x):
        return self.linear(x)    # NOTE: Softmax is build into CrossEntropyLoss

    def predict_proba(self, x):
        return self.softmax(self.forward(x))

    def predict(self, x, dim=1):
        return torch.argmax(self.forward(x), dim=dim)

    def get_loss(self, y_target, pred=None):
        return NegLogLikelihoodCost(input_to_output=self.predict_proba, y_target=y_
↪target)


if __name__ == "__main__":
    # Load data
    X, y = load_iris(True)
    X = X.astype(np.dtype(np.float32))

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=1)

    # numpy -> torch tensor
    x = torch.from_numpy(X_train)
    labels = torch.from_numpy(y_train)

    x_test = torch.from_numpy(X_test)
    y_test = torch.from_numpy(y_test)

    # Create and fit model
    model = Model(4, 3)

    learning_rate = 0.001
    momentum = 0.9
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,␣
↪momentum=momentum)

```

(continues on next page)

(continued from previous page)

```
    num_epochs = 800
    for epoch in range(num_epochs):
        optimizer.zero_grad()
        outputs = model(x)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # Evaluation
    y_pred = model.predict(x_test).detach().numpy()
    print("Accuracy: {0}".format(accuracy_score(y_test, y_pred)))

    # Select a data point whose prediction has to be explained
    x_orig = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict(torch.from_numpy(np.array([x_
↪orig])))))

    # Whitelist of features we can use/change when computing the counterfactual
    features_whitelist = [0, 2] # Use the first and third feature only

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    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-lever interface for tensorflow and nowadays part of tensorflow , we do not need to distinguish between keras and tensorflow models when using ceml.

Computing a counterfactual of a tensorflow/keras model is done by using the *ceml.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 *ceml.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 `ceml.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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

(continues on next page)

(continued from previous page)

```
from sklearn.metrics import accuracy_score

from ceml.tfkeras import generate_counterfactual
from ceml.backend.tensorflow.costfunctions import NegLogLikelihoodCost
from ceml.model import ModelWithLoss


# Neural network - Softmax regression
class Model(ModelWithLoss):
    def __init__(self, input_size, num_classes):
        super(Model, self).__init__()

        self.model = tf.keras.models.Sequential([
            tf.keras.layers.Dense(num_classes, activation='softmax', input_shape=(input_size,))
        ])

    def fit(self, x_train, y_train, num_epochs=800):
        self.model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

        self.model.fit(x_train, y_train, epochs=num_epochs, verbose=False)

    def predict(self, x):
        return np.argmax(self.model(x), axis=1)

    def predict_proba(self, x):
        return self.model(x)

    def __call__(self, x):
        return self.predict(x)

    def get_loss(self, y_target, pred=None):
        return NegLogLikelihoodCost(input_to_output=self.model.predict_proba, y_target=y_target)


if __name__ == "__main__":
    tf.random.set_seed(42)    # Fix random seed

    # Load data
    X, y = load_iris(True)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=1)

    # Create and fit model
    model = Model(4, 3)
    model.fit(X_train, y_train)

    # Evaluation
    y_pred = model.predict(X_test)
    print("Accuracy: {0}".format(accuracy_score(y_test, y_pred)))

    # Select a data point whose prediction has to be explained
    x_orig = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict(np.array([x_orig]))))
```

(continues on next page)

(continued from previous page)

```
    # Whitelist of features we can use/change when computing the counterfactual
    features_whitelist = None

    # Compute counterfactual
    optimizer = tf.compat.v1.train.GradientDescentOptimizer(learning_rate=1.0)     # Init optimization algorithm
    optimizer_args = {"max_iter": 1000}

    print("\nCompute counterfactual ....")
    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 dervied 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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import jax.numpy as npx
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB

from ceml.sklearn import generate_counterfactual
from ceml.backend.jax.costfunctions import CostFunctionDifferentiableJax


```

(continues on next page)

(continued from previous page)

```
# Custom implementation of the l2-regularization. Note that this regularization is differentiable
class MyRegularization(CostFunctionDifferentiableJax):
    def __init__(self, x_orig):
        self.x_orig = x_orig

        super(MyRegularization, self).__init__()

    def score_impl(self, x):
        return npx.sum(npx.square(x - self.x_orig)) # Note: This expression must be written in jax and it must be differentiable!


if __name__ == "__main__":
    # Load data
    X, y = load_iris(True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4242)

    # Whitelist of features - list of features we can change/use when computing a counterfactual
    features_whitelist = None   # All features can be used.

    # Create and fit the model
    model = GaussianNB()    # Note that ceml requires: multi_class='multinomial'
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Create custom regularization function
    regularization = MyRegularization(x)

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    print(generate_counterfactual(model, x, y_target=0, features_whitelist=features_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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import jax.numpy as npx
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import Ridge


from ceml.sklearn import generate_counterfactual
from ceml.sklearn import LinearRegression, LinearRegressionCounterfactual
from ceml.backend.jax.costfunctions import CostFunctionDifferentiableJax


# Custom implementation of the l2-regularization. Note that this regularization is differentiable.
class MyLoss(CostFunctionDifferentiableJax):
    def __init__(self, input_to_output, y_target):
        self.y_target = y_target

        super(MyLoss, self).__init__(input_to_output)

    def score_impl(self, y):
        return npx.abs(y - y_target)**4

# Derive a new class from ceml.sklearn.linearregression.LinearRegression and overwrite the get_loss method to use our custom loss MyLoss
class LinearRegressionWithMyLoss(LinearRegression):
    def __init__(self, model):
        super(LinearRegressionWithMyLoss, self).__init__(model)

    def get_loss(self, y_target, pred=None):
        if pred is None:
            return MyLoss(self.predict, y_target)
        else:
            return MyLoss(pred, y_target)

# Derive a new class from ceml.sklearn.linearregression.LinearRegressionCounterfactual that uses our new linear regression wrapper LinearRegressionWithMyLoss for computing counterfactuals
class MyLinearRegressionCounterfactual(LinearRegressionCounterfactual):
    def __init__(self, model):
        super(MyLinearRegressionCounterfactual, self).__init__(model)

    def rebuild_model(self, model):
        return LinearRegressionWithMyLoss(model)


if __name__ == "__main__":
    # Load data
    X, y = load_boston(True)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=4242)

    # Whitelist of features - list of features we can change/use when computing a counterfactual
    features_whitelist = None   # All features can be used.

    # Create and fit model
```

(continues on next page)

(continued from previous page)

```
    model = Ridge()
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Compute counterfactual
    print("\nCompute counterfactual ....")
    y_target = 25.0
    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.

    cf = MyLinearRegressionCounterfactual(model)    # Since we are using our own loss function, we can no longer use standard method generate_counterfactual
    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. Dervice 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:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from scipy.optimize import minimize

from ceml.sklearn import generate_counterfactual
from ceml.optim import Optimizer


# Custom optimization method that simply calls the BFGS optimizer from scipy
class MyOptimizer(Optimizer):
    def __init__(self):
        self.f = None
        self.f_grad = None
        self.x0 = None
        self.tol = None
        self.max_iter = None

        super(MyOptimizer, self).__init__()

```

(continues on next page)

(continued from previous page)

```
    def init(self, f, f_grad, x0, tol=None, max_iter=None):
        self.f = f
        self.f_grad = f_grad
        self.x0 = x0
        self.tol = tol
        self.max_iter = max_iter

    def is_grad_based(self):
        return True

    def __call__(self):
        optimum = minimize(fun=self.f, x0=self.x0, jac=self.f_grad, tol=self.tol,␣
↪options={'maxiter': self.max_iter}, method="BFGS")
        return np.array(optimum["x"])


if __name__ == "__main__":
    # Load data
    X, y = load_iris(True)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=4242)

    # Create and fit model
    model = LogisticRegression(solver='lbfgs', multi_class='multinomial')
    model.fit(X_train, y_train)

    # Select data point for explaining its prediction
    x = X_test[1,:]
    print("Prediction on x: {0}".format(model.predict([x])))

    # Compute counterfactual by using our custom optimizer 'MyOptimizer'
    print("\nCompute counterfactual ....")
    print(generate_counterfactual(model, x, y_target=0, optimizer=MyOptimizer(),␣
↪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 - 2020},
        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*, ***kwds*)

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'*, *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".

- **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*, ***kwds*)

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:

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

  The only reason for including this parameter is to match the signature of other *ceml.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 *ceml.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 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.

ceml.sklearn.knn

class ceml.sklearn.knn.**KNN**(*model*, *dist='l2'*, ***kwds*)

Bases: *ceml.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'*, ***kwds*)
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** (instace 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'*, *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:

  – 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 *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.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".

- **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*, ***kwds*)

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*, ***kwds*)

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'*, *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:

  – 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.

  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".

- **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.

ceml.sklearn.lvq

class ceml.sklearn.lvq.**CQPHelper**(*mymodel*, *x_orig*, *y_target*, *indices_other_prototypes*, *features_whitelist=None*, *regularization='l1'*, ***kwds*)

Bases: *ceml.optim.cvx.ConvexQuadraticProgram*

class ceml.sklearn.lvq.**LVQ**(*model*, *dist='l2'*, ***kwds*)

Bases: *ceml.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'>*, ***kwds*)

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** (instace 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*)

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.
- **tao** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.2
- **tao_max** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 100
- **mu** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.5

ceml.sklearn.lvq.**lvq_generate_counterfactual**(*model*, *x*, *y_target*, *features_whitelist=None*, *dist='l2'*, *regularization='l1'*, *C=1.0*, *optimizer='auto'*, *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.

- **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'*, *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:

  – 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.

  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:

  – 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.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".

- **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*, ***kwds*)

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 dependend priors.

Type *numpy.ndarray*

means
Class and feature dependend means.

Type *numpy.array*

variances
Class and feature dependend 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-likehood 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*, ***kwds*)

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*)

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.
- **tao** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.2
- **tao_max** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 100
- **mu** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.5

ceml.sklearn.naivebayes.**gaussiannb_generate_counterfactual**(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='auto'*, *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:

  – 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".

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.

- **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.**Lda**(*model*, ***kwds*)

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-likehood 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*, ***kwds*)
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'*, *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:

  – 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.

  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".
- **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.qda

class ceml.sklearn.qda.**Qda**(*model*, ***kwds*)

Bases: *ceml.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-likehood 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*, ***kwds*)

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*)

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.
- **tao** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.2

- **tao_max** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 100

- **mu** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.5

ceml.sklearn.qda.**qda_generate_counterfactual**(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='auto'*, *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:

  – 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.

  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.

- **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*, ***kwds*)

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:

  – 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 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'*, *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".

- **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*, ***kwds*)

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'*, *return_as_dict=True*, *done=None*)

Computes a counterfactual of a given input *x*.

Parameters

- **model** (a `sklearn.pipeline.Pipeline` instance.) – The modelpipeline 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.

- **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.sklearn.randomforest

class ceml.sklearn.randomforest.**EnsembleVotingCost**(*models*, *y_target*, *input_wrapper=None*, *epsilon=0*, ***kwds*)

Bases: *ceml.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*, ***kwds*)

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*, ***kwds*)

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'*, *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:

  – 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.

  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.

- **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'*, *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 "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.

  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.

- **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*, ***kwds*)

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'*, *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:

  – 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.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.

- **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'*, *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 "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.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.

- **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*, ***kwds*)
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 *ceml.sklearn.softmaxregression.SoftmaxRegression*

class ceml.sklearn.softmaxregression.**SoftmaxRegression**(*model*, ***kwds*)
Bases: *ceml.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-likehood 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*

ceml.sklearn.softmaxregression.**softmaxregression_generate_counterfactual**(*model*, *x*, *y_target*, *features_whitelist=None*, *regularization='l1'*, *C=1.0*, *optimizer='mp'*, *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:

  – 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.

  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".

- **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*, ***kwds*)
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:

  – 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 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

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:

  – 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.

  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*, ***kwds*)

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:

  – 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.

  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:

- l1: l1-norm

- l2: l2-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:

- 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).

  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 ceml.costfunctions

ceml.costfunctions.costfunctions

class ceml.costfunctions.costfunctions.**CostFunction**(*input_to_output=None*, ***kwds*)

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 ceml.costfunctions.costfunctions.**CostFunctionDifferentiable**(*input_to_output=None*, ***kwds*)

Bases: *ceml.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*, ***kwds*)

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**(***kwds*)

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**(***kwds*)
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**(***kwds*)
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*, ***kwds*)
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**(***kwds*)
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**(***kwds*)
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**(***kwds*)
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**(***kwds*)
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**(***kwds*)
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_optim**(*optim*, *f*, *x0*, *f_grad=None*, *tol=None*, *max_iter=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.
- **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.

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 proability 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 **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**(***kwds*)

Bases: abc.ABC

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*)

Builds and solves the convex quadratic optimization problem.

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.

- **mad** (*numpy.ndarray*, optional) – Weights for the weighted Manhattan distance.

  If *mad* is None, the Euclidean distance is used.

  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**(***kwds*)

Bases: object

Class for a difference-of-convex-quadratic program (DCQP) - for computing counterfactuals.

$$\min_{\vec{x} \in \mathbb{R}^d} \vec{x}^\top Q_0 \vec{x} + \vec{q}^\top \vec{x} + c - \vec{x}^\top Q_1 \vec{x} \quad \text{s.t. } \vec{x}^\top A0_i \vec{x} + \vec{x}^\top \vec{b_i} + r_i - \vec{x}^\top A1_i \vec{x} \leq 0 \; \forall \, i$$

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*)

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.

solve(*x0*, *tao=1.2*, *tao_max=100*, *mu=1.5*)
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.
- **tao** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.2
- **tao_max** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 100
- **mu** (*float*, optional) – Hyperparameter - see paper for details.

  The default is 1.5

class ceml.optim.cvx.**MathematicalProgram**(***kwds*)
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*, ***kwds*)
Bases: object

Implementation of the penalty convex-concave procedure for approximately solving a DCQP.

class ceml.optim.cvx.**SDP**(***kwds*)
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*)
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.

Returns

The solution of the optimization problem.

If no solution exists, *None* is returned.

Return type *numpy.ndarray*

1.7 ceml.backend.jax

1.7.1 ceml.backend.jax.costfunctions

class ceml.backend.jax.costfunctions.costfunctions.**CostFunctionDifferentiableJax**(*input_to_outp*
***kwds*)

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**(***kwds*)

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*, ***kwds*)

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*, ***kwds*)

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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)

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*, ***kwds*)

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*, ***kwds*)

Bases: object

Wrapper for an affine mapping (preprocessing)

class ceml.backend.jax.preprocessing.**MinMaxScaler**(*min_*, *scale*, ***kwds*)

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**(***kwds*)

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**(***kwds*)

Bases: *ceml.model.model.Model*

Wrapper for the normalizer.

predict(*x*)
Computes the forward pass.

class ceml.backend.jax.preprocessing.**PCA**(*w*, ***kwds*)
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*, ***kwds*)
Bases: *ceml.model.model.Model*

Wrapper for polynomial feature transformation.

predict(*x*)
Computes the forward pass.

class ceml.backend.jax.preprocessing.**StandardScaler**(*mu*, *sigma*, ***kwds*)
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**(***kwds*)
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**(***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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*, ***kwds*)
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**(***kwds*)
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**(**kw

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**(**kwds*)

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*, **kwds*)

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*, ***kwds*)

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*, ***kwds*)

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*, ***kwds*)

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*, ***kwds*)

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 ceml.backend.tensorflow.costfunctions.costfunctions.**SquaredError**(*y_target*, ***kwds*)

Bases: *ceml.backend.tensorflow.costfunctions.costfunctions.CostFunctionDifferentiableTf*

Squared error cost function.

score_impl(*y*)

Computes the loss - squared error.

1.9.2 ceml.backend.tensorflow.optimizer

class ceml.backend.tensorflow.optimizer.optimizer.**TfOptimizer**(***kwds*)

Bases: *ceml.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 `ceml.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.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

E

G

I

K

L

M

N

O

P

Q

R

S

T

V

W

X

Y