Solutions to Assignment 2 of CS-E4830 Kernel Methods in Machine Learning 2021

Kernel centering

Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel function and $\phi : \mathcal{X} \to F$ a feature map associated with this kernel. Let $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ be the set of training inputs.

Centering the data in the feature space moves the origin of the feature space to the center of mass of the training features $\frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}_i)$ and generally helps to improve the performance. After centering, the feature map is given by: $\phi_c(\mathbf{x}) = \phi(\mathbf{x}) - \frac{1}{N}\sum_{i=1}^{N}\phi(\mathbf{x}_i)$. We will see in this question that centering can be performed implicitly by transforming the kernel values.

Question 1: (2 points)

Show that

$$k_c(\mathbf{x}_i, \mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j) - \frac{1}{N}\sum_{p=1}^{N} k(\mathbf{x}_p, \mathbf{x}_j) - \frac{1}{N}\sum_{q=1}^{N} k(\mathbf{x}_i, \mathbf{x}_q) + \frac{1}{N^2}\sum_{p=1}^{N}\sum_{q=1}^{N} k(\mathbf{x}_p, \mathbf{x}_q),$$

where $k_c(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi_c(\mathbf{x}_i), \phi_c(\mathbf{x}_j) \rangle$ is the kernel value after centering.

Solution

$$\begin{aligned}
\kappa_c(\mathbf{x}_i, \mathbf{x}_j) &= \langle \phi_c(\mathbf{x}_i), \phi_c(\mathbf{x}_j) \rangle \\
&= \langle \phi(\mathbf{x}_i) - \frac{1}{N}\sum_{p=1}^{N}\phi(\mathbf{x}_p),\ \ \phi(\mathbf{x}_j) - \frac{1}{N}\sum_{q=1}^{N}\phi(\boldsymbol{x}_q) \rangle \\
&= \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle - \frac{1}{N}\langle \phi(\mathbf{x}_i), \sum_{q=1}^{N}\phi(\mathbf{x}_q) \rangle - \frac{1}{N}\langle \sum_{p=1}^{N}\phi(\mathbf{x}_p), \phi(\mathbf{x}_j) \rangle \\
&+ \frac{1}{N^2}\langle \sum_{p=1}^{N}\phi(\mathbf{x}_p), \sum_{q=1}^{N}\phi(\mathbf{x}_q) \rangle \\
\kappa_c(\mathbf{x}_i, \mathbf{x}_j) &= \kappa(\mathbf{x}_i, \mathbf{x}_j) - \frac{1}{N}\sum_{q=1}^{N}\kappa(\mathbf{x}_i, \mathbf{x}_q) - \frac{1}{N}\sum_{p=1}^{N}\kappa(\mathbf{x}_p, \mathbf{x}_j) + \frac{1}{N^2}\sum_{p,q=1}^{N}\kappa(\mathbf{x}_p, \mathbf{x}_q).
\end{aligned}$$

Question 2 (3 points):

Consider the binary classification as discussed in Lecture 4 and shown in Figure 1, where the probability densities, $p(x, C_1)$ and $p(x, C_2)$ for the two classes are known.

1. (1 point) For the point $\hat{x}$, compute the probability that it belongs to C_1, i.e., $P(y = C_1 | X = \hat{x})$.

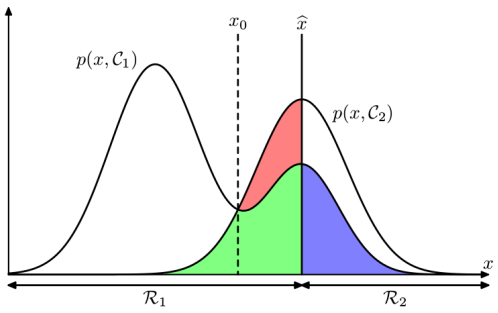


Figure 1: Data distribution for a binary classification problem

Solution

$$\begin{aligned} P(y = C_1 | X = \hat{x}) &= \frac{P(y = C_1, X = \hat{x})}{P(X = \hat{x})} \\ &= \frac{P(y = C_1, X = \hat{x})}{P(y = C_1, X = \hat{x}) + P(y = C_2, X = \hat{x})} \\ &= \frac{p(y = C_1, X = \hat{x})dx}{p(y = C_1, X = \hat{x})dx + p(y = C_2, X = \hat{x})dx} \\ &= \frac{p(y = C_1, X = \hat{x})}{p(y = C_1, X = \hat{x}) + p(y = C_2, X = \hat{x})} \end{aligned}$$

2. (2 points) Prove that the probability of the minimum mis-classification error satisfies this inequality:

$$P(\text{Minimum mis-classification error}) \leq \int_{x \in \mathcal{X}} (p(x, C_1)p(x, C_2))^{1/2} \, dx$$

Hint : In the proof you can apply the following inequality, for any $a \geq 0$ and $b \geq 0$ we have

$$\min(a, b) \leq (ab)^{1/2}.$$

Solution

The minimum mis-classification error corresponds to the union of the green and the blue area. That area covers the overlap between the density functions of the classes. It is the consequence of the fact that in the classification that class is chosen at a given x whose probability is the largest, thus the minimum error corresponds to the intersection of the areas under the curves of density functions, where both classes have the probability to occur. The area of the intersection is given by the integral

$$P(\text{Minimum mis-classification error}) = \int_{x \in \mathcal{X}} \min(p(x, C_1)p(x, C_2))dx$$

Now by exploiting the inequality $\min(a, b) \leq (ab)^{1/2}$, where $a = p(x, C_1)$ and $b = p(x, C_2)$ for any x, we can write

$$P(\text{Minimum misclassification error}) = \int_{x \in \mathcal{X}} \min(p(x, C_1)p(x, C_2))dx \leq \int_{x \in \mathcal{X}} (p(x, C_1)p(x, C_2))^{1/2} \, dx.$$

Question 3: (1 point)

Let $\mathrm{x}_i \in \mathcal{R}^d$ be an input example, and $\mathrm{w}_k \in \mathcal{R}^d$, $k = 1, ..., K$ a set of parameter vectors assigned to each class in multi-class classification. Let the probability $P(Y_i = k|X = x_i)$ of a class with respect to x_i be given by $\frac{1}{Z}\exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle)$, where Z is a normalization factor to guarantee that $\frac{1}{Z}$ is a probability.

The task is to suggest a multi-class desicion function for this concrete probability model, and derive the value of Z for a fixed number of classes.

Solution

Recall that given x class C_j is selected in the binary classification if $P(y = j|X = x) \geq 0.5$. It means $j = \mathrm{argmax}_k[P(y = k|X = x), k = 1, 2]$, thus class with the largest probability is chosen. We can follow the same approach in multi-class classification

$$j = \arg\max_k P(y = k|X = x), k = 1, ..., K$$

Since $P(y = k|X = x) = \frac{1}{Z}\exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle)$, therefore we can write

$$j = \arg\max_k \frac{1}{Z}\exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle), k = 1, ..., K$$

To compute Z we can expolit that the sum of the conditional probabilities of all classes has to be 1, thus we have

$$Z = \sum_{k=1}^{K} P(y = k|X = x) = \sum_{k=1}^{K} \exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle).$$

Finally the multi-class classification has this form

$$j = \arg\max_k \frac{\exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle)}{\sum_{k=1}^{K} \exp(\langle \mathrm{w_k}, \mathrm{x}_i \rangle)}, k = 1, ..., K$$

Question 4: (2 points)

Consider a random variable ϵ that takes the values $\{-1, +1\}$ with equal probability. Show that

$$\mathbb{E}[e^{\lambda\epsilon}] \leq e^{\frac{\lambda^2}{2}} \text{ for all } \lambda \in \mathbb{R}$$

where $\mathbb{E}[.]$ denotes the expectation w.r.t the random variable ϵ.

Hint : Use power series expansion of the exponential function.

Solution

$$\begin{aligned}
\mathbb{E}[e^{\lambda\epsilon}] &= \frac{1}{2}\{e^{\lambda} + e^{-\lambda}\} \\
&= \frac{1}{2}\sum_{k=0}^{\infty}\frac{(-\lambda)^k}{k!} + \frac{1}{2}\sum_{k=0}^{\infty}\frac{(\lambda)^k}{k!} \\
&= \sum_{k=0}^{\infty}\frac{\lambda^{2k}}{(2k)!} \\
&\leq 1 + \sum_{k=1}^{\infty}\frac{\lambda^{2k}}{2^k k!} \\
&= e^{\frac{\lambda^2}{2}}
\end{aligned}$$

Side note : The random variable ϵ as defined above is also known as Rademacher random variable, and is related to the Rademacher complexity of a function class. Let the training set be $\{(x_i, y_i)\}_{i=1}^n$. let $\epsilon_1, \ldots, \epsilon_n$ be independent random variables, which take values +1 or -1 with probability 0.5 each. For a function class $\mathcal{F}$, its Rademacher complexity is defined as

$$\mathcal{R}(\mathcal{F}) := E_\epsilon \sup_{f\in\mathcal{F}} \epsilon_i f(x_i)$$

Intuitively, it denotes the capacity of function class $\mathcal{F}$ to fit random noise.

KernelCourse2021_Exercise02

May 5, 2021

1 Exercise 02

Kernel Methods in Machine Learning (CS-E4830)

Tutorial session: 22nd April at 16:15-18:00

Submission deadline : 28th April at 4pm

Tasks:

1. Section **??** (**2 Point**)
2. Section **??**
3. Section **??** (**2 Points**)
4. Section **??** (**1 Point**)
5. Section **??**
6. Section **??** (**2 Point**)
7. Section **??**

Bonus Task:

4. Section **??**

Implement Task 3 using sklearn pipelines. You only need to consider the kernel centering as possible pre-processing step.

Version: 1.0

Version history:

- 1.0: Initial version

Please add you student number and email address to the notebook into the corresponding cell.

EMAIL: firstname.lastname@aalto.fi

STUDENT_NUMBER: 000000

1.1 Import required python packages

All tasks in this exercise can be solved by using only function and packages imported below. Please **do not** use any other imports. And place your code only in places where there read "YOUR CODE HERE".

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
import itertools as it

from sklearn.base import BaseEstimator, RegressorMixin, TransformerMixin, clone
from sklearn.metrics.pairwise import rbf_kernel as rbf_kernel_sk
from sklearn.metrics.pairwise import linear_kernel as linear_kernel_sk
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, KFold, GridSearchCV,␣
↪ParameterGrid
from sklearn.compose import TransformedTargetRegressor
from sklearn.preprocessing import StandardScaler, Normalizer
from sklearn.pipeline import Pipeline
```

Wrapper around the sklearn `rbf_kernel` function. In the lecture, we use the formulation of the Gaussian kernel which includes σ as parameter. However, sklearn uses a slighly different implementation, which is essentially just a re-parametrisation. We wrap the sklearn function, to be aligned with the lecture formulation.

[2]:
```
def gaussian_kernel_wrapper(X, Y=None, sigma=None):
    """
    Wrapper around the sklearn rbf-kernel function. It converts between the
    gamma parametrization (sklearn) and the sigma parametrization (lecture).
    """
    if sigma is None:
        sigma = np.sqrt(X.shape[1] / 2.)

    return rbf_kernel_sk(X, Y, gamma=(1. / (2. * (sigma**2))))
```

1.2 1. Kernel Ridge Regression (2 Point)

The Kernel Ridge Regression prediction model can be written as:

$$g(\mathbf{x}) = \sum_{i=1}^{n_{train}} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i) = \mathbf{k}(\mathbf{x})\boldsymbol{\alpha}$$

with: - $g : \mathbb{R}^d \to \mathbb{R}$ being the **prediction function** - $\mathbf{k}(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_{n_{train}})] \in \mathbb{R}^{1 \times n_{train}}$ being a row-vector containing all **similarities** between the new example $\mathbf{x}$ and the training examples $\mathbf{x}_i$ - $\boldsymbol{\alpha} \in \mathbb{R}^{n_{train}}$ column-vector containing the **dual variables**

Below you find the class-template for the Kernel Ridge Regression. It's functionality is split into three parts:

1.2.1 1. Intialization of Regressor Object using init()

A Kernel Ridge Regression instance can be created using its constructor, e.g.:

```
# using a kernel function
est = KernelRidgeRegression(kernel="gaussian")
```

```
# or using precomputed kernel matrices
est = KernelRidgeRegression(kernel="precomputed")
```

1.2.2 2. Model Training using fit()

This function takes as input:

- The features of the training examples $\mathbf{X}_{train}$ or a precomputed training kernel matrix $\mathbf{K}_{train} \in \mathbb{R}^{n_{train} \times n_{train}}$
- Their corresponding regression labels $\mathbf{y}_{train} \in \mathbb{R}^{n_{train}}$.

Using that, it estimates the dual variables α_i's. If needed, the kernel values between the training examples, i.e. $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ are calculated during the fitting process.

```
est.fit(X_train, y_train)

# or using precomputed kernel matrices
KX_train = my_kernel_function(X_train)
est.fit(KX_train, y_train)
```

1.2.3 3. Prediction for new Examples using predict()

When the model parameters are fitted, than we can make predictions for a new example $\mathbf{x}$ using the function $g(\mathbf{x})$.

```
y_test_pred = est.predict(X_test)

# or using precomputed kernel matrices
KX_test_train = my_kernel_function(X_test, X_train)
y_test_pred = est.predict(KX_test_train, y_train)

<b>Task:</b> Implement the missing code parts of <code>fit()</code> and <code>predict()</code>

<b>Hint:</b> An example how to circumvent the direct calculation of the matrix-inverse, can be
```

[3]:
```
class KernelRidgeRegression(BaseEstimator, RegressorMixin):
    def __init__(self, kernel="gaussian", beta=1.0, sigma=None):
        """
        Kernel Ridge Regression (KRR)

        :param kernel: string, specifying which kernel to use. Can be␣
↪'gaussian', 'linear' or 'precomputed'.
        :param beta: scalar, regularization parameter
        :param gamma: scalar, gaussian-kernel parameter
        """
        self.beta = beta
        self.X_train = None
        self.alphas = None

        # Set up kernel function
        self.kernel = kernel
```

```
        self.sigma = sigma

    def fit(self, X_train, y_train):
        """
        Fit a KRR model using training data.

        :param X_train: array-like, shape=(n_samples, n_features),␣
↪feature-matrix
            OR X_train: array-like, shape=(n_samples, n_samples), precomputed␣
↪kernel matrix
        :param y_train: array-like, shape=(n_samples,) or (n_samples, 1), label␣
↪vector
        """
        # Make label vector to column-vector
        if len(y_train.shape) == 1:
            y_train = y_train[:, np.newaxis]

        # Calculate training kernel
        if self.kernel != "precomputed":
            self.X_train = X_train
            KX_train = self._get_kernel(self.X_train)
        else:
            KX_train = X_train

        # Calculate the identity matrix scaled by the regularization parameter:␣
↪(beta * n_samples) * I
        # YOUR CODE HERE
        n_samples = KX_train.shape[0]
        reg = (self.beta * n_samples) * np.eye(n_samples)

        # Solve: alphas' = y' * inv(K + beta * n_samples * I), shape =␣
↪(n_samples, 1)
        # YOUR CODE HERE
        self.alphas = np.linalg.solve((KX_train + reg), y_train)

        return self

    def predict(self, X):
        """
        Predict using fitted KRR model for new data.

        :param X: array-like, shape=(n_samples_test, n_features),␣
↪feature-matrix of new data.
            OR X: array-like, shape=(n_samples_test, n_samples_train),␣
↪test-training kernel matrix.
```

```
        :return: array-like, shape=(n_samples_test,), predictions for all data␣
↪points
        """
        if self.alphas is None:
            raise RuntimeError("Call fit-function first.")

        if self.kernel != "precomputed":
            K_test_train = self._get_kernel(X, self.X_train)
        else:
            K_test_train = X
            if K_test_train.shape[1] != self.alphas.shape[0]:
                raise RuntimeError("Number of training examples does not match␣
↪the shape of the "
                                   "provided Test-Train-Kernel matrix.")

        # Calculate the value of the prediction function for each test example
        # YOUR CODE HERE
        g_X = K_test_train @ self.alphas

        return g_X.flatten()

    def _get_kernel(self, X, Y=None):
        """
        Calcualte kernel matrix using specified kernel-function and parameters.

        :param X: array-like, shape=(n_samples_A, n_features), feature-matrix␣
↪of set A
        :param Y: array-like, shape=(n_samples_B, n_features), feature-matrix␣
↪of set B or None, than Y = X
        :return: array-like, shape=(n_samples_A, n_samples_B), kernel matrix
        """
        if self.kernel == "gaussian":
            return gaussian_kernel_wrapper(X, Y, self.sigma)
        elif self.kernel == "linear":
            return linear_kernel_sk(X, Y)
        elif self.kernel == "precomputed":
            raise RuntimeError("Provide precomputed kernel matrix.")
        else:
            raise ValueError("Invalid kernel chosen.")
```

[4]:
```
from sklearn.kernel_ridge import KernelRidge as sk_KernelRidgeRegression

__X, __y = make_regression(n_samples=500, random_state=319)

# Split to train and test
__X_train, __X_test, __y_train, __y_test = train_test_split(__X, __y,␣
↪random_state=731)
```

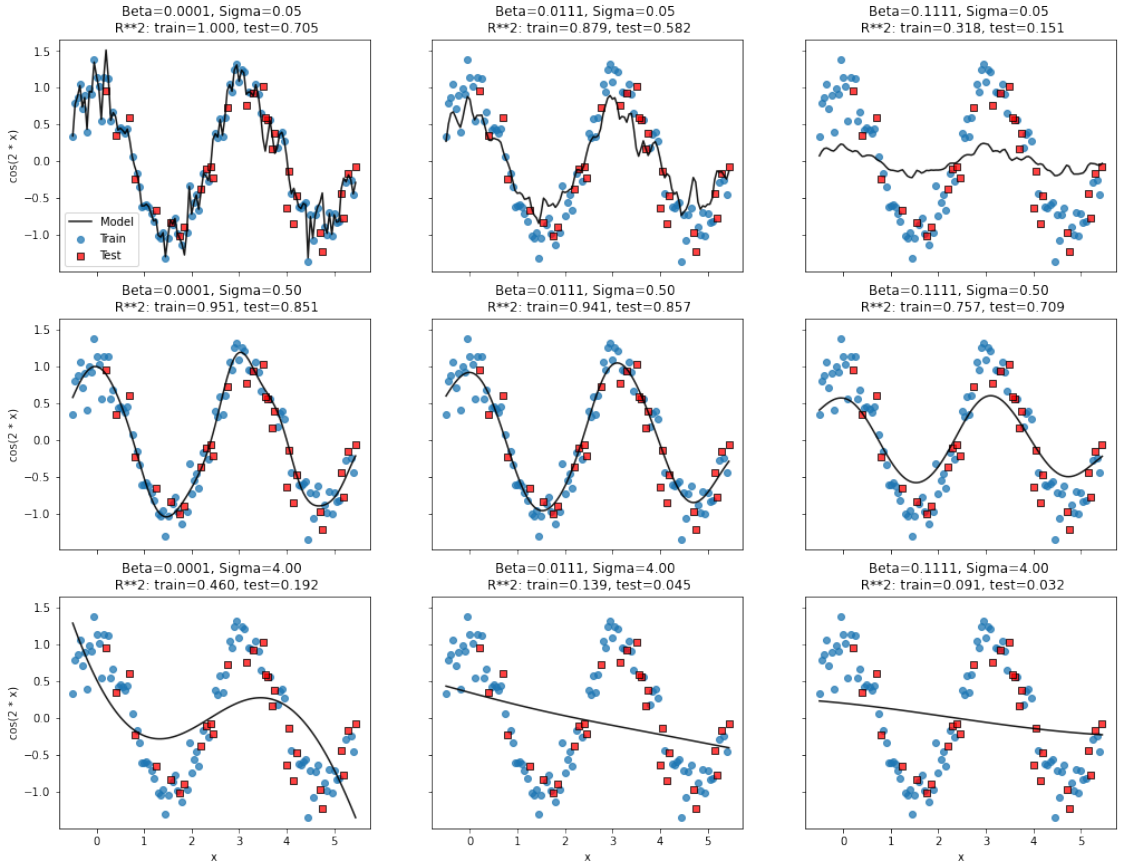
```
# Set up estimators
__gamma = 0.15
__KRR = KernelRidgeRegression(beta=0.5 / __X_train.shape[0], kernel="gaussian",␣
↪sigma=np.sqrt(1. / (2 * __gamma)))
__KRR_ref = sk_KernelRidgeRegression(alpha=0.5, kernel="rbf", gamma=__gamma)

# Fit models
__KRR.fit(__X_train, __y_train)
__KRR_ref.fit(__X_train, __y_train)
np.testing.assert_equal(__KRR.alphas.shape, (__X_train.shape[0], 1))

# Predict
__y_pred = __KRR.predict(__X_test)
__y_pred_ref = __KRR_ref.predict(__X_test)
np.testing.assert_allclose(__y_pred, __y_pred_ref)

# MSE
np.testing.assert_allclose(mean_squared_error(__y_test, __y_pred),
                           mean_squared_error(__y_test, __y_pred_ref))
```

Visually inspect your KRR implementation:

[5]:
```
rng = np.random.RandomState(320)

# Generate noise cosine curve
X = np.arange(-0.5, 5.5, 0.05)[:, np.newaxis]
y = np.cos(2*X) + rng.randn(X.shape[0], 1) * 0.2

# Split to train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=767)

sigmas = [0.05, 0.5, 4.]
betas = np.array([0.01, 1, 10]) / len(X_train)

fig, axr = plt.subplots(len(sigmas), len(betas),
                        figsize=(17, 13), sharex="all", sharey="all")

for idx, (s, b) in enumerate(it.product(sigmas, betas)):
    krr = KernelRidgeRegression(beta=b, sigma=s, kernel="gaussian")
    krr.fit(X_train, y_train)
    y_pred = krr.predict(X)

    # Calculate prediction score: sklear default for regression is R^2
    score_train = krr.score(X_train, y_train)
    score_test = krr.score(X_test, y_test)
```

```
    i, j = np.unravel_index(idx, dims=axr.shape)
    axr[i, j].plot(X, y_pred, label="Model", c="black", alpha=0.9)
    axr[i, j].scatter(X_train, y_train, label="Train", alpha=0.75)
    axr[i, j].scatter(X_test, y_test, label="Test", alpha=0.75, c="red",
                      marker="s", edgecolors="black")
    axr[i, j].set_title("Beta=%.4f, Sigma=%.2f \n R**2: train=%.3f, test=%.3f" %
                        (b, s, score_train, score_test))

    if i == (len(sigmas) - 1):
        axr[i, j].set_xlabel("x")
    if j == 0:
        axr[i, j].set_ylabel("cos(2 * x)")

_ = axr[0, 0].legend(loc="lower left")
```

```
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
<ipython-input-5-91656f350709>:25: DeprecationWarning: 'shape' argument should
be used instead of 'dims'
  i, j = np.unravel_index(idx, dims=axr.shape)
```



1.3 2. Kernel pre-processing

In this task you will implement two kernel (matrix) pre-processing steps. The first one is *kernel centering* and the second one *kernel normalization*. In practice this steps might lead to better regression models and you usually should try at least to use centered kernels.

1.3.1 A. Kernel centering (2 Points)

In the Pen & Paper exercise you showed, that the kernel centering (centering of the underlying feature vectors) can be done implicitly by transforming the kernel values. That means, you can calculate the simimlarity of the centered kernel between two samples $\kappa_c(\mathbf{x}_i, \mathbf{x}_j)$ using:

$$\kappa_c(\mathbf{x}_i, \mathbf{x}_j) = \kappa(\mathbf{x}_i, \mathbf{x}_j) - \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} \kappa(\mathbf{x}_i, \mathbf{x}_p) - \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} \kappa(\mathbf{x}_p, \mathbf{x}_j) + \frac{1}{n_{train}^2} \sum_{p \in \mathcal{I}_{train}} \sum_{q \in \mathcal{I}_{train}} \kappa(\mathbf{x}_p, \mathbf{x}_q)$$

Here we assume that $i \in \mathcal{I}_A$ and $j \in \mathcal{I}_B$ are samples indices from two samples sets A and B. The indices of the training examples are denoted with $\mathcal{I}_{train}$.

In practice we apply the centering to three matrices:

Kernel Matrix

Notation

$\mathcal{I}_A$

$\mathcal{I}_B$

Note

Training kernel matrix

$\mathbf{K}_{train} \in \mathbb{R}^{n_{train} \times n_{train}}$

$\mathcal{I}_{train}$

$\mathcal{I}_{train}$

Used for fitting.

Test-training kernel matrix

$\mathbf{K}_{test,train} \in \mathbb{R}^{n_{test} \times n_{train}}$

$\mathcal{I}_{test}$

$\mathcal{I}_{train}$

Used during prediction.

Test-test kernel matrix

$\mathbf{K}_{test,test} \in \mathbb{R}^{n_{test} \times n_{test}}$

$\mathcal{I}_{test}$

$\mathcal{I}_{test}$

Needed for kernel normalizaton only.

Implementing a kernel centering class We can implement the kernel centering as sklearn transformer class. It requires us to implement a `__init__()`, `fit()` and `transform()` function.

1. Intialization of transformer object using 'init()' We can get a KernelCentering instance using:

```
centerer = KernelCentering()
```

2. Fitting the centering statistics based on the training set using 'fit()' We fit the necessary centering statistics using a training kernel matrix:

```
centerer.fit(KX_train)
```

For that we calculate a the column-wise kernel value averages of the training kernel matrix $\mathbf{K}_{train}$ and store them in a vector $\bar{\mathbf{k}}_{train} \in \mathbb{R}^{1 \times n_{train}}$:

$$[\bar{\mathbf{k}}_{train}]_i = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} \kappa(\mathbf{x}_i, \mathbf{x}_p) = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} [\mathbf{K}_{train}]_{ip} \quad \forall_{i \in \mathcal{I}_{train}}.$$

We furthermore need to calculate the mean value of *all* kernel values $\mu_{train} \in \mathbb{R}$:

$$\mu_{train} = \frac{1}{n_{train}^2} \sum_{p \in \mathcal{I}_{train}} \sum_{q \in \mathcal{I}_{train}} \kappa(\mathbf{x}_p, \mathbf{x}_q) = \sum_{p \in \mathcal{I}_{train}} \sum_{q \in \mathcal{I}_{train}} [\mathbf{K}_{train}]_{pq}$$

3. Center train and test-train kernel matrix using 'transform()' We can transform center a given kernel matrix:

```
# Training kernel matrix
KX_train_c = centerer.transform(KX_train)  # I_A = I_train

# Test-Training kernel matrix
KX_test_train_c = centerer.transform(KX_test_train)  # I_A = I_test
```

For that we need to calculate the average kernel value between each example of $\mathcal{I}_A$ and the training set $\mathcal{I}_{train}$. We store this values in a column-vector $\bar{\mathbf{k}}_{A,train} \in \mathbb{R}^{n_A}$:

$$[\bar{\mathbf{k}}_{A,train}]_i = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} \kappa(\mathbf{x}_i, \mathbf{x}_p) = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} [\mathbf{K}_{A,train}]_{ip} \quad \forall_{i \in \mathcal{I}_A}.$$

4. Center test-test kernel matrix using 'transform_k_test()' To center the test-test kernel matirx, we $\mathbf{K}_{A,A}$ and $\mathbf{K}_{A,train}$:

```
centerer.transform_k_test(KX_test, KX_test_train)
```

For that we (again) need to calculate the average kernel value between each example of $\mathcal{I}_A$ and the training set $\mathcal{I}_{train}$. We store this values in a column-vector $\bar{\mathbf{k}}_{A,train} \in \mathbb{R}^{n_A}$:

$$[\bar{\mathbf{k}}_{A,train}]_i = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} \kappa(\mathbf{x}_i, \mathbf{x}_p) = \frac{1}{n_{train}} \sum_{p \in \mathcal{I}_{train}} [\mathbf{K}_{A,train}]_{ip} \quad \forall_{i \in \mathcal{I}_A}.$$

```
<b>Task:</b> Implement the missing code parts of <code>fit()</code>, <code>transform()</code> a

<b>Hints:</b>
```

- You can calculate the mean (average) value along different axes of a matrix using np.mean.
- Take a look on the NumPy broadcasting rules. What is the result of the following instructions:

```
M = np.arange(12).reshape((4, 3))
print(M)

a = np.array([[1, 2, 3]])  # shape = (1, 3)
print(a)

print(M - a)  # ???
```

[6]:
```
class KernelCenterer(BaseEstimator, TransformerMixin):
    def __init__(self):
        # Centering statistics
        self.k_bar_train = None
```

```
        self.mu_train = None

    def fit(self, K, y=None, **fit_params):
        """
        Extract the data statistics needed for the kernel centering.

        :param K: array-like, shape=(n_samples_train, n_samples_train), kernel␣
↪matrix
        :param y: array-like, shape=(n_samples,), target values. Will be␣
↪ignored.
        :return: self, returns an instance of it self.
        """
        # Calculate the statistics from the training set:
        # - k_bar_train, shape = (1, n_train)
        # - mu_train, scalar
        # YOUR CODE HERE
        self.k_bar_train = np.mean(K, axis=0)[np.newaxis, :]
        self.mu_train = np.mean(self.k_bar_train)

        return self  # allows to use the fit_transform method inhered from␣
↪TransformerMixin

    def transform(self, K, y=None, copy=True, K_test=False):
        """
        Apply centering to a kernel matrix.

        :param K: array-like, shape=(n_samples_A, n_samples_train)
        :param y: array-like, shape=(n_samples_A,), target values. Will be␣
↪ignored!
        :param copy: boolean, indicating whether the input kernel matrix should␣
↪be copied before so that the
            centering does not effect the original matrix.
        :return: array-like, shape=(n_samples_A, n_samples_train) centered␣
↪kernel matrix.
        """
        # Copy input array, if copy=True, i.e. original data is not modified.
        if copy:
            K = np.array(K)

        # Calculate: k_bar_A_train, shape = (n_test, 1) by averaging along the␣
↪rows
        # YOUR CODE HERE
        k_bar_A_train = np.mean(K, axis=1)[:, np.newaxis]
        assert(k_bar_A_train.shape == (K.shape[0], 1))
```

```
        # Center kernel matrix
        # YOUR CODE HERE
        K = K - (self.k_bar_train + k_bar_A_train) + self.mu_train

        return K

    def transform_K_test(self, K, K_test_train, copy=True):
        """
        Apply centering the test-test kernel matrix. This case needs to be␣
↪handled separetly.

        :param K: array-like, shape=(n_samples_A, n_samples_A), test-test␣
↪kernel matrix
        :param K_test_train: array-like, shape=(n_samples_A, n_samples_train)
        :param copy: boolean, indicating whether the input kernel matrix should␣
↪be copied before so that the
            centering does not effect the original matrix.
        :return: array-like, shape=(n_samples_A, n_samples_A) centered kernel␣
↪matrix
        """
        # Copy input array, if copy=True, i.e. original data is not modified.
        if copy:
            K = np.array(K)

        # Calculate: k_bar_A_train, shape = (n_test, 1) by averaging along the␣
↪rows of K_test_train
        # YOUR CODE HERE
        k_bar_A_train = np.mean(K_test_train, axis=1)[:, np.newaxis]
        assert(k_bar_A_train.shape == (K.shape[0], 1))

        # Center test-test kernel matrix K
        # YOUR CODE HERE
        K -= k_bar_A_train.T
        K -= k_bar_A_train
        K += self.mu_train

        return K

    @property
    def _pairwise(self):
        return True
```

[7]:
```
__rng = np.random.RandomState(890)
__X = __rng.rand(101, 31)
__KX = rbf_kernel_sk(__X, gamma=1.2)

# Fit your centerer using some training data
```

```
__centerer = KernelCenterer().fit(__KX[:75, :75])
np.testing.assert_equal(__centerer.k_bar_train.shape, (1, 75))
assert(np.isscalar(__centerer.mu_train))

# Transform your training data
__KX_train_c = __centerer.transform(__KX[:75, :75])
np.testing.assert_equal(__KX_train_c.shape, (75, 75))
np.testing.assert_allclose(np.sum(__KX_train_c, axis=0),
                           np.zeros((__KX_train_c.shape[0],)), atol=1e-12)
np.testing.assert_allclose(np.sum(__KX_train_c, axis=1),
                           np.zeros((__KX_train_c.shape[0],)), atol=1e-12)

# Transform your test-training data
__KX_test_train_c = __centerer.transform(__KX[75:, :75])
np.testing.assert_equal(__KX_test_train_c.shape, (26, 75))

# We test the test-test kernel centering by centering the features in the␣
 ↪original space and construct a
# linear kernel on top. In this case, centering the kernel implicitly is␣
 ↪equivalent to centering it in the
# feature space.
__X_train, __X_test = train_test_split(__X, random_state=380)

__feat_centerer = StandardScaler(with_std=False).fit(__X_train)
__X_train_c = __feat_centerer.transform(__X_train)
__X_test_c = __feat_centerer.transform(__X_test)

__KX_lin_train = linear_kernel_sk(__X_train, __X_train)
__KX_lin_test_train = linear_kernel_sk(__X_test, __X_train)
__KX_lin_test = linear_kernel_sk(__X_test, __X_test)

# build kernels based on centered feature vectors
__KX_lin_train_c = linear_kernel_sk(__X_train_c, __X_train_c)
np.testing.assert_allclose(np.sum(__KX_lin_train_c, axis=0),
                           np.zeros((__KX_lin_train_c.shape[0],)), atol=1e-12)

__centerer = KernelCenterer().fit(__KX_lin_train)
__KX_lin_test_c = __centerer.transform_K_test(__KX_lin_test,␣
 ↪__KX_lin_test_train)
__KX_lin_test_c_ref = linear_kernel_sk(__X_test_c, __X_test_c)

np.testing.assert_allclose(__KX_lin_test_c, __KX_lin_test_c_ref)
```

1.3.2 B. Kernel normalization (1 Point)

Using a normalized kernel can be in practice lead to better models, e.g. less senstive to outliers. As for the kernel centering, if we choose the type of normalization properly, we can calculate the

normalized kernel without explicit knowledge of the underlying featurespace.

Given a kernel matrix $\mathbf{K}_{A,B}$ between to sets of samples A and B, we can normalize its entries $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, with $i \in \mathcal{I}_A$ and $j \in \mathcal{I}_B$, as follows:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{\kappa(\mathbf{x}_i, \mathbf{x}_j)}{\sqrt{\kappa(\mathbf{x}_i, \mathbf{x}_i)\kappa(\mathbf{x}_j, \mathbf{x}_j)}}.$$

The feature-vectors $\phi(\mathbf{x})$ are implicitly normalized, such that:

$$\phi(\mathbf{x})_n = \frac{\phi(\mathbf{x})}{\|\phi(\mathbf{x})\|} = \frac{\phi(\mathbf{x})}{\sqrt{\phi(\mathbf{x})^T \phi(\mathbf{x})}}$$

Task: Implement the missing code parts of <code>normalize_kernel()</code>. Do not

Hint:

- Read the documentation of the NumPy function np.outer. How can it help you to construct a matrix $D \in \mathbb{R}^{n_A \times n_B}$ such that:

$$[D]_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_i)\kappa(\mathbf{x}_j, \mathbf{x}_j) \quad ?$$

- Make use of element-wise division.

[8]:
```
def normalize_kernel(K, d_A, d_B=None, copy=True):
    """
    Function implementing the kernel normalization.

    :param K: array-like, shape=(n_samples_A, n_samples_B), kernel matrix to␣
↪normalize
    :param d_A: array-like, shape=(n_samples_A,) diagonal of the kernel matrix␣
↪K_AA
    :param d_A: array-like, shape=(n_samples_B,) diagonal of the kernel matrix␣
↪K_BB
    :param copy: boolean, indicating whether the input kernel matrix should be␣
↪copied before
        normalization.
    :return: array-like, shape=(n_samples_A, n_samples_B) normalized kernel␣
↪matrix.
    """
    # Copy input array, if copy=True, i.e. original data is not modified.
    if copy:
        K = np.array(K)

    if K.shape[0] != len(d_A):
        raise ValueError("Number of elements of d_A must match the number of␣
↪rows if K.")

    if d_B is None:
        d_B = d_A
```

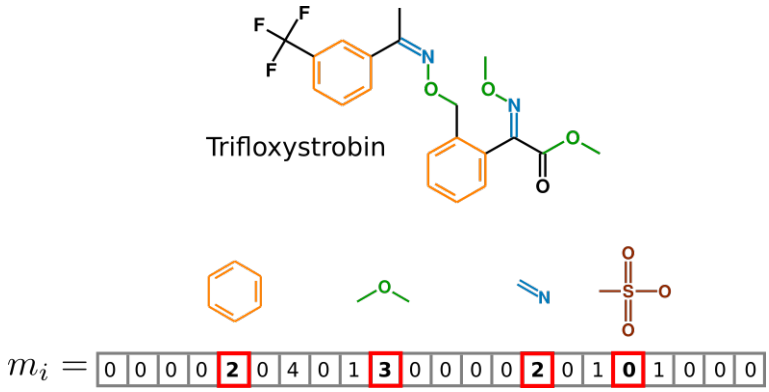
```
    if K.shape[1] != len(d_B):
        raise ValueError("Number of elements of d_B must match the number of␣
↪columns if K.")

    # Normalize kernel matrix
    # YOUR CODE HERE
    K /= np.sqrt(np.outer(d_A, d_B))

    return K
```

[9]:
```
__rng = np.random.RandomState(9212)
__X = __rng.rand(239, 22)
__X_n = Normalizer(norm="l2").fit_transform(__X)  # phi(x)_n = phi(x) /␣
↪||phi(x)||, linear kernel

__KX = linear_kernel_sk(__X, __X)
__KX_n_ref = linear_kernel_sk(__X_n, __X_n)

__KX_n = normalize_kernel(__KX, np.diag(__KX), np.diag(__KX))
assert(np.all(np.diag(__KX_n) == 1.))
np.testing.assert_allclose(__KX_n, __KX_n_ref)

__KX_test_train_n = normalize_kernel(__KX[75:, :75], np.diag(__KX[75:, 75:]),␣
↪np.diag(__KX[:75, :75]))
__KX_test_train_n_ref = linear_kernel_sk(__X_n[75:], __X_n[:75])
np.testing.assert_allclose(__KX_test_train_n, __KX_test_train_n_ref)
```

1.4 3. Predicting molecular properties using Kernel Ridge Regression

In this task you will predict so called logP-values for molecules. LogP is a molecular propery relevant for drug-research. To build a prediction model, you are given 300 experimental lopP values (y_i's) with their corresponding molecular structure. As features ($\mathbf{x}_i$'s) your are given so called molecular counting fingerprints. These are vectors, that represent the molecular graph as a vector indicating the number of occurances of certain substructures within the molecule.

Counting fingerprint illustration: Here $\mathbf{x}_i = m_i$



[10]:
```
def read_data(feat="maccs_fps.csv", n_samples=300, idir="/coursedata/exercise02/
↪"):
    """
    Read in logp data and molecule fingerprints.
    """
    X = np.genfromtxt(idir + "/" + feat, delimiter=",", dtype="int",␣
↪comments=None)[:, 1:]
    y = np.genfromtxt(idir + "/logp_values.csv", delimiter=",", dtype="float",
                      comments=None, usecols=(1))

    # Get a random set of samples
    rng = np.random.RandomState(90210)
    rnd_idx = rng.choice(X.shape[0], n_samples, replace=False)

    return X[rnd_idx], y[rnd_idx]
```

1.4.1 A. Implement the hyper parameter optimization (2 point)

Train (fit) your Kernel Ridge Regression (KRR) with Gaussian kernel on the training examples, i.e. `X_train`. To find the optimal hyper-parameter pair, i.e. KRR regularization β and Gaussian bandwidth parameter σ, we search a grid of different pairs and score each one using cross-validation.

Task: Implement the missing code parts of the <code>hyper_parameter_search_using_cv</co

Note:

- This function is slighly different from the one in Exercise 01, as we need to account for kernel centering and normalization.
- The kernel matrix must be pre-calculated and pre-processed **before** it is passed to the KRR (kernel="precomputed").
- When we center the input kernels, we usually also center the output (y_i's). You can read the decoumentation of TransformedTargetRegressor and StandardScaler to understand, how we can transparently do this our case.

[11]:
```
def hyper_parameter_search_using_cv(estimator, X, y, param_grid, n_cv_folds=5,
                                    center=False, normalize=False,␣
↪center_target=True,
```

```
                                    random_state=None):
    """
    Function calculating the estimator score for a grid of hyper parameters.

    :param estimator: object, subclass of RegressorMixin or ClassifierMixin and␣
↪BaseEstimator
    :param X: array-like, shape=(n_samples, n_features), feature-matrix used␣
↪for training
    :param y: array-like, shape=(n_samples,) or (n_samples, 1), label vector␣
↪used for training
    :param param_grid: dictionary,
        keys: different parameter names
        values: grid-values for each parameter
    :param n_cv_folds: scalar, a KFold cross-validation is performed where the␣
↪number of splits is equal the scalar.
    :param center: boolean, indicating whether the features (x) and targets (y)␣
↪should be centered
    :param normalize: boolean, indicating whether the features (x) should be␣
↪normalized
    :param center_target: boolean, indicating whether the targets (y) should be␣
↪centered. This can be used to overwrite
        'center' for the output.
    :param random_state: scalar, RandomState instance or None, optional,␣
↪default=None
        If int, random_state is the seed used by the random number generator;
        If RandomState instance, random_state is the random number generator;
        If None, the random number generator is the RandomState instance used
        by `np.random`.

    :return: tuple = (
            best estimator,
            param grid as list and corresponding evaluation scores,
            score of best parameter
            best parameter
            kernel centerer (or None if center == False)
            diagonal of the (centered) training kernel matrix (or None of␣
↪normalize == False)
        )
    """
    if estimator.kernel != "precomputed":
        raise ValueError("The estimator will get a pre-computed kernel matrix.")

    if estimator._estimator_type != "regressor":
        raise ValueError("Parameter estimation only supported for regression␣
↪classes.")
```

```
    # Get an iterator over all parameters
    param_grid_iter = ParameterGrid(param_grid)

    # Create cross-validation object
    cv = KFold(n_splits=n_cv_folds, random_state=random_state)

    # Store the valdidation set performance scores for all folds and parameters
    perf_scores = np.zeros((cv.get_n_splits(), len(param_grid_iter)))

    # If we center the input (kernel matrices) we need to center the output
    # as well (y). In order to make this transparent, we wrap the regressor
    # estimator into an TransformedTargetRegressor object. It takes care about
    # the propper output value transformation.
    estimator = TransformedTargetRegressor(
        regressor=estimator,
        transformer=StandardScaler(with_mean=(center and center_target),␣
↪with_std=False))

    # Iterate over the parameter grid. It contains the kernel parameters as␣
↪well.
    # We therefore first calculate the kernel matrix, and than we subset the␣
↪matrix
    # for training, validation, ...
    for idx, param in enumerate(param_grid_iter):
        # This implementation is currently restricted to the Gaussian kernel!
        K = gaussian_kernel_wrapper(X, sigma=param["sigma"])

        for fold, (train_set, val_set) in enumerate(cv.split(K, y)):
            # Separate K_train, K_val_train and K_val from K
            # YOUR CODE HERE
            K_train = K[train_set][:, train_set]
            K_val_train = K[val_set][:, train_set]
            K_val = K[val_set][:, val_set]
            assert(K_train.shape == (len(train_set), len(train_set)))

            # Separate the train and validation targets: y_train, y_val
            # YOUR CODE HERE
            y_train = y[train_set]
            y_val = y[val_set]

            # Center the kernel if needed:
            if center:
                # Fit the KernelCenterer using the training data
                # YOUR CODE HERE
                centerer = KernelCenterer().fit(K_train)

                # Center the different kernel matrices
```

```
                # - Center the training matrix: K_train
                # - Center the validation matrix: K_val (use:␣
↪transform_K_test())
                # - Center the validation-train matrix: K_val_test
                # YOUR CODE HERE
                K_train_c = centerer.transform(K_train)
                K_val_train_c = centerer.transform(K_val_train)
                K_val_c = centerer.transform_K_test(K_val, K_val_train)
            else:
                K_train_c, K_val_c, K_val_train_c = K_train, K_val, K_val_train␣
↪ # flat copies

            if normalize:
                # Extract the diagonals of the training and validation kernel
                # YOUR CODE HERE
                d_train_c = np.diag(K_train_c)
                d_val_c = np.diag(K_val_c)

                # Normalize the different kernel matrices
                # YOUR CODE HERE
                K_train_cn = normalize_kernel(K_train_c, d_train_c)
                K_val_train_cn = normalize_kernel(K_val_train_c, d_val_c,␣
↪d_train_c)
            else:
                K_train_cn, K_val_train_cn = K_train_c, K_val_train_c  # flat␣
↪copies

            # Clone the estimator object to get an un-initialized object
            est = clone(estimator)

            # Set model parameters, e.g. regularization for KRR
            est.set_params(**{"regressor__beta": param["beta"],
                              "regressor__sigma": param["sigma"]})

            # Fit the model using training set
            est.fit(K_train_cn, y_train)

            # Calculate the perf. score on validation set for current fold and␣
↪parameter index
            perf_scores[fold, idx] = est.score(K_val_train_cn, y_val)

    # Find best performing hyper-parameter
    # Average the perf. scores for each parameter across each fold
    avg_perf_scores = np.mean(perf_scores, axis=0)

    idx_best = np.argmax(avg_perf_scores)
    best_perf_score = avg_perf_scores[idx_best]
```

```
    best_param = param_grid_iter[idx_best]

    # Fit model using all data with the best parameters
    est = clone(estimator)
    est.set_params(**{"regressor__beta": best_param["beta"],
                      "regressor__sigma": best_param["sigma"]})

    # Pre-calculate kernel matrix for training the final model
    K = gaussian_kernel_wrapper(X, sigma=best_param["sigma"])

    # Pre-process kernel matrix
    centerer = None
    if center:
        # Fit the kernel centering and transform the training data
        # YOUR CODE HERE
        centerer = KernelCenterer().fit(K)
        K_c = centerer.transform(K)
    else:
        K_c = K

    d_K_c = None
    if normalize:
        # Normalize the kernel matrix
        # YOUR CODE HERE
        d_K_c = np.diag(K_c)
        K_cn = normalize_kernel(K_c, d_K_c)
    else:
        K_cn = K_c

    est.fit(K_cn, y)

    return (est, {"params": list(param_grid_iter), "scores": avg_perf_scores},
            best_perf_score, best_param, centerer, d_K_c)
```

[12]:
```
__rng = np.random.RandomState(320)

# Generate noise cosine curve
__X = np.arange(-0.5, 5.5, 0.05)[:, np.newaxis]
__y = np.cos(2*__X) + __rng.randn(__X.shape[0], 1) * 0.2

# Split to train and test
__X_train, __X_test, __y_train, __y_test = train_test_split(__X, __y,␣
↪random_state=767)

__sigmas = [0.05, 0.5, 4.]
__betas = np.array([0.01, 1, 10]) / __X_train.shape[0]
__param_grid = {"sigma": __sigmas, "beta": __betas}
```

```
# center = False, normalize = False
__est, __param_scores, __best_score, __best_param, __centerer, __d_K_train_c =␣
 ↪hyper_parameter_search_using_cv(
    KernelRidgeRegression(kernel="precomputed"), __X_train, __y_train,␣
 ↪__param_grid,
    center=False, normalize=False, random_state=345)

assert(__centerer is None)
assert(__d_K_train_c is None)
np.testing.assert_equal(np.round(__best_score, 4), 0.9317)
np.testing.assert_equal((__best_param["beta"] * __X_train.shape[0]), 1.0)
np.testing.assert_equal(__best_param["sigma"], 0.5)
np.testing.assert_equal(np.round(__param_scores["scores"], 3),
                        [0.7, 0.928, 0.414, 0.589, 0.932, 0.112, 0.177, 0.735,␣
 ↪0.08])
assert(isinstance(__est, TransformedTargetRegressor))

# center = True, normalize = False
__est, __param_scores, __best_score, __best_param, __centerer, __d_K_train_c =␣
 ↪hyper_parameter_search_using_cv(
    KernelRidgeRegression(kernel="precomputed"), __X_train, __y_train,␣
 ↪__param_grid,
    center=True, normalize=False, random_state=345)

assert(isinstance(__centerer, KernelCenterer))
assert(__d_K_train_c is None)
np.testing.assert_equal(np.round(__best_score, 4), 0.9316)
np.testing.assert_equal((__best_param["beta"] * __X_train.shape[0]), 1.0)
np.testing.assert_equal(__best_param["sigma"], 0.5)
np.testing.assert_equal(np.round(__param_scores["scores"], 3),
                        [0.698, 0.928, 0.415, 0.588, 0.932, 0.109, 0.175, 0.
 ↪735, 0.078])
assert(isinstance(__est, TransformedTargetRegressor))

# center = True, normalize = True
__est, __param_scores, __best_score, __best_param, __centerer, __d_K_train_c =␣
 ↪hyper_parameter_search_using_cv(
    KernelRidgeRegression(kernel="precomputed"), __X_train, __y_train,␣
 ↪__param_grid,
    center=True, normalize=True, random_state=345)

assert(isinstance(__centerer, KernelCenterer))
assert(len(__d_K_train_c) == __X_train.shape[0])
np.testing.assert_equal(np.round(__best_score, 4), 0.9331)
np.testing.assert_equal((__best_param["beta"] * __X_train.shape[0]), 1.0)
```

```
np.testing.assert_equal(__best_param["sigma"], 0.5)
np.testing.assert_equal(np.round(__param_scores["scores"], 3),
                        [0.698, 0.927, 0.695, 0.587, 0.933, 0.224, 0.176, 0.
 ↪776, -0.016])
assert(isinstance(__est, TransformedTargetRegressor))
```

```
/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(
/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(
/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(
```

Load data and split into train and test: We use the training set for hyper-parameter estimation using cross-validation. For the test set we calculate different perormance scores.

[13]:
```
X, y = read_data()
print("(#Molecules, #Features):", X.shape)

X_train, X_test, y_train, y_test, = train_test_split(X, y, random_state=777)

# Hyper parameter grid
param_grid = {"beta": np.array([0.001, 0.01, 0.1, 1.]) / X_train.shape[0],
              "sigma": [15., 20., 25., 30., 35.]}
```

```
(#Molecules, #Features): (300, 142)
```

1.4.2 B. Optimize the hyper-parameters and inspect predictions

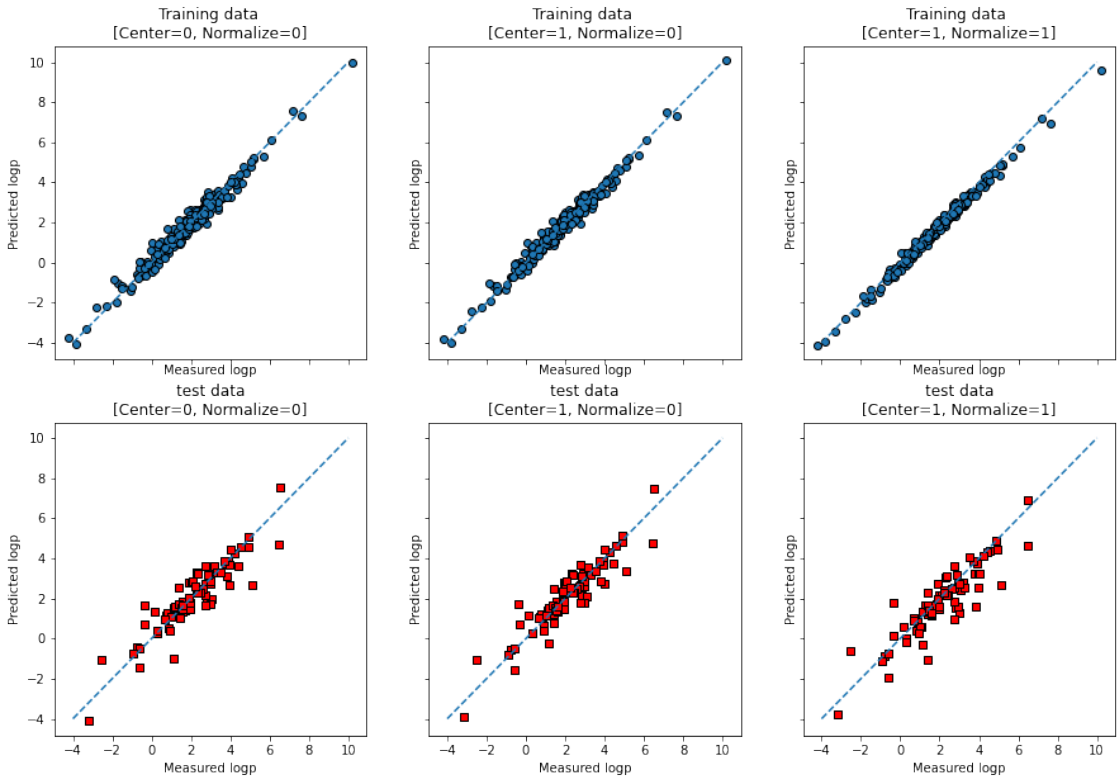
```
<b>Note:</b> We need to apply the same kernel pre-processing (centering or normalization) to t
```

[14]:
```
fig, axrr = plt.subplots(2, 3, figsize=(15, 10), sharex="all", sharey="all")

for idx, (cen, norm) in enumerate([[False, False], [True, False], [True,␣
 ↪True]]):
    print("[Center=%d, Normalize=%d]" % (cen, norm))

    # Find best hyper-parametrs
```

```
    est, param_scores, best_score, best_param, centerer, d_K_train_c =␣
↪hyper_parameter_search_using_cv(
        KernelRidgeRegression(kernel="precomputed"), X_train, y_train,␣
↪param_grid,
        center=cen, normalize=norm, random_state=373)

    print("\tBest parameter: beta=%f, sigma=%f" % (
        best_param["beta"] * X_train.shape[0], best_param["sigma"]))

    # Calculate kernels for prediction and print scores
    K_train = gaussian_kernel_wrapper(X_train, sigma=best_param["sigma"])
    K_test = gaussian_kernel_wrapper(X_test, sigma=best_param["sigma"])
    K_test_train = gaussian_kernel_wrapper(X_test, X_train,␣
↪sigma=best_param["sigma"])

    # Pre-process the kernels if needed
    if cen:
        K_train_c = centerer.transform(K_train)
        K_test_train_c = centerer.transform(K_test_train)
        K_test_c = centerer.transform_K_test(K_test, K_test_train)
    else:
        K_train_c, K_test_c, K_test_train_c = K_train, K_test, K_test_train

    if norm:
        K_train_cn = normalize_kernel(K_train_c, np.diag(K_train_c))
        K_test_train_cn = normalize_kernel(K_test_train_c, np.diag(K_test_c),␣
↪np.diag(K_train_c))
    else:
        K_train_cn, K_test_train_cn = K_train_c, K_test_train_c

    # Predict values
    y_train_pred = est.predict(K_train_cn)
    y_test_pred = est.predict(K_test_train_cn)

    print("\tR^2: train=%.3f, test=%.3f" % (
        est.score(K_train_cn, y_train), est.score(K_test_train_cn, y_test)))
    print("\tMSE: train=%.3f, test=%.3f" % (
        mean_squared_error(y_train_pred, y_train),␣
↪mean_squared_error(y_test_pred, y_test)))


    ##################################################
    # Plot scores for parameter pairs and predictions
    ##################################################
    axrr[0, idx].plot([-4, 10], [-4, 10], '--')
    axrr[0, idx].scatter(y_train, y_train_pred, edgecolors="black")
```

```
    axrr[0, idx].set_title("Training data \n[Center=%d, Normalize=%d]" % (cen,␣
↪norm))
    axrr[0, idx].set_xlabel("Measured logp")
    axrr[0, idx].set_ylabel("Predicted logp")
    axrr[1, idx].plot([-4, 10], [-4, 10], '--')
    axrr[1, idx].scatter(y_test, y_test_pred, c="red", marker="s",␣
↪edgecolors="black")
    axrr[1, idx].set_title("test data\n[Center=%d, Normalize=%d]" % (cen, norm))
    axrr[1, idx].set_xlabel("Measured logp")
    axrr[1, idx].set_ylabel("Predicted logp")
```

```
[Center=0, Normalize=0]

/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(

        Best parameter: beta=0.010000, sigma=25.000000
        R^2: train=0.971, test=0.820
        MSE: train=0.104, test=0.566
[Center=1, Normalize=0]

/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(

        Best parameter: beta=0.010000, sigma=20.000000
        R^2: train=0.979, test=0.854
        MSE: train=0.074, test=0.459
[Center=1, Normalize=1]

/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(

        Best parameter: beta=0.010000, sigma=20.000000
        R^2: train=0.983, test=0.767
        MSE: train=0.061, test=0.733
```



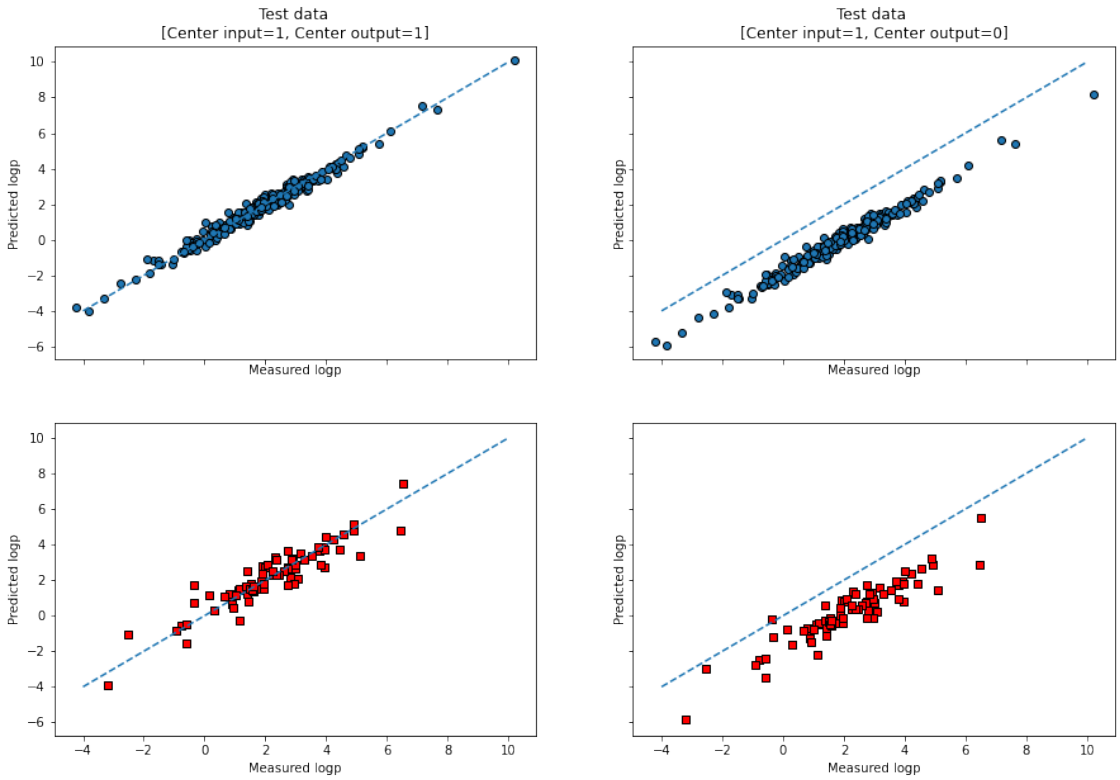
Center input kernel but not the targets: What happens if we do not center the ouputs?

[15]:
```
fig, axrr = plt.subplots(2, 2, figsize=(15, 10), sharex="all", sharey="all")

for idx, (cen_in, cen_out) in enumerate([[True, True], [True, False]]):
    print("[Center input=%d, Center output=%d]" % (cen_in, cen_out))

    # Find best hyper-parametrs
    est, param_scores, best_score, best_param, centerer, D_K_train_c =␣
↪hyper_parameter_search_using_cv(
        KernelRidgeRegression(kernel="precomputed"), X_train, y_train,␣
↪param_grid,
        center=cen_in, normalize=False, center_target=cen_out, random_state=373)

    print("\tBest parameter: beta=%f, sigma=%f" % (
        best_param["beta"] * X_train.shape[0], best_param["sigma"]))

    # Calculate kernels for prediction and print scores
    K_train = gaussian_kernel_wrapper(X_train, sigma=best_param["sigma"])
    K_test = gaussian_kernel_wrapper(X_test, sigma=best_param["sigma"])
    K_test_train = gaussian_kernel_wrapper(X_test, X_train,␣
↪sigma=best_param["sigma"])
```

```
    # Pre-process the kernels if needed
    if cen:
        K_train_c = centerer.transform(K_train)
        K_test_c = centerer.transform_K_test(K_test, K_test_train)
        K_test_train_c = centerer.transform(K_test_train)
    else:
        K_train_c, K_test_c, K_test_train_c = K_train, K_test, K_test_train

    # Predict values
    y_train_pred = est.predict(K_train_c)
    y_test_pred = est.predict(K_test_train_c)

    print("\tR^2: train=%.3f, test=%.3f" % (
        est.score(K_train_c, y_train), est.score(K_test_train_c, y_test)))
    print("\tMSE: train=%.3f, test=%.3f" % (
        mean_squared_error(y_train_pred, y_train),␣
↪mean_squared_error(y_test_pred, y_test)))

    ##################################################
    # Plot scores for parameter pairs and predictions
    axrr[0, idx].plot([-4, 10], [-4, 10], '--')
    axrr[0, idx].scatter(y_train, y_train_pred, edgecolors="black")
    axrr[0, idx].set_title("Training data \n[Center input=%d, Center␣
↪output=%d]" % (cen_in, cen_out))
    axrr[0, idx].set_xlabel("Measured logp")
    axrr[0, idx].set_ylabel("Predicted logp")
    axrr[1, idx].plot([-4, 10], [-4, 10], '--')
    axrr[1, idx].scatter(y_test, y_test_pred, c="red", marker="s",␣
↪edgecolors="black")
    axrr[0, idx].set_title("Test data \n[Center input=%d, Center output=%d]" %␣
↪(cen_in, cen_out))
    axrr[1, idx].set_xlabel("Measured logp")
    axrr[1, idx].set_ylabel("Predicted logp")
```

```
[Center input=1, Center output=1]

/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(

        Best parameter: beta=0.010000, sigma=20.000000
        R^2: train=0.979, test=0.854
        MSE: train=0.074, test=0.459
[Center input=1, Center output=0]

/opt/conda/lib/python3.8/site-packages/sklearn/model_selection/_split.py:293:
```

```
FutureWarning: Setting a random_state has no effect since shuffle is False. This
will raise an error in 0.24. You should leave random_state to its default
(None), or set shuffle=True.
  warnings.warn(

        Best parameter: beta=0.010000, sigma=20.000000
        R^2: train=-0.040, test=-0.300
        MSE: train=3.760, test=4.079
```



1.5 4. Sklearn pipelines

In the Section **??** you could see, that our hyper-parameter search function became relatively complicated. This eventually can lead to quite error-prone implementations. The sklearn package provides a nice tool to chain transformers and estimator using pipelines.

```
<b>Task:</b> Earn a bonus point, by implementing the hyper-parameter search using a chain of t
```

```
<b>Note / Hints:</b>
```

- You do not need integrate the kernel normalization, i.e. only centering is sufficient. Think about it: Why does kernel normalization not fits well to sklearn API?
- Read the documentation of GridSearchCV and Pipeline
- Your pipline should look like: GaussianKernel -> KernelCenterer -> TransformedTargetRegressor(KernelRidgeRegression, StandardScaler)
- Hyper-parameter names must be changed a bit when using pipelines.

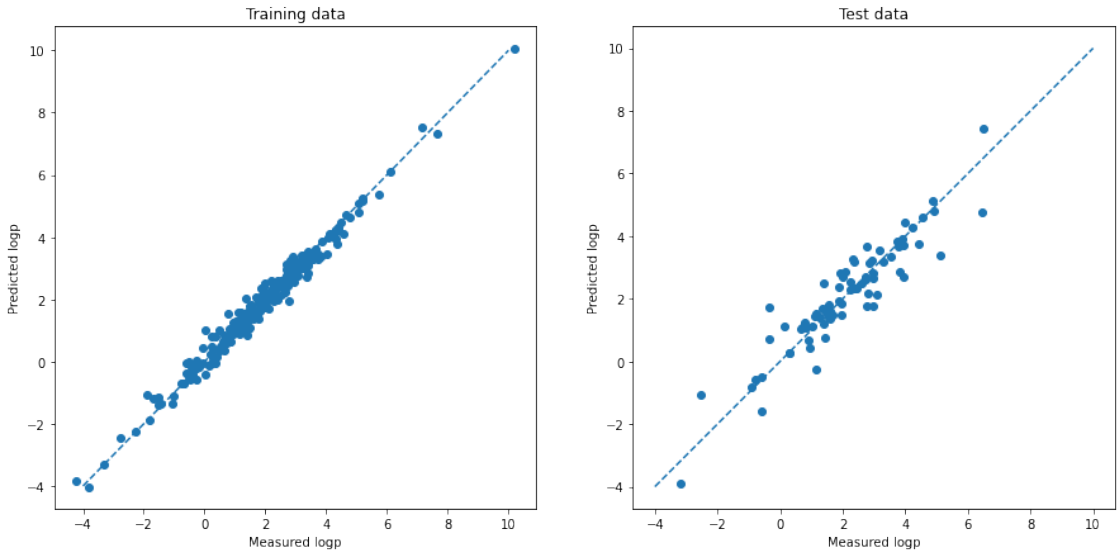
[16]:
```
class GaussianKernel(BaseEstimator, TransformerMixin):
    """
    Class to wrap the Gaussian kernel, so that it is an transformer estimator.
    This allows us to use the kernel in a pipeline.
    """
    def __init__(self, sigma=None):
        self.sigma = sigma
        self.X_train = None

    def fit(self, X, y=None):
        self.X_train = X

        return self

    def transform(self, X, y=None, copy=True):
        return gaussian_kernel_wrapper(X, self.X_train, sigma=self.sigma)
```

[17]:
```
# Build your pipeline
pipe = None
# YOUR CODE HERE
pipe = Pipeline([
    ("gaussian", GaussianKernel()),
    ("input_centering", KernelCenterer()),
    ("KRR",␣
 ↪TransformedTargetRegressor(KernelRidgeRegression(kernel="precomputed"),␣
 ↪StandardScaler(with_std=False)))])

# Set up the parameter grid. Note: Parameter names change a bit when using a␣
 ↪pipeline.
param_grid = None
betas = np.array([0.001, 0.01, 0.1, 1.]) / X_train.shape[0]
sigmas = [15., 20., 25., 30., 35.]
# YOUR CODE HERE
param_grid = {"KRR__regressor__beta": betas, "gaussian__sigma": sigmas}

# Fit a gridsearchcv object using 5-fold cv
gscv = None
# YOUR CODE HERE
gscv = GridSearchCV(pipe, param_grid=param_grid, cv=5, error_score="raise",␣
 ↪return_train_score=False)
_ = gscv.fit(X_train, y_train)
```

```
/opt/conda/lib/python3.8/site-packages/sklearn/utils/validation.py:67:
FutureWarning: Pass transformer=StandardScaler(with_std=False) as keyword args.
From version 0.25 passing these as positional arguments will result in an error
  warnings.warn("Pass {} as keyword args. From version 0.25 "
```

[18]:
```
print("Best params: beta=%f, sigma=%f" % (gscv.
 ↪best_params_["KRR__regressor__beta"] * X_train.shape[0], gscv.
 ↪best_params_["gaussian__sigma"]))
print("R^2: train=%.3f, test=%.3f" % (gscv.score(X_train, y_train), gscv.
 ↪score(X_test, y_test)))
print("MSE: train=%.3f, test=%.3f" % (
    mean_squared_error(gscv.predict(X_train), y_train),
    mean_squared_error(gscv.predict(X_test), y_test)))

fig, axrr = plt.subplots(1, 2, figsize=(15, 7))
axrr[0].scatter(y_train, gscv.predict(X_train))
axrr[0].plot([-4, 10], [-4, 10], '--')
axrr[0].set_title("Training data")
axrr[0].set_xlabel("Measured logp")
axrr[0].set_ylabel("Predicted logp")
axrr[1].scatter(y_test, gscv.predict(X_test))
axrr[1].plot([-4, 10], [-4, 10], '--')
axrr[1].set_title("Test data")
axrr[1].set_xlabel("Measured logp")
_ = axrr[1].set_ylabel("Predicted logp")
```

```
Best params: beta=0.010000, sigma=20.000000
R^2: train=0.979, test=0.854
MSE: train=0.074, test=0.459
```



[ ]: