UIC BUSINESS

Business Data Mining

IDS 472 (Spring 2024)

Instructor: Wenxin Zhou

Supervised Learning

UIC BUSINESS

Two main tasks of supervised learning: classification and regression

Scikit-Learn: the most popular Python-based machine learning software

A classification example

- data splitting, normalization
- train a classifier
- prediction
- evaluation

Supervised Learning

UIC BUSINESS

- Goal: learn a mapping between a vector of input variables (*predictors*/*feature vector*) and output variables (*target*/*response*/*outcome*)
- Assume a single output variable, y, is available
- Training data $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$

  where $\mathbf{x}_i \in \mathbb{R}^p, i = 1, \ldots, n,$ represents a vector of p feature variables, n is the number of observations (sample size)
  - In classification, values for y_i belong to a set of finite categories called *labels*
  - The goal is to assign a given feature vector to one of the class labels

Scikit-Learn

UIC BUSINESS

- In regression, y_i represents realizations of a numeric rv
- The goal is to estimate the target y for a given feature vector $\boldsymbol{x}$ (predicting y for a given $\boldsymbol{x}$)

- Scikit-Learn is a Python package that contains an efficient and uniform API to implement many ML methods

- Three fundamental objects in scikit-learn are *Estimators*, *Transformers*, and *Predictors*

Estimators

UIC BUSINESS

- Any object that can estimate some parameters based on a dataset is called an `estimator`
- All ML models, whether classifiers or regressors, are implemented in their own `Estimator` class
  - k-nearest neighbors (kNN) rule is implemented in the `KNeighborsClassifier` class from `sklearn.neighbors`
  - Perceptron is implemented in the `Perceptron` class in the `linear_model` module
- All estimators implement the `fit()` method that takes either one argument (data) or two (in supervised learning) where the second argument represents target values

`estimator.fit(data, targets)` or `estimator.fit(data)`

Transformers & Predictors

UIC BUSINESS

- Estimators that can also transform data are transformers and implement `transform()` or `fit_transform()` method to perform the transformation of data

```
new_data = transformer.transform(data)
new_data = transformer.fit_transform(data)
```

- Some estimators (predictors) can make predictions given a data, and implement `predict()` method to perform prediction

```
prediction = predictor.predict(data)
probability = predictor.predict_proba(data)
```

Iris Flower Classification

UIC BUSINESS

- Train a classifier that receives a feature vector containing morphologic measurements (length and width of petals and sepals in centimeters) of Iris flowers and classifies a given feature vector to one of the three Iris flower species: setosa, virginica and versicolor

- Underlying hypothesis: an Iris flower can be classified into its species based on its petal and sepal lengths and widths

- Feature vectors $\boldsymbol{x}_i$ are 4-dimensional ($p = 4$), y takes 3 values

- This is a *multiclass classification* (three-class) problem

Iris Flower Classification

UIC BUSINESS

- *Iris dataset* is a well-known dataset in ML
- This dataset is part of scikit-learn and can be accessed by importing its `datasets` module

```
from sklearn import datasets # sklearn is the Python name of␣
 ↪scikit-learn
iris = datasets.load_iris()
type(iris)
```

```
sklearn.utils.Bunch
```

- Datasets that are part of scikit-learn are stored as "Bunch" objects, that is, an object of class `sklearn.utils.Bunch`
  – It contains the actual data as well as some information about it
  – These information are stored in `Bunch` objects similar to dictionaries (using keys and values)
  – Use `key()` method to see all keys in a `Bunch` object

Data Preview

UIC BUSINESS

```
iris.keys()
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR',␣
↪'feature_names', 'filename'])
```

```
print(iris['target_names']) # or, equivalently, print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

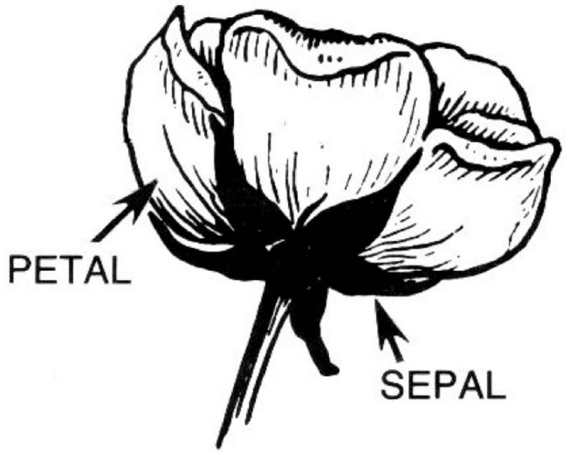
```
print(iris.DESCR[:500])
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
```



Data Preview

UIC BUSINESS

- All measurements (i.e. feature vectors) are stored as values of `data` key. For example, the first 10 feature vectors are

```
iris.data[:10]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1]])
```

- The matrix containing all feature vectors is called *data matrix* or *feature matrix*, which has shape sample size × feature size

Data Preview

UIC BUSINESS

- In this data there are 150 Iris flowers, and for each there are 4 features

```
iris.data.shape
```

```
(150, 4)
```

- The corresponding targets (in the same order as feature vectors) can be accessed through the `target` field

```
iris.target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

- The three classes are *encoded* as 0, 1, 2 (integer encoding)

Data Preview

UIC BUSINESS

- Use the `bincount()` method to count the number of samples in each class

```
import numpy as np
np.bincount(iris.target)
```

```
array([50, 50, 50])
```

- Check the type of data matrix

```
print('type of data: ' + str(type(iris.data))+ '\ntype of target: ' +␣
↪str(type(iris.target)))
```

```
type of data: <class 'numpy.ndarray'>
type of target: <class 'numpy.ndarray'>
```

- Check feature names

```
print(iris.feature_names) # the name of features
```

Test Set for Model Assessment

UIC BUSINESS

- After we train a model based on training data, a major question is how well the model performs (*predictive capacity* of the trained model)
- There are various evaluation rules. The most common one is to evaluate a trained model on a test data (*test set*)
- Using the `train_test_split` function from the `sklearn.model_selection` module, we randomly split the data into a training set and a test set
  - By default, `train_test_split` shuffles the data before splitting to avoid possible systematic biases in the data
  - The `test_size` argument represents the proportion of test set; default is 0.25
  - It is a good practice to keep the proportion of classes in both training and test sets in the whole data; `stratify=iris.target`

Test Set for Model Assessment

UIC BUSINESS

- Use stratified random split to divide the given data into 80% training and 20% test

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test= train_test_split(iris.data, iris.
↪target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +␣
↪str(X_test.shape)\
      + '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: '␣
↪+ str(y_test.shape))
```

```
X_train_shape: (120, 4)
X_test_shape: (30, 4)
y_train_shape: (120,)
y_test_shape: (30,)
```

```
np.bincount(y_train)
```

```
array([40, 40, 40])
```

Data Visualization

UIC BUSINESS

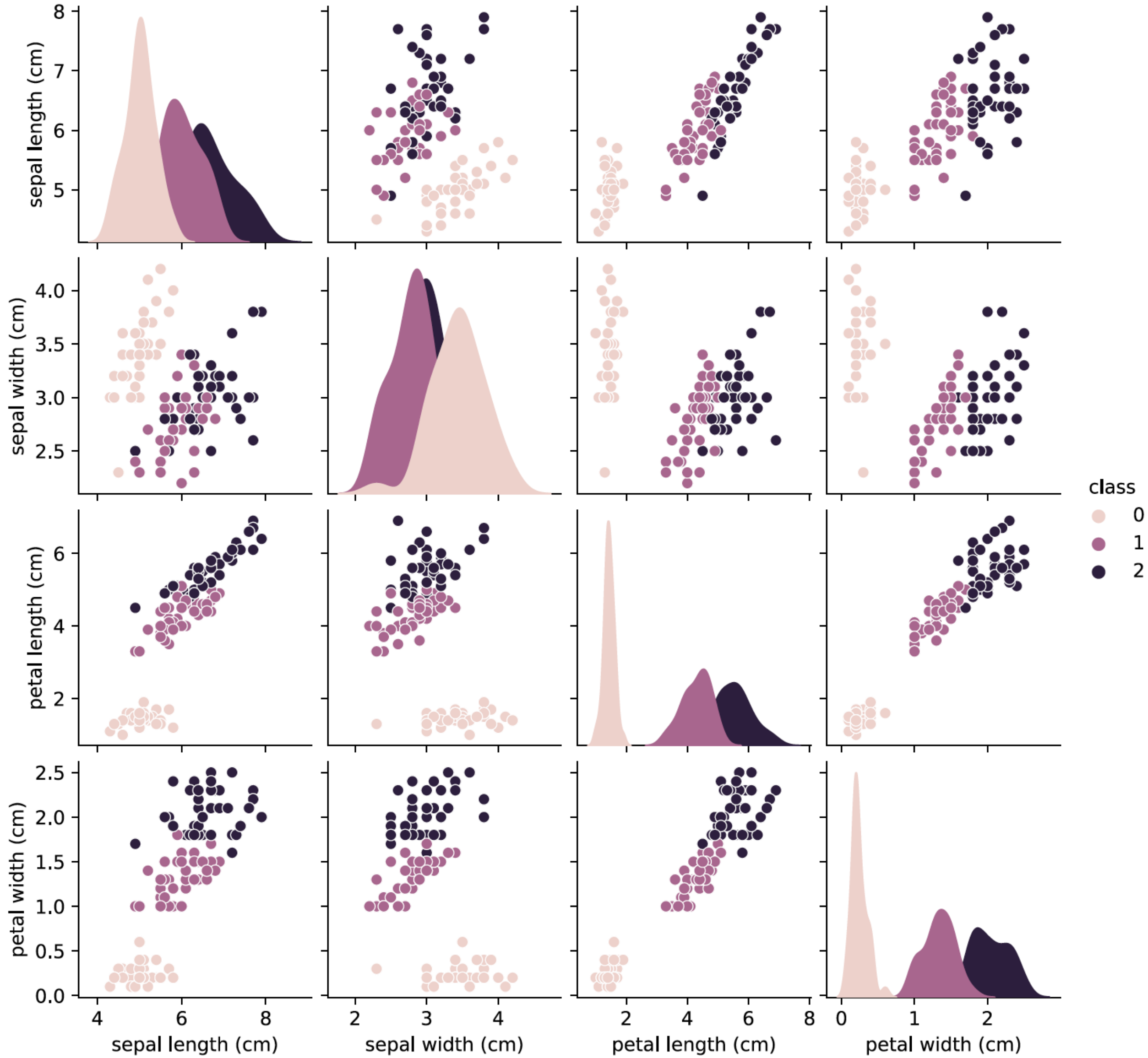
- Visualization is a good exploratory analysis: reveal possible abnormalities, provide an insight into the hypothesis behind the entire experiment
- *Pair plots* display scatter plots between all pairs of variables
- Use `pairplot()` function from `seaborn` library, which expects a `DataFrame` as input

```
import pandas as pd
X_train_df = pd.DataFrame(X_train, columns=iris.feature_names)
y_train_df = pd.DataFrame(y_train, columns=['class'])
X_y_train_df = pd.concat([X_train_df, y_train_df], axis=1)
```

```
import seaborn as sns
sns.pairplot(X_y_train_df, hue='class', height=2) # hue is set to the
 ↪class variable in the dataframe so that they are plotted in
 ↪different color and we are able to distinguish classes
```

Data Visualization

UIC BUSINESS



Feature Scaling/Normalization

UIC BUSINESS

- Values of features in a dataset often come in different scales, e.g. [0, 1] vs thousands or millions
- It's common to apply some *feature scaling/normalization* to the training data to make the scale of features "comparable"
- ML methods such as kNN, neural networks, ridge regression, etc benefit from feature scaling

Even if we are not sure whether a specific ML rule benefits from scaling or not, it would be helpful to still scale the data because scaling is not harmful and, at the same time, it facilitates comparing different types of models regardless of whether they can or can not benefit from scaling.

Feature Scaling/Normalization

UIC BUSINESS

- Two common way for feature scaling are *standardization* and *min-max scaling*
  - standardization: for each feature, subtract the mean (of that feature) and divide by its standard deviation
  - min-max scaling: subtract the minimum and divide by the range (maximum – minimum)

```
mean = X_train.mean(axis=0) # to take the mean across rows (for each␣
↪column)
std = X_train.std(axis=0) # to take the std across rows (for each␣
↪column)
X_train_scaled = X_train - mean # notice the utility of broadcasting
X_train_scaled /= std
```

```
X_train_scaled.mean(axis=0) # observe the mean is 0 now
```

```
array([-4.21884749e-16,  1.20042865e-16, -3.88578059e-16, -7.
↪04991621e-16])
```

Feature Scaling/Normalization

UIC BUSINESS

```
X_train_scaled.std(axis=0) # observe the std is 1 now
```

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

- A common *mistake* is to apply some data preprocessing such as normalization *before* splitting the entire data into training and test sets
- This causes data leakage: some information about the test set leak into the training process
- Once the relevant statistics (e.g. mean and standard deviation) are estimated from training set, they can be used to normalize test set

```
X_test_scaled = X_test - mean
X_test_scaled /= std
```

Feature Scaling/Normalization

UIC BUSINESS

- Use `StandardScaler` and `MinMaxScaler` classes from `sklearn.preprocessing` module

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

- To estimate the mean and std of each feature from training set, call the `fit()` method from the `scaler` object

```
scaler.fit(X_train)
```

```
StandardScaler()
```

- The `scaler` object holds any information that the standardization algorithm implemented in `StandardScaler` class extracts from `X_train`
- Next, call the `transform()` method of the `scaler` object to transform the training and test sets

Feature Scaling/Normalization

UIC BUSINESS

- For later use, the training and testing arrays can be saved using `numpy.save()` to binary files
- For saving multiple arrays, use `numpy.savez()`

```
np.savez('data/iris_train_scaled', X = X_train_scaled, y = y_train)
np.savez('data/iris_test_scaled', X = X_test_scaled, y = y_test)
```

- `numpy.save` and `numpy.savez` add `npy` and `npz` extensions to the name of a created file, respectively
- Later, arrays could be loaded by `numpy.load()`

```
tuple_X, tuple_y = np.load('data/iris_train_scaled.npz').items()
tuple_X
```

✓ 0.0s

```
('X',
 array([[-0.09321506, -1.12530064,  0.12847063, -0.01411418],
        [ 0.02706244, -0.18591924,  0.24099965,  0.37673996],
        [ 0.74872744, -0.18591924,  0.97243829,  0.7675941 ],
        [ 0.98928243,  0.04892611,  0.35352867,  0.24645525],
        [ 2.43261243,  1.69284358,  1.47881888,  1.02816352],
```

Model Training

UIC BUSINESS

- We use kNN classification rule to train an ML model
- To classify a test point, one can think that the kNN classifier grows a spherical region centered at the test point until it encloses k training samples, and classifies the test point to the majority class among these k training samples
- For example, 5NN assigns "green" to the test observation because within the 5 nearest observations to this test point, three are from the green class
- kNN classifier is implemented in the `KNeighborsClassifier` class in the `sklearn.neighbors` module

```
from sklearn.neighbors import KNeighborsClassifier as kNN # giving an␣
↪alias KNN for simplicity
knn = KNN(n_neighbors=3) # hyperparameter k is set to 3; that is, we␣
↪have 3NN
```

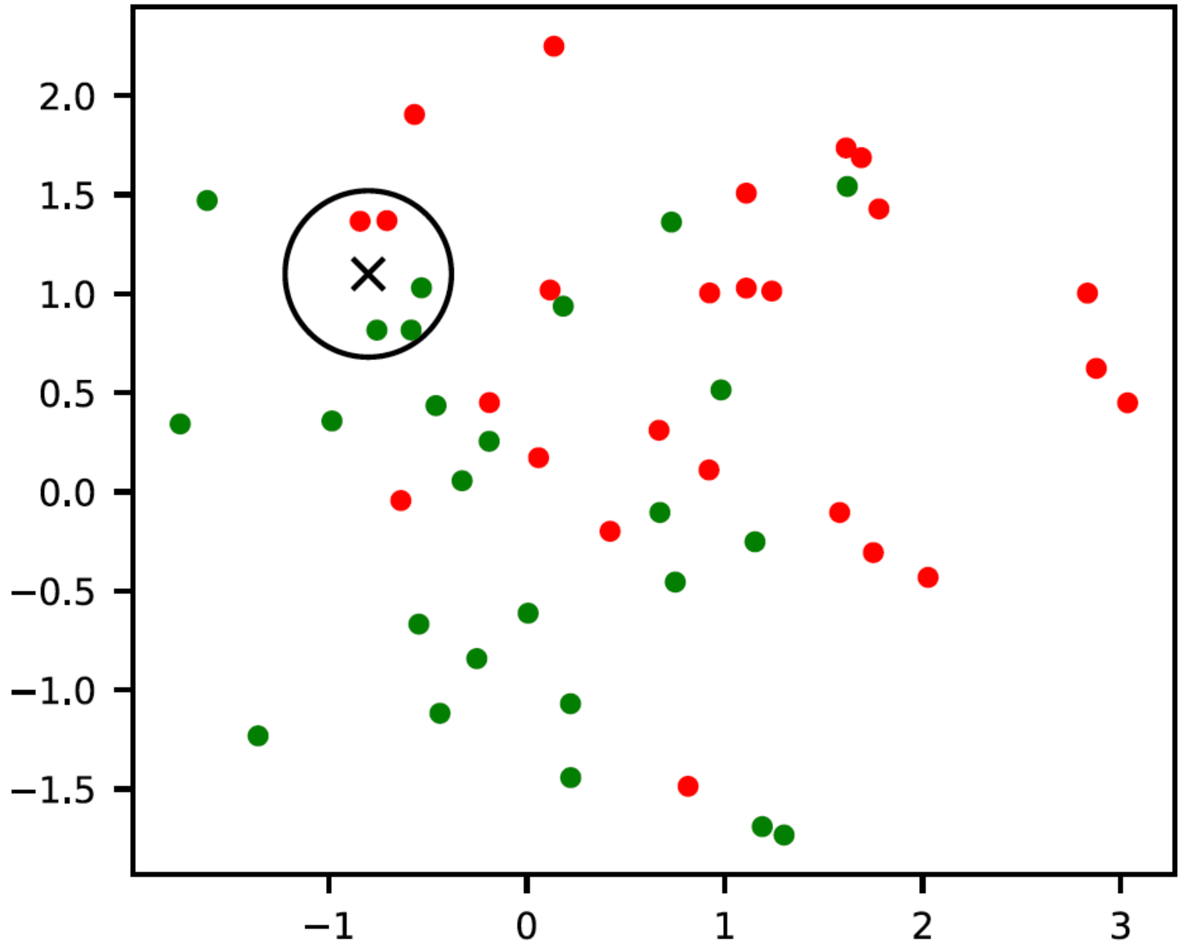
```
knn.fit(X_train_scaled, y_train)
```

```
KNeighborsClassifier(n_neighbors=3)
```

Model Training

UIC BUSINESS

- The working principle of kNN ($k = 5$): the test point is identified by $\times$; the circle encloses 5 nearest neighbors of the test point



Prediction using Trained Model

UIC BUSINESS

- Some estimator in scikit-learn are predictors, which can make prediction on a new data point by implementing the `predict()` method

```
x_test = np.array([[5.5, 2, 1.1, 0.6]]) # same as: np.array([5.5, 2, 1.
↪1, 0.6]).reshape(1,4)
x_test.shape
```

```
(1, 4)
```

```
x_test_scaled = scaler.transform(x_test)
x_test_scaled
```

```
array([[-0.45404756, -2.53437275, -1.50320017, -0.79582245]])
```

```
y_test_prediction = knn.predict(x_test_scaled)
print('knn predicts: ' + str(iris.target_names[y_test_prediction])) #␣
↪to convert the prediction (y_test_prediction) to the names of Iris␣
↪flower
```

```
knn predicts: ['versicolor']
```

Prediction using Trained Model

UIC BUSINESS

- Give several sample points as the argument to the `predict()` method, and receive assigned labels

```
y_test_predictions = knn.predict(X_test_scaled)
print('knn predicts: ' + str(iris.target_names[y_test_predictions])) #␣
↪fancy indexing in Section 3.1.4
```

```
knn predicts: ['versicolor' 'versicolor' 'versicolor' 'virginica'␣
↪'setosa' 'virginica' 'versicolor' 'setosa' 'versicolor' 'versicolor'␣
↪'versicolor' 'virginica' 'virginica' 'setosa' 'virginica' 'setosa'␣
↪'setosa' 'versicolor' 'setosa' 'virginica' 'setosa' 'versicolor'␣
↪'versicolor' 'setosa' 'versicolor' 'setosa' 'setosa' 'versicolor'␣
↪'virginica' 'versicolor']
```

```
y_test_predictions = KNN(n_neighbors=3).fit(X_train_scaled, y_train).
↪predict(X_test_scaled)
print('knn predicts: ' + str(iris.target_names[y_test_predictions]))
```

```
knn predicts: ['versicolor' 'versicolor' 'versicolor' 'virginica'␣
↪'setosa' 'virginica' 'versicolor' 'setosa' 'versicolor' 'versicolor'␣
↪'versicolor' 'virginica' 'virginica' 'setosa' 'virginica' 'setosa'␣
↪'setosa' 'versicolor' 'setosa' 'virginica' 'setosa' 'versicolor'␣
↪'versicolor' 'setosa' 'versicolor' 'setosa' 'setosa' 'versicolor'␣
↪'virginica' 'versicolor']
```

Model Evaluation

UIC BUSINESS

- The simplest and the most intuitive rule or metric to assess the performance of a classifier is the proportion of misclassified points in a test set (test-set estimator of *error rate*)
- The proportion of misclassified observations in the test set is an estimate of classification error rate denoted ε, defined as *the probability of misclassification by the trained classifier*
- $\mathbf{X}$: random feature vector, Y: binary random variable
- *Joint feature-label distribution* is

$$P(\mathbf{X} \in E, Y = i) = \int_E p(\mathbf{x}|Y = i)P(Y = i)d\mathbf{x}, \quad i = 0, 1,$$

  where $p(\mathbf{x}|Y = i)$ is the *class-conditional probability density function*, E represents an event
- $P(Y = i)$ is the *prior probability* of class i

Model Evaluation

UIC BUSINESS

- Given a training set $\mathbf{S}_{tr}$, train a classifier $\psi: \mathbb{R}^p \to \{0,1\}$, which maps realizations of $\mathbf{X}$ to realizations of Y
- Let E_0 denote events for which $\psi(\mathbf{X})$ gives label 0, that is, $E_0 = \{\psi(\mathbf{X}) = 0\}$. The joint prob of E_0 and $Y = 1$ is

$$P(\mathbf{X} \in E_0, Y = 1) \stackrel{1}{=} \int_{E_0} p(\mathbf{x}|Y = 1)P(Y = 1)d\mathbf{x} \equiv \int_{\psi(\mathbf{x})=0} p(\mathbf{x}|Y = 1)P(Y = 1)d\mathbf{x}$$

- Let E_1 denote events for which $\psi(\mathbf{X})$ gives label 1. Similarly,

$$P(\mathbf{X} \in E_1, Y = 0) = \int_{E_1} p(\mathbf{x}|Y = 0)P(Y = 0)d\mathbf{x} \equiv \int_{\psi(\mathbf{x})=1} p(\mathbf{x}|Y = 0)P(Y = 0)d\mathbf{x}$$

- The prob of misclassification ε is

$$\varepsilon = P(\mathbf{X} \in E_0, Y = 1) + P(\mathbf{X} \in E_1, Y = 0) \equiv P(\psi(\mathbf{X}) \neq Y)$$

Model Evaluation

UIC BUSINESS

- Apply the classifier on test set $\boldsymbol{S}_{te}$ that contains m observations with labels
- Let k be the num of observations in $\boldsymbol{S}_{te}$ that are misclassified
- The test-set error estimate $\hat{\varepsilon}_{te}$ is $\hat{\varepsilon}_{te} = \frac{k}{m}$
- The accuracy (acc) and its test-set estimate ($\widehat{acc}_{te}$) are

$$acc = 1 - \varepsilon,$$

$$a\hat{c}c_{te} = 1 - \hat{\varepsilon}_{te}$$

```
errors = (y_test_predictions != y_test)
errors
```

```
array([False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, False, False,
        True, False, False])
```

- In this case, $\hat{\varepsilon}_{te} = 3/30 = 0.1$

Model Evaluation

UIC BUSINESS

```
error_est = sum(errors)/errors.size
print('The error rate estimate is: {:.2f}'.format(error_est) + '\n'\
      'The accuracy is: {:.2f}'.format(1-error_est))
```

```
The error rate estimate is: 0.10
The accuracy is: 0.90
```

- Here we use the place holder { } and format specifier 0.2f to specify the number of digits after decimal point
- Many performance metrics can be easily calculated using scikit-learn built-in functions from `metrics` module

```
from sklearn.metrics import accuracy_score
print('The accuracy is {:.2f}'.format(accuracy_score(y_test,␣
 ↪y_test_predictions)))
```

```
The accuracy is 0.90
```

- All classifiers in scikit-learn have a `score` method that given a test data and its labels, returns the classifier accuracy

```
print('The accuracy is {:.2f}'.format(knn.score(X_test_scaled, y_test)))
```

```
The accuracy is 0.90
```