cicy3o_ml

April 12, 2020

1 Machine Learning for Complete Intersection Calabi-Yau Manifold

In the framework of String Theory, we apply **machine learning** (ML) techniques for the prediction of the **Hodge numbers** of *Complete Intersection Calabi-Yau* (CICY) 3-folds. The relevant quantities are therefore h_{11} and h_{21} which can be predicted starting from the configuration matrices of known manifolds.

We will use both **unsupervised** and **supervised** algorithms to produce an **engineered** dataset to improve the prediction abilities of different algorithms. We use libraries such as *Scikit-learn* (and *XGBoost* if needed) in order to fit the data and *Scikit-optimize* for Bayesan hyperparameter optimization.

1.1 Setup

We first setup the environment and import relevant packages which we will use in the analysis. We print their versions to keep track of changes and set the **random seed** of all random generators in order to get reproducible results.

[1]:
```
import sys

import matplotlib        as mpl
import matplotlib.pyplot as plt
import random            as rnd
import numpy             as np
import pandas            as pd
import sklearn           as skl
import skopt             as sko
import tensorflow        as tf
import xgboost           as xgb

from tensorflow       import keras
from tensorflow.keras import backend as K

import warnings
warnings.simplefilter(action='ignore', category=UserWarning) # ignore␣
 ↪UserWarning: I cannot really do anything about it...
```

```
print('Python version: {:d}.{:d}'      .format(sys.version_info.major, sys.
↪version_info.minor), flush=True)
print('Matplot version: {}'            .format(mpl.__version__),            ␣
↪                 flush=True)
print('Numpy version: {}'              .format(np.__version__),             ␣
↪                 flush=True)
print('Pandas version: {}'             .format(pd.__version__),             ␣
↪                 flush=True)
print('Scikit-learn version: {}'       .format(skl.__version__),            ␣
↪                 flush=True)
print('Scikit-optimize version: {}'    .format(sko.__version__),            ␣
↪                 flush=True)
print('Tensorflow version: {}'         .format(tf.__version__),             ␣
↪                 flush=True)
print('Keras version: {} (backend: {})'.format(keras.__version__, K.backend()),␣
↪                 flush=True)
print('XGBoost version: {}'            .format(xgb.__version__),            ␣
↪                 flush=True)

# fix random_seed
RAND = 42
rnd.seed(RAND)
np.random.seed(RAND)
tf.random.set_seed(RAND)
```

```
Python version: 3.7
Matplot version: 3.2.1
Numpy version: 1.18.1
Pandas version: 1.0.3
Scikit-learn version: 0.22.2.post1
Scikit-optimize version: 0.7.4
Tensorflow version: 2.0.0
Keras version: 2.2.4-tf (backend: tensorflow)
XGBoost version: 0.90
```

We also print the **hardware specifications** to have a representation of the current build.

[2]:
```
!echo "OS:  $(uname -o) - $(lsb_release -d| sed 's/^.*:\s*//g')"
!echo "CPU: $(lscpu| grep 'Model name'| sed 's/^.*:\s*//g')"
!echo "RAM: $(free --giga| awk '/^Mem/ {print $7}') GiB"
!echo "GPU: $(lspci | grep '3D controller' | sed 's/^.*controller:\s*//g')"
```

```
OS:  GNU/Linux - Arch Linux
CPU: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
RAM: 12 GiB
GPU: NVIDIA Corporation GM108M [GeForce 940MX] (rev a2)
```

In order to store information and results, we create a **logger** for the current Python session and a function to print information to the log file (or the standard output, if the logger is not defined).

[3]:
```python
import logging

from os import path, rename
from time import strftime, gmtime

def create_logfile(filename, name='logger', level=logging.INFO):
    """
    Create a logfile and rotate old logs.

    Required arguments:
        filename: the name of the file or path to the log

    Optional arguments
        name:     the name of the log session
        level:    the level of the information stores

    Returns:
        the log
    """

    # get current time to rename strings
    ctime = strftime('.%Y%m%d.%H%M%S', gmtime())

    # rotate log if it already exists
    if path.isfile(filename):
        print('Rotating existing logs...', flush=True)
        rename(filename, filename + ctime)

    # get a logging session by name
    log = logging.getLogger(name + ctime)
    log.setLevel(level)

    # define format
    fmt = logging.Formatter('%(asctime)s --> %(levelname)s: %(message)s')

    # add the log file
    han = logging.FileHandler(filename=filename)
    han.setLevel(level)
    han.setFormatter(fmt)

    # add handler for standard output
    std = logging.StreamHandler(sys.stdout)
    std.setLevel(level)
    std.setFormatter(fmt)
    # create the output
    log.addHandler(han)
    log.addHandler(std)

    print('Created new log file!', flush=True)
    return log

def logprint(string, stream='info', logger=None):
    """
    Decides whether to print on the logger or the standard output.

    Required arguments:
        string: the string to print

    Optional arguments:
        stream: standard input (info) or standard error (error)
        logger: the logger (None for standard output/error)
    """

    if logger is not None:
        if stream == 'info':
            logger.info(string)
        elif stream == 'error':
            logger.error(string)
        else:
            logger.debug(string)
    else:
        if stream == 'info':
            sys.stdout.write(string)
        elif stream == 'error':
            sys.stderr.write(string)
        else:
            sys.stdout.write(string)
```

1.2 Preparation and Tools

We first fetch the desired dataset and prepare the tools for the analysis. Specifically we need to:

1. define the names of the main **directories** and create them if non existent,
2. import the **database** and read the archive,
3. create tools for **visualisation** and **manipulation** of the dataset.

[4]:
```
import logging
from os import makedirs

ROOT_DIR = '.'      # root directory
IMG_DIR  = 'img'    # image directory
MOD_DIR  = 'models' # directory of saved models
```

```
LOG_DIR  = 'log'    # directory for logs

# name of the dataset to be considered
DB_NAME = 'cicy3o'
DB_FILE = DB_NAME + '.h5'                                    # full name with␣
 ↪extension
DB_PATH = path.join(ROOT_DIR, DB_FILE)                       # full path
DB_DIR  = 'original' if DB_NAME == 'cicy3o' else 'favourable' # subdir where to␣
 ↪store images, models, logs

# define full paths
IMG_PATH = path.join(ROOT_DIR, IMG_DIR, DB_DIR)
MOD_PATH = path.join(ROOT_DIR, MOD_DIR, DB_DIR)
LOG_PATH = path.join(ROOT_DIR, LOG_DIR, DB_DIR)

# create directories if non existent
if not path.isdir(IMG_PATH):
    makedirs(IMG_PATH, exist_ok=True)
if not path.isdir(MOD_PATH):
    makedirs(MOD_PATH, exist_ok=True)
if not path.isdir(LOG_PATH):
    makedirs(LOG_PATH, exist_ok=True)

# create logfile
logger = create_logfile(filename=path.join(LOG_PATH, DB_NAME + '_ml.log'),␣
 ↪name='CICY3', level=logging.INFO)
```

```
Created new log file!
```

The we import the database from lpthe.jussieu.fr:

[5]:
```
import tarfile

from urllib import request as rq

URL_ROOT  = 'http://www.lpthe.jussieu.fr/~erbin/files/data/' # root of the URL
TAR_FILE = DB_NAME + '_data.tar.gz'                          # name of the␣
 ↪tarball located at the URL
TAR_PATH = path.join(ROOT_DIR, TAR_FILE)                     # path where to␣
 ↪find the tarball in the local system

if not path.isfile(TAR_PATH): # fetch file only if not already downloaded
    logprint('Fetching dataset...\n', logger=logger)
    _, message = rq.urlretrieve(URL_ROOT + TAR_FILE, TAR_PATH)
    logger.info('Dataset fetched!')

if path.isfile(TAR_PATH):     # extract the tarball
```

```
    logprint('Extracting dataset from tarball...', logger=logger)
    with tarfile.open(TAR_PATH, 'r') as tar:
        tar.extract(DB_FILE, path=ROOT_DIR)
    logprint('Dataset extracted!', logger=logger)
else:
    logprint('Tarball non available: cannot extract tarball!', stream='error',␣
 ↪logger=logger)
```

```
2020-04-11 22:21:47,404 --> INFO: Extracting dataset from tarball…
2020-04-11 22:21:47,483 --> INFO: Dataset extracted!
```

We then prepare to load the dataset and prepare for the visualisation analysis.

[6]:
```
import pandas as pd

def load_dataset(filepath, mode='hdf5', shuffle=False, random_state=None,␣
 ↪logger=None):
    """
    Load a dataset given the path and the format.

    Required arguments:
        filepath: the path of the file

    Optional arguments:
        mode:         the format of the file
        shuffle:      whether to shuffle the file
        random_state: the seed of the random generator
        logger:       the logging session (None for standard output)

    Returns:
        the dataset
    """

    if path.isfile(filepath):
        logprint('Reading database...', logger=logger)
        if mode == 'hdf5':
            df = pd.read_hdf(filepath)
        elif mode == 'csv':
            df = pd.read_csv(filepath)
        logprint('Database loaded!', logger=logger)
    else:
        logprint('Database is not available: cannot load the database!',␣
 ↪stream='error', logger=logger)

    # shuffle the dataframe
    if shuffle and random_state is not None:
        logprint('Shuffling database...', logger=logger)
        df = skl.utils.shuffle(df, random_state=random_state)
        logprint('Database shuffled!', logger=logger)

    return df
```

We then define some functions we can use to extract and manipulate the database. We use the *Scikit-learn* API to create *Estimator* classes (inheriting the *Scikit* interface).

[7]:
```
from sklearn.base import BaseEstimator, TransformerMixin

# remove the outliers from a Pandas dataset
class RemoveOutliers(BaseEstimator, TransformerMixin):
    """
    Remove outlying data given a dataset and a dictionary containing the␣
 ↪intervals for each class.

    E.g.: if the two classes are 'h11' and 'h21', the dictionary will be:␣
 ↪{'h11': [1, 16], 'h21': [1, 86]}.

    Public methods:
        fit:           unused method
        transform:     remove data outside the given interval
        fit_transform: equivalent to transform(fit(...))
    """

    def __init__(self, filter_dict=None):
        """
        Constructor of the class.

        Optional arguments:
            filter_dict: the intervals to retain in the data
        """

        self.filter_dict = filter_dict

    def fit(self, X, y=None):
        """
        Unused method.
        """

        return self

    def transform(self, X):
        """
        Transform the input by deleting data outside the interval

        Required arguments:
            X: the dataset
        Returns:
            the transformed dataset
        """

        x = X.copy() # avoid overwriting

        if self.filter_dict is not None:
            for key in self.filter_dict:
                x = x.loc[x[key] >= self.filter_dict[key][0]]
                x = x.loc[x[key] <= self.filter_dict[key][1]]

        return x

# extract the tensors from a Pandas dataset
class ExtractTensor(BaseEstimator, TransformerMixin):
    """
    Extract a dense tensor from sparse input from a given dataset.

    Public methods:
        fit:           unused method
        transform:     extract dense tensor
        fit_transform: equivalent to transform(fit(...))
        get_shape:     compute the shape of the tensor
    """

    def __init__(self, flatten=False, shape=None):
        """
        Constructor of the class.

        Optional arguments:
            flatten: whether to flatten the output or keep the current shape
            shape:   force the computation with a given shape
        """

        self.flatten = flatten
        self.shape   = shape

    def fit(self, X, y=None):
        """
        Unused method.
        """

        return self

    def transform(self, X):
        """
        Compute the dense equivalent of the sparse input.

        Required arguments:
            X: the dataset

        Returns:
            the transformed input
        """

        x = X.copy() # avoid overwriting
        if self.shape is None:
            self.shape = x.apply(np.shape).max() # get the shape of the tensor

        if len(self.shape) > 0: # apply this to vectors and tensors
            offset = lambda s : [ (0, self.shape[i] - np.shape(s)[i]) for i in␣
↪range(len(self.shape)) ]
            x      = x.apply(lambda s: np.pad(s, offset(s), mode='constant'))

        if self.flatten and len(self.shape) > 0:
            return list(np.stack(x.apply(np.ndarray.flatten).values))
        else:
            return list(np.stack(x.values))

    def get_shape(self):
        """
        Compute the shape of the tensor.

        Returns:
            the shape of the tensor
        """

        return self.shape
```

We the define the functions we will use to evaluate and improve the algorithms. Even though the ultimate goal is a regression task, we will however predict integer values h_{11}, $h_{21} \in \mathbb{Z}$, thus we will evaluate the **accuracy** of the prediction, given the best estimate (in general we use the *mean squared error* to evaluate the algorithms, but we accept those with best *accuracy*).

[8]:
```
# get the accuracy (possibly after rounding)
def accuracy_score(y_true, y_pred, rounding=np.rint):
    """
    Compute the accuracy of the predictions after rounding.

    Required arguments:
        y_true: true values
        y_pred: predicted values
    Optional arguments:
        rounding: the Numpy function for rounding the predictions
    """

    assert np.shape(y_true)[0] == np.shape(y_pred)[0] # check if same length

    # if same length then proceed
    accuracy = 0
    if rounding is not None:
        for n in range(np.shape(y_true)[0]):
            accuracy = accuracy + 1 \
                       if int(y_true[n]) == int(rounding(y_pred[n])) \
                       else accuracy
    else:
        for n in range(np.shape(y_true)[0]):
            accuracy = accuracy + 1 \
                       if y_true[n] == y_pred[n] \
                       else accuracy
    return accuracy / np.shape(y_true)[0]

# get the error difference (possibly after rounding)
def error_diff(y_true, y_pred, rounding=np.rint):
    """
    Compute the error difference between true values and predictions (positive␣
 ↪values are overestimate and viceversa).

    Required arguments:
        y_true: true values
        y_pred: predicted values

    Optional arguments:
        rounding: the Numpy function for rounding the predictions
    """

    assert np.shape(y_true)[0] == np.shape(y_pred)[0] # check if same length

    # if same length then proceed
    err = y_true - rounding(y_pred)
    return np.array(err).astype(np.int8)

# print *SearchCV scores
def gridcv_score(estimator, rounding=np.rint, logger=None):
    """
    Print scores given by cross-validation and optimisation techniques.

    Required arguments:
        estimator: the estimator to be evaluated
    Optional arguments:
        rounding: the Numpy function for rounding the predictions
        logger:   the logging session (None for standard output)
    """

    best_params = estimator.best_params_               # get best parameters
    df          = pd.DataFrame(estimator.cv_results_) # dataframe with CV res.

    cv_best_res = df.loc[df['params'] == best_params] # get best results
    accuracy    = cv_best_res.loc[:, 'mean_test_score'].values[0]
    std         = cv_best_res.loc[:, 'std_test_score'].values[0]

    logprint('Best parameters: {}'.format(best_params), logger=logger)
    logprint('Accuracy ({}) of cross-validation: ({:.3f} ± {:.3f})%'.
 ↪format(rounding.__name__, accuracy*100, std*100), logger=logger)

# print the accuracy of the predictions
def prediction_score(estimator, X, y, use_best_estimator=False, rounding=np.
 ↪rint, logger=None):
    """
    Print the accuracy of the predictions.

    Required arguments:
        estimator: the estimator to be used for the predictions
        X: the features
        y: the labels (actual values)

    Optional arguments:
        use_best_estimator: whether to use the estimator.best_estimator_ or␣
 ↪just estimator
        rounding: the Numpy function for rounding the predictions
        logger:   the logging session (None for standard output)
    """

    if use_best_estimator:
        estimator = estimator.best_estimator_

    accuracy = accuracy_score(y, estimator.predict(X), rounding=rounding)
    logprint('Accuracy ({}) of the predictions: {:.3f}%'.format(rounding.
 ↪__name__, accuracy*100), logger=logger)
```

We use *Matplotlib* to plot the data and define a few functions which we can use during the analysis:

[9]:
```
# set label sizes
%matplotlib inline
mpl.rc('axes', labelsize=12)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# set building block sizes for the plot
mpl_width  = 6
mpl_height = 5

# save the current figure
def save_fig(filename, tight_layout=True, extension='png', resolution=96,␣
 ↪logger=None):
    """
    Save current figure to file.

    Required arguments:
        filename: the name of the file where to save the figure (without␣
 ↪extension)

    Optional arguments:
        tight_layout: whether to use the tight_layout
        extension:    extension of the file to use
        resolution:   resolution of the file
        logger:       the logging session (None for standard output)
    """

    filename = path.join(IMG_PATH, filename + '.' + extension)
    if tight_layout:
        plt.tight_layout()

    logprint('Saving {}...'.format(filename), logger=logger)
    plt.savefig(filename, format=extension, dpi=resolution)
    logprint('Saved {}!'.format(filename), logger=logger)

# get a generator to count the occurrencies
def get_counts(df, label, feature):
    """
    Generator to produce the count of unique occurrencies of the data.

    Required arguments:
        df:      the Pandas dataframe
        label:   the label to consider
        feature: the feature to consider

    Yields:
        [ unique feature, unique value, counts ]
    """

    for n in np.sort(df[feature].unique()):
        uniques, counts = np.unique(df[label].loc[df[feature] == n].values,␣
↪return_counts=True)
        for u, c in np.c_[uniques, counts]:
            yield [ n, u, c ]

# plot histogram of occurrencies
def count_plot(ax, data, title=None, xlabel=None, ylabel='N',
               legend=None, xlog=False, ylog=False, binstep=5,
               **kwargs):
    """
    Plot histogram of occurrencies (e.g.: frequency plot).

    Required arguments:
        ax:    the subplot ax where to plot data
        data: the data to plot

    Optional arguments:
        title:     the title of the plot
        xlabel:    the label of the x axis
        ylabel:    the label of the y axis
        legend:    the label for the legend in the plot
        xlog:      whether to use the log scale on the x axis
        ylog:      whether to use the log scale on the y axis
        binstep:   the distance between adjacent bins
        **kwargs: additional arguments to pass to plt.hist
    """

    min_tick = np.min(data) if np.min(data) > -100 else -100 # set a MIN cut
    max_tick = np.max(data) if np.max(data) < 100  else 100  # set a MAX cut

    ax.grid(alpha=0.2)                        # create a grid
    ax.set_title(title)                       # set title
    ax.set_xlabel(xlabel)                     # set a label for the x axis
    ax.set_ylabel(ylabel)                     # set a label for the y axis
    ax.set_xticks(np.arange(min_tick,         # set no. of ticks in the x axis
                            max_tick,
                            step=binstep
                           )
                 )

    if xlog:                                  # use log scale in x axis if needed
        ax.set_xscale('log')
    if ylog:                                  # use log scale in y axis if needed
        ax.set_yscale('log')

    ax.hist(data,                             # create histogram using 'step' funct.
            histtype='step',
            label=legend,
            **kwargs)

    if legend is not None:                # add legend
        ax.legend(loc='best')

    return ax

# plot labeled features and their values
def label_plot(ax, data, title=None, xlabel=None, ylabel='values',
               legend=None, xlog=False, ylog=False, binstep=1,
               **kwargs):
    """
    Plot values of labelled data (e.g.: variable ranking).

    Required arguments:
        ax:   the subplot ax where to plot data
        data: the data to plot

    Optional arguments:
        title:    the title of the plot
        xlabel:   the label of the x axis
        ylabel:   the label of the y axis
        legend:   the label for the legend in the plot
        xlog:     whether to use the log scale on the x axis
        ylog:     whether to use the log scale on the y axis
        binstep:  the distance between adjacent bins
        **kwargs: additional arguments to pass to plt.plot
    """

    labels      = [f[0] for f in data]    # labels vector
    importances = [f[1] for f in data]    # importances vector
    length      = len(labels)             # length of the labels vector

    ax.grid(alpha=0.2)                    # create a grid
    ax.set_title(title)                   # set title
    ax.set_xlabel(xlabel)                 # set a label for the x axis
    ax.set_ylabel(ylabel)                 # set a label for the x axis

    ax.set_xticks(np.arange(length,       # set no. of ticks in the x axis
                            step=binstep
                           )
                 )
    ax.set_xticklabels(labels,            # set name of labels of the x axis
                       ha='right',        # horizontal alignment
                       rotation=45        # rotation of the labels
                      )
    if xlog:                                 # use log scale in x axis if needed
        ax.set_xscale('log')
    if ylog:                                 # use log scale in y axis if needed
        ax.set_yscale('log')

    ax.plot(np.arange(length),               # plot data
            importances,
            label=legend,
            **kwargs)

    if legend is not None:                   # add legend
        ax.legend(loc='best')

    return ax

# plot the correlation matrix of a Pandas dataframe
def mat_plot(ax, matrix, labels, label='correlation matrix', **kwargs):
    """
    Plot the correlation matrix of a given dataframe.

    Required arguments:
        ax:     the subplot ax where to plot data
        matrix: the matrix to plot
        labels: the labels to show with the matrix

    Optional arguments:
        label:    the label to use for the colour bar
        **kwargs: additional arguments to pass to plt.matshow
    """

    ax.set_xticks(np.arange(len(labels), # set ticks for x axis
                  step=1)
                 )
    ax.set_xticklabels([''] + labels,    # set the name of the ticks
                       rotation=90
                      )

    ax.set_yticks(np.arange(len(labels), # set ticks for y axis
                  step=1)
                 )
    ax.set_yticklabels([''] + labels)    # set the name of the ticks

    matshow = ax.matshow(matrix,         # show the matrix
                         vmin=-1.0,
                         vmax=1.0,
                         **kwargs
                         )

    cbar = ax.figure.colorbar(matshow,   # create the colour bar
                              ax=ax,
                              fraction=0.05,
                              pad=0.05
                             )
    cbar.ax.set_ylabel(label,            # show the colour bar
                       va='bottom',      # vertical alignment
                       rotation=-90)     # rotation of the label

    return ax

# plot a scatter plot with colours and sizes
def scatter_plot(ax, data, title=None, xlabel=None, ylabel=None,
                 legend=None, xlog=False, ylog=False,
                 colour=True, size=True, colour_label='N', size_leg=0,
                 **kwargs):
    """
    Scatter plot of occurrencies with colour and size codes.

    Required arguments:
        ax:   the subplot ax where to plot data
        data: the data to plot

    Optional arguments:
        title:        the title of the plot
        xlabel:       the label of the x axis
        ylabel:       the label of the y axis
        legend:       the label for the legend in the plot
        xlog:         whether to use the log scale on the x axis
        ylog:         whether to use the log scale on the y axis
        colour:       whether to use colour codes
        size:         whether to use entries of different size
        colour_label: label to use for the colour code
        size_leg:     length of the legend of the size code
        **kwargs:     additional arguments to pass to plt.scatter
    """

    ax.grid(alpha=0.2)                   # create  a grid
    ax.set_xlabel(xlabel)                # set labels for the x axis
    ax.set_ylabel(ylabel)                # set labels for the y axis
    ax.set_title(title)                  # set title

    if xlog:                             # use log scale in x axis if needed
        ax.set_xscale('log')
    if ylog:                             # use log scale in y axis if needed
```

```
        ax.set_yscale('log')

    if colour:                              # create the plot with size and colours
        if size:
            scat = ax.scatter(data[0], data[1], s=data[2], c=data[2], **kwargs)
        else:
            scat = ax.scatter(data[0], data[1], c=data[2], **kwargs)
        cbar = ax.figure.colorbar(scat, ax=ax)
        cbar.ax.set_ylabel(colour_label, rotation=-90, va='bottom')
    else:
        if size:
            scat = ax.scatter(data[0], data[1], s=data[2], **kwargs)
        else:
            scat = ax.scatter(data[0], data[1], **kwargs)

    scat.set_label(legend)                  # set label of the plot
    if size_leg:                            # add the size legend if needed
        handles, labels = scat.legend_elements('sizes', num=size_leg)
        ax.legend(handles, labels, loc='lower center',
                  bbox_to_anchor=(0.5,-0.3), ncol=len(handles),
                  fontsize='medium', frameon=False)

    if legend:                              # show the legend
        ax.legend(loc='best')

    return ax

# plot a series with trivial x label
def series_plot(ax, data, title=None, xlabel='series', ylabel=None,
                legend=None, xlog=False, ylog=False,
                step=False, std=False,
                **kwargs):
    """
    Plot a series of data with ordered x axis (e.g.: epoch series).

    Required arguments:
        ax:   the subplot ax where to plot data
        data: the data to plot

    Optional arguments:
        title:    the title of the plot
        xlabel:   the label of the x axis
        ylabel:   the label of the y axis
        legend:   the label for the legend in the plot
        xlog:     whether to use the log scale on the x axis
        ylog:     whether to use the log scale on the y axis
        step:     whether to use a step function for the plot
        std:      highlight the strip of the standard deviation
        **kwargs: additional arguments to pass to plt.step or plot.plot
    """

    ax.grid(alpha=0.2)                         # create the grid
    ax.set_title(title)                        # set the title
    ax.set_xlabel(xlabel)                      # set labels for the x axis
    ax.set_ylabel(ylabel)                      # set labels for the y axis

    if xlog:                                   # use log scale in the x axis if needed
        ax.set_xscale('log')
    if ylog:                                   # use log scale in the y axis if needed
        ax.set_yscale('log')

    series = np.arange(len(data))              # create trivial x axis data
    if step:                                   # create the plot
        ax.step(series, data, label=legend, **kwargs)
    else:
        ax.plot(series, data, label=legend, **kwargs)

    if std:                                    # show coloured strip with std
        ax.fill_between(series,
                        data + np.std(data),
                        data - np.std(data),
                        alpha=0.2)

    if legend is not None:                     # show the legend
        ax.legend(loc='best')

    return ax
```

Before going further we also set the **memory growth** of the GPU RAM in order to avoid memory issues:

[10]:
```
# get the list of installed GPUs
gpus = tf.config.experimental.list_physical_devices('GPU')

if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True) # set memory␣
 ↪growth

        # get the list of logical devices (GPUs)
        logical_gpus = tf.config.experimental.list_logical_devices('GPU')
        logprint('GPU setup: {:d} physical GPUs, {:d} logical GPUs.'.
 ↪format(len(gpus), len(logical_gpus)), logger=logger)
```

```
    except RuntimeError as e:
        logprint(e, stream='error', logger=logger)
else:
    logprint('No GPUs in the setup!', stream='error', logger=logger)
```

```
2020-04-11 22:21:48,209 --> INFO: GPU setup: 1 physical GPUs, 1 logical GPUs.
```

1.3 Data Visualisation

We first visualise the data inside the dataset. We focus on manifolds which are **not direct products** of other spaces (i.e. we consider only entries with `isprod = 0`) and consider a restricted interval of the Hodge numbers, specifically $h_{11} \in [1, 16]$ and $h_{21} \in [1, 86]$, in order to avoid outliers which could spoil the training. We show the distribution of the labels both in their frequency and with respect to a few of the features.

[11]:
```
# load the database
df = load_dataset(DB_PATH, shuffle=True, random_state=RAND, logger=logger)
```

```
2020-04-09 11:51:29,235 --> INFO: Reading database…
2020-04-09 11:51:29,845 --> INFO: Database loaded!
2020-04-09 11:51:29,846 --> INFO: Shuffling database…
2020-04-09 11:51:29,853 --> INFO: Database shuffled!
```

As a reference we print the name of the columns and their respective *dtypes*:

[12]:
```
df.dtypes
```

[12]:
```
c2                  object
euler                int16
h11                  int16
h21                  int16
matrix              object
redun               object
size                object
num_cp                int8
num_eqs              int64
dim_cp              object
min_dim_cp           int64
max_dim_cp           int64
mean_dim_cp        float64
median_dim_cp      float64
num_dim_cp          object
num_cp_1              int8
num_cp_2              int8
num_cp_neq1           int8
num_over              int8
num_ex                int8
deg_eqs             object
```

```
min_deg_eqs          int64
max_deg_eqs          int64
mean_deg_eqs       float64
median_deg_eqs     float64
num_deg_eqs         object
rank_matrix           int8
norm_matrix        float64
dim_h0_amb          object
isprod               int64
favour               int64
dtype: object
```

We then extract only the entries such that `isprod = 0` and remove the **outliers**:

[13]:
```
# remove product spaces
logprint('Removing product spaces...', logger=logger)
df_noprod = df.loc[df['isprod'] == 0].drop(columns='isprod')
logprint('Product spaces removed!', logger=logger)

# remove outliers
filter_dict = {'h11': [1,16], 'h21': [1,86]}
logprint('Removing outliers...', logger=logger)
df_noprod_noout = RemoveOutliers(filter_dict=filter_dict).
 ↪fit_transform(df_noprod)
logprint('Outliers removed!', logger=logger)
```

```
2020-04-09 11:51:29,873 --> INFO: Removing product spaces…
2020-04-09 11:51:29,887 --> INFO: Product spaces removed!
2020-04-09 11:51:29,890 --> INFO: Removing outliers…
2020-04-09 11:51:29,926 --> INFO: Outliers removed!
```

1.3.1 Occurrencies of the Labels

We then plot the distributions for $h_{11}$:

[14]:
```
xplots  = 2
yplots  = 2
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax[0,0], df['h11'],               title='Distribution of $h_{11}$␣
 ↪(original dataset)',                  xlabel='$h_{11}$', ylog=True)
count_plot(ax[0,1], df_noprod['h11'],        title='Distribution of $h_{11}$ (w/
 ↪o product spaces)',                xlabel='$h_{11}$', ylog=True)
count_plot(ax[1,0], df_noprod_noout['h11'], title='Distribution of $h_{11}$ (w/
 ↪o product spaces and outliers)', xlabel='$h_{11}$', ylog=True)
```

```
count_plot(ax[1,1], df['h11'],              title='Distribution of $h_{11}$',  ␣
↪                                xlabel='$h_{11}$', ylog=True,␣
↪legend='original')
count_plot(ax[1,1], df_noprod['h11'],       title='Distribution of $h_{11}$',  ␣
↪                                xlabel='$h_{11}$', ylog=True, legend='no␣
↪prod')
count_plot(ax[1,1], df_noprod_noout['h11'], title='Distribution of $h_{11}$',  ␣
↪                                xlabel='$h_{11}$', ylog=True, legend='no␣
↪out')

save_fig('h11_distribution', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:51:31,215 --> INFO: Saving ./img/original/h11_distribution.png…
2020-04-09 11:51:31,790 --> INFO: Saved ./img/original/h11_distribution.png!
```

And the distributions for h_{21}:

[15]:
```
xplots  = 2
yplots  = 2
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax[0,0], df['h11'],               title='Distribution of $h_{21}$␣
 ↪(original dataset)',                  xlabel='$h_{21}$', ylog=True)
count_plot(ax[0,1], df_noprod['h21'],        title='Distribution of $h_{21}$ (w/
 ↪o product spaces)',                xlabel='$h_{21}$', ylog=True)
count_plot(ax[1,0], df_noprod_noout['h21'], title='Distribution of $h_{21}$ (w/
 ↪o product spaces and outliers)', xlabel='$h_{21}$', ylog=True)

count_plot(ax[1,1], df['h21'],               title='Distribution of $h_{21}$',  ␣
 ↪                                xlabel='$h_{21}$', ylog=True,␣
 ↪legend='original')
count_plot(ax[1,1], df_noprod['h21'],        title='Distribution of $h_{21}$',  ␣
 ↪                                xlabel='$h_{21}$', ylog=True, legend='no␣
 ↪prod')
count_plot(ax[1,1], df_noprod_noout['h21'], title='Distribution of $h_{21}$',  ␣
 ↪                                xlabel='$h_{21}$', ylog=True, legend='no␣
 ↪out')

save_fig('h21_distribution', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:51:32,993 --> INFO: Saving ./img/original/h21_distribution.png…
2020-04-09 11:51:33,532 --> INFO: Saved ./img/original/h21_distribution.png!
```

1.3.2 Distribution of the Labels

We now consider some of the scalar features and visualise the distribution of h_{11} and h_{21} as functions of those parameters.

[16]:
```
scat_features = [ 'num_cp', 'num_eqs', 'norm_matrix', 'rank_matrix' ]
scat_labels   = [ 'h11', 'h21' ]

fig, ax = plt.subplots(len(scat_features), len(scat_labels),␣
 ↪figsize=(len(scat_labels)*mpl_width, len(scat_features)*mpl_height))
fig.tight_layout()

for n in range(len(scat_features)):
    for m in range(len(scat_labels)):
        scatter_plot(ax[n,m], np.array(list(get_counts(df_noprod_noout,␣
 ↪scat_labels[m], scat_features[n]))).T,
                     title='Distribution of the labels',␣
 ↪xlabel=scat_features[n], ylabel=scat_labels[m],
                     colour_label='no. of occurrencies', size_leg=5)

save_fig('h11_h21_distribution', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:51:35,184 --> INFO: Saving
./img/original/h11_h21_distribution.png…
2020-04-09 11:51:36,078 --> INFO: Saved ./img/original/h11_h21_distribution.png!
```

1.4 Data Extraction and Unsupervised Preprocessing

We then extract usable data and start to analyse the features through unsupervised preprocessing of the data. We try both a **clustering** and a **principal component analysis** (PCA) approaches in order to simplify the data.

[17]:
```
# use only the dataframe without product spaces and outliers (and remove␣
↪irrelevant features)
df = df_noprod_noout.drop(labels=df_noprod_noout.
↪filter(regex='min|max|mean|media|c2|redun|size|euler|favour'), axis=1)

# divide features and labels
labels  = ['h11', 'h21']
logprint('Selecting labels...', logger=logger)
df_labs = df[labels]
logprint('Labels selected!', logger=logger)
df_feat = df.drop(labels=labels, axis=1)

# the only tensor feature
tensor_feat = ['matrix']

# then extract the others (scalars are columns of type int or float, while␣
↪object refers to vectors, but we must avoid counting the matrix twice)
logprint('Selecting features...', logger=logger)
scalar_feat = list(df_feat.select_dtypes(include=[np.int8, np.int16, np.int64,␣
↪np.float16, np.float32, np.float64, np.float128]).columns)
vector_feat = list(df_feat.drop(labels=tensor_feat, axis=1).
↪select_dtypes(include=['object']).columns)
logprint('Features selected!', logger=logger)
```

```
2020-04-09 11:51:37,275 --> INFO: Selecting labels…
2020-04-09 11:51:37,277 --> INFO: Labels selected!
2020-04-09 11:51:37,281 --> INFO: Selecting features…
2020-04-09 11:51:37,292 --> INFO: Features selected!
```

We then proceed to extract the features using the `ExtractTensor` transformer. In this case we do not need to flatten the result as we are only extracting the full matrices from the sparse factor.

[18]:
```
for feature in vector_feat:
    logprint('Extracting {} from vector features...'.format(feature),␣
↪logger=logger)
    df_feat[feature] = ExtractTensor(flatten=False).
↪fit_transform(df_feat[feature]) # do not flatten the output
logprint('Vector features have been extracted!', logger=logger)
```

```
for feature in tensor_feat:
    logprint('Extracting {} from tensor features...'.format(feature),␣
 ↪logger=logger)
    df_feat[feature] = ExtractTensor(flatten=False).
 ↪fit_transform(df_feat[feature]) # do not flatten the output
logprint('Tensor features have been extracted!', logger=logger)

df_feat = df_feat[scalar_feat + vector_feat + tensor_feat]
logprint('Features have been fully extracted!', logger=logger)
```

```
2020-04-09 11:51:37,301 --> INFO: Extracting dim_cp from vector features…
2020-04-09 11:51:37,631 --> INFO: Extracting num_dim_cp from vector features…
2020-04-09 11:51:37,916 --> INFO: Extracting deg_eqs from vector features…
2020-04-09 11:51:38,199 --> INFO: Extracting num_deg_eqs from vector features…
2020-04-09 11:51:38,533 --> INFO: Extracting dim_h0_amb from vector features…
2020-04-09 11:51:38,735 --> INFO: Vector features have been extracted!
2020-04-09 11:51:38,736 --> INFO: Extracting matrix from tensor features…
2020-04-09 11:51:39,372 --> INFO: Tensor features have been extracted!
2020-04-09 11:51:39,375 --> INFO: Features have been fully extracted!
```

We then show the correlation matrix between scalar features in order to better understand the relation between them.

[19]:
```
xplots  = 3
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

mat_plot(ax[0], df_labs.corr(), df_labs.columns.to_list())
mat_plot(ax[1], df_feat[scalar_feat].corr(), df_feat[scalar_feat].columns.
 ↪to_list())
mat_plot(ax[2],
         df_labs.join(df_feat)[['h11', 'h21', 'num_cp', 'num_eqs', 'num_over',␣
 ↪'num_ex', 'rank_matrix', 'norm_matrix']].corr(),
         df_labs.join(df_feat)[['h11', 'h21', 'num_cp', 'num_eqs', 'num_over',␣
 ↪'num_ex', 'rank_matrix', 'norm_matrix']].columns.to_list()
        )

save_fig('correlation_matrix', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:51:39,683 --> INFO: Saving
./img/original/correlation_matrix.png…
2020-04-09 11:51:39,933 --> INFO: Saved ./img/original/correlation_matrix.png!
```

1.4.1 Nearest Neighbours Clustering

We consider the `KMeans` clustering of the component of the configuration matrix to probe the distribution of its components.

[20]:
```python
from sklearn.cluster import KMeans

# flatten the matrix
flat_matrix       = ExtractTensor(flatten=True).fit_transform(df_feat['matrix'])
flat_matrix_shape = np.shape(flat_matrix)

# create an empty array to store the labels
cluster_range = range(2,20)
kmeans_labels = np.empty((flat_matrix_shape[0], cluster_range.stop -␣
 ↪cluster_range.start), dtype=np.int8)

# compute various clustering classifications
for n_clusters in cluster_range:
    logprint('Computing clustering for {:d} clusters...'.format(n_clusters),␣
 ↪logger=logger)
    kmeans = KMeans(n_clusters=n_clusters, random_state=RAND, n_jobs=-1)
    kmeans.fit_transform(flat_matrix)
    kmeans_labels[:,n_clusters - cluster_range.start] = kmeans.labels_
logprint('Clustering task ended!', logger=logger)
```

```
2020-04-09 11:51:40,767 --> INFO: Computing clustering for 2 clusters…
2020-04-09 11:51:41,811 --> INFO: Computing clustering for 3 clusters…
2020-04-09 11:51:42,327 --> INFO: Computing clustering for 4 clusters…
2020-04-09 11:51:43,215 --> INFO: Computing clustering for 5 clusters…
2020-04-09 11:51:44,390 --> INFO: Computing clustering for 6 clusters…
2020-04-09 11:51:45,614 --> INFO: Computing clustering for 7 clusters…
2020-04-09 11:51:46,837 --> INFO: Computing clustering for 8 clusters…
2020-04-09 11:51:48,101 --> INFO: Computing clustering for 9 clusters…
2020-04-09 11:51:49,506 --> INFO: Computing clustering for 10 clusters…
2020-04-09 11:51:50,933 --> INFO: Computing clustering for 11 clusters…
2020-04-09 11:51:52,394 --> INFO: Computing clustering for 12 clusters…
```

```
2020-04-09 11:51:54,010 --> INFO: Computing clustering for 13 clusters…
2020-04-09 11:51:55,756 --> INFO: Computing clustering for 14 clusters…
2020-04-09 11:51:57,165 --> INFO: Computing clustering for 15 clusters…
2020-04-09 11:51:58,829 --> INFO: Computing clustering for 16 clusters…
2020-04-09 11:52:00,598 --> INFO: Computing clustering for 17 clusters…
2020-04-09 11:52:02,207 --> INFO: Computing clustering for 18 clusters…
2020-04-09 11:52:04,052 --> INFO: Computing clustering for 19 clusters…
2020-04-09 11:52:06,200 --> INFO: Clustering task ended!
```

We then include this into the dataframe:

[21]:
```
# add clustering to dataframe
df_feat['clustering'] = list(kmeans_labels)
logprint('Adding clustering labels to dataframe...', logger=logger)

# reorder the dataframe
df_feat = df_feat[scalar_feat + ['clustering'] + vector_feat + tensor_feat]
logprint('Labels have been added!', logger=logger)
```

```
2020-04-09 11:52:06,228 --> INFO: Adding clustering labels to dataframe…
2020-04-09 11:52:06,254 --> INFO: Labels have been added!
```

1.4.2 Principal Components Analysis

We then proceed with the PCA on the configuration matrix. We compute the algorithms with 2 principal components only for plotting purposes but we will the discard the results. Instead we will keep the 99% variance PCA.

[22]:
```
from sklearn.decomposition import PCA

# compute the PCA
logprint('Computing PCA with 2 components...', logger=logger)
pca_2 = PCA(n_components=2, random_state=RAND)
matrix_pca_2 = pca_2.fit_transform(flat_matrix)
logprint('PCA with 2 components computed!', logger=logger)
logprint('Ratio of the variance retained for each component: {:.2f}%, {:.2f}%'.
 ↪format(pca_2.explained_variance_ratio_[0]*100, pca_2.
 ↪explained_variance_ratio_[1]*100), logger=logger)

# plot the labels with respect to the principal components
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

scatter_plot(ax[0], [matrix_pca_2[:,0], matrix_pca_2[:,1], df_labs['h11']],␣
 ↪title='Distribution of $h_{11}$ w.r.t. PCA(n = 2)', xlabel='PCA #1',␣
 ↪ylabel='PCA #2', size=True)
```

```
scatter_plot(ax[1], [matrix_pca_2[:,0], matrix_pca_2[:,1], df_labs['h21']],␣
↪title='Distribution of $h_{21}$ w.r.t. PCA(n = 2)', xlabel='PCA #1',␣
↪ylabel='PCA #2', size=True)

save_fig('h11_h21_pca_2_comp', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:06,294 --> INFO: Computing PCA with 2 components…
2020-04-09 11:52:06,467 --> INFO: PCA with 2 components computed!
2020-04-09 11:52:06,476 --> INFO: Ratio of the variance retained for each
component: 9.09%, 5.59%
2020-04-09 11:52:06,712 --> INFO: Saving
./img/original/h11_h21_pca_2_comp.png…
2020-04-09 11:52:07,282 --> INFO: Saved ./img/original/h11_h21_pca_2_comp.png!
```

Then compute the "good" PCA:

[23]:
```
# compute the PCA with 99% variance
logprint('Computing PCA with 99% variance...', logger=logger)
matrix_pca99 = PCA(n_components=0.99, random_state=42).
↪fit_transform(flat_matrix)
logprint('PCA with 99% variance computed!', logger=logger)

# analyse the result
matrix_pca99_shape = matrix_pca99.shape
logprint('No. of components of the PCA: {:d}'.format(matrix_pca99_shape[1]),␣
↪logger=logger)

# save the results inside the dataframe
df_feat['matrix_pca99'] = list(matrix_pca99)
```

```
logprint('Adding PCA to dataframe...', logger=logger)

# reorder the dataframe
df_feat = df_feat[scalar_feat + ['clustering'] + vector_feat + tensor_feat +␣
↪['matrix_pca99']]
logprint('PCA has been added!', logger=logger)
```

```
2020-04-09 11:52:07,761 --> INFO: Computing PCA with 99% variance…
2020-04-09 11:52:07,860 --> INFO: PCA with 99% variance computed!
2020-04-09 11:52:07,861 --> INFO: No. of components of the PCA: 81
2020-04-09 11:52:07,866 --> INFO: Adding PCA to dataframe…
2020-04-09 11:52:07,875 --> INFO: PCA has been added!
```

1.5 Variable Ranking and Feature Selection

Using the engineered features including `PCA` and `KMeans`, we then train a RandomForestRegressor in order to extract the **variables ranking** for the dataset.

[24]:
```
# create a dictionary of shapes of the features
features_shapes = {}

# scalars
for feature in scalar_feat:
    features_shapes[feature] = 1

# clustering
features_shapes['clustering'] = df_feat['clustering'].apply(np.shape).
↪unique()[0][0] # unique() returns an array, thus we need to take only one␣
↪element (the first)

rnd_for_features = np.c_[df_feat[scalar_feat].values,
                         ExtractTensor(flatten=True).
↪fit_transform(df_feat['clustering']),
                        ] # create features input

# vectors
for feature in vector_feat:
    features_shapes[feature] = df_feat[feature].apply(np.shape).unique()[0][0]
    rnd_for_features = np.c_[rnd_for_features, ExtractTensor(flatten=True).
↪fit_transform(df_feat[feature])]

# tensors
for feature in tensor_feat:
    features_shapes[feature] = df_feat[feature].apply(np.shape).unique()[0][0]␣
↪* df_feat[feature].apply(np.shape).unique()[0][1]
    rnd_for_features = np.c_[rnd_for_features, ExtractTensor(flatten=True).
↪fit_transform(df_feat[feature])]
```

```
# pca
features_shapes['matrix_pca99'] = df_feat['matrix_pca99'].apply(np.shape).
 ↪unique()[0][0]
rnd_for_features = np.c_[rnd_for_features, ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['matrix_pca99'])]
```

We then proceed with the decision trees:

[25]:
```
from sklearn.ensemble import RandomForestRegressor

# for the time being there is no need for optimization
rnd_for_param = {'criterion': 'mse',
                 'n_estimators': 50,
                 'n_jobs':       -1,
                 'random_state': RAND
                }

logprint('Computing random forest for h11...', logger=logger)
rnd_for_h11 = RandomForestRegressor(**rnd_for_param)
rnd_for_h11.fit(rnd_for_features, df_labs['h11'])
logprint('Random forest for h11 completed!', logger=logger)
logprint('Accuracy of the random forest for h11: {:.3f}%'.
 ↪format(accuracy_score(df_labs['h11'].values, rnd_for_h11.
 ↪predict(rnd_for_features), rounding=np.floor)*100), logger=logger)

logprint('Computing random forest for h21...', logger=logger)
rnd_for_h21 = RandomForestRegressor(**rnd_for_param)
rnd_for_h21.fit(rnd_for_features, df_labs['h21'])
logprint('Random forest for h21 completed!', logger=logger)
logprint('Accuracy of the random forest for h21: {:.3f}%'.
 ↪format(accuracy_score(df_labs['h21'].values, rnd_for_h21.
 ↪predict(rnd_for_features), rounding=np.floor)*100), logger=logger)
```

```
2020-04-09 11:52:10,103 --> INFO: Computing random forest for h11…
2020-04-09 11:52:18,369 --> INFO: Random forest for h11 completed!
2020-04-09 11:52:18,500 --> INFO: Accuracy of the random forest for h11: 72.297%
2020-04-09 11:52:18,501 --> INFO: Computing random forest for h21…
2020-04-09 11:52:26,352 --> INFO: Random forest for h21 completed!
2020-04-09 11:52:26,497 --> INFO: Accuracy of the random forest for h21: 41.689%
```

We then consider the features importance and plot them for comparison.

[26]:
```
# create a list with all the components
extended_features = []
for feature in features_shapes:
    extended_features.append(feature)
    for _ in range(features_shapes[feature] - 1):
        extended_features.append('')

# list of feature importances
feat_imp_h11 = list(zip(extended_features, rnd_for_h11.feature_importances_))
feat_imp_h21 = list(zip(extended_features, rnd_for_h21.feature_importances_))
```

Consider the scalar features first:

```
[27]: xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

label_plot(ax, feat_imp_h11[0:np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])], title='Scalar Features', ylabel='Variable Ranking',␣
 ↪legend='$h_{11}$')
label_plot(ax, feat_imp_h21[0:np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])], title='Scalar Features', ylabel='Variable Ranking',␣
 ↪legend='$h_{21}$')

save_fig('feat_imp_scalars', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:26,871 --> INFO: Saving ./img/original/feat_imp_scalars.png…
2020-04-09 11:52:26,972 --> INFO: Saved ./img/original/feat_imp_scalars.png!
```

Then consider vector features and clustering:

```
[28]: xplots  = 2
      yplots  = 1
      fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
       ↪yplots*mpl_height))
      fig.tight_layout()

      label_plot(ax[0], feat_imp_h11[np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     :np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     +np.sum([features_shapes[feature] for feature in␣
       ↪vector_feat])], title='Vector Features (per component)', ylabel='Variable␣
       ↪Ranking', legend='$h_{11}$')
```

```
label_plot(ax[0], feat_imp_h21[np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              +features_shapes['clustering']
                              :np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              +features_shapes['clustering']
                              +np.sum([features_shapes[feature] for feature in␣
 ↪vector_feat])], title='Vector Features (per component)', ylabel='Variable␣
 ↪Ranking', legend='$h_{21}$')

label_plot(ax[1], feat_imp_h11[np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              :np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              +features_shapes['clustering']], title='Cluster␣
 ↪Features', ylabel='Variable Ranking', legend='$h_{11}$')
label_plot(ax[1], feat_imp_h21[np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              :np.sum([features_shapes[feature] for feature in␣
 ↪scalar_feat])
                              +features_shapes['clustering']], title='Cluster␣
 ↪Features', ylabel='Variable Ranking', legend='$h_{21}$')

save_fig('feat_imp_vectors_clustering', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:27,237 --> INFO: Saving
./img/original/feat_imp_vectors_clustering.png…
2020-04-09 11:52:27,463 --> INFO: Saved
./img/original/feat_imp_vectors_clustering.png!
```

Notice that even the combined sum of the clusters, does not reach a sensible threshold to be considered:

```
[29]: logprint('Cluster importance for h11: {:.3f}%'.format(np.sum([f[1] for f in␣
       ↪feat_imp_h11[np.sum([features_shapes[feature] for feature in scalar_feat]):
       ↪np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])+features_shapes['clustering']]])*100), logger=logger)
      logprint('Cluster importance for h21: {:.3f}%'.format(np.sum([f[1] for f in␣
       ↪feat_imp_h21[np.sum([features_shapes[feature] for feature in scalar_feat]):
       ↪np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])+features_shapes['clustering']]])*100), logger=logger)
```

```
2020-04-09 11:52:27,769 --> INFO: Cluster importance for h11: 0.858%
2020-04-09 11:52:27,770 --> INFO: Cluster importance for h21: 0.423%
```

Then consider the matrix and PCA:

```
[30]: xplots  = 2
      yplots  = 1
      fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
       ↪yplots*mpl_height))
      fig.tight_layout()

      label_plot(ax[0], feat_imp_h11[np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     +np.sum([features_shapes[feature] for feature in␣
       ↪vector_feat])
                                     :np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     +np.sum([features_shapes[feature] for feature in␣
       ↪vector_feat])
                                     +features_shapes['matrix']], title='Matrix␣
       ↪Features', ylabel='Variable Ranking (per component)', legend='$h_{11}$',␣
       ↪binstep=10)
      label_plot(ax[0], feat_imp_h21[np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     +np.sum([features_shapes[feature] for feature in␣
       ↪vector_feat])
                                     :np.sum([features_shapes[feature] for feature in␣
       ↪scalar_feat])
                                     +features_shapes['clustering']
                                     +np.sum([features_shapes[feature] for feature in␣
       ↪vector_feat])
```

```
                               +features_shapes['matrix']], title='Matrix␣
↪Features', ylabel='Variable Ranking (per component)', legend='$h_{21}$',␣
↪binstep=10)

label_plot(ax[1], feat_imp_h11[np.sum([features_shapes[feature] for feature in␣
↪scalar_feat])
                               +features_shapes['clustering']
                               +np.sum([features_shapes[feature] for feature in␣
↪vector_feat])
                               +features_shapes['matrix']
                               :], title='PCA Features', ylabel='Variable␣
↪Ranking', legend='$h_{11}$', binstep=10)
label_plot(ax[1], feat_imp_h21[np.sum([features_shapes[feature] for feature in␣
↪scalar_feat])
                               +features_shapes['clustering']
                               +np.sum([features_shapes[feature] for feature in␣
↪vector_feat])
                               +features_shapes['matrix']
                               :], title='PCA Features', ylabel='Variable␣
↪Ranking', legend='$h_{21}$', binstep=10)

save_fig('feat_imp_matrix_pca', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:28,135 --> INFO: Saving
./img/original/feat_imp_matrix_pca.png…
2020-04-09 11:52:28,269 --> INFO: Saved ./img/original/feat_imp_matrix_pca.png!
```

For comparison, we can plot the separate sum of each vector and tensor feature:

[31]:
```
# sum of the scalars
scalar_h11_sum = np.sum([f[1] for f in feat_imp_h11[0:np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])]])
scalar_h21_sum = np.sum([f[1] for f in feat_imp_h21[0:np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])]])

# sum of dim_cp importances
dim_cp_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                          ␣
 ↪+features_shapes['clustering']
                                                                           :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                          ␣
 ↪+features_shapes['clustering']
                                                                          ␣
 ↪+features_shapes['dim_cp']]])
dim_cp_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                          ␣
 ↪+features_shapes['clustering']
                                                                           :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                          ␣
 ↪+features_shapes['clustering']
                                                                          ␣
 ↪+features_shapes['dim_cp']]])

# sum of num_dim_cp importances
num_dim_cp_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                              ␣
 ↪+features_shapes['clustering']
                                                                              ␣
 ↪+features_shapes['dim_cp']
                                                                               :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                              ␣
 ↪+features_shapes['clustering']
                                                                              ␣
 ↪+features_shapes['dim_cp']
                                                                              ␣
 ↪+features_shapes['num_dim_cp']]])
num_dim_cp_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
```

```
                                                                      ␣
 ↪+features_shapes['clustering']
                                                                      ␣
 ↪+features_shapes['dim_cp']
                                                                       :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                      ␣
 ↪+features_shapes['clustering']
                                                                      ␣
 ↪+features_shapes['dim_cp']
                                                                      ␣
 ↪+features_shapes['num_dim_cp']]])

# sum of deg_eqs importances
deg_eqs_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                   ␣
 ↪+features_shapes['clustering']
                                                                   ␣
 ↪+features_shapes['dim_cp']
                                                                   ␣
 ↪+features_shapes['num_dim_cp']
                                                                    :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                   ␣
 ↪+features_shapes['clustering']
                                                                   ␣
 ↪+features_shapes['dim_cp']
                                                                   ␣
 ↪+features_shapes['num_dim_cp']
                                                                   ␣
 ↪+features_shapes['deg_eqs']]])
deg_eqs_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                   ␣
 ↪+features_shapes['clustering']
                                                                   ␣
 ↪+features_shapes['dim_cp']
                                                                   ␣
 ↪+features_shapes['num_dim_cp']
                                                                    :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                   ␣
 ↪+features_shapes['clustering']
```

```
                                                                        ␣
↪+features_shapes['dim_cp']
                                                                        ␣
↪+features_shapes['num_dim_cp']
                                                                        ␣
↪+features_shapes['deg_eqs']]]])

# sum of num_deg_eqs importances
num_deg_eqs_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                            ␣
↪+features_shapes['clustering']
                                                                            ␣
↪+features_shapes['dim_cp']
                                                                            ␣
↪+features_shapes['num_dim_cp']
                                                                            ␣
↪+features_shapes['deg_eqs']
                                                                             :np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                            ␣
↪+features_shapes['clustering']
                                                                            ␣
↪+features_shapes['dim_cp']
                                                                            ␣
↪+features_shapes['num_dim_cp']
                                                                            ␣
↪+features_shapes['deg_eqs']
                                                                            ␣
↪+features_shapes['num_deg_eqs']]]])
num_deg_eqs_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                            ␣
↪+features_shapes['clustering']
                                                                            ␣
↪+features_shapes['dim_cp']
                                                                            ␣
↪+features_shapes['num_dim_cp']
                                                                            ␣
↪+features_shapes['deg_eqs']
                                                                             :np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                            ␣
↪+features_shapes['clustering']
```

```
                                                                              ␣
 ↪+features_shapes['dim_cp']
                                                                              ␣
 ↪+features_shapes['num_dim_cp']
                                                                              ␣
 ↪+features_shapes['deg_eqs']
                                                                              ␣
 ↪+features_shapes['num_deg_eqs']]]])

# sum of dim_h0_amb importances
dim_h0_amb_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                             ␣
 ↪+features_shapes['clustering']
                                                                             ␣
 ↪+features_shapes['dim_cp']
                                                                             ␣
 ↪+features_shapes['num_dim_cp']
                                                                             ␣
 ↪+features_shapes['deg_eqs']
                                                                             ␣
 ↪+features_shapes['num_deg_eqs']
                                                                              :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                             ␣
 ↪+features_shapes['clustering']
                                                                             ␣
 ↪+features_shapes['dim_cp']
                                                                             ␣
 ↪+features_shapes['num_dim_cp']
                                                                             ␣
 ↪+features_shapes['deg_eqs']
                                                                             ␣
 ↪+features_shapes['num_deg_eqs']
                                                                             ␣
 ↪+features_shapes['dim_h0_amb']]]])
dim_h0_amb_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                             ␣
 ↪+features_shapes['clustering']
                                                                             ␣
 ↪+features_shapes['dim_cp']
                                                                             ␣
 ↪+features_shapes['num_dim_cp']
```

```
                                                                               ␣
 ↪+features_shapes['deg_eqs']
                                                                               ␣
 ↪+features_shapes['num_deg_eqs']
                                                                                :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                               ␣
 ↪+features_shapes['clustering']
                                                                               ␣
 ↪+features_shapes['dim_cp']
                                                                               ␣
 ↪+features_shapes['num_dim_cp']
                                                                               ␣
 ↪+features_shapes['deg_eqs']
                                                                               ␣
 ↪+features_shapes['num_deg_eqs']
                                                                               ␣
 ↪+features_shapes['dim_h0_amb']]])

# sum of the vector importances
vector_h11_sum = dim_cp_h11_sum + num_dim_cp_h11_sum + deg_eqs_h11_sum +␣
 ↪num_deg_eqs_h11_sum + dim_h0_amb_h11_sum
vector_h21_sum = dim_cp_h21_sum + num_dim_cp_h21_sum + deg_eqs_h21_sum +␣
 ↪num_deg_eqs_h21_sum + dim_h0_amb_h21_sum

# sum of the importance of the matrix components
matrix_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                           ␣
 ↪+features_shapes['clustering']
                                                                            +np.
 ↪sum([features_shapes[feature] for feature in vector_feat])
                                                                            :np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                           ␣
 ↪+features_shapes['clustering']
                                                                            +np.
 ↪sum([features_shapes[feature] for feature in vector_feat])
                                                                           ␣
 ↪+features_shapes['matrix']]])
matrix_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
 ↪sum([features_shapes[feature] for feature in scalar_feat])
                                                                           ␣
 ↪+features_shapes['clustering']
```

```
                                                            +np.
↪sum([features_shapes[feature] for feature in vector_feat])
                                                            :np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                           ␣
↪+features_shapes['clustering']
                                                            +np.
↪sum([features_shapes[feature] for feature in vector_feat])
                                                           ␣
↪+features_shapes['matrix']]]])

# sum of the importance of the PCA components
pca_h11_sum = np.sum([f[1] for f in feat_imp_h11[np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                        ␣
↪+features_shapes['clustering']
                                                         +np.
↪sum([features_shapes[feature] for feature in vector_feat])
                                                        ␣
↪+features_shapes['matrix']
                                                         :]])
pca_h21_sum = np.sum([f[1] for f in feat_imp_h21[np.
↪sum([features_shapes[feature] for feature in scalar_feat])
                                                        ␣
↪+features_shapes['clustering']
                                                         +np.
↪sum([features_shapes[feature] for feature in vector_feat])
                                                        ␣
↪+features_shapes['matrix']
                                                         :]])

# plot the sum of vector and tensor features
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
↪yplots*mpl_height))
fig.tight_layout()

label_plot(ax,
           [('dim_cp',       dim_cp_h11_sum),
            ('num_dim_cp',   num_dim_cp_h11_sum),
            ('deg_eqs',      deg_eqs_h11_sum),
            ('num_deg_eqs',  num_deg_eqs_h11_sum),
            ('dim_h0_amb',   dim_h0_amb_h11_sum),
            ('matrix',       matrix_h11_sum),
            ('matrix_pca99', pca_h11_sum)
           ],
           title='Vector and Tensor Features (sum of the components)',
           ylabel='Variable Ranking',
           legend='$h_{11}$'
          )
label_plot(ax,
           [('dim_cp',       dim_cp_h21_sum),
            ('num_dim_cp',   num_dim_cp_h21_sum),
            ('deg_eqs',      deg_eqs_h21_sum),
            ('num_deg_eqs',  num_deg_eqs_h21_sum),
            ('dim_h0_amb',   dim_h0_amb_h21_sum),
            ('matrix',       matrix_h21_sum),
            ('matrix_pca99', pca_h21_sum)
           ],
           title='Vector and Tensor Features (sum of the components)',
           ylabel='Variable Ranking',
           legend='$h_{21}$'
          )

save_fig('feat_imp_vector_tensor_sum', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:28,599 --> INFO: Saving
./img/original/feat_imp_vector_tensor_sum.png…
2020-04-09 11:52:28,685 --> INFO: Saved
./img/original/feat_imp_vector_tensor_sum.png!
```

For better visualisation, we can also plot the sum of scalar, vector and tensor features:

```
[32]: # plot the sum of vector and tensor features
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

label_plot(ax,
           [('scalars',      scalar_h11_sum),
            ('vectors',      vector_h11_sum),
            ('matrix',       matrix_h11_sum),
            ('matrix_pca99', pca_h11_sum)
           ],
           title='Sum of the Features (sum of the components)',
           ylabel='Variable Ranking',
           legend='$h_{11}$'
```

```
          )
label_plot(ax,
           [('scalars',      scalar_h21_sum),
            ('vectors',      vector_h21_sum),
            ('matrix',       matrix_h21_sum),
            ('matrix_pca99', pca_h21_sum)
           ],
           title='Sum of the Features (sum of the components)',
           ylabel='Variable Ranking',
           legend='$h_{21}$'
          )

save_fig('feat_imp_sum', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-09 11:52:28,867 --> INFO: Saving ./img/original/feat_imp_sum.png…
2020-04-09 11:52:28,948 --> INFO: Saved ./img/original/feat_imp_sum.png!
```

Given the previous results, we select only those features which where able to reach at least the threshold of 5% in the feature importances. The choice has to be absolute as not to introduce arbitrariness in the computations. We therefore select:

- `num_cp`, `dim_cp`, `matrix_pca99` for h_{11}
- `num_cp`, `dim_cp`, `dim_h0_amb`, `matrix_pca99` for h_{21}

We will however keep a copy of the `matrix` (not flattened) to feed the neural networks later on.

[33]:
```
# save the dataset to file
logprint('Saving engineered dataset to file...', logger=logger)
df_labs.join(df_feat[['num_cp', 'dim_cp', 'dim_h0_amb', 'matrix_pca99']]).
 ↪to_hdf(path.join(ROOT_DIR, DB_NAME + '_eng.h5'), key=DB_NAME + '_eng',␣
 ↪complevel=9, complib='bzip2')
logprint('Engineered dataset saved to file!', logger=logger)

logprint('Saving matrix to file...', logger=logger)
df_labs.join(df_feat['matrix']).to_hdf(path.join(ROOT_DIR, DB_NAME + '_matrix.
 ↪h5'), key=DB_NAME + '_matrix', complevel=9, complib='bzip2')
logprint('Matrix saved to file!', logger=logger)
```

```
2020-04-09 11:52:29,046 --> INFO: Saving engineered dataset to file…
2020-04-09 11:52:29,168 --> INFO: Engineered dataset saved to file!
2020-04-09 11:52:29,169 --> INFO: Saving matrix to file…
2020-04-09 11:52:29,232 --> INFO: Matrix saved to file!
```

```
/home/riccardo/anaconda/envs/ml/lib/python3.7/site-
packages/pandas/core/generic.py:2505: PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block2_values]
[items->Index(['dim_cp', 'dim_h0_amb', 'matrix_pca99'], dtype='object')]

  encoding=encoding,
/home/riccardo/anaconda/envs/ml/lib/python3.7/site-
packages/pandas/core/generic.py:2505: PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed,key->block1_values]
[items->Index(['matrix'], dtype='object')]

  encoding=encoding,
```

1.6 Machine Learning Analysis

We now apply several ML algorithms to the engineered dataset and try to get valuable predictions on the labels. We first reload the datasets in order for this section to be independent on the previous one and then proceed to decide the cross-validation and evaluation strategy.

[12]:
```
# reload the dataset
DF_ENG_PATH = path.join(ROOT_DIR, DB_NAME + '_eng.h5')
```

```
DF_MAT_PATH = path.join(ROOT_DIR, DB_NAME + '_matrix.h5')

# load featured engineered set
if path.isfile(DF_ENG_PATH):
    df = load_dataset(DF_ENG_PATH, logger=logger)

    # extract labels and features
    df_labs = df[['h11', 'h21']]
    df_feat = df.drop(labels=['h11', 'h21'], axis=1)

    # divide features according to training sets
    logprint('Extracting features from the engineered database...',␣
 ↪logger=logger)
    num_cp       = ExtractTensor(flatten=True).fit_transform(df_feat['num_cp'])
    dim_cp       = ExtractTensor(flatten=True).fit_transform(df_feat['dim_cp'])
    features_h11 = np.c_[ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['num_cp']),
                         ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['dim_cp']),
                         ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['matrix_pca99'])
                        ]
    features_h21 = np.c_[ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['num_cp']),
                         ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['dim_cp']),
                         ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['dim_h0_amb']),
                         ExtractTensor(flatten=True).
 ↪fit_transform(df_feat['matrix_pca99'])
                        ]
    logprint('Features extracted!', logger=logger)
else:
    logprint('File is not present: cannot read engineered database!',␣
 ↪stream='error', logger=logger)

# load matrix
if path.isfile(DF_MAT_PATH):
    df = load_dataset(DF_MAT_PATH, logger=logger)

    # extract labels and features
    df_labs = df[['h11', 'h21']]
    df_feat = df.drop(labels=['h11', 'h21'], axis=1)

    # divide features according to training sets
    logprint('Extracting features from the matrix database...', logger=logger)
    matrix = ExtractTensor(flatten=True).fit_transform(df_feat['matrix'])
    logprint('Features extracted!', logger=logger)
else:
    logprint('File is not present: cannot read matrix database!',␣
 ↪stream='error', logger=logger)
```

```
2020-04-11 22:22:43,492 --> INFO: Reading database…
2020-04-11 22:22:43,650 --> INFO: Database loaded!
2020-04-11 22:22:43,655 --> INFO: Extracting features from the engineered
database…
2020-04-11 22:22:45,306 --> INFO: Features extracted!
2020-04-11 22:22:45,306 --> INFO: Reading database…
2020-04-11 22:22:45,392 --> INFO: Database loaded!
2020-04-11 22:22:45,403 --> INFO: Extracting features from the matrix
database…
2020-04-11 22:22:45,785 --> INFO: Features extracted!
```

Then build a training and a test sets and define a cross-validation strategy. We use 10% of the dataset as *test set*, while the remaining 90% will be trated as *training set*. We implement **cross-validation** with a `KFold` splitting strategy: we split the training set into 9 folds and cross-evaluate the training algorithms on each one of them. For each fold the training is therefore performed on 80% of the total dataset (including the test set), while the validation uses 10% of it.

```
[13]: from sklearn.model_selection import KFold, train_test_split
from sklearn.metrics         import make_scorer
from sklearn.model_selection import GridSearchCV, cross_val_score
from skopt                   import gp_minimize
from skopt.plots             import plot_convergence
from skopt.space             import Categorical, Integer, Real
from skopt.utils             import use_named_args

# split into training and test sets
features_h11_train, features_h11_test, \
features_h21_train, features_h21_test, \
num_cp_train, num_cp_test, \
dim_cp_train, dim_cp_test, \
matrix_train, matrix_test, \
h11_train, h11_test, \
h21_train, h21_test = train_test_split(features_h11,
                                       features_h21,
                                       num_cp,
                                       dim_cp,
                                       matrix,
                                       df_labs['h11'].values,
                                       df_labs['h21'].values,
                                       test_size=0.1,        # keep 10% as␣
 ↪test set
```

```
                                        shuffle=False          # for consistency␣
↪with other datasets: it has already been shuffled!
                                       )
# cross-validation strategy
cv     = KFold(n_splits=9, shuffle=False)
scorer = lambda x: make_scorer(accuracy_score,
                               greater_is_better=True,
                               rounding=x)             # create a scoring␣
↪function for GridSearchCV and RandomizedSearchCV (and skopt but with its own␣
↪interface)

# hyperparameter optimization iterations
n_iter = 100
```

At the end of each training we will then save the models trained on the feature engineered sets.

[14]:
```
import joblib

# save the models
def save_model(filename, estimator):
    """
    Save trained models to file.

    Required arguments:
        filename:  the name of the file (w/o extension)
        estimator: the model to save
    """

    MOD_FILE = path.join(MOD_PATH, filename + '.joblib.xz')

    logprint('Saving the estimator to {}.joblib.xz...'.format(filename),␣
↪logger=logger)
    joblib.dump(estimator, MOD_FILE, compress=('xz',9))
    logprint('Saved {}.joblib.xz!'.format(filename), logger=logger)
```

1.6.1 Linear Regression

We start with simplest algorithm. We use a **linear regression** with multiple features. Since the number of tunable parameters is limited, we implement a `GridSearchCV` evaluation.

[37]:
```
from sklearn.linear_model import LinearRegression
logprint('Evaluating Linear Regression...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = {'fit_intercept': [ True, False ],
                 'normalize':     [ True, False ]
```

```
                }
rounding = np.floor

#------------------------------
# MATRIX BASELINE
#------------------------------

# define the GridSearchCV strategy for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
lin_reg_h11 = GridSearchCV(LinearRegression(),
                           search_params,
                           scoring=scorer(rounding),
                           n_jobs=-1,
                           refit=True,
                           cv=cv)

# fit the algorithm
lin_reg_h11.fit(matrix_train, h11_train)

# read scores and predictions
gridcv_score(lin_reg_h11, rounding=rounding, logger=logger)
prediction_score(lin_reg_h11, matrix_test, h11_test, use_best_estimator=True,␣
↪rounding=rounding, logger=logger)

# define the GridSearcCV strategy for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
lin_reg_h21 = GridSearchCV(LinearRegression(),
                           search_params,
                           scoring=scorer(rounding),
                           n_jobs=-1,
                           refit=True,
                           cv=cv)

# fit the algorithm
lin_reg_h21.fit(matrix_train, h21_train)

# read scores and predictions
gridcv_score(lin_reg_h21, rounding=rounding, logger=logger)
prediction_score(lin_reg_h21, matrix_test, h21_test, use_best_estimator=True,␣
↪rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define GridSearchCV strategy for h11
logprint('Fitting the feature engineered dataset for h11...', logger=logger)
```

```
lin_reg_h11 = GridSearchCV(LinearRegression(),
                           search_params,
                           scoring=scorer(rounding),
                           n_jobs=-1,
                           refit=True,
                           cv=cv)

# fit the algorithm
lin_reg_h11.fit(features_h11_train, h11_train)

# read scores and predictions
gridcv_score(lin_reg_h11, rounding=rounding, logger=logger)
prediction_score(lin_reg_h11, features_h11_test, h11_test,␣
 ↪use_best_estimator=True, rounding=rounding, logger=logger)

# define GridSearchCV strategy for h21
logprint('Fitting the feature engineered dataset for h21...', logger=logger)
lin_reg_h21 = GridSearchCV(LinearRegression(),
                           search_params,
                           scoring=scorer(rounding),
                           n_jobs=-1,
                           refit=True,
                           cv=cv)

# fit the algorithm
lin_reg_h21.fit(features_h21_train, h21_train)

# read scores and predictions
gridcv_score(lin_reg_h21, rounding=rounding, logger=logger)
prediction_score(lin_reg_h21, features_h21_test, h21_test,␣
 ↪use_best_estimator=True, rounding=rounding, logger=logger)

# plot the error difference of the feature engineered dataset
logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      lin_reg_h11.best_estimator_.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
```

```
count_plot(ax,
           error_diff(h21_test,
                      lin_reg_h21.best_estimator_.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')

save_fig('lin_reg_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('lin_reg_h11', lin_reg_h11.best_estimator_)
save_model('lin_reg_h21', lin_reg_h21.best_estimator_)
```

```
2020-04-09 11:52:31,874 --> INFO: Evaluating Linear Regression…
2020-04-09 11:52:31,875 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 11:52:39,809 --> INFO: Best parameters: {'fit_intercept': True,
'normalize': True}
2020-04-09 11:52:39,810 --> INFO: Accuracy (floor) of cross-validation: (50.531
± 1.045)%
2020-04-09 11:52:39,817 --> INFO: Accuracy (floor) of the predictions: 51.399%
2020-04-09 11:52:39,818 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 11:52:47,942 --> INFO: Best parameters: {'fit_intercept': True,
'normalize': True}
2020-04-09 11:52:47,943 --> INFO: Accuracy (floor) of cross-validation: (10.927
± 0.780)%
2020-04-09 11:52:47,950 --> INFO: Accuracy (floor) of the predictions: 11.196%
2020-04-09 11:52:47,951 --> INFO: Fitting the feature engineered dataset for
h11…
2020-04-09 11:52:49,158 --> INFO: Best parameters: {'fit_intercept': True,
'normalize': True}
2020-04-09 11:52:49,159 --> INFO: Accuracy (floor) of cross-validation: (52.555
± 1.253)%
2020-04-09 11:52:49,170 --> INFO: Accuracy (floor) of the predictions: 50.891%
2020-04-09 11:52:49,171 --> INFO: Fitting the feature engineered dataset for
h21…
2020-04-09 11:52:50,942 --> INFO: Best parameters: {'fit_intercept': True,
'normalize': True}
2020-04-09 11:52:50,951 --> INFO: Accuracy (floor) of cross-validation: (19.887
± 1.065)%
2020-04-09 11:52:50,963 --> INFO: Accuracy (floor) of the predictions: 17.939%
2020-04-09 11:52:50,965 --> INFO: Plotting error distribution…
2020-04-09 11:52:51,146 --> INFO: Saving ./img/original/lin_reg_error_eng.png…
2020-04-09 11:52:51,356 --> INFO: Saved ./img/original/lin_reg_error_eng.png!
```

```
2020-04-09 11:52:51,535 --> INFO: Saving the estimator to
lin_reg_h11.joblib.xz…
2020-04-09 11:52:51,573 --> INFO: Saved lin_reg_h11.joblib.xz!
2020-04-09 11:52:51,574 --> INFO: Saving the estimator to
lin_reg_h21.joblib.xz…
2020-04-09 11:52:51,614 --> INFO: Saved lin_reg_h21.joblib.xz!
```

1.6.2 Lasso

We then move to a different algorithm which applies **L1** regularisation to a linear model. We implement the Bayesan search strategy for the optimal parameters.

[38]:
```
from sklearn.linear_model import Lasso
logprint('Evaluating Lasso...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = [Real(1.0e-6, 5.0e-1, base=10, prior='log-uniform',␣
↪name='alpha'),
                Integer(False, True,                                       ␣
 ↪name='fit_intercept'),
                Integer(False, True,                                       ␣
 ↪name='normalize'),
                Integer(False, True,                                       ␣
 ↪name='positive')
               ]
rounding = np.floor

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
lasso_h11 = Lasso(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    lasso_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lasso_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
lasso_h11_res = gp_minimize(objective_mat_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lasso_h11_res.x[n] for n in␣
 ↪range(len(lasso_h11_res.x))}
lasso_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(lasso_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h11)*100.0,
                                                                                                    np.
 ↪std(cross_score_h11)*100.0
                                                                                                   ),
        logger=logger
        )

# compute the accuracy of the predictions
lasso_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=lasso_h11, X=matrix_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
lasso_h21 = Lasso(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    lasso_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lasso_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
lasso_h21_res = gp_minimize(objective_mat_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lasso_h21_res.x[n] for n in␣
 ↪range(len(lasso_h21_res.x))}
lasso_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(lasso_h21, matrix_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                                                    np.
 ↪mean(cross_score_h21)*100.0,
                                                                                                    np.
 ↪std(cross_score_h21)*100.0
                                                                                                   ),
         logger=logger
        )

# compute the accuracy of the predictions
lasso_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=lasso_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
lasso_h11 = Lasso(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    lasso_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lasso_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
lasso_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lasso_h11_res.x[n] for n in␣
 ↪range(len(lasso_h11_res.x))}
lasso_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(lasso_h11, features_h11_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                       np.
 ↪mean(cross_score_h11)*100.0,
                                                                       np.
 ↪std(cross_score_h11)*100.0
```

```
                                                                          ),
         logger=logger
        )

# compute the accuracy of the predictions
lasso_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=lasso_h11, X=features_h11_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
lasso_h21 = Lasso(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    lasso_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lasso_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
lasso_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lasso_h21_res.x[n] for n in␣
 ↪range(len(lasso_h21_res.x))}
lasso_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(lasso_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                           np.
 ↪mean(cross_score_h21)*100.0,
                                                                           np.
 ↪std(cross_score_h21)*100.0
                                                                          ),
         logger=logger
        )
```

```
# compute the accuracy of the predictions
lasso_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=lasso_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(lasso_h11_res, ax=ax[0])
plot_convergence(lasso_h21_res, ax=ax[1])

save_fig('lasso_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      lasso_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      lasso_h21.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')

save_fig('lasso_error_eng', logger=logger)
plt.show()
```

```python
plt.close(fig)

# Saving the feature engineered models
save_model('lasso_h11', lasso_h11)
save_model('lasso_h21', lasso_h21)
```

```
2020-04-09 11:52:51,652 --> INFO: Evaluating Lasso…
2020-04-09 11:52:51,657 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 11:58:37,327 --> INFO: Best parameters: {'alpha':
6.537447325215235e-05, 'fit_intercept': 1, 'normalize': 1, 'positive': 1}
2020-04-09 11:58:39,167 --> INFO: Accuracy (floor) of the cross-validation:
51.352% ± 1.211%
2020-04-09 11:58:39,310 --> INFO: Accuracy (floor) of the predictions: 50.509%
2020-04-09 11:58:39,311 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 12:12:43,321 --> INFO: Best parameters: {'alpha':
0.001005300150246659, 'fit_intercept': 1, 'normalize': 0, 'positive': 0}
2020-04-09 12:12:45,628 --> INFO: Accuracy (floor) of the cross-validation:
11.168% ± 0.534%
2020-04-09 12:12:45,833 --> INFO: Accuracy (floor) of the predictions: 10.305%
2020-04-09 12:12:45,834 --> INFO: Fitting the feature engineered set for h11…
2020-04-09 12:15:09,505 --> INFO: Best parameters: {'alpha':
0.07465180326075849, 'fit_intercept': 0, 'normalize': 1, 'positive': 0}
2020-04-09 12:15:09,701 --> INFO: Accuracy (floor) of the cross-validation:
63.765% ± 1.529%
2020-04-09 12:15:09,719 --> INFO: Accuracy (floor) of the predictions: 63.613%
2020-04-09 12:15:09,720 --> INFO: Fitting the feature engineered set for h21…
2020-04-09 12:27:50,367 --> INFO: Best parameters: {'alpha':
1.8441967978871898e-05, 'fit_intercept': 1, 'normalize': 1, 'positive': 0}
2020-04-09 12:27:54,078 --> INFO: Accuracy (floor) of the cross-validation:
20.226% ± 0.715%
2020-04-09 12:27:55,257 --> INFO: Accuracy (floor) of the predictions: 17.430%
2020-04-09 12:27:55,258 --> INFO: Plotting convergence progression…
2020-04-09 12:27:55,466 --> INFO: Saving
./img/original/lasso_conv_prog_eng.png…
2020-04-09 12:27:55,602 --> INFO: Saved ./img/original/lasso_conv_prog_eng.png!
```

```
2020-04-09 12:27:55,781 --> INFO: Plotting error distribution…
2020-04-09 12:27:55,869 --> INFO: Saving ./img/original/lasso_error_eng.png…
2020-04-09 12:27:55,976 --> INFO: Saved ./img/original/lasso_error_eng.png!
```

```
2020-04-09 12:27:56,084 --> INFO: Saving the estimator to lasso_h11.joblib.xz…
2020-04-09 12:27:56,123 --> INFO: Saved lasso_h11.joblib.xz!
2020-04-09 12:27:56,124 --> INFO: Saving the estimator to lasso_h21.joblib.xz…
2020-04-09 12:27:56,171 --> INFO: Saved lasso_h21.joblib.xz!
```

1.6.3 Ridge

We consider a different algorithm which applies **L2** regularisation to a linear model. We implement the Bayesan search strategy for the optimal parameters as in the previous case.

[39]:
```
from sklearn.linear_model import Ridge
logprint('Evaluating Ridge...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = [Real(1.0e-3, 1.0e3, base=10, prior='log-uniform',␣
↪name='alpha'),
                 Integer(False, True,                            ␣
↪name='fit_intercept'),
                 Integer(False, True,                            ␣
↪name='normalize')
                ]
rounding = np.floor

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
ridge_h11 = Ridge(tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    ridge_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(ridge_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
ridge_h11_res = gp_minimize(objective_mat_h11, search_params, n_calls=n_iter,␣
↪random_state=RAND)
```

```
# return the best parameters and print them
best_params = {search_params[n].name: ridge_h11_res.x[n] for n in␣
 ↪range(len(ridge_h11_res.x))}
ridge_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(ridge_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                           np.
 ↪mean(cross_score_h11)*100.0,
                                                                           np.
 ↪std(cross_score_h11)*100.0
                                                                          ),
         logger=logger
        )

# compute the accuracy of the predictions
ridge_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=ridge_h11, X=matrix_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
ridge_h21 = Ridge(tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    ridge_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(ridge_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
ridge_h21_res = gp_minimize(objective_mat_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
```

```
best_params = {search_params[n].name: ridge_h21_res.x[n] for n in␣
 ↪range(len(ridge_h21_res.x))}
ridge_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(ridge_h21, matrix_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h21)*100.0,
                                                                          np.
 ↪std(cross_score_h21)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
ridge_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=ridge_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
ridge_h11 = Ridge(tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    ridge_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(ridge_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
ridge_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)
```

```
# return the best parameters and print them
best_params = {search_params[n].name: ridge_h11_res.x[n] for n in␣
↪range(len(ridge_h11_res.x))}
ridge_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(ridge_h11, features_h11_train, h11_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
↪format(rounding.__name__,
                                                                            np.
↪mean(cross_score_h11)*100.0,
                                                                            np.
↪std(cross_score_h11)*100.0
                                                                           ),
         logger=logger
        )

# compute the accuracy of the predictions
ridge_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=ridge_h11, X=features_h11_test,␣
↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
ridge_h21 = Ridge(tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    ridge_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(ridge_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
ridge_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: ridge_h21_res.x[n] for n in␣
↪range(len(ridge_h21_res.x))}
```

```
ridge_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(ridge_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h21)*100.0,
                                                                          np.
 ↪std(cross_score_h21)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
ridge_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=ridge_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(ridge_h11_res, ax=ax[0])
plot_convergence(ridge_h21_res, ax=ax[1])

save_fig('ridge_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
            error_diff(h11_test,
                        ridge_h11.predict(features_h11_test),
                        rounding=rounding),
            title='Error distribution on the test set',
            xlabel='Difference from real value',
            legend='$h_{11}$')
count_plot(ax,
            error_diff(h21_test,
                        ridge_h21.predict(features_h21_test),
                        rounding=rounding),
            title='Error distribution on the test set',
            xlabel='Difference from real value',
            legend='$h_{21}$')

save_fig('ridge_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('ridge_h11', ridge_h11)
save_model('ridge_h21', ridge_h21)
```

```
2020-04-09 12:27:56,216 --> INFO: Evaluating Ridge…
2020-04-09 12:27:56,219 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 12:32:13,758 --> INFO: Best parameters: {'alpha':
0.024433312339777795, 'fit_intercept': 1, 'normalize': 1}
2020-04-09 12:32:15,568 --> INFO: Accuracy (floor) of the cross-validation:
51.649% ± 1.323%
2020-04-09 12:32:15,596 --> INFO: Accuracy (floor) of the predictions: 51.527%
2020-04-09 12:32:15,597 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 12:36:14,557 --> INFO: Best parameters: {'alpha': 2.3386114231851183,
'fit_intercept': 1, 'normalize': 0}
2020-04-09 12:36:16,359 --> INFO: Accuracy (floor) of the cross-validation:
11.196% ± 0.596%
2020-04-09 12:36:16,391 --> INFO: Accuracy (floor) of the predictions: 10.305%
2020-04-09 12:36:16,392 --> INFO: Fitting the feature engineered set for h11…
2020-04-09 12:37:31,617 --> INFO: Best parameters: {'alpha': 987.3896902033662,
'fit_intercept': 0, 'normalize': 1}
2020-04-09 12:37:31,737 --> INFO: Accuracy (floor) of the cross-validation:
60.396% ± 0.770%
2020-04-09 12:37:31,746 --> INFO: Accuracy (floor) of the predictions: 58.142%
2020-04-09 12:37:31,747 --> INFO: Fitting the feature engineered set for h21…
2020-04-09 12:38:53,706 --> INFO: Best parameters: {'alpha':
0.0023883783005328252, 'fit_intercept': 1, 'normalize': 1}
2020-04-09 12:38:53,820 --> INFO: Accuracy (floor) of the cross-validation:
20.212% ± 1.024%
2020-04-09 12:38:53,833 --> INFO: Accuracy (floor) of the predictions: 16.921%
```

```
2020-04-09 12:38:53,835 --> INFO: Plotting convergence progression…
2020-04-09 12:38:53,992 --> INFO: Saving
./img/original/ridge_conv_prog_eng.png…
2020-04-09 12:38:54,125 --> INFO: Saved ./img/original/ridge_conv_prog_eng.png!
```

```
2020-04-09 12:38:54,294 --> INFO: Plotting error distribution…
2020-04-09 12:38:54,390 --> INFO: Saving ./img/original/ridge_error_eng.png…
2020-04-09 12:38:54,513 --> INFO: Saved ./img/original/ridge_error_eng.png!
```

```
2020-04-09 12:38:54,604 --> INFO: Saving the estimator to ridge_h11.joblib.xz…
2020-04-09 12:38:54,635 --> INFO: Saved ridge_h11.joblib.xz!
2020-04-09 12:38:54,636 --> INFO: Saving the estimator to ridge_h21.joblib.xz…
2020-04-09 12:38:54,669 --> INFO: Saved ridge_h21.joblib.xz!
```

1.6.4 Elastic Net

As a comparison, we also use a different algorithm which uses both **L1** and **L2** resularisations of a linear model. This way we can directly see the relevance of the two kind of approaches.

[40]:
```
from sklearn.linear_model import ElasticNet
logprint('Evaluating Elastic Net...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
 ↪accuracy
search_params = [Real(1.0e-7, 1.0e-1, base=10, prior='log-uniform',␣
 ↪name='alpha'),
                 Integer(False, True,                                  ␣
 ↪name='fit_intercept'),
                 Integer(False, True,                                    ␣
 ↪name='normalize')
                ]
rounding = np.floor

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
el_net_h11 = ElasticNet(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    el_net_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(el_net_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
el_net_h11_res = gp_minimize(objective_mat_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: el_net_h11_res.x[n] for n in␣
 ↪range(len(el_net_h11_res.x))}
el_net_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(el_net_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h11)*100.0,
                                                                          np.
 ↪std(cross_score_h11)*100.0
                                                                         ),
        logger=logger
       )
```

```
# compute the accuracy of the predictions
el_net_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=el_net_h11, X=matrix_test,␣
↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
el_net_h21 = ElasticNet(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    el_net_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(el_net_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
el_net_h21_res = gp_minimize(objective_mat_h21, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: el_net_h21_res.x[n] for n in␣
↪range(len(el_net_h21_res.x))}
el_net_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(el_net_h21, matrix_train, h21_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
↪format(rounding.__name__,
                                                                          np.
↪mean(cross_score_h21)*100.0,
                                                                          np.
↪std(cross_score_h21)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
el_net_h21.fit(matrix_train, h21_train)
```

```
preds_score_h21 = prediction_score(estimator=el_net_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#-----------------------------
# FEATURE ENGINEERED SET
#-----------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
el_net_h11 = ElasticNet(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    el_net_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(el_net_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
el_net_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: el_net_h11_res.x[n] for n in␣
 ↪range(len(el_net_h11_res.x))}
el_net_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(el_net_h11, features_h11_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                           np.
 ↪mean(cross_score_h11)*100.0,
                                                                           np.
 ↪std(cross_score_h11)*100.0
                                                                          ),
         logger=logger
        )

# compute the accuracy of the predictions
```

```
el_net_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=el_net_h11, X=features_h11_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
el_net_h21 = ElasticNet(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    el_net_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(el_net_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
el_net_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: el_net_h21_res.x[n] for n in␣
 ↪range(len(el_net_h21_res.x))}
el_net_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(el_net_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                           np.
 ↪mean(cross_score_h21)*100.0,
                                                                           np.
 ↪std(cross_score_h21)*100.0
                                                                          ),
         logger=logger
        )

# compute the accuracy of the predictions
el_net_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=el_net_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)
```

```
#-------------------------------
# PLOTS
#-------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(el_net_h11_res, ax=ax[0])
plot_convergence(el_net_h21_res, ax=ax[1])

save_fig('el_net_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      el_net_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      el_net_h21.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')

save_fig('el_net_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('el_net_h11', el_net_h11)
save_model('el_net_h21', el_net_h21)
```

```
2020-04-09 12:38:54,716 --> INFO: Evaluating Elastic Net…
2020-04-09 12:38:54,722 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 12:54:27,858 --> INFO: Best parameters: {'alpha':
6.489338358646725e-06, 'fit_intercept': 1, 'normalize': 1}
2020-04-09 12:54:30,618 --> INFO: Accuracy (floor) of the cross-validation:
51.465% ± 1.247%
2020-04-09 12:54:31,119 --> INFO: Accuracy (floor) of the predictions: 51.654%
2020-04-09 12:54:31,120 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 13:38:44,317 --> INFO: Best parameters: {'alpha':
4.266318343793527e-06, 'fit_intercept': 1, 'normalize': 1}
2020-04-09 13:38:50,168 --> INFO: Accuracy (floor) of the cross-validation:
11.281% ± 0.974%
2020-04-09 13:38:51,981 --> INFO: Accuracy (floor) of the predictions: 10.814%
2020-04-09 13:38:51,982 --> INFO: Fitting the feature engineered set for h11…
2020-04-09 13:42:57,850 --> INFO: Best parameters: {'alpha': 0.0999222657985683,
'fit_intercept': 0, 'normalize': 0}
2020-04-09 13:42:58,096 --> INFO: Accuracy (floor) of the cross-validation:
62.930% ± 1.349%
2020-04-09 13:42:58,123 --> INFO: Accuracy (floor) of the predictions: 64.122%
2020-04-09 13:42:58,124 --> INFO: Fitting the feature engineered set for h21…
2020-04-09 14:21:14,918 --> INFO: Best parameters: {'alpha':
2.732330584792332e-07, 'fit_intercept': 1, 'normalize': 1}
2020-04-09 14:21:24,838 --> INFO: Accuracy (floor) of the cross-validation:
20.127% ± 0.968%
2020-04-09 14:21:28,138 --> INFO: Accuracy (floor) of the predictions: 17.430%
2020-04-09 14:21:28,139 --> INFO: Plotting convergence progression…
2020-04-09 14:21:28,327 --> INFO: Saving
./img/original/el_net_conv_prog_eng.png…
2020-04-09 14:21:28,515 --> INFO: Saved ./img/original/el_net_conv_prog_eng.png!
```

```
2020-04-09 14:21:28,769 --> INFO: Plotting error distribution…
2020-04-09 14:21:28,900 --> INFO: Saving ./img/original/el_net_error_eng.png…
```

```
2020-04-09 14:21:29,047 --> INFO: Saved ./img/original/el_net_error_eng.png!
```

```
2020-04-09 14:21:29,188 --> INFO: Saving the estimator to
el_net_h11.joblib.xz…
2020-04-09 14:21:29,234 --> INFO: Saved el_net_h11.joblib.xz!
2020-04-09 14:21:29,235 --> INFO: Saving the estimator to
el_net_h21.joblib.xz…
2020-04-09 14:21:29,283 --> INFO: Saved el_net_h21.joblib.xz!
```

1.6.5 Linear Support Vector Machine

We then change completely the model and start to analyse **support vector machines**. In particular we start from a linear implementation of the algorithm in order to have a common ground with the previous analysis. In this case the number of hyperparameters is usually larger than before, thus we can expect an increase in the time spent to train the algorithm.

[41]:
```
from sklearn.svm import LinearSVR
logprint('Evaluating Linear SVR...', logger=logger)
```

```
# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = [Real(1.0e-2, 2.0e2, base=10, prior='log-uniform',                    ␣
↪     name='C'),
                 Real(1e-3, 1e3, base=10, prior='log-uniform',                        ␣
↪     name='intercept_scaling'),
                 Integer(False, True,                                                 ␣
↪     name='fit_intercept'),
                 Categorical(['epsilon_insensitive',␣
↪'squared_epsilon_insensitive'], name='loss')
                ]
rounding = np.floor

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
lin_svr_h11 = LinearSVR(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    lin_svr_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lin_svr_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
lin_svr_h11_res = gp_minimize(objective_mat_h11, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lin_svr_h11_res.x[n] for n in␣
↪range(len(lin_svr_h11_res.x))}
lin_svr_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(lin_svr_h11, matrix_train, h11_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
```

```
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
↪format(rounding.__name__,
                                                                                          np.
↪mean(cross_score_h11)*100.0,
                                                                                          np.
↪std(cross_score_h11)*100.0
                                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
lin_svr_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=lin_svr_h11, X=matrix_test,␣
↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
lin_svr_h21 = LinearSVR(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    lin_svr_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lin_svr_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
lin_svr_h21_res = gp_minimize(objective_mat_h21, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lin_svr_h21_res.x[n] for n in␣
↪range(len(lin_svr_h21_res.x))}
lin_svr_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(lin_svr_h21, matrix_train, h21_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
↪format(rounding.__name__,
                                                                                     np.
↪mean(cross_score_h21)*100.0,
                                                                                     np.
↪std(cross_score_h21)*100.0
                                                                                    ),
        logger=logger
       )

# compute the accuracy of the predictions
lin_svr_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=lin_svr_h21, X=matrix_test,␣
↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
lin_svr_h11 = LinearSVR(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    lin_svr_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lin_svr_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
lin_svr_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lin_svr_h11_res.x[n] for n in␣
↪range(len(lin_svr_h11_res.x))}
lin_svr_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(lin_svr_h11, features_h11_train, h11_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
```

```
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                                np.
 ↪mean(cross_score_h11)*100.0,
                                                                                np.
 ↪std(cross_score_h11)*100.0
                                                                               ),
         logger=logger
        )

# compute the accuracy of the predictions
lin_svr_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=lin_svr_h11, X=features_h11_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
lin_svr_h21 = LinearSVR(max_iter=15000, tol=0.001, random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    lin_svr_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(lin_svr_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
lin_svr_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: lin_svr_h21_res.x[n] for n in␣
 ↪range(len(lin_svr_h21_res.x))}
lin_svr_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(lin_svr_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                            np.
 ↪mean(cross_score_h21)*100.0,
                                                                            np.
 ↪std(cross_score_h21)*100.0
                                                                           ),
         logger=logger
        )

# compute the accuracy of the predictions
lin_svr_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=lin_svr_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(lin_svr_h11_res, ax=ax[0])
plot_convergence(lin_svr_h21_res, ax=ax[1])

save_fig('lin_svr_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      lin_svr_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      lin_svr_h21.predict(features_h21_test),
                       rounding=rounding),
            title='Error distribution on the test set',
            xlabel='Difference from real value',
            legend='$h_{21}$')

save_fig('lin_svr_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('lin_svr_h11', lin_svr_h11)
save_model('lin_svr_h21', lin_svr_h21)
```

```
2020-04-09 14:21:29,329 --> INFO: Evaluating Linear SVR…
2020-04-09 14:21:29,338 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 14:42:34,274 --> INFO: Best parameters: {'C': 0.024150170295267344,
'intercept_scaling': 0.28602907950666967, 'fit_intercept': 1, 'loss':
'squared_epsilon_insensitive'}
2020-04-09 14:42:36,395 --> INFO: Accuracy (floor) of the cross-validation:
51.437% ± 1.678%
2020-04-09 14:42:36,436 --> INFO: Accuracy (floor) of the predictions: 51.908%
2020-04-09 14:42:36,438 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 14:58:44,224 --> INFO: Best parameters: {'C': 10.107677022985344,
'intercept_scaling': 1.6338298132220626, 'fit_intercept': 1, 'loss':
'epsilon_insensitive'}
2020-04-09 14:58:46,761 --> INFO: Accuracy (floor) of the cross-validation:
11.465% ± 0.630%
2020-04-09 14:58:47,004 --> INFO: Accuracy (floor) of the predictions: 12.087%
2020-04-09 14:58:47,005 --> INFO: Fitting the feature engineered set for h11…
2020-04-09 15:49:52,752 --> INFO: Best parameters: {'C': 0.01,
'intercept_scaling': 0.4608885326772953, 'fit_intercept': 0, 'loss':
'squared_epsilon_insensitive'}
2020-04-09 15:49:52,961 --> INFO: Accuracy (floor) of the cross-validation:
54.876% ± 0.851%
2020-04-09 15:49:53,018 --> INFO: Accuracy (floor) of the predictions: 53.053%
2020-04-09 15:49:53,019 --> INFO: Fitting the feature engineered set for h21…
2020-04-09 17:45:20,178 --> INFO: Best parameters: {'C': 3.9318912970679647,
'intercept_scaling': 0.17583193101026398, 'fit_intercept': 1, 'loss':
'epsilon_insensitive'}
2020-04-09 17:46:52,754 --> INFO: Accuracy (floor) of the cross-validation:
21.840% ± 1.229%
2020-04-09 17:47:18,464 --> INFO: Accuracy (floor) of the predictions: 19.847%
2020-04-09 17:47:18,465 --> INFO: Plotting convergence progression…
2020-04-09 17:47:18,651 --> INFO: Saving
./img/original/lin_svr_conv_prog_eng.png…
2020-04-09 17:47:18,780 --> INFO: Saved
./img/original/lin_svr_conv_prog_eng.png!
```

```
2020-04-09 17:47:18,948 --> INFO: Plotting error distribution…
2020-04-09 17:47:19,048 --> INFO: Saving ./img/original/lin_svr_error_eng.png…
2020-04-09 17:47:19,163 --> INFO: Saved ./img/original/lin_svr_error_eng.png!
```

```
2020-04-09 17:47:19,259 --> INFO: Saving the estimator to
lin_svr_h11.joblib.xz…
2020-04-09 17:47:19,291 --> INFO: Saved lin_svr_h11.joblib.xz!
2020-04-09 17:47:19,292 --> INFO: Saving the estimator to
lin_svr_h21.joblib.xz…
2020-04-09 17:47:19,326 --> INFO: Saved lin_svr_h21.joblib.xz!
```

1.6.6 Support Vector Machine (with Gaussian Kernel)

In the framework of **support vector machines** we consider a different choice of the kernel function, namely the Gaussian kernel (`SVR(kernel='rbf'` in terms of *scikit-learn* implementation). This is a very powerful technique and given its geometric interpretation, we can probably expect to find the best results. Notice that in this case we adopt a different rounding function. We specifically choose to `np.rint` function to improve the results.

[42]:
```
from sklearn.svm import SVR
logprint('Evaluating SVR...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = [Real(1.0e-1, 1.0e2,  base=10, prior='log-uniform', name='C'),
                 Real(1.0e-3, 1.0e-1, base=10, prior='log-uniform',␣
↪name='gamma'),
                 Real(1.0e-6, 1.0e-1, base=10, prior='log-uniform',␣
↪name='epsilon'),
                 Integer(False, True,                              ␣
↪name='shrinking')
                ]
rounding = np.rint

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
svr_rbf_h11 = SVR(kernel='rbf', tol=0.001)

@use_named_args(search_params)
def objective_mat_h11(**params):
    svr_rbf_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(svr_rbf_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                                cv=cv,
                                                n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
svr_rbf_h11_res = gp_minimize(objective_mat_h11, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: svr_rbf_h11_res.x[n] for n in␣
 ↪range(len(svr_rbf_h11_res.x))}
svr_rbf_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(svr_rbf_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h11)*100.0,
                                                                          np.
 ↪std(cross_score_h11)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
svr_rbf_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=svr_rbf_h11, X=matrix_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
svr_rbf_h21 = SVR(kernel='rbf', tol=0.001)

@use_named_args(search_params)
def objective_mat_h21(**params):
    svr_rbf_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(svr_rbf_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))
```

```
# fit the algorithm and minimize the objective function for h21
svr_rbf_h21_res = gp_minimize(objective_mat_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: svr_rbf_h21_res.x[n] for n in␣
 ↪range(len(svr_rbf_h21_res.x))}
svr_rbf_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(svr_rbf_h21, matrix_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                              np.
 ↪mean(cross_score_h21)*100.0,
                                                                              np.
 ↪std(cross_score_h21)*100.0
                                                                             ),
         logger=logger
        )

# compute the accuracy of the predictions
svr_rbf_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=svr_rbf_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
svr_rbf_h11 = SVR(kernel='rbf', tol=0.001)

@use_named_args(search_params)
def objective_h11(**params):
    svr_rbf_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(svr_rbf_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))
```

```
# fit the algorithm and minimize the objective function for h11
svr_rbf_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter,␣
↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: svr_rbf_h11_res.x[n] for n in␣
↪range(len(svr_rbf_h11_res.x))}
svr_rbf_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(svr_rbf_h11, features_h11_train, h11_train,␣
↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
↪format(rounding.__name__,
                                                                              np.
↪mean(cross_score_h11)*100.0,
                                                                              np.
↪std(cross_score_h11)*100.0
                                                                             ),
         logger=logger
        )

# compute the accuracy of the predictions
svr_rbf_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=svr_rbf_h11, X=features_h11_test,␣
↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
svr_rbf_h21 = SVR(kernel='rbf', tol=0.001)

@use_named_args(search_params)
def objective_h21(**params):
    svr_rbf_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(svr_rbf_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
```

```
svr_rbf_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: svr_rbf_h21_res.x[n] for n in␣
 ↪range(len(svr_rbf_h21_res.x))}
svr_rbf_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(svr_rbf_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h21)*100.0,
                                                                          np.
 ↪std(cross_score_h21)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
svr_rbf_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=svr_rbf_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(svr_rbf_h11_res, ax=ax[0])
plot_convergence(svr_rbf_h21_res, ax=ax[1])

save_fig('svr_rbf_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
```

```
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      svr_rbf_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      svr_rbf_h21.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')

save_fig('svr_rbf_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('svr_rbf_h11', svr_rbf_h11)
save_model('svr_rbf_h21', svr_rbf_h21)
```

```
2020-04-09 17:47:19,382 --> INFO: Evaluating SVR…
2020-04-09 17:47:19,386 --> INFO: Fitting the matrix baseline for h11…
2020-04-09 20:14:16,112 --> INFO: Best parameters: {'C': 35.33852281770326,
'gamma': 0.035409984260578134, 'epsilon': 1e-06, 'shrinking': 0}
2020-04-09 20:16:04,038 --> INFO: Accuracy (rint) of the cross-validation:
70.318% ± 1.277%
2020-04-09 20:16:52,156 --> INFO: Accuracy (rint) of the predictions: 68.575%
2020-04-09 20:16:52,157 --> INFO: Fitting the matrix baseline for h21…
2020-04-09 21:51:04,308 --> INFO: Best parameters: {'C': 18.41049109945271,
'gamma': 0.0675734377514334, 'epsilon': 1e-06, 'shrinking': 0}
2020-04-09 21:51:54,808 --> INFO: Accuracy (rint) of the cross-validation:
22.619% ± 1.183%
2020-04-09 21:52:16,713 --> INFO: Accuracy (rint) of the predictions: 24.555%
2020-04-09 21:52:16,713 --> INFO: Fitting the feature engineered set for h11…
2020-04-10 00:01:34,691 --> INFO: Best parameters: {'C': 57.1755329164205,
'gamma': 0.027645527499421556, 'epsilon': 1e-06, 'shrinking': 1}
2020-04-10 00:03:25,030 --> INFO: Accuracy (rint) of the cross-validation:
72.456% ± 1.404%
2020-04-10 00:04:12,845 --> INFO: Accuracy (rint) of the predictions: 73.664%
```

```
2020-04-10 00:04:12,846 --> INFO: Fitting the feature engineered set for h21…
2020-04-10 01:05:03,071 --> INFO: Best parameters: {'C': 19.678296692699618,
'gamma': 0.006868285877846671, 'epsilon': 1e-06, 'shrinking': 1}
2020-04-10 01:05:38,004 --> INFO: Accuracy (rint) of the cross-validation:
37.636% ± 1.370%
2020-04-10 01:05:53,178 --> INFO: Accuracy (rint) of the predictions: 35.623%
2020-04-10 01:05:53,178 --> INFO: Plotting convergence progression…
2020-04-10 01:05:53,315 --> INFO: Saving
./img/original/svr_rbf_conv_prog_eng.png…
2020-04-10 01:05:53,475 --> INFO: Saved
./img/original/svr_rbf_conv_prog_eng.png!
```

```
2020-04-10 01:05:53,640 --> INFO: Plotting error distribution…
2020-04-10 01:05:54,987 --> INFO: Saving ./img/original/svr_rbf_error_eng.png…
2020-04-10 01:05:55,071 --> INFO: Saved ./img/original/svr_rbf_error_eng.png!
```

```
2020-04-10 01:05:55,165 --> INFO: Saving the estimator to
svr_rbf_h11.joblib.xz…
2020-04-10 01:05:56,477 --> INFO: Saved svr_rbf_h11.joblib.xz!
2020-04-10 01:05:56,478 --> INFO: Saving the estimator to
svr_rbf_h21.joblib.xz…
2020-04-10 01:05:57,950 --> INFO: Saved svr_rbf_h21.joblib.xz!
```

1.6.7 Random Forest (*Scikit* implementation)

We then consider the family of decision tree algorithms. In particular we start with a **random forest** regressor with the *Scikit* implementation. Given the large number of hyperaparameters to optimise, it may be necessary to reduce the no. of iterations of the Bayesan optimisation.

[43]:
```
from sklearn.ensemble import RandomForestRegressor
logprint('Evaluating Random Forest...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
↪accuracy
search_params = [Integer(30, 75, prior='uniform',     name='n_estimators'),
                 Integer(2, 12, prior='uniform',      name='min_samples_split'),
                 Integer(2, 25, prior='uniform',      name='min_samples_leaf'),
                 Integer(2, 20, prior='uniform',      name='max_depth'),
                 Categorical(['friedman_mse', 'mae'], name='criterion')
                ]
rounding   = np.floor
n_iter_mod = int(n_iter / 4)

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
rnd_for_h11 = RandomForestRegressor(random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    rnd_for_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(rnd_for_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
rnd_for_h11_res = gp_minimize(objective_mat_h11, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: rnd_for_h11_res.x[n] for n in␣
 ↪range(len(rnd_for_h11_res.x))}
rnd_for_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(rnd_for_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                           np.
 ↪mean(cross_score_h11)*100.0,
                                                                           np.
 ↪std(cross_score_h11)*100.0
```

```
                                                                            ),
        logger=logger
       )

# compute the accuracy of the predictions
rnd_for_h11.set_params(n_jobs=-1)
rnd_for_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=rnd_for_h11, X=matrix_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
rnd_for_h21 = RandomForestRegressor(random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    rnd_for_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(rnd_for_h21,
                                         matrix_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
rnd_for_h21_res = gp_minimize(objective_mat_h21, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: rnd_for_h21_res.x[n] for n in␣
 ↪range(len(rnd_for_h21_res.x))}
rnd_for_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(rnd_for_h21, matrix_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                             np.
 ↪mean(cross_score_h21)*100.0,
                                                                             np.
 ↪std(cross_score_h21)*100.0
                                                                            ),
        logger=logger
        )

# compute the accuracy of the predictions
rnd_for_h21.set_params(n_jobs=-1)
rnd_for_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=rnd_for_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
rnd_for_h11 = RandomForestRegressor(random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    rnd_for_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(rnd_for_h11,
                                         features_h11_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
rnd_for_h11_res = gp_minimize(objective_h11, search_params, n_calls=n_iter_mod,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: rnd_for_h11_res.x[n] for n in␣
 ↪range(len(rnd_for_h11_res.x))}
rnd_for_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(rnd_for_h11, features_h11_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                              np.
 ↪mean(cross_score_h11)*100.0,
                                                                              np.
 ↪std(cross_score_h11)*100.0
```

```
                                                                                    ),
         logger=logger
        )

# compute the accuracy of the predictions
rnd_for_h11.set_params(n_jobs=-1)
rnd_for_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=rnd_for_h11, X=features_h11_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
rnd_for_h21 = RandomForestRegressor(random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    rnd_for_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(rnd_for_h21,
                                         features_h21_train,
                                         h21_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
rnd_for_h21_res = gp_minimize(objective_h21, search_params, n_calls=n_iter_mod,␣
 ↪random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: rnd_for_h21_res.x[n] for n in␣
 ↪range(len(rnd_for_h21_res.x))}
rnd_for_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(rnd_for_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                                     np.
 ↪mean(cross_score_h21)*100.0,
                                                                                     np.
 ↪std(cross_score_h21)*100.0
                                                                                    ),
         logger=logger
        )

# compute the accuracy of the predictions
rnd_for_h21.set_params(n_jobs=-1)
rnd_for_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=rnd_for_h21, X=features_h21_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(rnd_for_h11_res, ax=ax[0])
plot_convergence(rnd_for_h21_res, ax=ax[1])

save_fig('rnd_for_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      rnd_for_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      rnd_for_h21.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')
```

```
save_fig('rnd_for_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('rnd_for_h11', rnd_for_h11)
save_model('rnd_for_h21', rnd_for_h21)
```

```
2020-04-10 01:05:57,981 --> INFO: Evaluating Random Forest…
2020-04-10 01:05:57,986 --> INFO: Fitting the matrix baseline for h11…
2020-04-10 02:06:15,695 --> INFO: Best parameters: {'n_estimators': 75,
'min_samples_split': 2, 'min_samples_leaf': 2, 'max_depth': 20, 'criterion':
'friedman_mse'}
2020-04-10 02:06:24,459 --> INFO: Accuracy (floor) of the cross-validation:
62.067% ± 2.203%
2020-04-10 02:06:25,400 --> INFO: Accuracy (floor) of the predictions: 60.051%
2020-04-10 02:06:25,402 --> INFO: Fitting the matrix baseline for h21…
2020-04-10 04:19:41,414 --> INFO: Best parameters: {'n_estimators': 30,
'min_samples_split': 2, 'min_samples_leaf': 2, 'max_depth': 20, 'criterion':
'mae'}
2020-04-10 04:24:30,249 --> INFO: Accuracy (floor) of the cross-validation:
15.088% ± 1.133%
2020-04-10 04:25:01,952 --> INFO: Accuracy (floor) of the predictions: 15.267%
2020-04-10 04:25:01,953 --> INFO: Fitting the feature engineered set for h11…
2020-04-10 09:53:53,885 --> INFO: Best parameters: {'n_estimators': 30,
'min_samples_split': 12, 'min_samples_leaf': 2, 'max_depth': 6, 'criterion':
'mae'}
2020-04-10 10:05:07,317 --> INFO: Accuracy (floor) of the cross-validation:
64.544% ± 1.701%
2020-04-10 10:06:19,004 --> INFO: Accuracy (floor) of the predictions: 63.486%
2020-04-10 10:06:19,005 --> INFO: Fitting the feature engineered set for h21…
2020-04-10 14:08:58,795 --> INFO: Best parameters: {'n_estimators': 30,
'min_samples_split': 2, 'min_samples_leaf': 2, 'max_depth': 20, 'criterion':
'friedman_mse'}
2020-04-10 14:09:32,213 --> INFO: Accuracy (floor) of the cross-validation:
20.326% ± 1.372%
2020-04-10 14:09:35,349 --> INFO: Accuracy (floor) of the predictions: 17.176%
2020-04-10 14:09:35,354 --> INFO: Plotting convergence progression…
2020-04-10 14:09:35,702 --> INFO: Saving
./img/original/rnd_for_conv_prog_eng.png…
2020-04-10 14:09:35,925 --> INFO: Saved
./img/original/rnd_for_conv_prog_eng.png!
```

```
2020-04-10 14:09:36,214 --> INFO: Plotting error distribution…
2020-04-10 14:09:36,539 --> INFO: Saving ./img/original/rnd_for_error_eng.png…
2020-04-10 14:09:36,674 --> INFO: Saved ./img/original/rnd_for_error_eng.png!
```

```
2020-04-10 14:09:36,821 --> INFO: Saving the estimator to
rnd_for_h11.joblib.xz…
2020-04-10 14:09:36,994 --> INFO: Saved rnd_for_h11.joblib.xz!
2020-04-10 14:09:36,995 --> INFO: Saving the estimator to
rnd_for_h21.joblib.xz…
2020-04-10 14:09:39,214 --> INFO: Saved rnd_for_h21.joblib.xz!
```

1.6.8 Gradient Boosting (*Scikit* implementation)

We then consider the *Scikit* implementation of boosted trees to improve the previous results. Due to computational power and the very time consuming operation, we restrict the number of boostings but try to optimize a larger number of hyperparameters.

[15]:
```
from sklearn.ensemble import GradientBoostingRegressor
logprint('Evaluating Gradient Boosting...', logger=logger)

# define search parameters and the rounding function for the computation of the␣
 ↪accuracy
search_params = [Categorical(['ls', 'lad'],                          name='loss'),
                 Categorical(['friedman_mse', 'mae'],               ␣
 ↪name='criterion'),
                 Real(0.6, 1.0,             prior='uniform',        ␣
 ↪name='subsample'),
                 Real(1e-5, 1e-1, base=10, prior='log-uniform',␣
 ↪name='learning_rate'),
                 Integer(2, 10,             prior='uniform',        ␣
 ↪name='min_samples_split'),
                 Integer(1, 50,             prior='uniform',        ␣
 ↪name='min_samples_leaf'),
                 Integer(2, 20,             prior='uniform',        ␣
 ↪name='max_depth')
                ]
rounding   = np.floor
n_iter_mod = int(n_iter / 4)

#------------------------------
# MATRIX BASELINE
#------------------------------

# define objective function for h11
logprint('Fitting the matrix baseline for h11...', logger=logger)
grd_boost_h11 = GradientBoostingRegressor(n_estimators=45, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h11(**params):
    grd_boost_h11.set_params(**params)
    return 1.0 - np.mean(cross_val_score(grd_boost_h11,
                                         matrix_train,
                                         h11_train,
                                         scoring=scorer(rounding),
                                         cv=cv,
                                         n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
grd_boost_h11_res = gp_minimize(objective_mat_h11, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: grd_boost_h11_res.x[n] for n in␣
 ↪range(len(grd_boost_h11_res.x))}
grd_boost_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(grd_boost_h11, matrix_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                            np.
 ↪mean(cross_score_h11)*100.0,
                                                                            np.
 ↪std(cross_score_h11)*100.0
                                                                           ),
         logger=logger
        )

# compute the accuracy of the predictions
grd_boost_h11.fit(matrix_train, h11_train)
preds_score_h11 = prediction_score(estimator=grd_boost_h11, X=matrix_test,␣
 ↪y=h11_test, use_best_estimator=False, rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the matrix baseline for h21...', logger=logger)
grd_boost_h21 = GradientBoostingRegressor(n_estimators=45, random_state=RAND)

@use_named_args(search_params)
def objective_mat_h21(**params):
    grd_boost_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(grd_boost_h21,
                                         matrix_train,
                                                h21_train,
                                                scoring=scorer(rounding),
                                                cv=cv,
                                                n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
grd_boost_h21_res = gp_minimize(objective_mat_h21, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: grd_boost_h21_res.x[n] for n in␣
 ↪range(len(grd_boost_h21_res.x))}
grd_boost_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(grd_boost_h21, matrix_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                          np.
 ↪mean(cross_score_h21)*100.0,
                                                                          np.
 ↪std(cross_score_h21)*100.0
                                                                         ),
         logger=logger
        )

# compute the accuracy of the predictions
grd_boost_h21.fit(matrix_train, h21_train)
preds_score_h21 = prediction_score(estimator=grd_boost_h21, X=matrix_test,␣
 ↪y=h21_test, use_best_estimator=False, rounding=rounding, logger=logger)

#------------------------------
# FEATURE ENGINEERED SET
#------------------------------

# define objective function for h11
logprint('Fitting the feature engineered set for h11...', logger=logger)
grd_boost_h11 = GradientBoostingRegressor(n_estimators=45, random_state=RAND)

@use_named_args(search_params)
def objective_h11(**params):
    grd_boost_h11.set_params(**params)

    return 1.0 - np.mean(cross_val_score(grd_boost_h11,
                                          features_h11_train,
                                          h11_train,
                                          scoring=scorer(rounding),
                                          cv=cv,
                                          n_jobs=-1))

# fit the algorithm and minimize the objective function for h11
grd_boost_h11_res = gp_minimize(objective_h11, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: grd_boost_h11_res.x[n] for n in␣
 ↪range(len(grd_boost_h11_res.x))}
grd_boost_h11.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h11 = cross_val_score(grd_boost_h11, features_h11_train, h11_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                            np.
 ↪mean(cross_score_h11)*100.0,
                                                                            np.
 ↪std(cross_score_h11)*100.0
                                                                           ),
         logger=logger
        )

# compute the accuracy of the predictions
grd_boost_h11.fit(features_h11_train, h11_train)
preds_score_h11 = prediction_score(estimator=grd_boost_h11,␣
 ↪X=features_h11_test, y=h11_test, use_best_estimator=False,␣
 ↪rounding=rounding, logger=logger)

# define objective function for h21
logprint('Fitting the feature engineered set for h21...', logger=logger)
grd_boost_h21 = GradientBoostingRegressor(n_estimators=45, random_state=RAND)

@use_named_args(search_params)
def objective_h21(**params):
    grd_boost_h21.set_params(**params)

    return 1.0 - np.mean(cross_val_score(grd_boost_h21,
                                          features_h21_train,
                                          h21_train,
                                                scoring=scorer(rounding),
                                                cv=cv,
                                                n_jobs=-1))

# fit the algorithm and minimize the objective function for h21
grd_boost_h21_res = gp_minimize(objective_h21, search_params,␣
 ↪n_calls=n_iter_mod, random_state=RAND)

# return the best parameters and print them
best_params = {search_params[n].name: grd_boost_h21_res.x[n] for n in␣
 ↪range(len(grd_boost_h21_res.x))}
grd_boost_h21.set_params(**best_params)
logprint('Best parameters: {}'.format(best_params), logger=logger)

# compute the cross-validation accuracy
cross_score_h21 = cross_val_score(grd_boost_h21, features_h21_train, h21_train,␣
 ↪scoring=scorer(rounding), cv=cv, n_jobs=-1)
logprint('Accuracy ({}) of the cross-validation: {:.3f}% ± {:.3f}%'.
 ↪format(rounding.__name__,
                                                                        np.
 ↪mean(cross_score_h21)*100.0,
                                                                        np.
 ↪std(cross_score_h21)*100.0
                                                                       ),
         logger=logger
        )

# compute the accuracy of the predictions
grd_boost_h21.fit(features_h21_train, h21_train)
preds_score_h21 = prediction_score(estimator=grd_boost_h21,␣
 ↪X=features_h21_test, y=h21_test, use_best_estimator=False,␣
 ↪rounding=rounding, logger=logger)

#------------------------------
# PLOTS
#------------------------------
logprint('Plotting convergence progression...', logger=logger)
xplots  = 2
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

plot_convergence(grd_boost_h11_res, ax=ax[0])
plot_convergence(grd_boost_h21_res, ax=ax[1])
```

```
save_fig('grd_boost_conv_prog_eng', logger=logger)
plt.show()
plt.close(fig)

logprint('Plotting error distribution...', logger=logger)
xplots  = 1
yplots  = 1
fig, ax = plt.subplots(yplots, xplots, figsize=(xplots*mpl_width,␣
 ↪yplots*mpl_height))
fig.tight_layout()

count_plot(ax,
           error_diff(h11_test,
                      grd_boost_h11.predict(features_h11_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{11}$')
count_plot(ax,
           error_diff(h21_test,
                      grd_boost_h21.predict(features_h21_test),
                      rounding=rounding),
           title='Error distribution on the test set',
           xlabel='Difference from real value',
           legend='$h_{21}$')

save_fig('grd_boost_error_eng', logger=logger)
plt.show()
plt.close(fig)

# Saving the feature engineered models
save_model('grd_boost_h11', grd_boost_h11)
save_model('grd_boost_h21', grd_boost_h21)
```

```
2020-04-11 22:23:09,180 --> INFO: Evaluating Gradient Boosting…
2020-04-11 22:23:09,186 --> INFO: Fitting the matrix baseline for h11…
2020-04-12 01:02:49,945 --> INFO: Best parameters: {'loss': 'ls', 'criterion':
'mae', 'subsample': 1.0, 'learning_rate': 0.1, 'min_samples_split': 2,
'min_samples_leaf': 1, 'max_depth': 20}
2020-04-12 01:22:19,542 --> INFO: Accuracy (floor) of the cross-validation:
55.159% ± 2.581%
2020-04-12 01:28:49,940 --> INFO: Accuracy (floor) of the predictions: 55.471%
2020-04-12 01:28:49,941 --> INFO: Fitting the matrix baseline for h21…
2020-04-12 04:37:28,027 --> INFO: Best parameters: {'loss': 'ls', 'criterion':
'mae', 'subsample': 1.0, 'learning_rate': 0.1, 'min_samples_split': 2,
'min_samples_leaf': 1, 'max_depth': 20}
2020-04-12 05:02:42,665 --> INFO: Accuracy (floor) of the cross-validation:
```

```
16.957% ± 0.844%
2020-04-12 05:13:10,022 --> INFO: Accuracy (floor) of the predictions: 17.939%
2020-04-12 05:13:10,022 --> INFO: Fitting the feature engineered set for h11…
2020-04-12 10:44:02,273 --> INFO: Best parameters: {'loss': 'ls', 'criterion':
'friedman_mse', 'subsample': 0.9545470855056722, 'learning_rate': 0.1,
'min_samples_split': 3, 'min_samples_leaf': 1, 'max_depth': 2}
2020-04-12 10:44:13,316 --> INFO: Accuracy (floor) of the cross-validation:
59.604% ± 1.316%
2020-04-12 10:44:17,671 --> INFO: Accuracy (floor) of the predictions: 58.397%
2020-04-12 10:44:17,672 --> INFO: Fitting the feature engineered set for h21…
2020-04-12 18:40:55,175 --> INFO: Best parameters: {'loss': 'ls', 'criterion':
'friedman_mse', 'subsample': 1.0, 'learning_rate': 0.1, 'min_samples_split': 10,
'min_samples_leaf': 1, 'max_depth': 17}
2020-04-12 18:41:59,817 --> INFO: Accuracy (floor) of the cross-validation:
23.241% ± 1.471%
2020-04-12 18:42:25,732 --> INFO: Accuracy (floor) of the predictions: 23.791%
2020-04-12 18:42:25,733 --> INFO: Plotting convergence progression…
2020-04-12 18:42:26,072 --> INFO: Saving
./img/original/grd_boost_conv_prog_eng.png…
2020-04-12 18:42:26,211 --> INFO: Saved
./img/original/grd_boost_conv_prog_eng.png!
```

```
2020-04-12 18:42:26,372 --> INFO: Plotting error distribution…
2020-04-12 18:42:26,443 --> INFO: Saving
./img/original/grd_boost_error_eng.png…
2020-04-12 18:42:26,520 --> INFO: Saved ./img/original/grd_boost_error_eng.png!
```

```
2020-04-12 18:42:26,601 --> INFO: Saving the estimator to
grd_boost_h11.joblib.xz…
2020-04-12 18:42:26,646 --> INFO: Saved grd_boost_h11.joblib.xz!
2020-04-12 18:42:26,647 --> INFO: Saving the estimator to
grd_boost_h21.joblib.xz…
2020-04-12 18:42:28,932 --> INFO: Saved grd_boost_h21.joblib.xz!
```

In this case we can also plot the loss function as a function of the boosting rounds:

[24]:
```
xplot = 1
yplot = 1
fig, ax = plt.subplots(xplot, yplot, figsize=(xplot*mpl_width,␣
 ↪yplot*mpl_height))

series_plot(ax, grd_boost_h11.train_score_, title='Loss function', xlabel='no.␣
 ↪of boosting rounds', ylabel='loss', legend='$h_{11}$')
series_plot(ax, grd_boost_h21.train_score_, title='Loss function', xlabel='no.␣
 ↪of boosting rounds', ylabel='loss', legend='$h_{21}$')
```

```
save_fig('grd_boost_loss_function', logger=logger)
plt.show()
plt.close(fig)
```

```
2020-04-12 19:21:48,129 --> INFO: Saving
./img/original/grd_boost_loss_function.png…
2020-04-12 19:21:48,248 --> INFO: Saved
./img/original/grd_boost_loss_function.png!
```