MLAI 504
NEURAL NETWORKS & DEEP LEARNING

MLAI – 2023/2024

Dr. Zein Al Abidin IBRAHIM

zein.ibrahim@ul.edu.lb

IMPLEMENTATION OF ANN USING SCIKIT-LEARN

NEURAL NETWORK

SCIKIT-LEARN

- Scikit-learn, numpy, matplotlib and pandas → common tools for data science in python

```
import sklearn
import numpy as np
import matplotlib.pyplot as plt
```

- sklearn has many of the tools needed to set up a data analysis pipeline:
  - Dataset loading
  - Preprocessing
  - Models' creation
  - Models' fitting
  - Evaluation

DATA LOADING

DATASET LOADING (1/3)

- Built-in Datasets:

Python

```python
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels

print(f"Number of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
print(f"Target classes: {iris.target_names}")
```

DATASET LOADING (2/3)

- **Load_csv**:

Python

```python
from sklearn.datasets import load_csv

data = load_csv("my_data.csv")
X = data["data"]  # Features (assuming "data" column)
y = data["target"]  # Labels (assuming "target" column)

print(f"Number of samples: {X.shape[0]}")
print(f"Number of features: {X.shape[1]}")
```

- **read_csv** function→ Similar to **load_csv** with offers more customization options.

DATASET LOADING (3/3)

- **fetch_openml** → Downloads and loads datasets from OpenML repository

Python

```python
from sklearn.datasets import fetch_openml

text_data = fetch_openml("bbc", version=1, as_frame=True)
X = text_data["text"]
y = text_data["category"]

print(f"Number of documents: {X.shape[0]}")
print(f"Document lengths: {X.shape[1]}")
print(f"Target categories: {y.unique()}")
```

- **read_text** function → Reads text files line by line.

DATA PREPROCESSING

SCIKIT-LEARN

- Preprocessors include:
  - **standardScaler**: shifts and scale the data to have mean 0 and standard deviation 1.
  - **Normalizer**: normalizes the features for each data sample to have unit length
  - **MinMaxScaler**: shifts and scales the data so it fits in a given interval
  - **OneHotEncoder**: transforms class labels to a one-hot encoded matrix of 0 or 1 values
  - **PolynomialFeatures**: Creates polynomial features
  - ….

SCIKIT-LEARN: PREPROCESSING

- Standardization, or mean removal and variance scaling

- ```
  >>> 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)
  >>> scaler
  StandardScaler()

  >>> scaler.mean_
  array([1. ..., 0. ..., 0.33...])

  >>> scaler.scale_
  array([0.81..., 0.81..., 1.24...])

  >>> X_scaled = scaler.transform(X_train)
  >>> X_scaled
  array([[ 0.  ..., -1.22...,  1.33...],
         [ 1.22...,  0.  ..., -0.26...],
         [-1.22...,  1.22..., -1.06...]])
  ```

```
>>> X_scaled.mean(axis=0)
array([0., 0., 0.])

>>> X_scaled.std(axis=0)
array([1., 1., 1.])
```
```

SCIKIT-LEARN: PREPROCESSING

- Scaling features to a range [0, 1]
-
```
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
...
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[0.5       , 0.        , 1.        ],
       [1.        , 0.5       , 0.33333333],
       [0.        , 1.        , 0.        ]])
```

SCIKIT-LEARN: PREPROCESSING

- **Normalization →** process of scaling individual samples to have unit norm.

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> X_normalized = preprocessing.normalize(X, norm='l2')

>>> X_normalized
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])
```

`l1`, `l2`, or `max` norms

```
>>> normalizer = preprocessing.Normalizer().fit(X)  # fit does nothing
>>> normalizer
Normalizer()
```

The normalizer instance can then be used on sample vectors as any transformer:

```
>>> normalizer.transform(X)
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])

>>> normalizer.transform([[-1.,  1., 0.]])
array([[-0.70...,  0.70...,  0.  ...]])
```

SCIKIT-LEARN: PREPROCESSING

- OneHotEncoder:

Python

```python
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse=False)  # sparse=True creates sparse matrix if memory concerns
```

2. Fit the encoder to your data:

Python

```python
colors = ["red", "blue", "green", "red", "blue"]
encoded_colors = encoder.fit_transform(colors.reshape(-1, 1))
```

Result:

```
encoded_colors:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]]
```

SCIKIT-LEARN: PREPROCESSING

- Generating polynomial features
- Here degree =2

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

The features of X have been transformed from (X_1, X_2) to $(1, X_1, X_2, X_1^2, X_1X_2, X_2^2)$.

SCIKIT-LEARN: PREPROCESSING

- Generating polynomial features
- Here degree =3 but only interaction terms are generated

```
>>> X = np.arange(9).reshape(3, 3)
>>> X
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> poly = PolynomialFeatures(degree=3, interaction_only=True)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   2.,   0.,   0.,   2.,   0.],
       [  1.,   3.,   4.,   5.,  12.,  15.,  20.,  60.],
       [  1.,   6.,   7.,   8.,  42.,  48.,  56., 336.]])
```

(X_1, X_2, X_3) to $(1, X_1, X_2, X_3, X_1X_2, X_1X_3, X_2X_3, X_1X_2X_3)$.

SCIKIT-LEARN: PREPROCESSING

- Handling missing values → simple

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
SimpleImputer()
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[4.          2.        ]
 [6.          3.666...]
 [7.          6.        ]]
```

```
>>> import scipy.sparse as sp
>>> X = sp.csc_matrix([[1, 2], [0, -1], [8, 4]])
>>> imp = SimpleImputer(missing_values=-1, strategy='mean')
>>> imp.fit(X)
SimpleImputer(missing_values=-1)
>>> X_test = sp.csc_matrix([[-1, 2], [6, -1], [7, 6]])
>>> print(imp.transform(X_test).toarray())
[[3. 2.]
 [6. 3.]
 [7. 6.]]
```

```
>>> import pandas as pd
>>> df = pd.DataFrame([["a", "x"],
...                    [np.nan, "y"],
...                    ["a", np.nan],
...                    ["b", "y"]], dtype="category")
...
>>> imp = SimpleImputer(strategy="most_frequent")
>>> print(imp.fit_transform(df))
[['a' 'x']
 ['a' 'y']
 ['a' 'y']
 ['b' 'y']]
```

SCIKIT-LEARN: PREPROCESSING

- Iterative imputer→ models each feature with missing values as a function of other features, and uses that estimate for imputation

```
>>> import numpy as np
>>> from sklearn.experimental import enable_iterative_imputer
>>> from sklearn.impute import IterativeImputer
>>> imp = IterativeImputer(max_iter=10, random_state=0)
>>> imp.fit([[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]])
IterativeImputer(random_state=0)
>>> X_test = [[np.nan, 2], [6, np.nan], [np.nan, 6]]
>>> # the model learns that the second feature is double the first
>>> print(np.round(imp.transform(X_test)))
[[ 1.  2.]
 [ 6. 12.]
 [ 3.  6.]]
```

MODEL CREATION

SCIKIT-LEARN: MODEL

- **sklearn.linear_model.Perceptron** → one neuron
- Parameters:
  - **eta**: Learning rate (step size for weight updates).
  - **n_iter**: Maximum number of training iterations.
  - **tol**: Tolerance for stopping training (epsilon value for convergence).
  - **random_state**: Seed for random initialization.
  - **fit_intercept**: Whether to learn an intercept term (bias).
  - **warm_start**: Option to use previously learned weights as initialization.
- Methods
  - **fit(X, y)**: Trains the perceptron on data X and labels y.
  - **predict(X)**: Predicts class labels for new data X.
  - **score(X, y)**: Calculates the model accuracy on data X and labels y.

SCIKIT-LEARN: MODEL

- **sklearn.linear_model.Perceptron** → one neuron
- Attributes:
  - **classes_**: List of class labels (usually [0, 1] for binary classification).
  - **coef_**: Weight vector of the trained hyperplane.
  - **intercept_**: Intercept term of the hyperplane (if fit_intercept=True).
  - **n_iter_**: Actual number of training iterations performed.
  - **loss_function_: T**he function that determines the loss

SCIKIT-LEARN: MODEL

Python

```python
from sklearn.datasets import make_blobs
from sklearn.linear_model import Perceptron

X, y = make_blobs(n_samples=100, centers=2, random_state=42)
model = Perceptron(max_iter=1000)
model.fit(X, y)
y_pred = model.predict(X)

accuracy = model.score(X, y)
print(f"Model accuracy: {accuracy:.4f}")
```

Python

```python
from sklearn.model_selection import train_test_split

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

model = Perceptron(warm_start=True)
for epoch in range(10):
    model.partial_fit(X_train, y_train)

y_pred = model.predict(X_test)
accuracy = model.score(X_test, y_test)
print(f"Perceptron accuracy on test set: {accuracy:.4f}")
```

SCIKIT-LEARN: MODEL

```
fit(X, y, coef_init=None, intercept_init=None, sample_weight=None)
```

Fit linear model with Stochastic Gradient Descent.

Parameters:	**X : *{array-like, sparse matrix}, shape (n_samples, n_features)*** Training data. **y : *ndarray of shape (n_samples,)*** Target values. **coef_init : *ndarray of shape (n_classes, n_features), default=None*** The initial coefficients to warm-start the optimization. **intercept_init : *ndarray of shape (n_classes,), default=None*** The initial intercept to warm-start the optimization. **sample_weight : *array-like, shape (n_samples,), default=None*** Weights applied to individual samples. If not provided, uniform weights are assumed. These weights will be multiplied with class_weight (passed through the constructor) if class_weight is specified.
Returns:	**self : *object*** Returns an instance of self.

SCIKIT-LEARN: MODEL

`predict(X)`

Predict class labels for samples in X.

Parameters:	**X :** ***{array-like, sparse matrix} of shape (n_samples, n_features)*** The data matrix for which we want to get the predictions.
Returns:	**y_pred :** ***ndarray of shape (n_samples,)*** Vector containing the class labels for each sample.

SCIKIT-LEARN: MODEL

score(*X*, *y*, *sample_weight=None*)

Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:	**X : *array-like of shape (n_samples, n_features)*** Test samples. **y : *array-like of shape (n_samples,) or (n_samples, n_outputs)*** True labels for `X`. **sample_weight : *array-like of shape (n_samples,), default=None*** Sample weights.
Returns:	**score : *float*** Mean accuracy of `self.predict(X)` w.r.t. `y`.

SCIKIT-LEARN: MODEL

- **sklearn.neural_network.MLPClassifier** → MLP

class **sklearn.neural_network.MLPClassifier**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

Python

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

# Load data and split
mnist = fetch_openml("mnist_784", version=1)
X = mnist.data / 255.0  # Normalize pixel values
y = mnist.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Define and train the network
model = MLPClassifier(hidden_layer_sizes=(128,), activation="relu", solver="adam")
model.fit(X_train, y_train)

# Evaluate and predict
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model accuracy on test set: {accuracy:.4f}")
```

SCIKIT-LEARN: MODEL

- **sklearn.neural_network.MLPClassifier** → MLP

Python

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

# Load and split data
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Define and train the network
model = MLPClassifier(hidden_layer_sizes=(16, 8), activation="tanh", solver="sgd")
model.fit(X_train, y_train)

# Evaluate and predict
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model accuracy on test set: {accuracy:.4f}")
```

SCIKIT-LEARN: MODEL

- **sklearn.neural_network.MLPClassifier** → MLP

Python

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

# Load and split data
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Define and train the network
model = MLPClassifier(hidden_layer_sizes=(16, 8), activation="tanh", solver="sgd")
model.fit(X_train, y_train)

# Evaluate and predict
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model accuracy on test set: {accuracy:.4f}")
```

```python
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# 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)

# Create an MLPClassifier with RMSProp optimizer and non-default parameters
mlp_classifier = MLPClassifier(
    hidden_layer_sizes=(100,),  # Adjust as needed
    max_iter=500,
    solver='rmsprop',           # Use RMSProp as the optimizer
    learning_rate_init=0.001,   # Initial learning rate
    alpha=0.001,                # L2 regularization parameter
    random_state=42
)

# Train the MLPClassifier on the training data
mlp_classifier.fit(X_train, y_train)

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

# Evaluate the performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Display classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```