CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep-konsep dari pembelajaran mesin untuk kemudian bisa diterapkan pada berbagai permasalahan– (C2);
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin sehingga bisa mendapatkan pemecahan masalah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan perbaikan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 2
Data Preprocessing

DESKRIPSI TEMA

Mahasiswa mempelajari tentang data, fitur, dan teknik-teknik pra proses data sebelum dijadikan masukkan dari mesin.

CAPAIAN PEMBELAJARAN MINGGUAN (SUB-CAPAIAN PEMBELAJARAN)

Mahasiswa dapat menjelaskan dan menerapkan teknik praproses data untuk menghasilkan masukkan yang layak dengan bahasa pemrograman python – SCPMK-02

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is absolutely critical that we make sure to examine and preprocess a dataset before we feed it to a learning algorithm.

Dealing with Missing Data

It is not uncommon in real world application that our samples are missing one or more values for various reasons. There could have been an error in the data collection process, certain measurement are not applicable, particular fields could have been simply left blank in survey, for example. We typically see missing values as the blank spaces in our data table or as placeholder string such as NaN (not a number). Before we discuss several techniques for dealing with missing values,

1. lets create a simple data from a CSV (comma-separated values) file to get a better graps of the problem.

```
from IPython.display import Image
%matplotlib inline
```

```
import pandas as pd
from io import StringIO
import sys

csv_data = \
'''A,B,C,D
1.0,2.0,3.0,4.0
5.0,6.0,,8.0
10.0,11.0,12.0,'''

df = pd.read_csv(StringIO(csv_data))
df
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

2. working with numpy array, it can sometimes be more convenient to preprocess data using pandas DataFrame.

```
#Access the underlying numpy array
#via the 'values' attribute
df.values
```

```
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6., nan,  8.],
       [10., 11., 12., nan]])
```

Eliminating Samples of Features With Missing Values

3. One of the easiest ways to deal with missing data is to simply remove the corresponding features (columns) or samples (rows) from the dataset entirely; rows with missing values can be easily dropped via the dropna methods :

```
# remove rows that contain missing values
df.dropna(axis=0)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

Using the preceding code, we read CSV_formatted data into a pandas DataFrame via the read_csv function and noticed that the two missing cells were replaced by NaN. The StringIO function in the preceding code example was simply used for the purposes of illustration. It allows us to read the string assigned to csv_data into a pandas DataFrame as if was regular CSV File on our hard drive.
For a larger DataFrame, it can be tedious to look for missing values manually; in this case, we can use the isnull method to return a DataFrame with Boolean values that indicate whether a cell contains a numeric value (False) or if data is missing (True).

4. Using the sum method, we can return the number of missing values per columns as follows :

```
df.isnull().sum()
```

```
A    0
B    0
C    1
D    1
dtype: int64
```

5. Although, scikit-learn was developed for Similarly, we can drop columns that have at least one NaN in any row by setting the axis argument to 1

```
# remove columns that contain missing values
df.dropna(axis=1)
```

	A	B
0	1.0	2.0
1	5.0	6.0
2	10.0	11.0

6. The dropna method supports several additional parameters that can come in handy :

```
# only drop rows where all columns are NaN
df.dropna(how='all')
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

```
# drop rows that have fewer than 3 real values
df.dropna(thresh=4)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

```
# only drop rows where NaN appear in specific columns (here: 'C')
df.dropna(subset=['C'])
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
2	10.0	11.0	12.0	NaN

Imputing Missing Values

Often the removal of samples of dropping of entire feature columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training samples in our data set. One of the most common interpolation techniques is mean imputation, where we simply replace the missing value by the mean value of the entire feature columns.

7. A convenient way to achieve this is by using the Imputer class from scikit-learn as shown in the following code :

```
# again: our original array
df.values
```

```
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6., nan,  8.],
       [10., 11., 12., nan]])
```

```
# impute missing values via the column mean

from sklearn.impute import SimpleImputer
import numpy as np

imr = SimpleImputer(missing_values=np.nan, strategy='mean')
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data
```

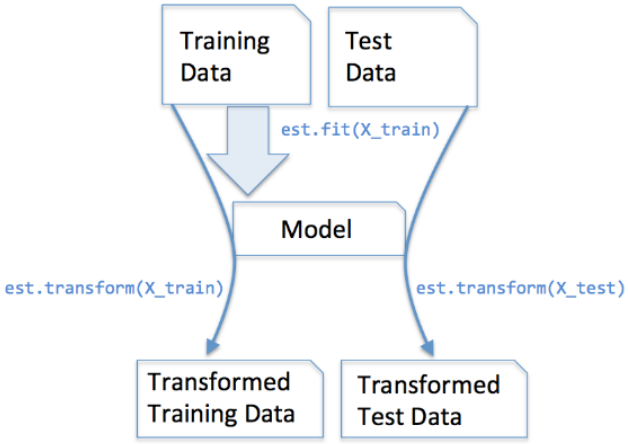
```
array([[ 1. ,  2. ,  3. ,  4. ],
       [ 5. ,  6. ,  7.5,  8. ],
       [10. , 11. , 12. ,  6. ]])
```

```
df.fillna(df.mean())
```

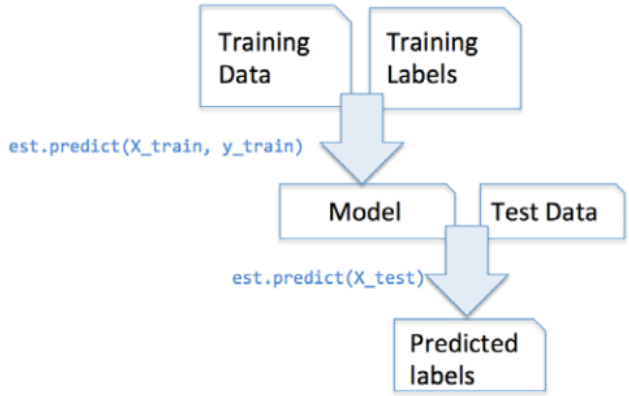
	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

Understanding the Scikit-Learn Estimator API

The two essential methods of those estimators are fit and transform. The fit method is used to learn the parameters from the training data, and the transform method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model. The following figure illustrates how a transformer fitted on the training data is used to transform a training dataset as well as a new test dataset:



Estimators have a predict method but can also have a transform method, as we will see later. As you may recall, we also used the fit method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new data samples via the predict method, as illustrated in the following figure:



Handling Categorical Data

So far, we have been working with numerical data. However, it is not uncommon that real-world datasets contain one or more categorical features columns. The categorical data type contain nominal and ordinal features.

8. Lets create a new data frame to illustrate the problem (different between data type)

```
import pandas as pd

df = pd.DataFrame([['green', 'M', 10.1, 'class2'],
                   ['red', 'L', 13.5, 'class1'],
                   ['blue', 'XL', 15.3, 'class2']])

df.columns = ['color', 'size', 'price', 'classlabel']
df
```

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

The newly created DataFrame contains a nominal feature (color), an ordinal feature (size) and numerical feature (price). The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column.

Mapping Ordinal Features

9. Make sure that the learning algorithms interprets the ordinal features correctly, we need to convert the categorical string values into integers. Thus, we have to define the mapping manually. In the example, lets assume that we know the difference between features.

```
size_mapping = {'XL': 3,
                'L': 2,
                'M': 1}

df['size'] = df['size'].map(size_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

10. If we want to transform the integer values back to the ordinal string representation at a later stage, we can simply define a reverse mapping dictionary inv_size_mapping. That can be used via the pandas map method on the transformed feature column similar to the size_mapping dictionary that we used previously.

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df['size'].map(inv_size_mapping)
```

```
0     M
1     L
2    XL
Name: size, dtype: object
```

Encoding Class Label

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously.

11. We need to remember that class labels are not ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels starting at 0 :

```
import numpy as np

# create a mapping dict
# to convert class labels from strings to integers
class_mapping = {label: idx for idx, label in enumerate(np.unique(df['classlabel']))}
class_mapping
```

```
{'class1': 0, 'class2': 1}
```

12. Next we can use the mapping dictionary to transform the class labels into integers :

```
# to convert class labels from strings to integers
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	1
1	red	2	13.5	0
2	blue	3	15.3	1

13. We can reverse the key – value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation :

```
# reverse the class label mapping
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

14. There is a convenient LabelEncoder class directly implemented in scikit-learn to achieve the same :

```
from sklearn.preprocessing import LabelEncoder

# Label encoding with sklearn's LabelEncoder
class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
y
```

```
array([1, 0, 1])
```

15. Note that the fit_transform method is just a shortcut for calling fit and transform separately and we can use the inverse_transform method to transform the integer class labels back into their original string representation :

```
# reverse mapping
class_le.inverse_transform(y)
```

```
array(['class2', 'class1', 'class2'], dtype=object)
```

Performing one-hot encoding on nominal features

16. We used the convenient LabelEncoder class to encode the string labels into integers. It may appear that we could use a similar approach to transform the nomimal color columns for our dataset, as follows :

```
X = df[['color', 'size', 'price']].values
color_le = LabelEncoder()
X[:, 0] = color_le.fit_transform(X[:, 0])
X
```

```
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

After executing the preceding code, the first column of the numpy array X now holds the new color values, which encoded as follows :

- Blue → 0
- Green → 1
- Red → 2

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, a learning algorithm will now assume that green is larger than blue, and red is larger than green. Although this assumption is incorrect, the algorithm could still produce useful results. However, those results would not be optimal.

17. The solution technique called one-hot encoding. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature columns. Here, we would convert the color feature into three new features (blue, green, red). Binary values can then be used to indicate the particular color of a sample. To perform this transformation, we can use the OneHotEncoder that is implemented in the scikit-learn.preprocessing module :

```
from sklearn.preprocessing import OneHotEncoder

X = df[['color', 'size', 'price']].values
color_ohe = OneHotEncoder()
color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
```

```
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

```
from sklearn.compose import ColumnTransformer

X = df[['color', 'size', 'price']].values
c_transf = ColumnTransformer([ ('onehot', OneHotEncoder(), [0]),
                               ('nothing', 'passthrough', [1, 2])])
c_transf.fit_transform(X).astype(float)
```

```
array([[ 0. ,  1. ,  0. ,  1. , 10.1],
       [ 0. ,  0. ,  1. ,  2. , 13.5],
       [ 1. ,  0. ,  0. ,  3. , 15.3]])
```

18. Applied on a DataFrame, the get_dummies (implemented in pandas) method will only convert string columns and leave all other columns unchanged :

```
# one-hot encoding via pandas
pd.get_dummies(df[['price', 'color', 'size']])
```

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

```
#multicollinearity guard in get_dummies
pd.get_dummies(df[['price', 'color', 'size']], drop_first=True)
```

	price	size	color_green	color_red
0	10.1	1	1	0
1	13.5	2	0	1
2	15.3	3	0	0

```
# multicollinearity guard for the OneHotEncoder
color_ohe = OneHotEncoder(categories='auto', drop='first')
c_transf = ColumnTransformer([ ('onehot', color_ohe, [0]),
                               ('nothing', 'passthrough', [1, 2])])
c_transf.fit_transform(X).astype(float)
```

```
array([[ 1. ,  0. ,  1. , 10.1],
       [ 0. ,  1. ,  2. , 13.5],
       [ 0. ,  0. ,  3. , 15.3]])
```

Partitioning Dataset in Training and Test Sets

19. In this section, we will prepare a new dataset, the Wine Dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset. Using the pandas library, we will directly read in the open source Wine dataset from the UCI machine learning repository :

```
df_wine = pd.read_csv('https://archive.ics.uci.edu/'
                      'ml/machine-learning-databases/wine/wine.data',
                      header=None)

# if the Wine dataset is temporarily unavailable from the
# UCI machine learning repository, un-comment the following line
# of code to load the dataset from a local path:

# df_wine = pd.read_csv('wine.data', header=None)


df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']

print('Class labels', np.unique(df_wine['Class label']))
df_wine.head()
```

```
Class labels [1 2 3]
```

20. A convenient way to randomly partition this dataset into a separate test and training dataset is to use the train_test_split function from scikit-learn cross_validation submodule.
    First, we assigned the NumPy array representation of feature columns 1-13 to the variable X, and we assigned the class labels from the first column to the variable y. Then, we used the train_test_split function to randomly split X and y into separate training and test datasets. By setting test_size=0.3 we assigned 30 percent of the wine samples to X_test and y_test, and the remaining 70 percent of the samples were assigned to X_train and y_train, respectively.

```
from sklearn.model_selection import train_test_split

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test =\
    train_test_split(X, y,
                     test_size=0.3,
                     random_state=0,
                     stratify=y)
```

Bringing Features Onto the Same Scale

There are two common approach to bringing different features onto the same scale : normalization and standardization. Those term are often quite loosely in different fields, and the same meaning has to be derived from the context

21. Normalization : refers to the rescaling of the features to a range of [0, 1] which is a special case of mix-max scaling. The formula of normalization can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Here, $x_{norm}^{(i)}$ is a new value of a sample $x^{(i)}$, x_{min} is the smallest value in a feature column and x_{max} the largest value, respectively.
The min-max scaling procedure is implemented in scikit-learn and can be used as follows :

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

22. Standardization : we center the feature columns at mean 0 with standard deviation 1 so that the feature columns take the norm of a normal distribution which makes it easier to learn the weight. The procedure of standardization can be expressed by the following equation :

1. $x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$
2. Here, μ_x is the sample mean of a particular feature column and the $\sigma_x$the corresponding standard deviation, respectively.
3. Similar to MinMaxScaler, scikit-learn also implements a class for standardization :

```
from sklearn.preprocessing import StandardScaler

stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

23. Here, a visual example of normalization and standardization :

```
ex = np.array([0, 1, 2, 3, 4, 5])

print('standardized:', (ex - ex.mean()) / ex.std())

# normalize
print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
```

```
standardized: [-1.46385011 -0.87831007 -0.29277002  0.29277002  0.87831007  1.46385011]
normalized: [0.  0.2 0.4 0.6 0.8 1. ]
```

REFERENSI

1. Geron A. 2017. Hands on Machine Learning wirh Scikit Learn and TensorFlow. O Reilly Media Inc
2. Raschka S, Mirjailili. 2017. Phython Machine Leraning 2nd edition. Packt Publishing