Scikit-learn and Tour of Classifiers

First ML pipeline with Scikit-learn

Tour of Classifiers in scikit-learn

- Discuss a **selection** of **popular** and **powerful** machine learning algorithms **commonly** used in academia and industry
- Learn about the **differences** between several **supervised learning** algorithms for **classification** and their individual **strengths** and **weaknesses**
  - Logistic regression
  - Support vector machines
  - Decision trees
  - Random forest
  - K-nearest neighbours
- **First steps** with the scikit-learn library

Tour of Classifiers in scikit-learn

- Choosing an **appropriate** classification algorithm for a particular problem task requires practice
- **No Free Lunch theorem**: no single classifier works best across all possible scenarios
- “The **average performance** of **any pair** of algorithms across **all possible** problems is **identical**.” (Wolpert, D.H., Macready, W.G. (1997), No Free Lunch Theorems for Optimization, IEEE)
- If a specific algorithm seems to **outperform** another in a certain situation, it is a consequence of its **fit to the particular problem**, **not** the general superiority of the algorithm

Tour of Classifiers in scikit-learn

- Compare the performance of at least a **handful of different learning algorithms** to select the **best** model for the **particular** problem
- The problem at hand may be influenced by a number of contextual settings
  - Number of features
  - Number of samples
  - Distribution of the data
  - Amount of noise in the data
  - Whether classes are linearly separable or not
  - etc.

Tour of Classifiers in scikit-learn

- Performance of classifier:
  - Computational performance
  - Predictive power

Summary of five main steps in training a ML algorithm

1. **Selecting features** and **collecting training samples**
2. Choosing a **performance metric**
3. Choosing a **classifier** and **optimisation algorithm**
4. Evaluating the **performance of the model**
5. Tuning the **algorithm**

Training a perceptron with scikit-learn

Training Perceptron on Iris data set

- Imports

```
# ==============================================================================
# Import modules
# ==============================================================================

import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

scikit_learn_intro.ipynb

Training Perceptron on Iris data set

- Use two features (easy visualization)

```
# ==============================================================================
# Load data and select features
# ==============================================================================

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
```

scikit_learn_intro.ipynb

Training Perceptron – class label encoding

- Class labels: Scikit-learn also works with strings as class labels
- Converting to integers is recommended due to memory & runtime performance

Multi-class classification with OvR

Multi-class

We only discussed binary classifier, what about multiple classes?

OvR (One-vs-Rest) also called OvA (One-vs-All)

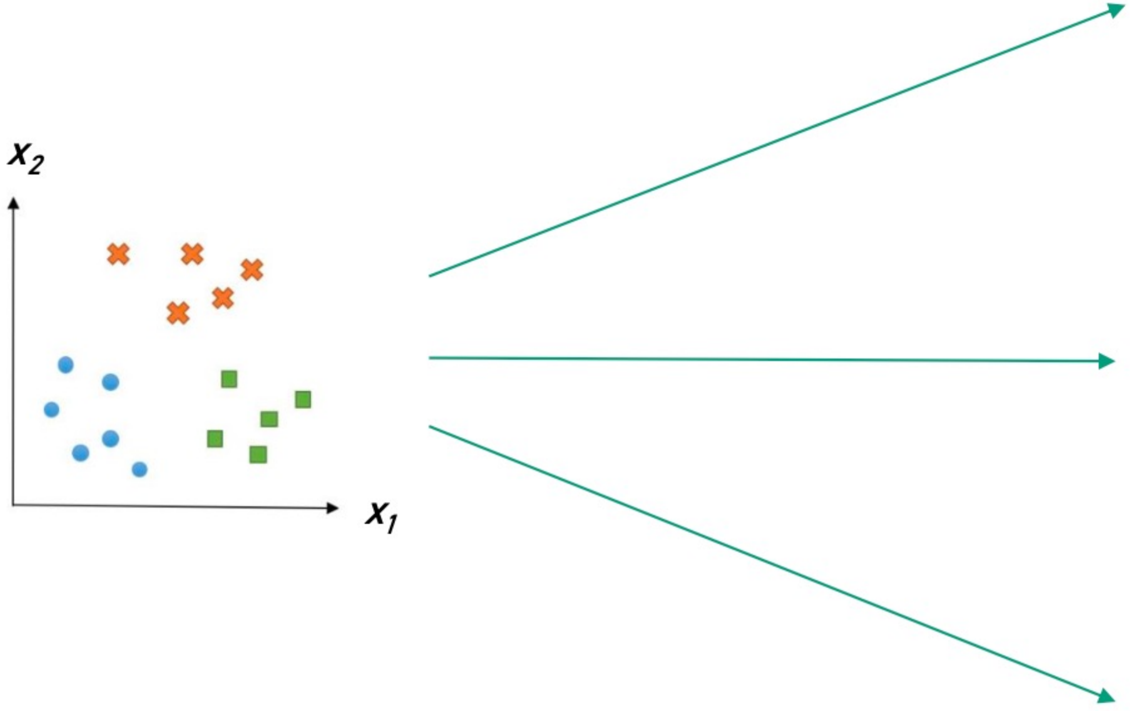
- a technique that allows us to extend a binary classifier to multi-class problems
- Using OvA, we can train one classifier per class, where a particular class is treated as the positive class and the samples from all other classes are considered negative classes

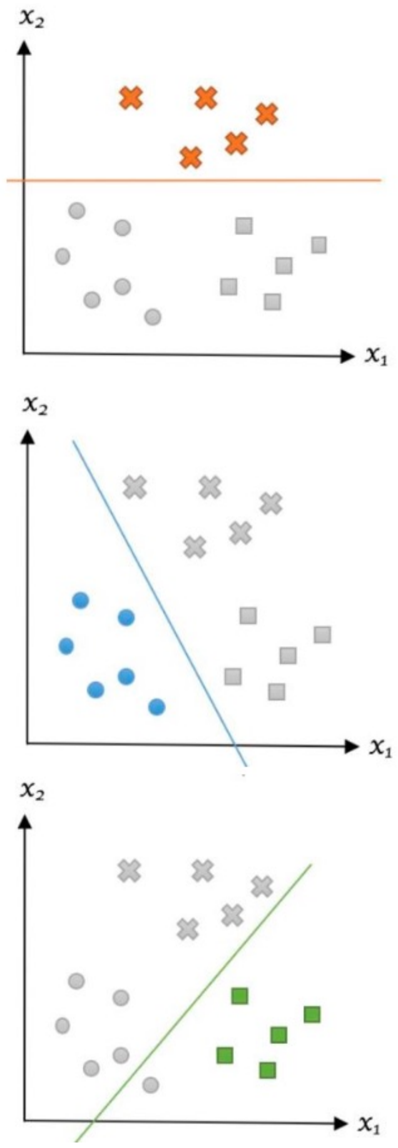
Classification of a new sample:

- use the *n* trained classifiers, where *n* is the number of class labels
- assign the class label with the highest confidence
- we usually want a classifier for this that can output probabilities (e.g. logistic regression)
- Perceptron: choose the class label that is associated with the largest net input value z

Multi-class

OvR (One-vs-Rest) also called OvA (One-vs-All)





Classifiers in scikit-learn automatically support this

Test/Train split

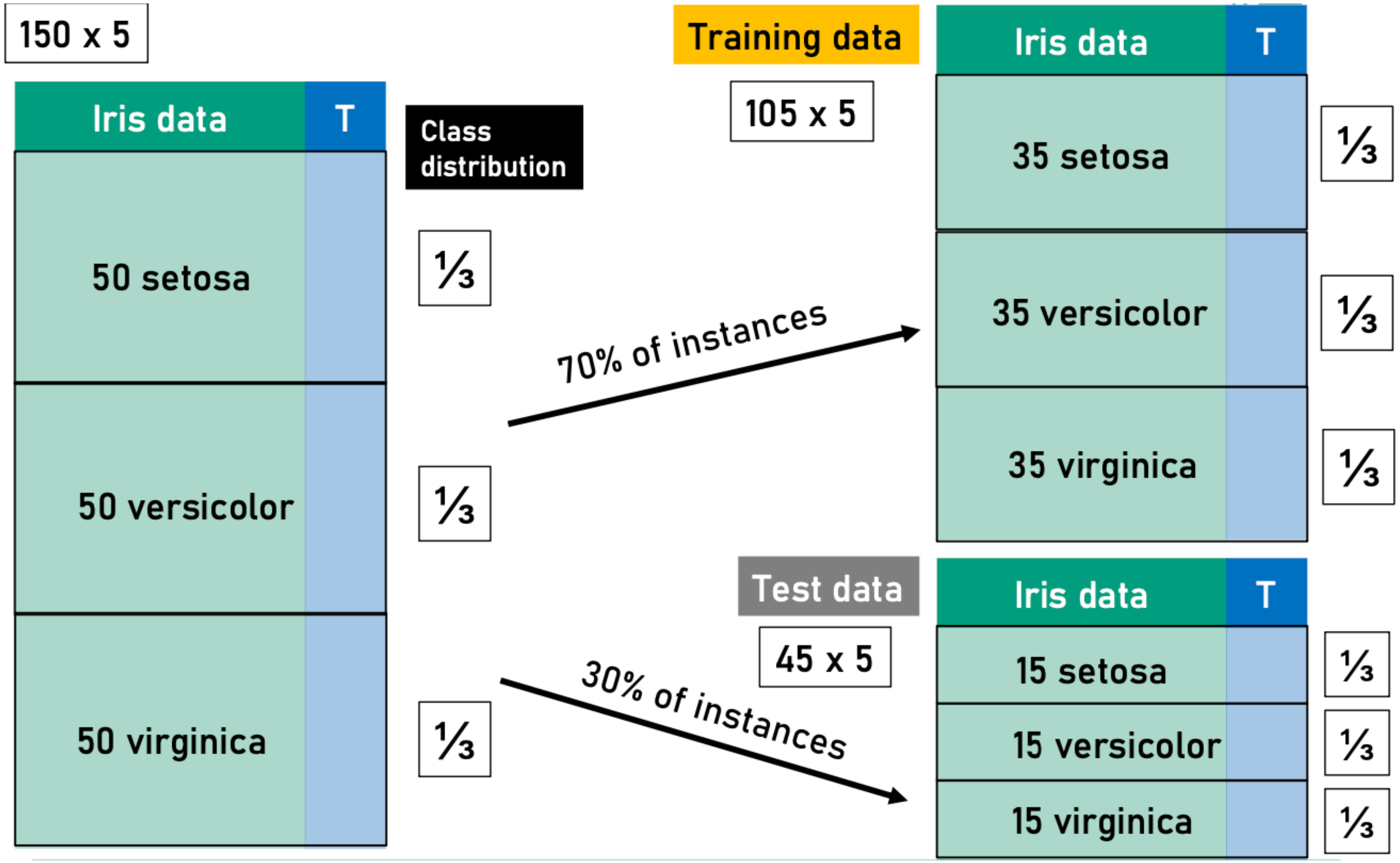
Training Perceptron – test/train split

- Split data into separate training and test datasets
- More details later in “Best practices for evaluation and hyperparameter tuning”
- Using the `train_test_split` function from scikit-learn's `model_selection` module
- Randomly split the X and y arrays into 30 percent test data (45 samples) and 70 percent training data (105 samples)

```
# Split data into training and test data (70% training, 30% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y
)
```

`scikit_learn_intro.ipynb`

Training Perceptron – test/train split



Training Perceptron – test/train split

- Properties of the `train_test_split` function
  - **Shuffle training** set internally **before** splitting → we don't need to worry about ordering of the samples prior to splitting
- Provides a `random_state` parameter for a **fixed random seed** (default: `random_state=0`) → «**reproducible**» shuffling of samples (handy when training multiple times)
- Built-in support for stratification with `stratify=y` → `train_test_split` method returns training and test subsets that have the **same proportions of class labels** as the **input dataset**

```
# Split data into training and test data (70% training, 30% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1, stratify=y
)
```

Training Perceptron – test/train split

- Checking distribution of classes in training and test set with numpy

```
# Show distribution of classes in input data, training data and test data
print(f"Labels counts in y: {np.bincount(y)}")
print(f"Labels counts in y_train: {np.bincount(y_train)}")
print(f"Labels counts in y_test: {np.bincount(y_test)}")
```

- Output:

```
Labels counts in y: [50 50 50]
Labels counts in y_train: [35 35 35]
Labels counts in y_test: [15 15 15]
```

scikit_learn_intro.ipynb

Training Perceptron – feature scaling

- Many machine learning and optimisation algorithms also require feature scaling for optimal performance
- Standardise the features using the `StandardScaler` class from scikit-learn's `preprocessing` module

```
# Initialise standard scaler and compute mean and stddev from training data
sc = StandardScaler()
sc.fit(X_train)

# Transform (standardise) both X_train and X_test with mean and stddev from
# training data
X_train_sc = sc.transform(X_train)
X_test_sc = sc.transform(X_test)
```

`scikit_learn_intro.ipynb`

Training Perceptron – feature scaling

Person	Height (cm)	Weight (kg)	Shoe size
Person A	174	55	46
Person B	188	92	45
Person C	158	65	42
Person D	202	110	49
Person E	171	96	44
Person F	193	79	48
Mean	181	82.833333	45.6667
STD	16.198765	20.507722	2.58199

Original data

Person	Height (cm)	Weight (kg)	Shoe size
Person A	-7	-27.833333	0.33333
Person B	7	9.1666667	-0.66667
Person C	-23	-17.833333	-3.66667
Person D	21	27.166667	3.33333
Person E	-10	13.166667	-1.66667
Person F	12	-3.8333333	2.33333
Mean	0.00	0.00	0.00
STD	16.198765	20.507722	2.58199

Centred data

Person	Height (cm)	Weight (kg)	Shoe size
Person A	-0.4321317	-1.3572123	0.1291
Person B	0.4321317	0.4469861	-0.2582
Person C	-1.4198613	-0.8695911	-1.42009
Person D	1.2963951	1.3247043	1.29099
Person E	-0.617331	0.6420346	-0.6455
Person F	0.7407972	-0.1869215	0.9037
Mean	0.00	0.00	0.00
STD	1	1	1

Standardised (scaled) data

`scikit_learn_intro.ipynb`

Training Perceptron – feature scaling

- Important: Scale test and training data with scaling obtained from training data!
- Use fit method, StandardScaler estimated the parameters μ (sample mean) and σ (standard deviation) for **each feature** dimension from the **training data**
- Call transform method: standardises training and test data using estimated the parameters μ and σ acquired from **training data** → values from training and test data are comparable to each other

```
# Initialise standard scaler and compute mean and stddev from training data
sc = StandardScaler()
sc.fit(X_train)

# Transform (standardise) both X_train and X_test with mean and stddev from
# training data
X_train_sc = sc.transform(X_train)
X_test_sc = sc.transform(X_test)
```

scikit_learn_intro.ipynb

Choose model

Training Perceptron

- Load the `Perceptron` class from the `linear_model` module
- Initialise a new Perceptron object
  - Parameter `eta0` defines learning rate
    Parameter `max_iter` defines the number of epochs (passes over the training set)
    - Train the model using the `fit` method

```
# Initialise the model
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_sc, y_train)
```

`scikit_learn_intro.ipynb`

Training Perceptron

- Finding an appropriate learning rate requires some experimentation
- If the learning rate is too large, the algorithm will overshoot the global cost minimum
- If the learning rate is too small, the algorithm requires more epochs until convergence → can make the learning slow—especially for large datasets
- The `random_state` parameter ensures the reproducibility of the initial shuffling of the training dataset after each epoch

```
# Initialise the model
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_sc, y_train)
```

`scikit_learn_intro.ipynb`

Evaluation/Prediction
Performance metrics: accuracy

Training Perceptron – Predictions

```
# Predict classes for samples in test set and print number of misclassfications
y_pred = ppn.predict(X_test_sc)
print("Misclassified samples: {0}".format((y_test != y_pred).sum()))
```

```
Misclassified samples: 3
```

```
# Print accuracy computed from predictions on the test set
print("Accuracy: {0:.2f}".format(accuracy_score(y_test, y_pred)))


# Print accuracy computed from predictions on the test set
print("Accuracy: {0:.2f}".format(ppn.score(X_test_sc, y_test)))
```

Misclassification error: 3 out of 45→3 / 45 ≈ 0.067 or approx. 6.7%
Often the **accuracy** is reported instead of misclassification error:
1 – error = 0.933 = 93.3%

scikit_learn_intro.ipynb

Training Perceptron – Accuracy

- Simple performance metric; method `score` also outputs accuracy

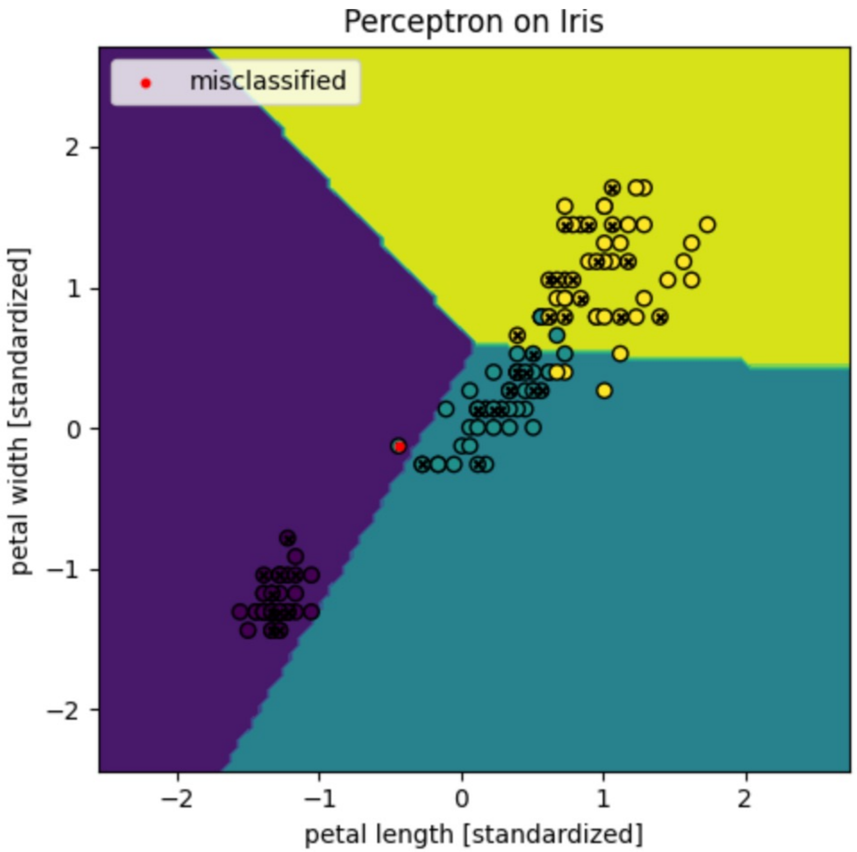
```
# Print accuracy computed from predictions on the test set
print("Accuracy: {0:.2f}".format(accuracy_score(y_test, y_pred)))


# Print accuracy computed from predictions on the test set
print("Accuracy: {0:.2f}".format(ppn.score(X_test_sc, y_test)))
```

`scikit_learn_intro.ipynb`

Training Perceptron – Plot decision boundary

- The three flower classes cannot be perfectly separated by a linear decision boundary
- Algorithm would go on forever, scikit-learn has a stopping criterion if no improvement



scikit_learn_intro.ipynb

Training Perceptron – Exercise

- Apply what we saw to Wine data set
- Train a `Perceptron` on the wine data (`datasets.load_wine`) that is integrated in scikit-learn
  - Use two features of the dataset: ‘`alcohol`’ and ‘`hue`’
  - Train – test – split: 60 percent of data for training, random state should be set to 5
  - Perceptron: Set maximum number of iterations to 100
  - Perceptron: Set learning rate to 0.05
  - Perceptron: Set random state to 77
  - Plot decision regions for training and test data

  - Extra (look in online documentation of Perceptron): 1. What is the test accuracy? 2. How many epochs for training? 3. How often were the weights updated? 4. What are the final weights?

`scikit_learn_wine_solution.ipynb`

Training Perceptron – Exercise 2

- Train `LogisticRegression` on the wine data set (`datasets.load_wine`)
  - Use two features of the dataset: '`alcohol`' and '`hue`'
  - Train – test – split: 60 percent of data for training, random state should be set to 5
  - `LogisticRegression`: Set maximum number of iterations to 100
  - `LogisticRegression`: Set random state to 77
  - Plot decision regions for training and test data

`scikit_learn_wine_logreg_solution.ipynb`