Title: A primer about Machine Learning in Catalysis - A tutorial with code

Authors: Stefan Palkovits

To be cited as: *ChemCatChem* 10.1002/cctc.202000234

Link to VoR: http://dx.doi.org/10.1002/cctc.202000234

10.1002/cctc.202000234

A primer about Machine Learning in Catalysis - A tutorial with code

Dr. Stefan Palkovits

RWTH Aachen University
Institute for Technical and Macromolecular Chemistry
Worringerweg 2
52074 Aachen
E-mail: stefan.palkovits@itmc.rwth-aachen.de
Twitter: @PalkovitsLab

Abstract

Based on a well edited dataset from literature by Schmack et al. [1] this manuscript tries to give a tutorial like introduction to Machine Learning (ML) and Data Science (DS) based on the actual programming code in the Python programming language. The study will not only try to illustrate a ML workflow but will also show important tasks like hyperparameter tuning and data preprocessing which often cover much time of an actual study. Moreover the study spans from classical ML methods to Deep Learning with Neural Networks.

Introduction

Machine Learning (ML) is a growing area in nearly all fields of science but also in public in general. Everyone is using for example recommendation systems from Netflix or Amazon. Other examples are personal assistants like Siri or Alexa or the algorithms behind the search engines from Google. The examples seem to be endless and continue to applications like self driving cars. But also in Chemistry and Catalysis more and more applications for Machine Learning appear. Reviewing and conceptual publications [2-4] do not move away from Computational Chemistry or experimental Catalysis but shift more weight on exploration of the available data and the definition of proper catalyst descriptors.

Very often ML is used at the border of Computational and Solid State Chemistry [5, 6] as the descriptors of the materials are well defined and larger material libraries are easy to prepare. ML is even more prominent in the field of Computational (Heterogeneous) Catalysis and Chemistry [7] as this field of interest has the power to generate large datasets in silico. Here the limiting factor is often not the ML part of the studies but the computational expensive Quantum Chemical calculations. But there are also studies from other fields of Chemistry and Catalysis. With a suitable dataset water oxidation catalysts are predictable with ML [8] and even for approaches from Organic Synthesis there are approaches to make ML based predictions [9]

But especially for people newly approaching this field with a view from their respective discipline Data Science (DS) and Machine Learning sometimes seem to be some kind of arcane art. But when comparing for example a Neural Network with the math in Computational Chemistry then the latter is way more complex. This study will try to convince the inclined reader that ML and DS can be a valuable addition to the Chemists toolbox. The respective algorithms will of course not solve every problem in Chemistry but can be a help and guidance to see and visualize trends that are sometimes well hidden in the data.

As starting point of this study no artificial data will be used but a well edited dataset compiled from experiments from literature initially collected by Zavyalova et al. [10]. The dataset used in this manuscript deals with the oxidative coupling of methane (OCM)

$$2CH_4 + O_2 \rightarrow C_2H4 + 2H_2O$$

and has more than 1.000 entries. One conclusion drawn based on the data is that there are 18 key elements for OCM, namely Sr, Ba, Mg, Ca, La, Nd, Sm, Ga, Bi, Mo, W, Mn, Re, Li, Na, Cs, F and Cl which turned out to be important for a good performance of the OCM reaction. In a first publication based on the original data Kondratenko et. al [11] just used a fraction of the dataset to gain more insights. Finally Schmack et al. [1] came to an even more curated dataset based on the original version. Based on their statistical analysis they showed that a good OCM catalyst contains at least two elements and one of it must be able to form a carbonate at the reaction temperature of the OCM reaction. The second element must be thermally stable at the relevant conditions. This leads to catalysts for the OCM composed of a thermally stable oxide support together with an active species being able to form a carbonate. More information about the original data can be found in the respective manuscripts. The OCM reaction is still under active research because of the relative abundance of methane and there are for example newer studies combining DS approaches with High-Throughput Screening [12].

This study will not reveal new trends from the history of the OCM but draw some conclusions just based on the published data and if possible with cross-reference to the original manuscript. The author will try to illustrate a ML workflow with a decent choice of algorithms and tools, knowing that the same is also achievable with other tools, for example Matlab instead of Python.

Python and Jupyter Notebooks

Especially the draft version of this manuscript was written in a **Jupyter Notebook** which is a browser based front-end originally developed for the programming languages Julia, Python [13] and R and this is also where the name came from. The back-end to the browser based interface is then connected to a kernel from which right now about 100 exist. For this manuscript one of the most frequently used kernels, the Python kernel, will be used. Important to notice is also that the kernel does not have to be on same computer than the browser interface but can also live on a remote high performance machine. Please notice that Jupyter notebooks have the power to mix programming code, Markdown text (a HTML variant) and LaTeX for the narrative around the code. Moreover, it is possible to add videos, pictures, widgets and so on for a rich user experience. Apart from being a tool for programming Jupyter notebooks are especially a useful tool for teaching. For DS and ML Python is close to the standard programming language.

This manuscript is meant to be somewhere between a concept and a tutorial. So there will be things that cannot be explained in depth because of length restrictions. Python and Jupyter Notebooks are well documented and the author encourages the reader to learn more in books or online resources. Now and then semicolons are added to the source code to make the output better readable. They are not needed for functional code (most of the time). The choice of the used libraries is intentionally kept simple. For example for the plotting Matplotlib is chosen instead of for example Altair or Bokeh. The experience from lecturing topics related to programming shows that one main issue is to get started and simplicity helps in the first place.

Data preprocessing

As first step the source data has to be imported and preprocessed in a way that the ML algorithm of choice can use the dataset. This is often a very important step in each ML study because the preprocessing step helps to get an overview over the data, to find outliers and more peculiarities of the respective dataset. To start the project we will first import some libraries like Pandas [14] to work with the tabulated data, NumPy [15, 16] for array type elements and the respective math and Matplotlib [17] for the visualization of the data.

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
```

The next code block is included to print out the version numbers of the libraries used to simplify the reproducibility of this study.

```
import numpy as numpy
import pandas as pandas
import matplotlib as matplotlib
import sklearn as sklearn
import tensorflow as tensorflow

print('Numpy Version', numpy.__version__)
print('Pandas Version', pandas.__version__)
print('Matplotlib Version', matplotlib.__version__)
print('Scikit-Learn Version', sklearn.__version__)
print('Tensorflow Version', tensorflow.__version__)
```

```
Numpy Version 1.15.4
Pandas Version 0.25.0
Matplotlib Version 3.1.0
Scikit-Learn Version 0.21.2
Tensorflow Version 1.13.1
```

The code below with % is a 'magic command' in the Jupyter Notebook and is used to make all plots appear in the browser.

```
%matplotlib inline
```

Although ML methods are often statistical methods it is important to end up with a reproducible and deterministic study. Therefore it is important to provide seeds for the random number generators working behind the scenes in some algorithms and procedures. If a dataset is very large and very balanced this is sometimes not important but for medium and small datasets this can be an issue. Therefore we will fix the global random seed here first to an arbitrary value of 42. Additional seeds will be fixed later in the study if needed and there will be some additional comments.

```
np.random.seed(42)
```

Now we will restrict the Pandas library to show only five columns of the respective tabular output. This restriction is only necessary in the context of this publication to be better readable.

```
pd.set_option('display.max_columns', 5)
```

From the original study [1] the authors did not only publish the manuscript and an electronic supplementory file but there is also an Excel file available online. This link is put now into the variable *url*.

```
url = 'https://static-content.springer.com/esm/art%3A10.1038%2Fs41467-019-08325-8/MediaObjects/41467_2019_8325_MOESM3_ESM.xls'
```

Now we load the actual dataset from the respective website. For this the *read_excel* function from Pandas is used. It takes the URL where the file lives or simply a filename and the name of the sheet we want to use as arguments. Of course there are more possible arguments for fine tuning. The loaded table is stored in the Pandas dataframe *raw_data*. No we *print* the first five rows of *raw_data* with the Pandas *head* function.

```
raw_data = pd.read_excel(url, sheet_name=1)
print(raw_data.head())
```

```
   Catalyst Nr  Nr of publication  ...   S(C2), %  Y(C2), %
0            1                  1  ...  45.500000    5.0050
1            2                  1  ...  40.000000    4.0000
2            3                  1  ...   4.800000    0.4032
3            4                  1  ...  44.099998    2.9988
4            5                  1  ...   1.300000    0.1040

[5 rows x 33 columns]
```

The original table holds one entry/sample per row and of course at a first glance it is not necessary to have a column for every used element and its respective amount. For a ML approach it would help to have a table with columns for each element first and its amount in the sample, no matter if it is a cation, an anion, support material or something else. With a table like that we could use the data as **features** meaning the input data for ML algorithms. To get a table like that we will first create pivoted tables with Pandas for all ions and the supports. The *pivot* function will sort every original column for us with respect to the elements.

```
cation1 = raw_data.pivot(columns='Cation 1', values='Cation 1 mol%')
cation2 = raw_data.pivot(columns='Cation 2', values='Cation 2 mol%')
cation3 = raw_data.pivot(columns='Cation 3', values='Cation 3 mol%')
cation4 = raw_data.pivot(columns='Cation 4', values='Cation 4 mol%')
anion1 = raw_data.pivot(columns='Anion 1', values='Anion 1 mol%')
anion2 = raw_data.pivot(columns='Anion 2', values='Anion 2 mol%')
```

10.1002/cctc.202000234

```
support1 = raw_data.pivot(columns='Support 1', values='Support 1 mol%')
support2 = raw_data.pivot(columns='Support 2', values='Support 2 mol%')
```

Now we create a Python *pivot_list* holding the single pivoted tables.

```
pivot_lists = [cation1, cation2, cation3, cation4, anion1, anion2, support1,
support2]
```

And now we combine all of the pivoted tables with the Pandas *concat* function to a huge list.

```
concat_pivot_lists = pd.concat(pivot_lists, axis=1, sort=True)
```

Pandas like Numpy is array oriented. So we can simply divide the *concat_pivot_lists* dataframe by 100 and put it into a new dataframe called *composition*.

```
composition = concat_pivot_lists.groupby(level=0, axis=1).sum()/100
```

When we print this dataframe we can see that it now formated by the chemical elements with the molar ratio below.

```
print(composition.head())
```

```
    Ag     Al  ...   Zn   Zr
0  0.0  0.908  ...  0.0  0.0
1  0.0  0.953  ...  0.0  0.0
2  0.0  0.955  ...  0.0  0.0
3  0.0  0.896  ...  0.0  0.0
4  0.0  0.972  ...  0.0  0.0

[5 rows x 68 columns]
```

For an overall dataframe we extract now the left over publication numbers and the reaction conditions. To do so we use the Pandas *iloc* function which allows us to extract data in the same way like from a Numpy array. The respective data is the put in the dataframes called *pub_nr* and *reaction_data*.

```
pub_nr = raw_data.iloc[:,0]
reaction_data = raw_data.iloc[:,19:]
```

Now we create a list called *cleaned_list* holding all the single data parts and finally we *concat* the three array into one single dataset called *data_cleaned*.

```
cleaned_list = [pub_nr, composition, reaction_data]
data_cleaned = pd.concat(cleaned_list, axis=1, sort=True)
```

The print out of this arrays indicates that now all information is stored in single columns which is easier to analyze than a mixture of columns and rows. Optionally we can still continue to work with the single arrays like *composition* which are still kept in memory.

```
print(data_cleaned.head())
```

```
   Catalyst Nr   Ag  ...   S(C2), %  Y(C2), %
0            1  0.0  ...  45.500000    5.0050
1            2  0.0  ...  40.000000    4.0000
```

```
2             3  0.0  ...   4.800000    0.4032
3             4  0.0  ...  44.099998    2.9988
4             5  0.0  ...   1.300000    0.1040

[5 rows x 83 columns]
```

Data inspection

Now we can start to work with the data. For example let us calculate the *mean* of all the elements in the *composition* dataframe. Next we *sort_values* in a descending order to make the elements with the highest amount appear first. From this we take the first 18 rows and by using the *keys* we get the names for the elements. All this can be done in a single line of code leading to Mg, Ca and Si being the most prominent elements in the composition array. We will compare this 18 elements with the findings of one of the original publications [10] a little later.

```
print(composition.mean(axis=0).sort_values(ascending=False).head(18).keys())

Index(['Mg', 'Ca', 'Si', 'La', 'Al', 'Li', 'Ba', 'Sr', 'Zr', 'Na', 'Ti', 
'Mn',
       'Nd', 'Cl', 'Sm', 'Y', 'Zn', 'Ce'],
      dtype='object')
```

Visualization is a key part in every ML workflow and so it is here for a first inspection of some part of the data. So let us take the nine elements being present in the largest amount in the dataset. We will loop over this nine entries and plot for each of this the selectivity to C2 components on the y-axis and the respective molar amount on the x-axis. In every plot all samples from the dataset are occurring.

```
count = 1
max_val = 
len(composition.mean(axis=0).sort_values(ascending=False).head(9).keys())

plt.figure(figsize=(9,9))
for element in 
composition.mean(axis=0).sort_values(ascending=False).head(9).keys():
    plt.subplot(np.sqrt(max_val),np.sqrt(max_val), count)
    plt.title(element)
    plt.ylabel('molar amount, %')
    plt.ylabel('S(C2), %')
    plt.scatter(data_cleaned[element], data_cleaned['S(C2), %'])
    count += 1
plt.tight_layout()
```

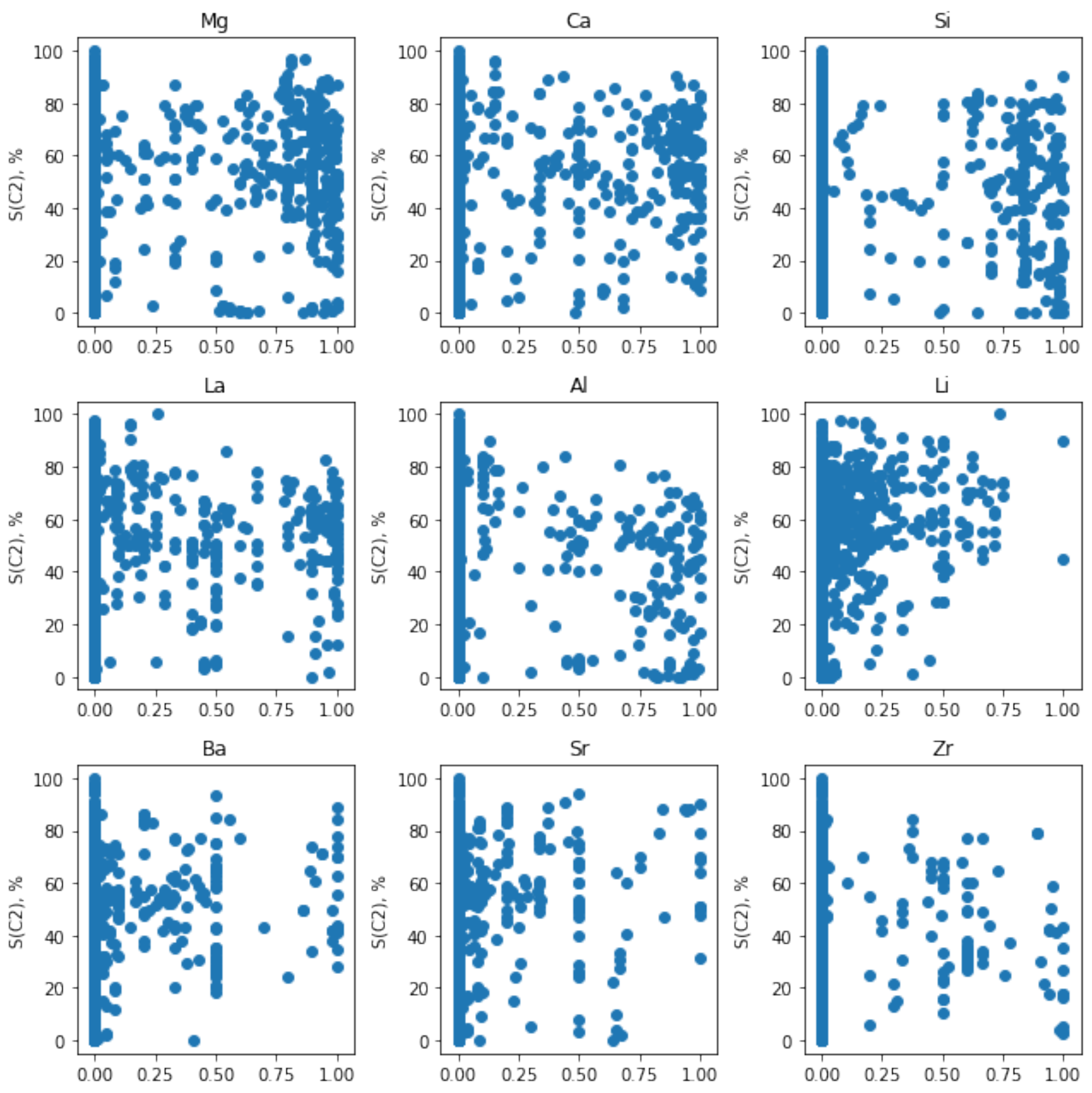


Figure 1: Initial inspection of the dataset.

From the inspection (Figure 1) there are no clear trends to see. For all the nine elements the dots are spread widely over the whole composition axis and for all samples there are clearly good candidates and also bad ones. In the next step we will use ML algorithms to try to see some trends.

Unsupervised learning

In Unsupervised Learning the respective algorithms never gets to see the targets (output values) of the dataset. It only gets to see the features (input variables, like e.g. the composition). This results in a situation were the algorithm has to figure out any dependencies in the data by itself. In the next passage several algorithms will be combined with each other:

- First a K-Means clustering algorithm to group similar samples together;

- Second a Principle Component Analysis (PCA) and a t-Distributed Stochastic Neighbor Embedding [18] (TSNE) to reduce the dimensionality of the dataset.

To use the respective algorithm the corresponding libraries will be imported from Scikit-Learn [19]. Scikit-Learn is a ML library being popular in the Python community for its easy application even for people not familiar with the field. It is even quite popular within more experienced users.

```
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

Now that we have imported the respective libraries for the algorithms we have to make some use of them. Let us start with the K-Means clustering. The algorithms tries to separate the data into clusters where each data point has a minimal distance to a cluster center. But how to find the right amount of clusters? A typical approach is to plot an “Elbow curve”. To do so we loop over several cluster amounts *n_cluster*, in this case between 1 and 20 clusters and in every pass of the loop we initialize the K-Means algorithm with a different cluster amount. Next the algorithm is fit to the composition array and last we calculate a score for each pass and store it in a list. One of the comfortable things about Scikit-Learn is that most of the algorithms have a scoring method included to evaluate the quality of the fitting. All scoring methods give results between 0 and 1 and the higher the better. So after this step we have a score for each cluster amount.

```
k_ellbow = []

for E in range(1,21):
    kmeans = KMeans(n_clusters=E)
    kmeans.fit(composition)
    score = kmeans.score(composition)
    k_ellbow.append(score)
```

Now the score is plotted over the respective cluster amount. To make the “Elbow curve” look nicer the gradient of the score is calculated via Numpy.

```
plt.figure(figsize=(12,9))
plt.title('Elbow curve')
plt.xlabel('n Cluster')
plt.ylabel('grad Score')
plt.plot([7,7],[0, np.max(np.gradient(k_ellbow))])
plt.plot(range(1,21), np.gradient(k_ellbow), 'k');
```

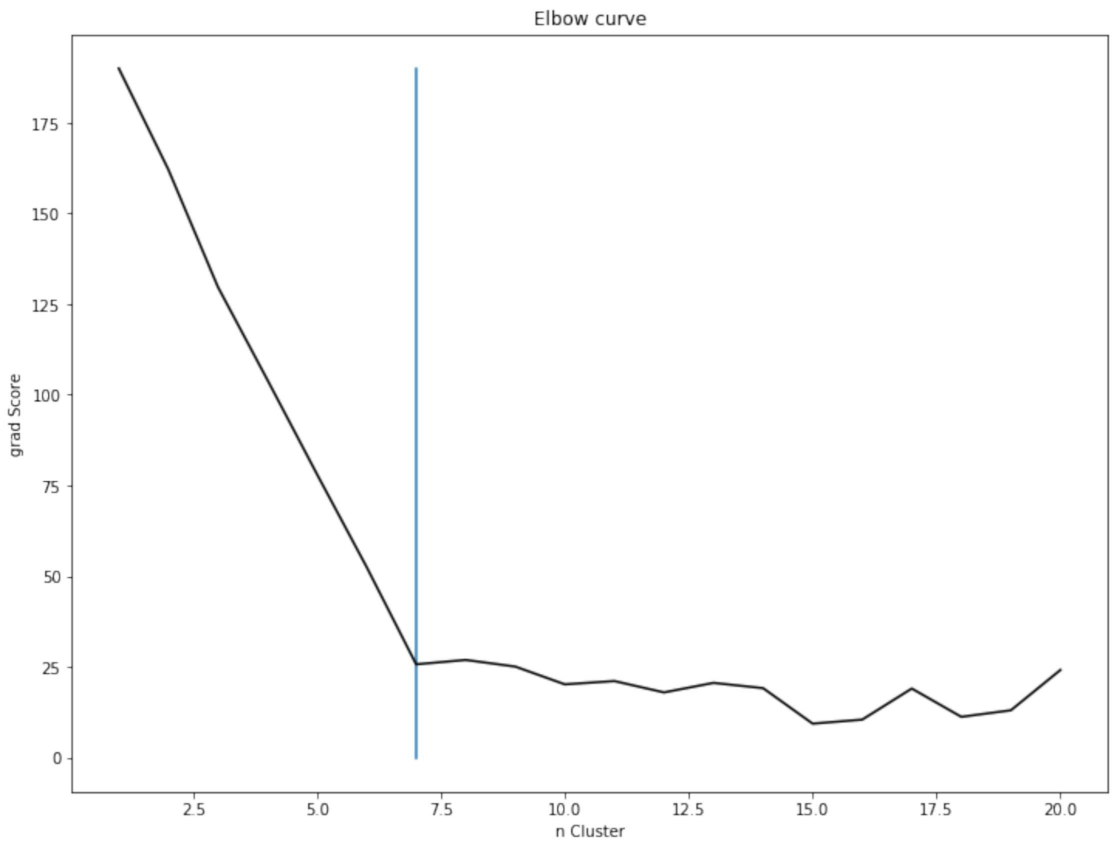


Figure 2: Elbow curve of the scores of the K-Means clustering algorithm.

It can be seen that the line goes all the way down until a cluster amount of around 7 (Figure 2). Choosing fewer clusters will result in a worse result and choosing a cluster amount of more than 7 clusters does not lead to a better result. So a cluster amount of 7 is chosen and a final algorithm is fit with this value. Indeed in the original publications about 10 groups of similar catalysts were found with completely different methods. Having this in mind we are in good agreement with the previous knowledge.

The next code also shows nicely how most Scikit-Learn algorithms work

- First an instance of the respective algorithm is initialized, here *KMeans*, and the parameters are set,
- next the instance of the algorithm is fit to the respective data,
- and last some predictions, transformations or scores are calculated depending on the algorithm.

In this case we end up with a cluster number for each observation in the dataset so that we know which observation belongs to which cluster.

```
cluster = KMeans(n_clusters=7)
cluster.fit(composition)
cluster_predictions = cluster.predict(composition)
```

Unfortunately the dataframe for the elemental composition has 68 columns and 1802 entries now which means we have a 68 dimensional dataset and it is not straight forward to do a 2D visualization directly. But Unsupervised Learning also includes many algorithms for dimensional reduction. In this work first a classical one like Principal Component Analysis (PCA) will be used. The algorithm tries to project the variables to a new coordinate system where the axes, the principal components, have a high variance. Each axis of the PCA represents a linear combination of the input features and there are ways to get even more information from these linear combinations. This will be neglected for now because then the PCA has to be programmed a little different and we will only use the power of PCA to reduce dimensions.

To use a PCA first an *PCA* instance is initialized with two components and then is fitted to the composition dataframe and finally it is transformed into the 2D space.

```
pca = PCA(n_components=2)
reducer_01 = pca.fit_transform(composition)
```

An alternative algorithm is the t-Distributed Stochastic Neighbor Embedding (TSNE). It searches for a probability distribution in the high dimension which data points are neighboring and the algorithm tries to project the resulting probability distribution down to the 2D space. The programming procedure is apart from the arguments set the same like for the PCA above.

```
tsne = TSNE(n_components=2, random_state=42, init='pca')
reducer_02 = tsne.fit_transform(composition)
```

It is now possible to plot the dimensional reduction from both algorithms and color the data points according to the cluster that they belong to (Figure 3). From the PCA in the left graph it seems that there is not too much variance in the data at all and all the clusters are nicely separable. Also the TSNE algorithm nicely reduces the 68 dimensional data into the plane and separates them into the calculated clusters. Of course it is not perfect but especially for TSNE no hyperparameters were adjusted which could even improve the separability.

```
fig, axs = plt.subplots(1,2,figsize=(15,9))
axs[0].set_title('Principal Components')
axs[1].set_title('t-SNE')
plot1 = axs[0].scatter(reducer_01[:,0], reducer_01[:,1],
c=cluster_predictions, cmap=plt.cm.get_cmap('viridis', 7))
plot2 = axs[1].scatter(reducer_02[:,0], reducer_02[:,1],
c=cluster_predictions, cmap=plt.cm.get_cmap('viridis', 7))
fig.colorbar(plot2, ax=axs);
```

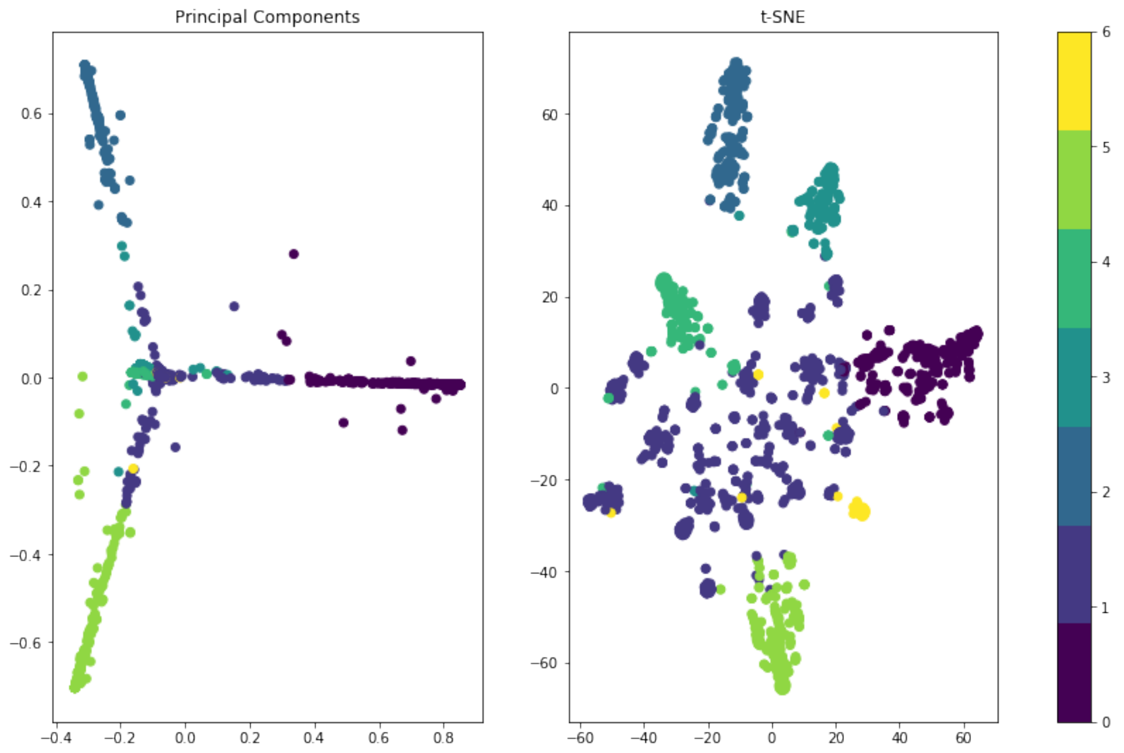


Figure 3: Dimensional reduction of the dataset by Principal Components Analysis (left), t-Distributed Stochastic Neighbor Embedding (right) and superposition the results from K-Means clustering (color bar).

What is lost by the dimensional reduction algorithms is the information about the chemistry and the composition. But the original composition can be colored with respect to the cluster number we found (Figure 4). When we do so we can for example recognize that one cluster (first row, first column, cluster 0) is rich in Magnesium, one cluster (first row, second column, cluster 5) includes lots of Calcium and one cluster (second row, second column, cluster 3) is rich in Aluminium. With this neat little trick we know now much more about the elemental composition of each of the clusters we found. It is clear to see that every cluster always contains more than one element. The clustering is not heading for classes containing just one element but for multi-component catalyst classes. This makes the interpretation sometimes difficult.

Coming back to the original publication [10] and the proposed 18 key elements for the OCM reaction and comparing them with the elements available in the largest proportions in this study. It shows that Sr, Ba, Mg Ca, La, Nd, Sm, Mn, Li, Na, Cl are present in both studies. So even by looking at just the data without any prior knowledge about OCM we would be able to make an educated guess for some OCM catalysts. Looking at the hypothesis from Schmack et al. [1] about one stable oxide support and one active species that could form a carbonate we could also guess a support like Alumina and active metals like Li, Mg, Ca Sr and Ba from the visualizations we just did. Schmack et al. also verified their hypothesis experimentally and an initial guess based on the cluster mapping below would be a valuable experimental starting point.

```
count = 0
elements =
composition.mean(axis=0).sort_values(ascending=False).head(9).keys()

fig, axs = plt.subplots(3,3, figsize=(15,12))
fig.subplots_adjust(wspace=0.3, hspace=0.3)

for ax in axs.flat:
    ax.set_title(elements[count])
    ax.set_ylabel('molar amount, %')
    ax.set_ylabel('S(C2), %')
    plot = ax.scatter(data_cleaned[elements[count]], data_cleaned['S(C2),
%'],
                      c=cluster_predictions, cmap=plt.cm.get_cmap('viridis',
7))
    count += 1
cb = fig.colorbar(plot, ax=axs)
plt.savefig('toc.png');
```

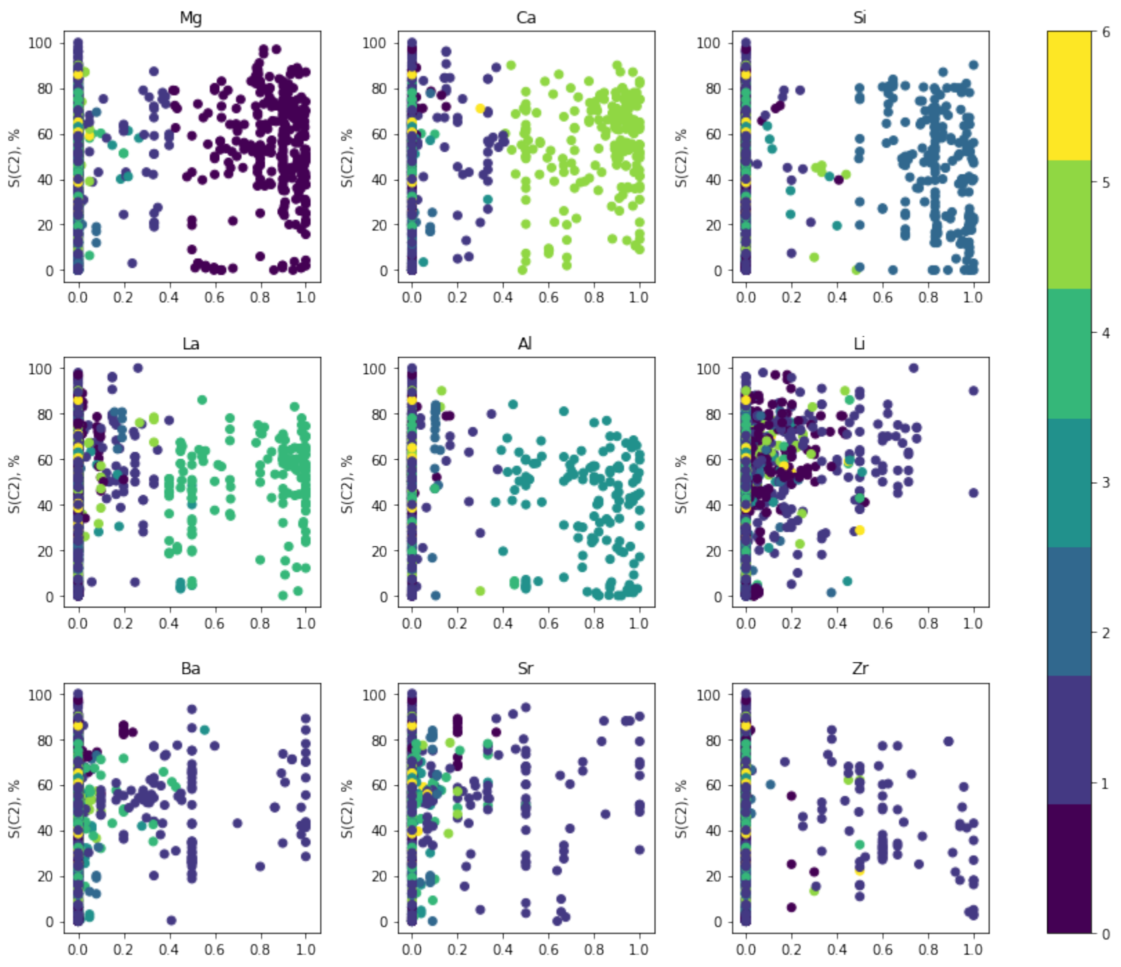


Figure 4: Coloring the elements occurring with highest amount in the dataset with respect to the cluster analysis.

10.1002/cctc.202000234

Supervised Learning

So now we gained quite some insight into the dataset even if we have a 68 dimensional dataset but now it is time to use Supervised algorithms to make predictions based on that dataset. It can be quite challenging to choose a good algorithm but there are some guidelines like the one from Scikit-Learn [22] which mentions more algorithms like the ones used in this study.

For most Supervised algorithms it is common practice to split the dataset into a training set and a test set. The idea behind is that the test set tries to mimic new data that the trained algorithm has never seen it in order to evaluate its performance. Of course it is possible to do the splitting, shuffeling and so on by hand but there is a method available from Scikit-Learn called *train_test_split*.

```
from sklearn.model_selection import train_test_split
```

So first we put all the features, the composition, into a feature array *X* and fill empty values with zeros. In the same manor we put the selectivity to C2 products into a target array *y* and divide all values by 100 to be in the range between 0 and 1. The feature and target arrays are then split into a training and a test set with *train_test_split*. Very often the shuffling of the data before the split is an issue and so it is here. Fixing the *random_state* for the shuffling to 42 leads to more stable results and is already part of the hyperparameter tuning. The intention behind this is the same as fixing the global random seed. A typical test set contains between 20 and 30% of the samples. More samples in the test set makes the risk of overfitting higher because of a smaller training set.

```
X = composition.fillna(value=0).astype(np.float)
y = data_cleaned.iloc[:,-2].fillna(value=0).astype(np.float)/100

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Support Vector Machine

Now that we have a decently split dataset we will use a Support Vector Machine (SVM, SVR) [21] as a first model. Unfortunately the default parameters are not very useful so they have to be adjusted. Therefore the function *RandomizedSearchCV* will used to help us with the hyperparameter search.

```
from sklearn.svm import SVR
from sklearn.model_selection import RandomizedSearchCV
```

Although we could just initiate an instances of the SVM and set some values like before, fit to the data and look at the score and do so as long as we have some good values it would be a little tedious. Therefore hyperparameter optimization is of great importance for ML. To make the search for good hyperparameters easier we can make use of one of the methods available in Scikit-Learn like *RandomSearchCV*. To do so we have to create a dictionary with the hyperparameters to search, then an instance of the SVM is initiated with static parameters that will not be changed during the search. Next an instance of *RandomizedSearchCV* will be initiated with the SVM and the hyperparameters as arguments and then upon the use of the *fit* method this instance will use the training data to look for the best hyperparameters with respect to the score. There are other methods around like *GridSearchCV*. The difference is that the randomized search takes random combinations from the proposed hyperparameters and the grid methods tries all in a brute force fashion. The randomized search is chosen here because it is the faster one when no parameters are known.

```
hyper_params_svr = {'gamma':np.random.normal(12, 2, 10000),
                    'C':np.random.normal(1, 0.2, 10000)
                   }

svr_tune = SVR('rbf')
g_search = RandomizedSearchCV(svr_tune, hyper_params_svr, cv=5, n_jobs=-1,
random_state=42)
g_search.fit(X_train, y_train);
```

This search leads to a C constant of near 1 and a gamma value of about 13.

```
print('Best Estimator ', g_search.best_estimator_)
```

```
Best Estimator  SVR(C=1.3052088684920178, cache_size=200, coef0=0.0,
degree=3, epsilon=0.1,
    gamma=12.393654443958436, kernel='rbf', max_iter=-1, shrinking=True,
    tol=0.001, verbose=False)
```

Now we fix the parameters to the best found ones and initiate a new SVM instance with exactly these values.

```
best_C = g_search.best_estimator_.C
best_gamma = g_search.best_estimator_.gamma

svr_rbf = SVR(kernel='rbf', C=best_C, gamma=best_gamma)
```

And now we train the algorithm again on the training dataset.

```
y_svr = svr_rbf.fit(X_train, y_train)
```

With the ready trained SVM at hand we can now make predictions first on the training set and then on the up to now unknown test dataset. To see the performance of the trained algorithm we take a look at the scores.

```
predict_svr_train = y_svr.predict(X_train)
predict_svr = y_svr.predict(X_test)

print('Score on training set:', y_svr.score(X_train, y_train))
print('Score on test set:', y_svr.score(X_test, y_test))
```

```
Score on training set: 0.6279883460520106
Score on test set: 0.27599607709842233
```

10.1002/cctc.202000234

A score of about 62% is at least better than guessing the right combination from 68 elements but not really good and the score for the test set is even worse. Visualizing the trend in a parity plot shows the same (Figure 5).

```
plt.figure(figsize=(12,9))

plt.scatter(y_test, predict_svr)
plt.title('Support Vector Machine, S(C2), %')
plt.xlabel('True Values [%]')
plt.ylabel('Predictions [%]')
plt.axis('equal')
plt.xlim(plt.xlim())
plt.ylim(plt.ylim())
plt.plot([-1000, 1000], [-1000, 1000]);
```

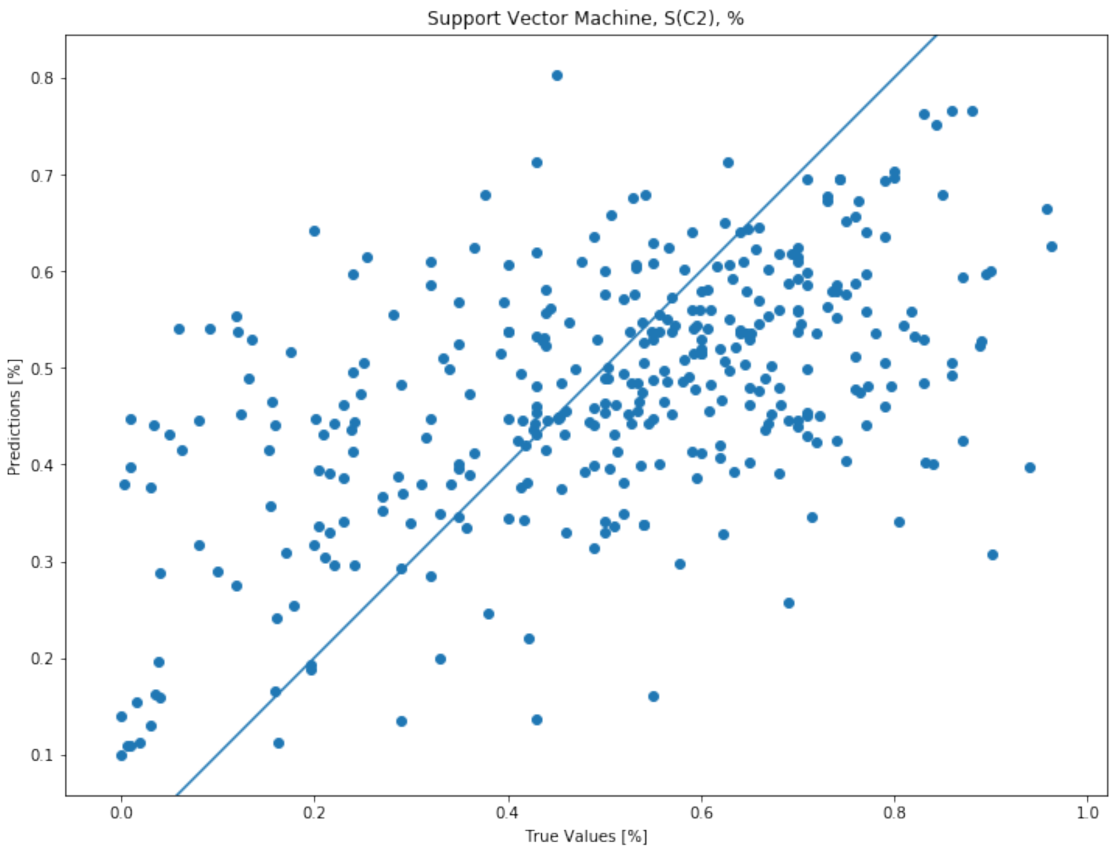


Figure 5: Parity plot for the results of the Support Vector Machine on the test set.

Random Forests

Maybe it is not the right algorithm for this kind of dataset, and this will happen quite often on real live examples like here. SVM is an algorithm that is often well suited for smaller datasets like this one here. Another aspect that is not regarded in this study is the data itself. Here only the composition of the catalyst is used but the reaction parameters are completely neglected. For example, the reaction temperature is a very important variable in OCM and there is a high probability that including the reaction parameters will improve the study. Please keep that in mind when we stay now with the elemental composition. We will try a Random Forests algorithm instead of an SVM. Random Forests is known to work even with very large datasets, and it is quite popular in Chemistry as there is a good chance that the result is easy to interpret.

```
from sklearn.ensemble import RandomForestRegressor
```

After the import of the *RandomForestRegressor* we can now look for decent hyperparameters like we did before by first defining a dictionary of possible parameters, then the regressor and after that the search algorithm will be initiated. Finally the regressor will be fitted to the training data.

```
hyper_params_rdf = {'n_estimators':np.arange(1, 25, 1)}

rdf_tune = RandomForestRegressor()
g_search_rdf = RandomizedSearchCV(rdf_tune, hyper_params_rdf, cv=5, n_jobs=-
1, random_state=42)
g_search_rdf.fit(X_train, y_train);
```

We always get *n_estimators* of around 20 as best result and this is now fixed for the training of the final Random Forest.

```
print('Best Estimator ', g_search_rdf.best_estimator_)

estimators = g_search_rdf.best_estimator_.n_estimators
```

```
Best Estimator  RandomForestRegressor(bootstrap=True, criterion='mse',
max_depth=None,
                      max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=22,
                      n_jobs=None, oob_score=False, random_state=None,
                      verbose=0, warm_start=False)
```

Now we can again fit the final regressor to the training data.

```
regressor = RandomForestRegressor(n_estimators=estimators, random_state=42)
regressor.fit(X_train, y_train);
```

Of course we can make predictions with the trained algorithm and calculate the score for the training and the test dataset.

```
y_pred_train = regressor.predict(X_train)
y_pred = regressor.predict(X_test)

print('Score on training set:', regressor.score(X_train, y_train))
print('Score on test set:', regressor.score(X_test, y_test))
```

```
Score on training set: 0.8286827831069986
Score on test set: 0.2853065599532085
```

Plotting the results from Random Forests looks a little bit better though (Figure 6).

```
plt.figure(figsize=(12,9))

plt.scatter(y_test, y_pred)
plt.title('Random Forests, S(C2), %')
plt.xlabel('True Values [%]')
plt.ylabel('Predictions [%]')
plt.axis('equal')
plt.xlim(plt.xlim())
plt.ylim(plt.ylim())
plt.plot([-1000, 1000], [-1000, 1000]);
```

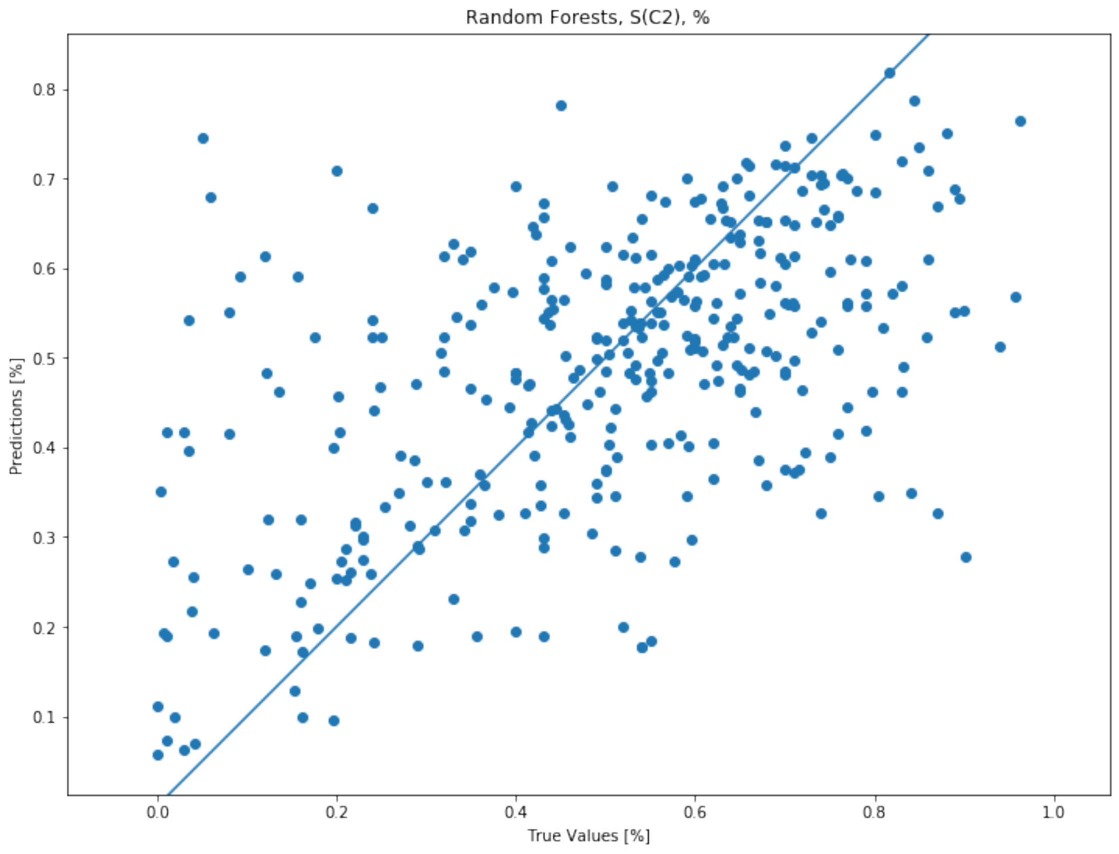


Figure 6: Parity plot for the results of the Random Forests algorithm on the test set.

To visualize the performance of both of the algorithms with respect to each other some more metrics are calculated. The first metric is a simple *mean_squared_error* and the second the *r2_score* which is basically the same score we already calculated but here we explicitly use the *r2_score* to be concise. The scores are calculated for both the training and the testing datasets for comparison.

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

mse_rdf_train = mean_squared_error(y_train,y_pred_train)
mse_rdf_test = mean_squared_error(y_test,y_pred)
```

```
mse_svr_train = mean_squared_error(y_train,predict_svr_train)
mse_svr_test = mean_squared_error(y_test,predict_svr)

r2_rdf_train = r2_score(y_train,y_pred_train)
r2_rdf_test = r2_score(y_test,y_pred)

r2_svr_train = r2_score(y_train,predict_svr_train)
r2_svr_test = r2_score(y_test,predict_svr)
```

What we see when plotting the metrics is again that the Random Forests is slightly better than the SVM (Figure 7). Another thing to mention is that the scores on the test data is for both a little worse than for the training data. This can be a hint that both algorithms are not overfitting and are still able to generalize on new and unknown data which exactly the aim of Supervised Learning.

```
import matplotlib.patches as mpatches

ann = mpatches.Patch(color='k', label='Random Forests')
svr = mpatches.Patch(color='m', label='Support Vector Regression')

names = ['train', 'test', 'train', 'test']
pos = range(len(names))
colors = ['k', 'k', 'm', 'm']
rotation = 0

plt.figure(figsize=(12,9))
plt.suptitle('Overall metrics', y=1.02)
plt.subplot(121)
plt.bar(pos, [mse_rdf_train, mse_rdf_test, mse_svr_train, mse_svr_test], 
color=colors)
plt.xticks(pos, names,rotation=rotation)
plt.ylabel('mean squared error')

plt.subplot(122)
plt.bar(pos, [r2_rdf_train,r2_rdf_test, r2_svr_train, r2_svr_test], 
color=colors)
plt.xticks(pos, names, rotation=rotation)
plt.ylabel('R2 score')

plt.tight_layout()
plt.legend(loc='upper center', handles=[ann, svr], ncol=3, bbox_to_anchor=(-
0.1, -0.05));
```

10.1002/cctc.202000234

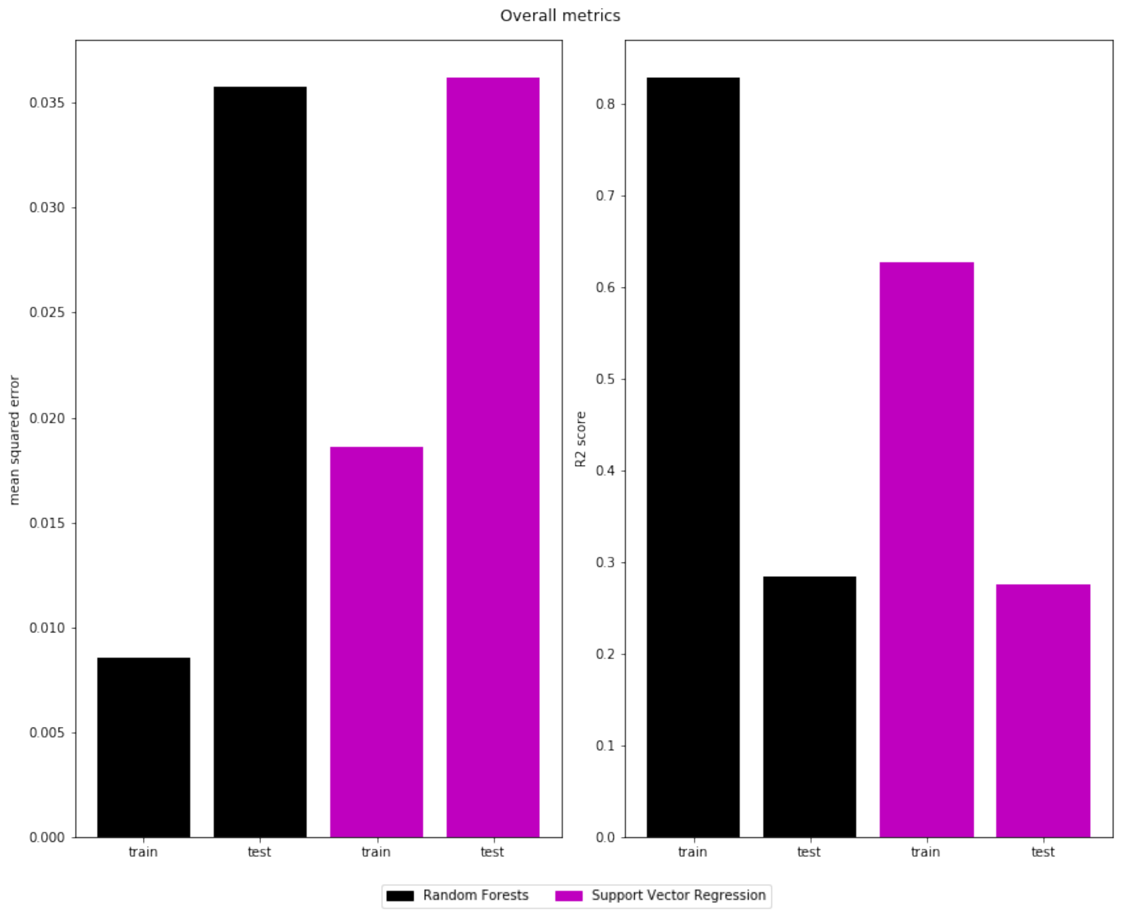


Figure 7: Overall metrics for the Support Vector Machine (black) and the Random Forests algorithm (magenta) on the training and the test set.

But are there measures to be sure that the algorithms are not overfitting? The randomized search includes a concept called cross-validation. For each tested parameter combination, the training data is split into parts, five parts is a common choice. Four of the parts serve as new training set and one as test set for the parameter search. Then every parameter combination is tested five times and every time another fraction becomes the test set. Then we can evaluate the mean score from each of the calculations and find the best combination.

When we take a look at the mean test score results (Figure 8) for the Support Vector Machine we see that the score is always around 25% and this is close to the value of the final version of the SVM with respect to the test set. For Random Forests it looks a little different because the scores vary much more. Good scores with about 27 to 30% come close to the final regressor on the test set. The bad values are a hint that for Random Forests a good performance depends very much on the shuffling of this actual dataset and indeed when the random state in the *train_test_split* is set to different seeds the values vary even more. We will continue with a random seeds of 42 and more stable results. After evaluating the cross-validation results we can now be quite sure that both algorithms are not overfitting.

```
svr_mean_cv_score = g_search.cv_results_['mean_test_score']
rdf_mean_cv_score = g_search_rdf.cv_results_['mean_test_score']
```

```
plt.figure(figsize=(12,9))

plt.subplot(121)
plt.xlabel('Cross-validation step')
plt.ylabel('Mean test score from cross-validation')
plt.title('Support Vector Machine')
plt.bar(x=range(0, len(svr_mean_cv_score)), height=svr_mean_cv_score)
plt.subplot(122)
plt.xlabel('Cross-validation step')
plt.title('Random Forests')
plt.bar(x=range(0, len(rdf_mean_cv_score)), height=rdf_mean_cv_score);
```

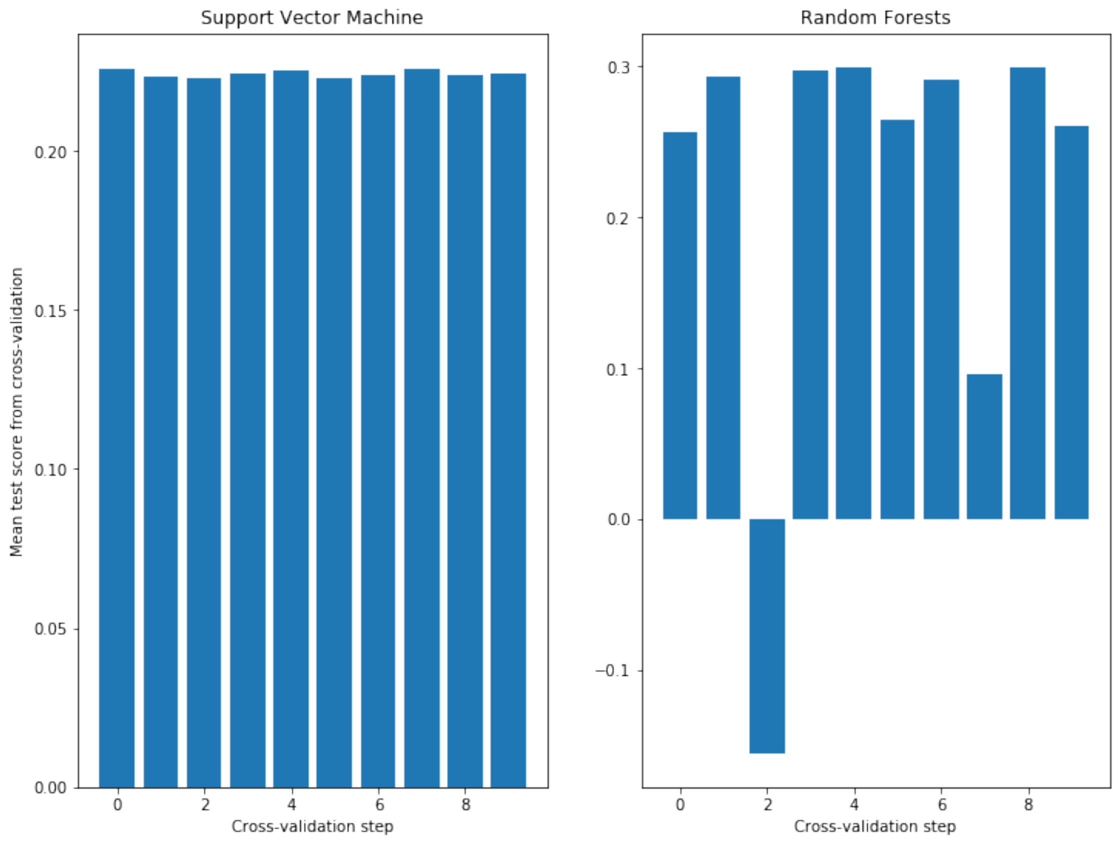


Figure 8: Results of the cross-validation of the Support Vector Machine (left) and the Random Forests algorithm (right).

Especially Random Forests can give some more information about the features we put into the algorithm. For example we can ask which of the 73 elements in the dataset are the most decisive and have the highest impact on the predictions of the algorithm. To get this information we can use the *SelectFromModel* method from Scikit-Learn. It gives us just a selection of the most important features in the dataset which is in this case a subgroup of about 20 elements and the output is ranked so that the feature with the highest importance comes first. Again, this observation is well in line with the findings from [10] about the key elements and from [1] for the combinations of support and active components. For example Ca on Alumina would be a good experimental candidate and the experimental data from [1] supports this guess. This is the initial step of a process called feature selection and becomes more and more important the more features (input variables) a dataset has. Of course it can be beneficial to have more features but at some point the calculation times then get too long and it becomes mandatory to select the most important features.

```
from sklearn.feature_selection import SelectFromModel

select_features =
SelectFromModel(RandomForestRegressor(n_estimators=estimators))
select_features.fit(X_train, y_train)
selected_features= X_train.columns[(select_features.get_support())]

print(selected_features)

Index(['Al', 'Ba', 'C', 'Ca', 'Ce', 'Cl', 'Co', 'Fe', 'K', 'La', 'Li', 'Mg',
       'Mn', 'Na', 'Ni', 'P', 'Pb', 'Pr', 'Si', 'Sr', 'Ti', 'W', 'Zr'],
      dtype='object')
```

Up to now we could not beat 80% accuracy of the Random Forest algorithm on the training set. Can we get any better?

Deep Learning to the rescue?

What can we try now to improve the quality of our prediction? Well, everybody is talking about Deep Learning nowadays, so maybe we should try this? We will try soon but first, what is it all about? Machine Learning is available now for quite a while and indeed some algorithms are also pretty old. But there are also more techniques like for example Neural Networks (NN). Also NN's exist for quite some time and a simple NN looks more or less this way:

- An input layer of artificial neurons,
- a hidden layer where the values of the input layer are multiplied with a weight array. If the value in a neuron is high enough it gets activated by an activation function and the value of this neuron propagates to
- an output neuron delivering the final value.

In principle one can model any arbitrary function with a NN but it was hard to adjust the weighting arrays in the middle of the networks in the past. But then an idea came up how to adjust the weights with something called back-propagation [22] and combine it with a gradient descent optimization. This made NN's better usable but is was still computational expensive until the use of accelerator units like Graphics Processing Units (GPUs) which lead to a tremendous speedup. This enabled researchers to add more and more hidden layers to their NNs and creating more complex structures. If we now have more than one hidden layer in a NN we are generally talking about Deep Learning [23]. We will now try to see if Deep Learning can solve our problem more efficient than classical methods.

Nowadays it is not important to program a NN from scratch but there are libraries out to do the heavy lifting for the researcher. Examples are PyTorch (by Facebook) or Tensorflow (by Google) [24] but there are more libraries available. Here we will use Tensorflow with the Keras backend. This means the AI library Tensorflow will do the calculations and the Keras library helps to define the structure of the NN. This is a user friendly way to get started. But first the libraries needs to be imported.

```
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
```

The next line is intended to clear the Tensorflow model. This is just done for the manuscript to have a fresh model in each pass of code.

```
tf.keras.backend.clear_session()
```

The imports could be a little easier as it would be sufficient to just import *tensorflow* with a suitable alias but this would lead to much longer code lines in the box below. Moreover it is a little more narrative to import additional libraries with a proper alias. Now we define a model with the *keras* functional API, which is an easy way to get started. The *model* contains

- one input layer with 73 neurons
- two hidden layers again with 73 neurons each
- one output layer
- Dropout layers after each dense layer
- ReLu is used as activation function.

Dense layer means that every neuron in one layer is connected with every neuron in the next layer, this is often displayed by an arrow. Dropout layers can be imagined like a stroke. When the NN is trained the connection to some neurons is cut in each training pass forcing the NN to learn alternative ways through the network. This is a measure against overfitting. The output layer gathers everything together and tells us the selectivity we are aiming at.

The structure of a NN is one of the main hyperparameters to adjust in such a study.

```
model = keras.Sequential([
    layers.Dense(73, activation='relu', input_shape=[X_train.shape[1]]),
    layers.Dropout(0.2),
    layers.Dense(73, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(73, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1)
  ])
```

Now the model gets compiled with a *loss* function being the measure of optimization, in this case it is mean absolute error *mae*. Moreover we need a suitable optimizer. For a regression problem like we are dealing with right here *rmsprop* works good. Of course the parameters can always be optimized but for the time being we can imagine *rmsprop* as a suitable version of gradient descent.

```
model.compile(loss='mae', optimizer='rmsprop');
```

Now we can take a look what the NN *model* looks like.

10.1002/cctc.202000234

```
model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense (Dense)                (None, 73)                5037      
_________________________________________________________________
dropout (Dropout)            (None, 73)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 73)                5402      
_________________________________________________________________
dropout_1 (Dropout)          (None, 73)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 73)                5402      
_________________________________________________________________
dropout_2 (Dropout)          (None, 73)                0         
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 74        
=================================================================
Total params: 15,915
Trainable params: 15,915
Non-trainable params: 0
_________________________________________________________________
```

Now it is time to train the NN with more than 15.000 parameters. To do so we call *fit* on the model and feed it with the training features and targets. The training iterates several times over the data which is called *epochs* and is set to 500. In the *validation_split* a fraction of the training data is put aside to monitor the performance of the training. The loss function will always just go down but on the validation dataset we can have indications that we are for example overfitting or underfitting. The use of a validation set follows the same idea as the test set, but it is only used internally for the fitting of the NN and not equal to the test set. Although a size of about 20 to 30% from the training set would be good for the validation set one often finds smaller validation datasets. Especially when working with smaller datasets this is quite common in order not to lose to many samples that are needed for a proper training. Here 10% were reasonable.

```
model.fit(X_train, y_train, epochs=500, validation_split = 0.1, verbose=0);
```

Now we make a plot of the loss function and the validation loss function (Figure 9). We can see that over the epochs the loss function always goes down, leading to a higher accuracy. On the other side the validation loss has a slight upward trend after about 70 epochs. This can be a hint that the NN is overfitting which means it learns by heart because it is large enough to store all training data in the huge amount of parameters. The validation data mimics unknown test data and therefore we will get a low score on the test data with a NN that is overfitting but it will look good on the training data.

```
plt.figure(figsize=(12,6))

plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.plot(model.history.epoch, np.array(model.history.history['loss']),
'k',label='Train set')
plt.plot(model.history.epoch, np.array(model.history.history['val_loss']),
'm', label='Validation set')
plt.legend()

plt.tight_layout()
```

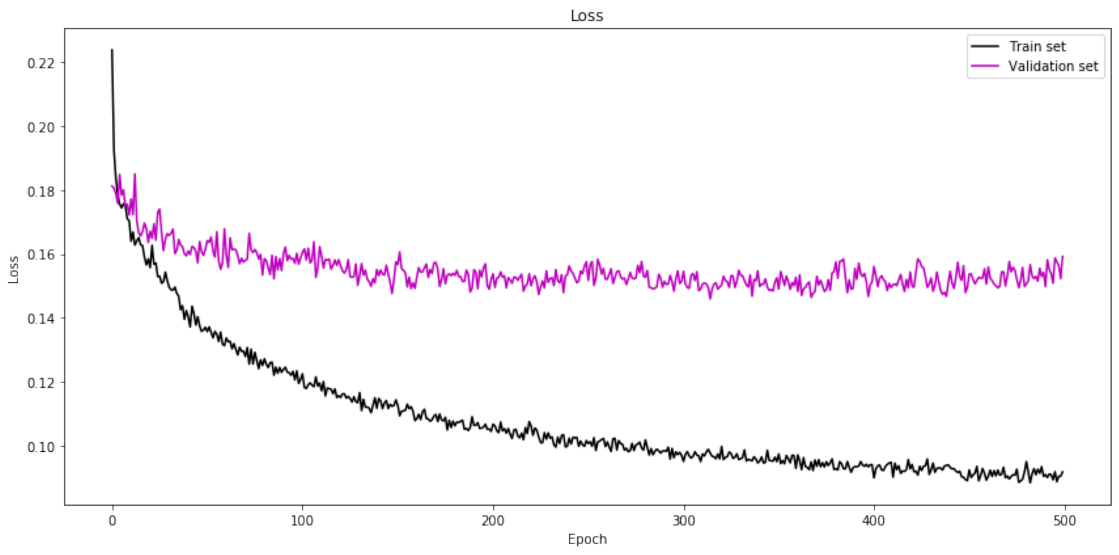


Figure 9: Course of the training and the validation loss functions of the proposed Neural Network during the training.

To test this, we make some predictions with the ready trained NN on the training and the test data and get the score.

```
predict_on_train = model.predict(X_train).flatten()
predict_on_test = model.predict(X_test).flatten()

print('Score on training set:', r2_score(y_train, predict_on_train))
print('Score on test set:', r2_score(y_test, predict_on_test))
```

```
Score on training set: 0.6854732737389186
Score on test set: 0.1684929186415739
```

And we get a score slightly around 70% on the training data and about 23% on the test data. So the training data looks good but the test data not. What went wrong? When we do the training again with just 80 epochs we will end up with a 50% score for the training data and around 22% for the test data which is a clear hint that we are overfitting with the model we build and training it over 500 epochs.

When we now make a parity plot for the test data it does not look much better than for example the SVM (Figure 10). On the other hand the tree based algorithm looks better and is easier to interpret than the NN. Of course the NN is not fully optimized by the author but it clearly shows that just using Deep Learning is not a warranty for success and can make the study even very complicated.

10.1002/cctc.202000234

```
plt.figure(figsize=(12,9))

plt.scatter(y_test, predict_on_test)
plt.title('S(C2), %')
plt.xlabel('True Values [%]')
plt.ylabel('Predictions [%]')
plt.axis('equal')
plt.xlim(plt.xlim())
plt.ylim(plt.ylim())
plt.plot([-1000, 1000], [-1000, 1000]);
```

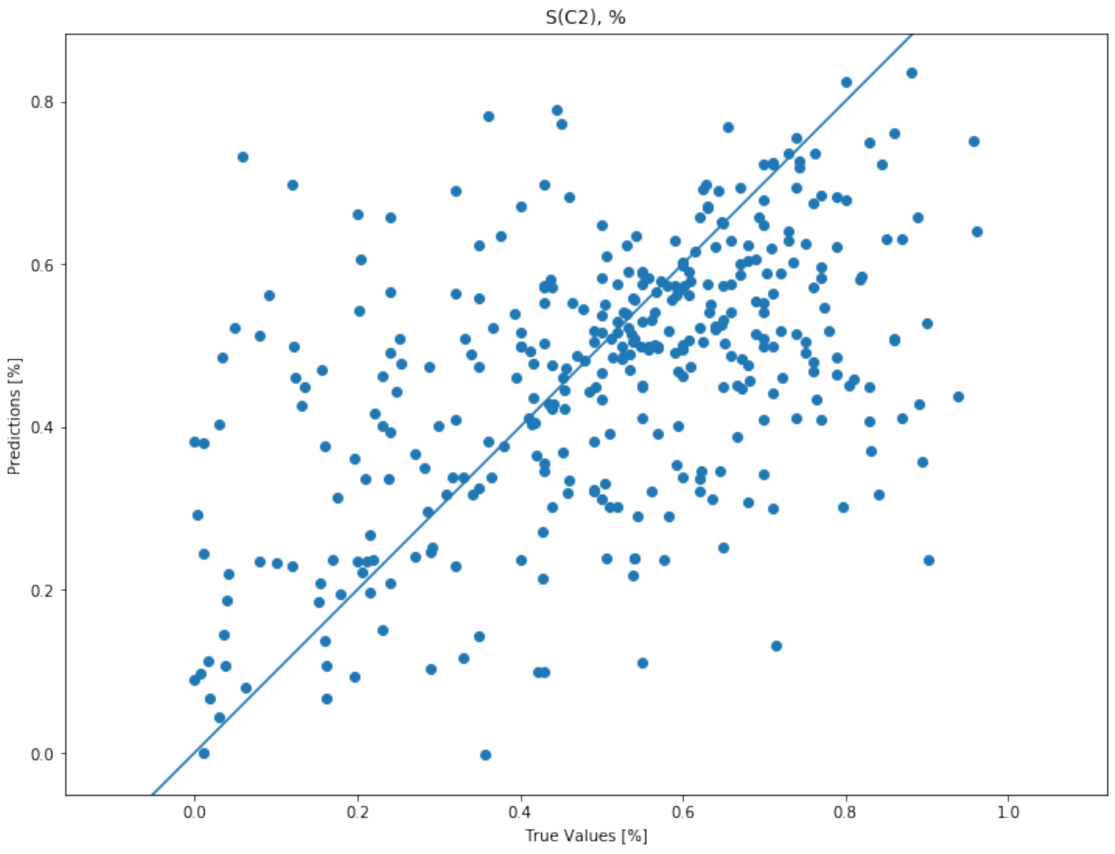


Figure 10: Parity plot for the results of the Neural Network on the test set.

Conlusions

I hope I could guide you through a Data Science study based on literature data illustrating also some pitfalls in such a study. I hope I could show you that you have to spend much time about the data in Data Science with preprocessing, inspection and visualization of the data and that you will need your scientific knowledge for the science fraction in the phrase Data Science to gather more knowledge. I wanted to illustrate that the tuning of the various hyperparameters is very important but can also be very tedious. Moreover I wanted to illustrate that Deep Learning is indeed very powerful but because of its complexity it can be prone to errors and has to be used prudently. Finally I hope I could convince you that Data Science and Machine Learning can be a valuable addition also to a Chemists toolbox and that there is more than Excel and Origin out there.

Literature

[1] R. Schmack, A. Friedrich, E. V. Kondratenko, J. Polte, A. Werwatz, R. Kraehnert, “A meta-analysis of catalytic literature data reveals property-performance correlations for the OCM reaction”, Nature Communications 2019, 10, 441

[2] J. K. Nørskov, T. Bligaard, “The Catalyst Genome”, Angew. Chem. Int. Ed. 2013, 52, 776-777

[3] K. Takahashi, L. Takahashi, I. Miyazato, J. Fujima, Y. Tanaka, T. Uno, H. Satoh, K. Ohno, M. Nishida, K. Hirai, J. Ohyama, T. N. Nguyen, S. Nishimura, T. Taniike, “The Rise of Catalyst Informatics: Towards Catalyst Genomics”, ChemCatChem 2019, 11, 1146-1152

[4] K. Takahashi, Y. Tanaka, “Materials informatics: a journey towards material design and synthesis”, Dalton Trans. 2016, 45, 10497-10499

[5] K. Takahashi, L. Takahashi, “Creating Machine Learning-Driven Material Recipes Based on Crystal Structure”, J. Phys. Chem. Lett. 2019, 10, 2, 283-288

[6] K. Takahashi, L. Takahashi, “Data Driven Determination in Growth of Silver from Clusters to Nanoparticles and Bulk”, J. Phys. Chem. Lett. 2019, 10, 14, 4063-4068

[7] P. Schlexer Lamoureux, K. T. Winther, J. A. Garrido Torres, V. Streibel, M. Zhao, M. Bajdich, F. Abild-Pedersen, T. Bligaard, “Machine Learning for Computational Heterogeneous Catalysis”, ChemCatChem 2019, 11, 3581-3601

[8] R. Palkovits, S. Palkovits, “Using Artificial Intelligence To Forecast Water Oxidation Catalysts”, ACS Catal. 2019, 9, 8383-8387

[9] J. N. Wei, D. Duvenaud, A. Aspuru-Guzik, “Neural Networks for the Prediction of Organic Chemistry Reactions”, ACS Cent. Sci. 2016, 2, 10, 725-732

[10] U. Zavyalova, M. Holena, R. Schlögl, M. Baerns, “Statistical Analysis of Past Catalytic Data on Oxidative Methane Coupling for New Insights into the Composition of High-Performance Catalysts”, ChemCatChem 2011, 3, 1935–1947.

[11] E. V. Kondratenko, M. Schlüter, M. Baerns, D. Linke, M. Holena, “Developing catalytic materials for the oxidative coupling of methane through statistical analysis of literature data”, Catal. Sci. Technol. 2015, 5, 1668-1677

[12] T. N. Nguyen, T. T. Phuong Nhat, K. Takimoto, A. Thakur, S. Nishimura, J. Ohyama, I. Miyazato, L. Takahashi, J. Fujima, K. Takahashi, T. Taniike,‘High-Throughput Experimentation and Catalyst Informatics for Oxidative Coupling of Methane’, ACS Catal. 2020, 10, 921-932

[13] G. van Rossum, “Python Tutorial”, Centrum Voor Wiskunde En Informatica (CWI) Amsterdam 1995.

[14] W. McKinney, “Data Structures for Statistical Computing in Python”, Proc. of the 9th Python in Science Conf. 2010, 51-56

[15] T. E. Oliphant, “A guide to NumPy”, USA: Trelgol Publishing 2006

[16] S. van der Walt, S. C. Colbert, G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation", Comp. Sci. Eng. 2011, 13, 22-30

[17] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Comput. Sci. Eng. 2007, 9, 3, 90-95, 2007.

[18] L. van der Maaten, G. E. Hinton, "Visualizing Data using t-SNE" J. Mach. Learn. Res. 2008, 9, 2579–2605.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. "Scikit-learn: Machine Learning in Python", J. Mach. Learn. Res. 2011, 12, 2825–2830

[22] https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

[21] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, "LIBLINEAR: A Library for Large Linear Classification", J. Mach. Learn. Res. 2008, 9, 1871–1874

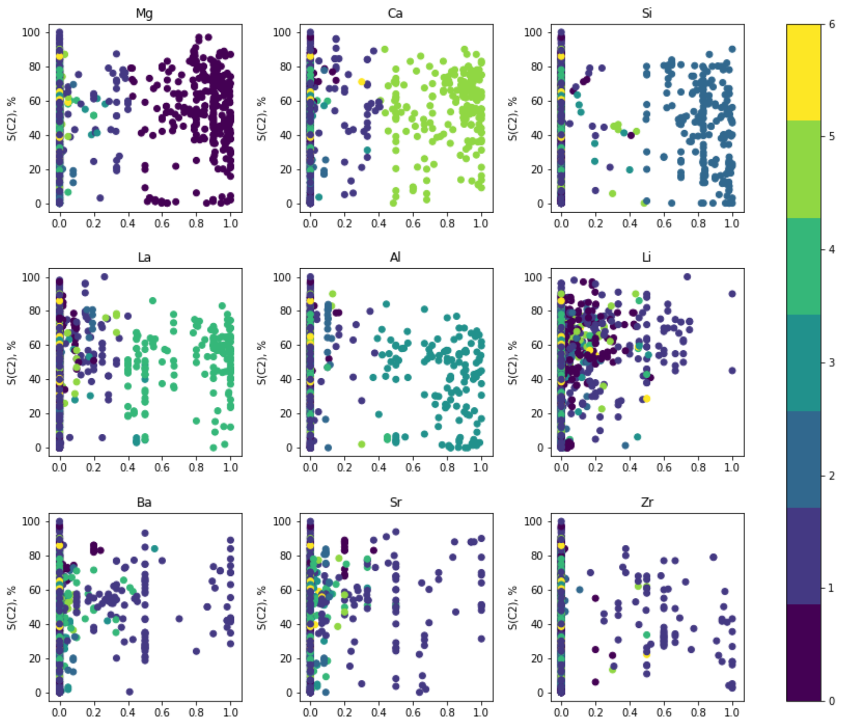
[22] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning representations by back-propagating errors", Nature 1986, 323, 533-536

[23] Y. LeCun, Y. Bengio, G. Hinton, "Deep Learning", Nature 2015, 521, 436-444

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems", 2015.

10.1002/cctc.202000234

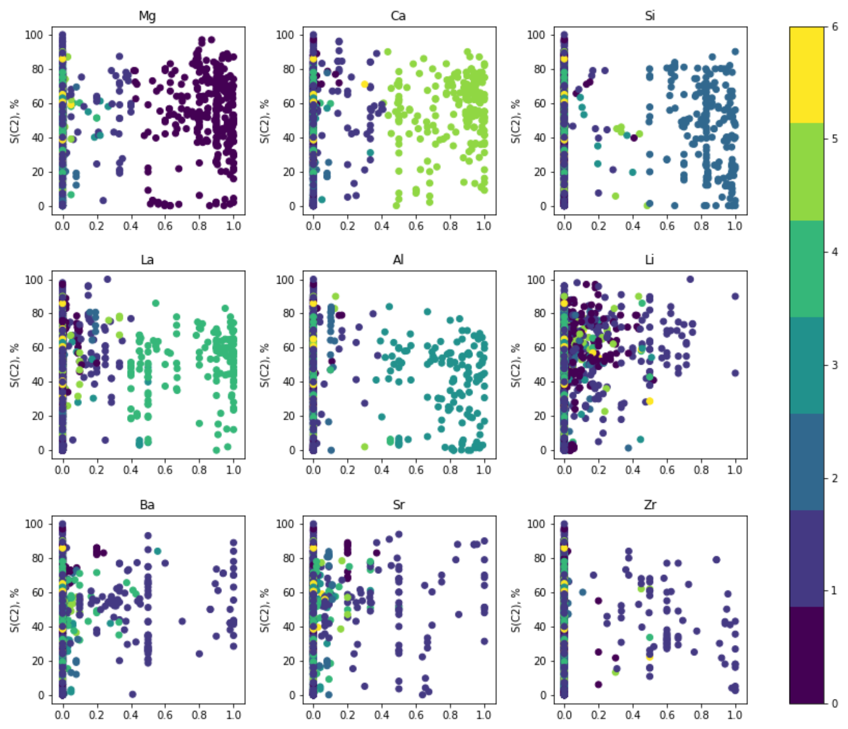
Frontispiece



10.1002/cctc.202000234

Table of Contents

In a tutorial style article with programming code a Machine Learning workflow will be illustrated based on literature known data for the oxidative coupling of methane.


Mg
Ca
Si
La
Al
Li
Ba
Sr
Zr
S(C2), %