A selection of Python examples for TKJ4175

0. How to access the documentation

You can access documentation for a Python object (variable, function, class, module) by typing its name followed by a question mark and then executing the cell:

In [ ]:
```
import numpy as np

np.linspace?
```

Alternatively, you can also type `help()`:

In [ ]:
```
help(np.linspace)
```

Another alternative is using **Shift+Tab**:

1. Place your cursor: Type the name of a Python object (variable, function, class, module) in a code cell.
2. Press Shift + Tab: With the cursor on or immediately after the object's name (or within the parentheses of a function call), press the Shift key and the Tab key simultaneously.
3. A tooltip appears: A small tooltip window will pop up, displaying:
    - The object's docstring (if it has one).
    - The object's signature (for functions and methods), showing arguments and return values.
4. Press Shift + Tab multiple times:
    - Pressing Shift + Tab once usually shows a concise view.
    - Pressing Shift + Tab a second time often expands the tooltip to show more details from the docstring.
    - Pressing Shift + Tab a third or fourth time might open a larger, scrollable pane with the full documentation (similar to the ? pager, but often embedded).

1. Reading data with Pandas

In [ ]:
```
import pandas as pd
import numpy as np
```

```
# To read a comma separated file (cvs: comma-separated values):
data = pd.read_csv("data_file.csv")
# Show the first few rows:
data.head()
```

In [ ]:
```
# Read example data and extract some columns:
import pandas as pd

data = pd.read_csv("data_file.csv")
# Select X and Y variables:
x_variables = ["x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9", "x10"]
X = data[x_variables].to_numpy()
Y = data[["y1", "y2", "y3"]].to_numpy()
x = data[["x1"]].to_numpy().flatten()
y = data[["y1"]].to_numpy().flatten()
```

2. Some simple plotting

2.1 Scatter plots

In [ ]:
```
from matplotlib import pyplot as plt

xdata = np.linspace(-np.pi, np.pi, 50)
ydata = np.sin(xdata)

fig, ax = plt.subplots()
ax.scatter(xdata, ydata, label="Here are some points")
ax.set_xlabel("xdata")
ax.set_ylabel("ydata")
ax.legend()
```

2.2 Line plots

In [ ]:
```
from matplotlib import pyplot as plt

xdata = np.linspace(-np.pi, np.pi, 50)
ydata = np.sin(xdata)

fig, ax = plt.subplots()
ax.plot(xdata, ydata, label="Here is a line")
ax.set_xlabel("xdata")
ax.set_ylabel("ydata")
ax.legend()
```

In [ ]:
```
from matplotlib import pyplot as plt

xdata = np.linspace(-np.pi, np.pi, 50)
ydata = np.sin(xdata)

fig, ax = plt.subplots()
ax.plot(xdata, ydata, marker="o", label="Here are some points on a line")
ax.set_xlabel("xdata")
ax.set_ylabel("ydata")
ax.legend()
```

In [ ]:
```
from matplotlib import pyplot as plt

xdata = np.linspace(-np.pi, np.pi, 50)
ydata = np.sin(xdata)
```

```
yerror = np.full_like(ydata, 0.2)

fig, ax = plt.subplots()
ax.errorbar(
    xdata,
    ydata,
    yerr=yerror,
    marker="o",
    label="Here are some points on a line with error bars",
)
ax.set_xlabel("xdata")
ax.set_ylabel("ydata")
ax.legend()
```

2.3 Saving a figure

In [ ]:
```
from matplotlib import pyplot as plt

xdata = np.linspace(-np.pi, np.pi, 50)
ydata = np.sin(xdata)

fig, ax = plt.subplots()
ax.plot(xdata, ydata, marker="o", label="Here are some points on a line")
ax.set_xlabel("xdata")
ax.set_ylabel("ydata")
ax.legend()
fig.savefig("figure-name.png")
# or (to store the figure that is currently open):
plt.savefig("figure-name2.png")
# Note: These figures are stored in the same folder as the notebook.
```

Alternative for a notebook: Right-click on the image and select "Save Image As".

3. Preprocessing

3.1 Splitting into training and testing

In [ ]:
```
from sklearn.model_selection import train_test_split

# Split into training and test sets. Use 33% of the data for the test set.
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33)

# Note: You can use the "stratify" parameter if you need to make
# sure class distributions are correctly reflected in the test and
# training sets.
```

3.2 Standardising data

In [ ]:
```
from sklearn.preprocessing import StandardScaler

# The StandardScaler will standardise the features by
# removing the mean and scaling to unit variance.

# Create a scaler for X and one for Y:
scale_X = StandardScaler().fit(X_train)
scale_Y = StandardScaler().fit(Y_train)

# Apply the scaler:
```

```
X_train_scaled = scale_X.transform(X_train)
X_test_scaled = scale_X.transform(X_test)
# If we are doing regression, we can scale Y.
# For classification, it does not make sense to scale Y.
Y_train_scaled = scale_Y.transform(Y_train)
Y_test_scaled = scale_Y.transform(Y_test)
```

In [ ]:
```
from sklearn.preprocessing import scale

# Using scale is shorter than using the StandardScaler:
X_scaled = scale(X)
Y_scaled = scale(Y)

# Note: If you have a training set and test set, the best practice is to
# create a StandardScaler, fit it to the traning data and then transform
# both the training and test data using the same scaling parameters.
```

4. Regression methods

4.1 Least squares regression

In [ ]:
```
from sklearn.linear_model import LinearRegression

# Create a model, use a intercept:
model = LinearRegression(fit_intercept=True)

# Fit the model:
model.fit(X, y)

# Use the model for prediction:
y_hat = model.predict(X)

# Show coefficients:
print("Coefficients:", model.coef_)
# Show intercept:
print("Intercept:", model.intercept_)
```

4.2 Least squares polynomial regression

In [ ]:
```
import numpy as np

# Fit a fith-order polynomial in x to y:
poly = np.polyfit(x, y, deg=5)
# Coefficints for polynomial:
print("Polynomial coefficients:", poly)

# Use polynomial for prediction:
y_hat = np.polyval(poly, x)

# Polynomial regression can also be done with scikit-learn:
X_ = np.column_stack((x, x**2, x**3, x**4, x**5))
model = LinearRegression(fit_intercept=True)
model.fit(X_, y)
# Show coefficients:
print("Coefficients:", model.coef_)
# Show intercept:
print("Intercept:", model.intercept_)
```

4.3 Non-linear least squares regression

In [ ]:
```python
import numpy as np
from scipy.optimize import minimize


# We first define our model, this is for a model of exponential decay.
def model(x, params):
    """Calculate y using the given parameters

    Args:
        x: The independent variable.
        params: A list of parameters:

    Returns:
        The  value of the function at x.
    """
    a = params[0]  # Amplitude
    tau = params[1]  # Time constant
    return a * np.exp(-x / tau)


# Next, we define the objective function we will minimize:
def objective(params, x, y):
    y_fit = model(x, params)
    return np.sum((y - y_fit) ** 2)  # Return sum of squared errors


# Define data to fit:
xdata = np.linspace(0, 100)
ydata = 5.0 * np.exp(-xdata / 10.0)

# Starting point for optimization
initial_guess = [1.0, 1.0]
# Set up boundaries for the coefficients, these
# are on form (min, max) for each parameter
bounds = [
    (0.0, 100),
    (0.0, np.inf),
]


result = minimize(
    objective,
    initial_guess,
    args=(xdata, ydata),
    # bounds=bounds,  # Add this to use boundaries
    options={
        "disp": True,
        "maxiter": 5000,
    },  # Print information, and do maximum 5000 iterations
    method="Nelder-Mead",
)
print("The results are:")
print(result)
print("The optimized parameters are:", result.x)
```

In [ ]:
```python
from matplotlib import pyplot as plt

# Use the optimized model:
param_opt = result.x

y_hat = model(xdata, param_opt)

fig, ax = plt.subplots()
ax.scatter(xdata, ydata, label="Raw data")
```

```
ax.plot(xdata, y_hat, color="red", label="Fitted model")
ax.legend()
```

4.4 Partial least squares regression

In [ ]:
```
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

# Split the data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.33, random_state=42
)

# Scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Create and fit the PLS model
model = PLSRegression(
    n_components=2,  # The number of components.
    scale=False,  # Do not scale the input data.
)
# We set scale=False since we handle the scaling in a separate step.
model.fit(X_train_scaled, Y_train)

# Make predictions
Y_pred = model.predict(X_test_scaled)

# Evaluate the model
mse = mean_squared_error(Y_test, Y_pred)
print("Mean Squared Error:", mse)

# Show coefficients:
print("Coefficients:", model.coef_)
# Show intercept:
print("Intercept:", model.intercept_)
```

4.5 Computing regression metrics

R^2, MAE, MSE, RMSE

In [ ]:
```
from sklearn.metrics import (
    r2_score,
    mean_absolute_error,
    mean_squared_error,
    root_mean_squared_error,
)

Y_hat = model.predict(X)

# 1. Calculate R²:
r2 = r2_score(Y, Y_hat)
print(f"R² = {r2}")

# 2. Calculate the mean absolute error:
mae = mean_absolute_error(Y, Y_hat)
print(f"MAE = {mae}")
```

```
# 2. Calculate the mean squared error:
mse = mean_squared_error(Y, Y_hat)
print(f"MSE = {mse}")

# 3. Calculate the root mean squared error:
rmse = root_mean_squared_error(Y, Y_hat)
print(f"RMSE = {rmse}")
```

The observed vs. predicted plot

In [ ]:
```
from matplotlib import pyplot as plt

model = LinearRegression(fit_intercept=True)
model.fit(X, y)
y_hat = model.predict(X)


fig, ax = plt.subplots()
ax.scatter(y, y_hat)
ax.set_xlabel("Observed Values (y)")
ax.set_ylabel("Predicted Values (ŷ)")
ax.set_title("Observed vs. Predicted Values")

# Add x=y line:
min_val = min(min(y), min(y_hat))
max_val = max(max(y), max(y_hat))
ax.plot(
    [min_val, max_val],
    [min_val, max_val],
    "r--",
    linewidth=2,
    label="Perfect Fit",
)
ax.set_xlim([min_val, max_val])
ax.set_ylim([min_val, max_val])
```

Plot of residuals

In [ ]:
```
from matplotlib import pyplot as plt

model = LinearRegression(fit_intercept=True)
model.fit(X, y)
y_hat = model.predict(X)

residual = y - y_hat

fig, ax = plt.subplots()
ax.scatter(y_hat, residual)
ax.set_xlabel("Predicted Values (ŷ)")
ax.set_ylabel("Residuals (y-ŷ)")
ax.set_title("Residuals vs. Predicted Values")
# Add line at y=0 as a reference line:
ax.axhline(y=0, ls="--", color="black")
```

5. Classification

In [ ]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import (
```

```
    accuracy_score,
    precision_score,
    recall_score,
    ConfusionMatrixDisplay,
)

# 1. Generate a synthetic dataset with two classes
X, y = make_classification(
    n_samples=1000,
    n_features=20,
    n_informative=10,
    n_redundant=0,
    n_classes=2,
    random_state=42,
)

# 2. Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# 3. Create and train a decision tree classifier
model = DecisionTreeClassifier(max_depth=4, random_state=42)
model.fit(X_train, y_train)

# 4. Make predictions on the test set
y_pred = model.predict(X_test)

# 5. Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)

print("Accuracy:", accuracy)
print("Precision", precision)
print("Recall", recall)

ConfusionMatrixDisplay.from_estimator(
    model,
    X_test,
    y_test,
)
```

6. Principal component analysis

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# 1. Load the Iris dataset (a classic example for PCA)
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names

# 2. Scale the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. Apply PCA
pca = PCA(n_components=3)  # Reduce to 3 principal components
```

```
scores = pca.fit_transform(X_scaled)

# 4. Plot the scores
# Percentage of variance explained, used for labels:
percent = pca.explained_variance_ratio_ * 100

fig, ax = plt.subplots()
ax.scatter(scores[:, 0], scores[:, 1], c=y)
ax.set_xlabel(f"Scores, PC1 ({percent[0]:.2f}%)")
ax.set_ylabel(f"Scores, PC2 ({percent[1]:.2f}%)")

# 5. Plot the loadings
loadings = pca.components_.T
fig, ax = plt.subplots()
ax.scatter(loadings[:, 0], loadings[:, 1])
for i, variable in enumerate(feature_names):
    ax.text(loadings[i, 0], loadings[i, 1], variable)

ax.set_xlabel(f"Loadings, PC1 ({percent[0]:.2f}%)")
ax.set_ylabel(f"Loadings, PC2 ({percent[1]:.2f}%)")
```

7. Clustering

In [ ]:

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# 1. Generate synthetic data
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# 2. Scale the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. Try different numbers of clusters and calculate Within-Cluster Sum of Squares
wcss = []  # Within-Cluster Sum of Squares
cluster_range = range(1, 11)  # Try clusters from 1 to 10

for k in cluster_range:
    kmeans = KMeans(n_clusters=k, random_state=0)
    kmeans.fit(X_scaled)
    # or if you did not scale:
    # kmeans.fit(X)
    wcss.append(kmeans.inertia_)  # Inertia is the WCSS

# 4. Plot the Elbow Method
fig, ax = plt.subplots()
ax.plot(cluster_range, wcss, marker="o")
ax.set_title("Elbow Method for Optimal k")
ax.set_xlabel("Number of Clusters (k)")
ax.set_ylabel("Within-Cluster Sum of Squares (WCSS)")
ax.set_xticks(cluster_range)


# 5. Apply KMeans with the chosen k (e.g., 4 based on the elbow method)
chosen_k = 4  # Adjust this based on your elbow plot
kmeans_final = KMeans(n_clusters=chosen_k, random_state=0)
clusters = kmeans_final.fit_predict(X_scaled)

# 6. Visualize the clusters with the chosen k
fig, ax = plt.subplots()
ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap="viridis")
```

```
ax.scatter(
    kmeans_final.cluster_centers_[:, 0],
    kmeans_final.cluster_centers_[:, 1],
    s=300,
    c="red",
    marker="*",
    label="Centroids",
)
ax.set_title(f"KMeans Clustering (k={chosen_k})")
ax.set_xlabel("Feature 1 (scaled)")
ax.set_ylabel("Feature 2 (scaled)")
ax.legend()


# 7. Print the cluster centers for the chosen k
print(f"Cluster Centers (k={chosen_k}):\n", kmeans_final.cluster_centers_)
```