COP528 Applied Machine Learning (LABs)

LAB Day 02. Data

Introduction

The aim of this session is to understand the manipulation of data, including populating data, normalising data, pre-processing data, visualising data and sampling data. The experiment includes **6** tutorial examples and **4** tasks. You may follow the tutorials to learn the functions and then complete the tasks.

Tutorial 01. Filling of missing data

Many real-world applications require you to deal with missing data problems. Both Pandas and Sklearn offer you methods to fill the missing data. Please run the following examples to ensure that you are familiar with the method in Pandas or Sklearn to deal with missing data.

(1) Fill the missing data by 0, predefined values, average values in Pandas.

```
import numpy as np
import pandas as pd
df = pd.DataFrame([[np.nan, 2, np.nan, 0], [3, 4, np.nan, 1], [np.nan, np.nan, np.nan, 5],
[np.nan, 3, np.nan, 4]], columns=list('ABCD'))
df.head(4)
filleddf = df.fillna(0)
values = {'A': 0, 'B':1, 'C':2, 'D':3}
df_predefine = df.fillna(value = values)
df_mean = df.fillna(df.mean())
print(df)
print(filleddf)
print(df_predefine)
print(df_mean)
```

(2) Fill in the missing data by using Sklearn functions.

```
import numpy as np
from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
imp_mean.fit([[7, 2, 3], [4, np.nan, 6], [10, 5, 9]])
X = [[np.nan, 2, 3], [4, np.nan, 6], [10, np.nan, 9]]
X_filled = imp_mean.transform(X)
print(X_filled)
```

(2) Optional: investigate other filling methods in sklearn:

https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html

https://scikit-learn.org/stable/modules/impute.html#impute

Task 01. Exercises to fill in missing data

Practice the missing data filling methods that you learned to fill the missing data generated by the following code (note that you could work on the Numpy arrays if you decide to use functions in Sklearn, or you could use the Pandas Dataframe if you would like to process the missing data by using methods in Pandas). Here is the code to generate two datasets with missing data.

```
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.datasets import load_diabetes

rng = np.random.RandomState(42)
X_diabetes, y_diabetes = load_diabetes(return_X_y=True)
X_california, y_california = fetch_california_housing(return_X_y=True)
X_california = X_california[:400]
y_california = y_california[:400]
def add_missing_values(X_full, y_full):
   n_samples, n_features = X_full.shape
   # Add missing values in 75% of the lines
   missing_rate = 0.75
   n_missing_samples = int(n_samples * missing_rate)
   missing_samples = np.zeros(n_samples, dtype=bool)
   missing_samples[: n_missing_samples] = True
   rng.shuffle(missing_samples)
   missing_features = rng.randint(0, n_features, n_missing_samples)
   X_missing = X_full.copy()
   X_missing[missing_samples, missing_features] = np.nan
   y_missing = y_full.copy()

   return X_missing, y_missing
X_miss_california, y_miss_california = add_missing_values(
   X_california, y_california)
X_miss_diabetes, y_miss_diabetes = add_missing_values(
   X_diabetes, y_diabetes)

import pandas as pd
diabetes_pddata = pd.DataFrame(data= X_miss_diabetes)
diabetes_pddata['target'] = pd.Series(y_miss_diabetes)

print(diabetes_pddata.head(5))
```

Tutorial 02. Data pre-processing

Data in real-world applications requires preprocessing stages: e.g., (1) many machine learning algorithms are sensitive to the data range, data standardization is a normal preprocessing step. For instance, many elements used in the objective function of a learning algorithm (such as the RBF kernel of Support Vector Machines or the l1 and l2 regularizers of linear models) assume that all features are centred around zero and have variance in the same order. and (2) some features are provided as categorical data, e.g., [male, female], education level, encoding categorical features is another important preprocessing step. Sklearn provides functions for doing these preprocessing tasks.

(1) Normalization based on mean and standard deviation.

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.], [ 2.,  0.,  0.], [ 0.,  1., -1.]])
scaler = preprocessing.StandardScaler().fit(X_train)

vector_unnorm = np.array([[0.3, 0.5, 3]])
scaler.transform(vector_unnorm)
```

(2) Minmax standardisation to normalise data in range [0,1].

```
from sklearn.preprocessing import MinMaxScaler
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
scaler = MinMaxScaler()
scaler.fit(data)
print(scaler.data_max_)
print(scaler.transform(data))
```

(3) OrdinalEncoder:

```
enc = preprocessing.OrdinalEncoder()
X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
enc.fit(X)
enc.transform([['female', 'from US', 'uses Safari']])
```

(4) One Hot Encoder

```
enc = preprocessing.OneHotEncoder()
X = [['male', 'from US', 'uses Safari'], ['female', 'from Europe', 'uses Firefox']]
enc.fit(X)
enc.transform([['female', 'from US', 'uses Safari'],
          ['male', 'from Europe', 'uses Safari']]).toarray()
```

Tutorial 03. Distance metrics

Distance metrics play important role in machine learning algorithms. Many metrics are implemented in sklearn.Metrics.DistanceMetric package. Please read the following document (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html?highlight=distancemetric#sklearn.metrics.DistanceMetric) so that you know how to call functions to calculate distances between samples.

(1) EuclideanDistance calculation.

```
from sklearn.metrics import DistanceMetric
dist = DistanceMetric.get_metric('euclidean')
X= [[0,1,2],[3,4,5]]
print(dist.pairwise(X))
```

(2) mahalanobis Distance calculation.

```
from sklearn.neighbors import DistanceMetric
import numpy as np

X = [[1, 2, 2], [ -1,7, 5]]
V=np.cov(X)

dist = DistanceMetric.get_metric('mahalanobis', V=np.cov(X))
print(dist.pairwise(V))
```

Available Metrics

The following lists the string metric identifiers and the associated distance metric classes:

Metrics intended for real-valued vector spaces:

identifier	class name	args	distance function
“euclidean”	EuclideanDistance	•	sqrt(sum((x - y)^2))
“manhattan”	ManhattanDistance	•	sum(|x - y|)
“chebyshev”	ChebyshevDistance	•	max(|x - y|)
“mahalanobis”	MahalanobisDistance	V or VI	sqrt((x - y)' V^-1 (x - y))

Task 02. Distances and standardization

Use the following code to load diabetes datasets, and calculate the pairwise distances between the 1st, 10th, 20th, 50th and 100th samples. Have a comparison on the results of using different metrics (you may pick 3 metrics from the above table).

```
import numpy as np
from sklearn.datasets import load_diabetes


X_diabetes, y_diabetes = load_diabetes(return_X_y=True)
```

(2) Run a standardization method first and compare the difference with the one by using original data.

Tutorial 04. Matplotlib Basics

Data visualization is an important skill to possess for anyone trying to extract and communicate insights from data. In the field of machine learning, visualization plays a key role throughout the entire process of analysis.

This tutorial introduces Python data visualization based on the **Pandas**, **Matplotlib**, and **Seaborn** libraries. **Pandas** is a data analysis library that implements some basic plotting methods. **Matplotlib** and **Seaborn** are the most commonly used visualization tools in Python, and Python data visualization is generally implemented through the lower level Matplotlib library and the higher level Seaborn library.

(1) **The basic form of the Matplotlib function:**

Matplotlib drawing functions are generally of the following form:

plt.plotName(x, y, 'colour marker line type')

You can see what each function does by commenting out the function

```
# Import packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

# In order to display graphs in the jupyter notebook inline pages, you can turn on the following switch
# If the switch is not turned on, you will have to use the function: plt.show() each time the graph is displayed
%matplotlib inline


x = np.linspace(0, 2 * np.pi, 20)
plt.figure(figsize=(6, 4))# create a new image and set the image size
plt.plot(x, np.sin(x), 'ro-', label='sinx')# set colour, marker symbol, line type, legend label
plt.plot(x, np.cos(x), 'b*--', label='cosx')
plt.title('plot curve', fontsize=25)# title
plt.xlim(-1, 7)# x-axis range
plt.ylim(-1.5, 1.5)# y-axis range
plt.xlabel('x', fontsize=20)# x-axis label
plt.ylabel('y', fontsize=20)# y-axis label
plt.legend(loc='best')# legend
plt.show()
```

(2) **The relationship between figure and axes**

In Matplotlib, a whole image is a Figure object. The figure object can contain one or more Axes objects. Each Axes object is a plotting area with its coordinate system.

Accordingly, matplotlib has two ways of drawing plots, the Pyplot method and the axes method. Pyplot code is simple and suitable for use in cases where subplots are not involved; Axes is suitable for use in cases where subplots are drawn.

The Pyplot method is as follows.

```
# pyplot graphing
plt.figure()
plt.plot(np.arange(6), np.arange(6), color='r')
plt.scatter([0.5, 4.1, 4.8, 3], [6, 8, 3.1, 1.2], color='b', marker='^')
plt.title('pyplot')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

The Axes method is as follows. Note: axes graphing sets the graph parameters via ax.set_{something}()

```
# axes drawing I
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(np.arange(6), np.arange(6), color='r')
ax.scatter([0.5, 4.1, 4.8, 3], [6, 8, 3.1, 1.2], color='b', marker='^')
ax.set_title('axes1')
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()

# axes drawing  II
fig, ax = plt.subplots()
ax.plot(np.arange(6), np.arange(6), color='r')
ax.scatter([0.5, 4.1, 4.8, 3], [6, 8, 3.1, 1.2], color='b', marker='^')
ax.set_title('axes2')
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
```

(3) **Subplotting**

There are two ways of drawing subplots, the Pyplot way and the Axes object-oriented way.

```
# pyplot
plt.figure()
X = np.arange(0.01, 10, 0.01)
# Divided into 2*2, occupying the 1st subplots
plt.subplot(221)
plt.plot(X, np.sin(X), 'r-')

# Divided into 2*2, occupying the 2nd subplots
plt.subplot(222)
plt.plot(X, np.cos(X), 'g-')

# Divided into 2*1, occupying the 2nd subplot (i.e. occupying the 2*2 3rd and 4th subplots)
plt.subplot(212)
plt.bar(np.arange(6), np.array([2, 4, 1, 6, 3, 8]))

plt.suptitle('pyplot')
plt.show()
```

```
# axes way one: add_subplot
# Same parameters as plt.subplot
fig = plt.figure()
ax1 = fig.add_subplot(221)
ax1.plot(X, np.sin(X), 'r-')

ax2 = fig.add_subplot(222)
ax2.plot(X, np.cos(X), 'g-')

ax3 = fig.add_subplot(212)
ax3.bar(np.arange(6), np.array([2, 4, 1, 6, 3, 8]))

fig.suptitle('add_subplot')
plt.show()
```

(4) **Exporting vector graphics.**

```
plt.figure()
plt.plot(np.arange(6))
# pdf
plt.savefig('./filename.pdf',format='pdf')
# svg
plt.savefig('./filename.svg',format='svg')
```

Tutorial 05. Visualisation examples

Seaborn Basics

Seaborn requires a DataFrame or Numpy array of pandas as input type for the raw data, and the drawing function is generally of the following form:

sns.plotName(x='X-axis column name', y='Y-axis column name', data=original data df object)

sns.plotName(x='X-axis column name', y='Y-axis column name', hue='group plotting parameters', data=original data df object)

sns.plotname(x=np.array, y=np.array[, ...])

Seaborn has five pre-defined themes (styles): darkgrid, whitegrid, dark, white, and ticks (surrounded by borders and scales). They are each suitable for different applications and personal preferences. The default theme is darkgrid.

There are four pre-defined contexts, in order of relative size: paper, notebook, talk, and poster.

The default size is notebook. The colours (palette) are: muted (common), RdBu, Blues_d, Set1, etc.

Set the theme, context, palette, etc. with sns.set(), e.g.

sns.set(style='white', context='talk', palette="muted", color_codes=True)

Follow the tutorial below to learn about basic data graphing by comparing the three graphing methods, Pandas, Matplotlib and Seaborn, to draw different images. Remember: there is no universal best way to visualise data, and different questions are best answered by different visualisations.

(1) Loading data

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Loading data
iris = sns.load_dataset('iris')
titanic = sns.load_dataset('titanic')
tips = sns.load_dataset('tips')

iris.head()
titanic.head()
tips.head()
```

(1)Line drawings

```
#pandas
test_dict = {'Sales':[1000,2000,5000,2000,4000,3000],'Collection':[1500,2300,3500,2400,1900,3000]}
line = pd.DataFrame(test_dict,index=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])
line.plot()
```

```
# matplotlib
plt.figure(figsize=(6, 4))# create a new image and set the image size
index=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
plt.plot(index, test_dict['Sales'], 'ro-', label='Sales')# set colour, marker symbol, line type, legend label
plt.plot(index, test_dict['Collection'], 'bo-', label='Sales')

plt.legend(loc='best')# legend
plt.show()
```

```
# seaborn
d = {'Sales':test_dict['Sales'], 'Collection':test_dict['Collection'],'date': index}
line=pd.DataFrame(d)
sns.set(style='darkgrid')
sns.lineplot(x="date",y='Sales',data=line)
sns.lineplot(x="date",y='Collection',data=line)
```

(2) Scatterplot.

```
# pandas
iris.plot(x='sepal_length', y='sepal_width', kind='scatter')
```

```
# matplotlib
plt.scatter(iris['sepal_length'], iris['sepal_width'])
```

```
#seaborn--Common Scatter Plot
sns.stripplot(x="sepal_length",y="sepal_width", data=iris)
```

```
#seaborn--A scatter plot of the distribution density can be seen
sns.swarmplot(y="sepal_width", data=iris)
```

(3) Bar Chart.

```
#pandas
titanic.pclass.value_counts().plot(kind='bar')
```

```
#matplotlib
plt.bar(np.arange(8), np.array([1, 4, 2, 3, 3, 5, 6, 3]))
```

```
# Statistical function for barplot, default is the mean of the variable estimator=np.mean
sns.barplot(x='day', y='total_bill', data=tips)
```

```
# Plot a bar graph of the median of the variables, with estimator specifying the statistical function
sns.barplot(x='day', y='total_bill', hue='sex', data=tips, estimator= np.median)
```

(4) Histograms.

To create a histogram, first divide the range of values on the x-axis into equal intervals, then count the number of values contained in each interval and use that number as the value for the y-axis.

```
#Pandas
iris.hist('sepal_width', by='species',layout=(1,3), bins=8)
iris.plot(y='sepal_width', kind='hist')
```

```
#matplotlib
plt.hist(iris['sepal_width'])# If parameter density=True is set, the probability density value is calculated
```

```
#seaborn
sns.histplot(x=iris['sepal_width'],bins=10)
```

(5) pairplot plots

Multiple comparison plots by combining multiple features in pairs.

```
sns.pairplot(iris, vars=['sepal_length', 'sepal_width', 'petal_length'], hue="species")
```

(6) parallel_coordinates

```
import pandas as pd
df = pd.read_csv(
   'https://raw.github.com/pandas-dev/'
    'pandas/main/pandas/tests/io/data/csv/iris.csv'
)
pd.plotting.parallel_coordinates(
    df, 'Name', color=('#556270', '#4ECDC4', '#C7F464')
)
```

Task 03. parallel_coordinates Visualisation

Visualizing the wine dataset using parallel coordinates and interpreting the visualization results.

The wine dataset is imported as follows.

Check out https://pandas.pydata.org/docs/reference/api/pandas.plotting.parallel_coordinates.html for a better understanding of the meaning of the arguments to the parallel coordinates function

```
from sklearn import datasets
wine = datasets.load_wine()
df = pd.DataFrame(wine.data, columns=wine.feature_names)
```

Tutorial 06. Cross-validation: evaluating estimator performance

(1) K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k-1 remaining folds form the training set.

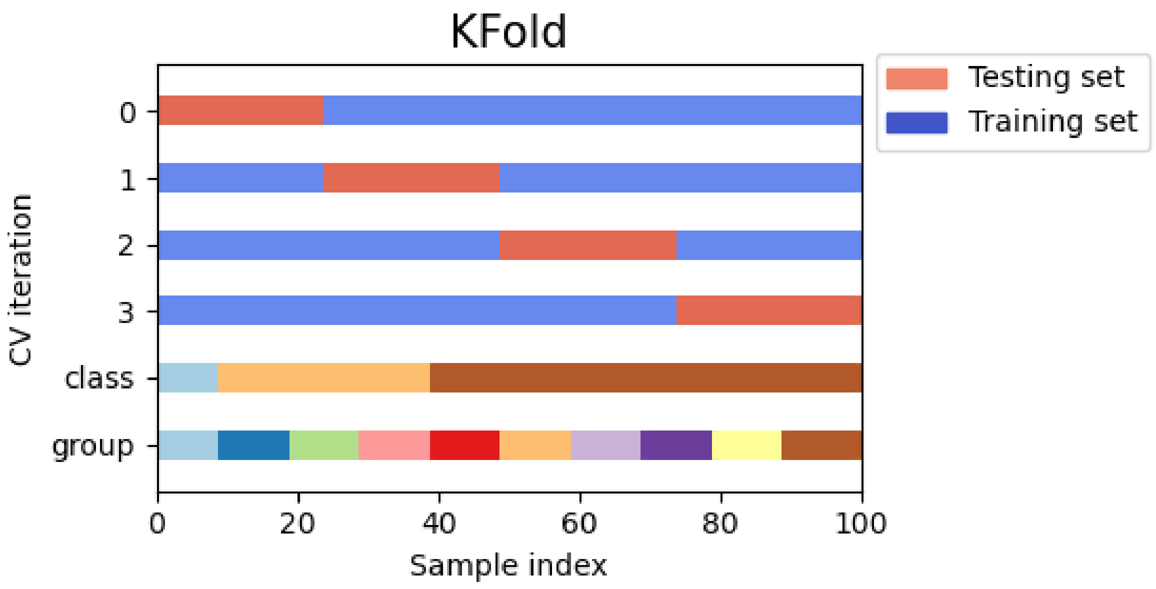
(1) An example of the KFold function in sklrean is as follows.

```
import numpy as np
from sklearn.model_selection import KFold
X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
y = np.array([1, 2, 3, 4])
kf = KFold(n_splits=2)
kf.get_n_splits(X)
print(kf)

for train_index, test_index in kf.split(X):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

(2) Example of 2-fold cross-validation on a dataset with 4 samples:

```
import numpy as np
from sklearn.model_selection import KFold

X = ["a", "b", "c", "d"]
kf = KFold(n_splits=2)
for train, test in kf.split(X):
    print("%s %s" % (train, test))
```

Here is a visualization of the cross-validation behavior. Note that KFold is not affected by classes or groups.



(2) Cross-validation: evaluating estimator performance

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called overfitting. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_test, y_test.

(1) In scikit-learn a random split into training and test sets can be quickly computed with the train_test_split helper function. Let's load the iris data set to fit a linear support vector machine on it:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn import svm

X, y = datasets.load_iris(return_X_y=True)
X.shape, y.shape
```

(2) We can now quickly sample a training set while holding out 40% of the data for testing (evaluating) our classifier:

```
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.4, random_state=0)

X_train.shape, y_train.shape
X_test.shape, y_test.shape

clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
clf.score(X_test, y_test)
```

(3) The simplest way to use cross-validation is to call the **cross_val_score** helper function on the estimator and the dataset.

The following example demonstrates how to estimate the accuracy of a linear kernel support vector machine on the iris dataset by splitting the data, fitting a model and computing the score 5 consecutive times (with different splits each time):

```
from sklearn.model_selection import cross_val_score
clf = svm.SVC(kernel='linear', C=1, random_state=42)
scores = cross_val_score(clf, X, y, cv=5)
scores
```

(4) The mean score and the standard deviation are hence given by:

```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

(5) By default, the score computed at each CV iteration is the score method of the estimator. It is possible to change this by using the scoring parameter:

```
from sklearn import metrics
scores = cross_val_score(
   clf, X, y, cv=5, scoring='f1_macro')
scores
```

Task 04. K-Folds cross experiment

Breast Cancer dataset is a binary classification task. Breast Cancer dataset is a binary classification task. Please use a simple logistic regression classifier (hint: day01-Task01) and a K-flod to measure for the classification experiment.

(1) Let’s load the Breast Cancer dataset.

```
from sklearn.datasets import load_breast_cancer
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import numpy as np

X, y = load_breast_cancer(return_X_y=True)
```