# Section 9 Introduction to Data Science

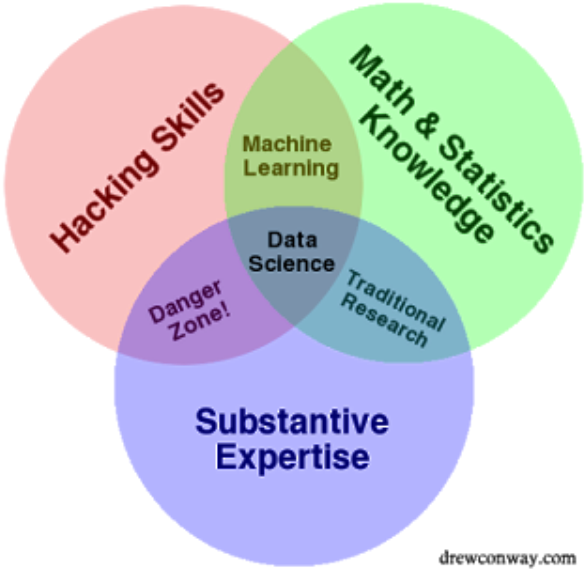
# What is Data Science?

Three correlated concepts:

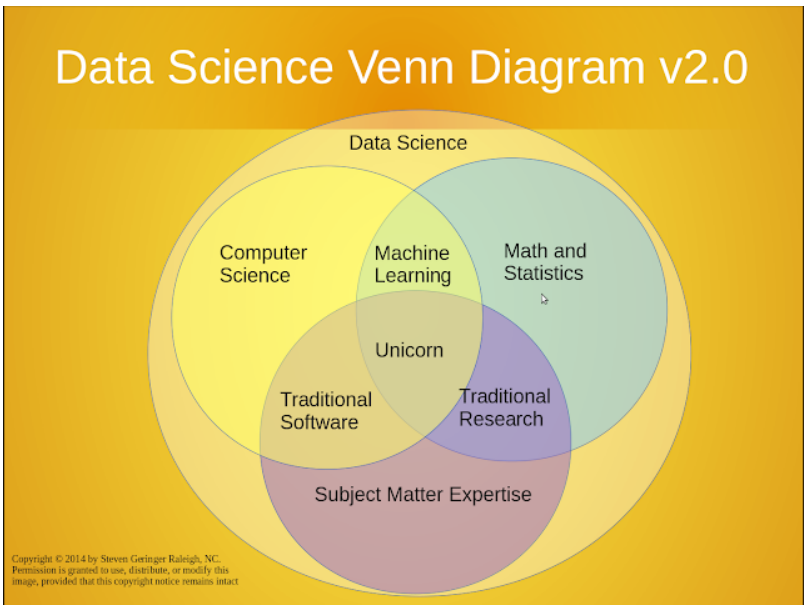
- Data Science
- Artificial Intelligence
- Machine Learning

Battle of the Data Science Venn Diagrams

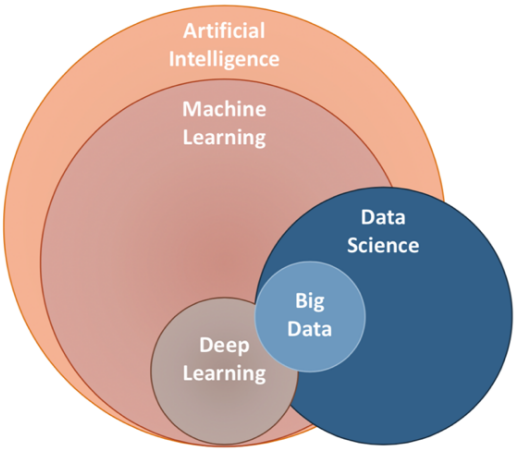
The original Venn diagram from Drew Conway:



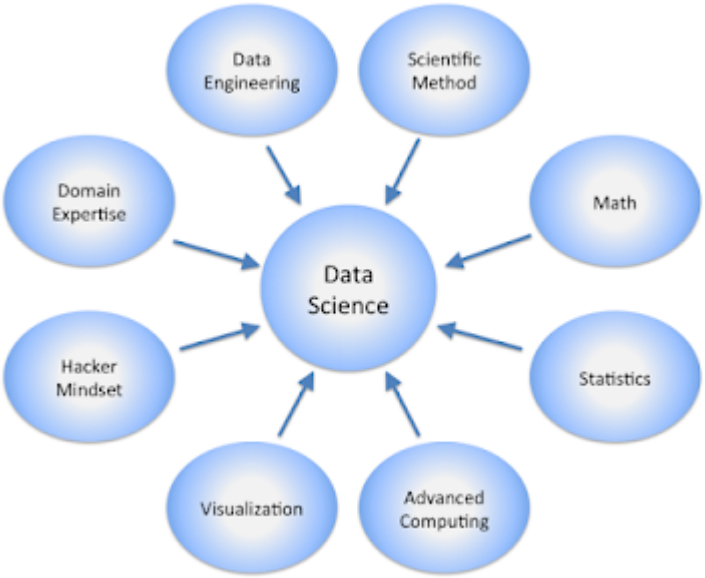

Another diagram from Steven Geringer:

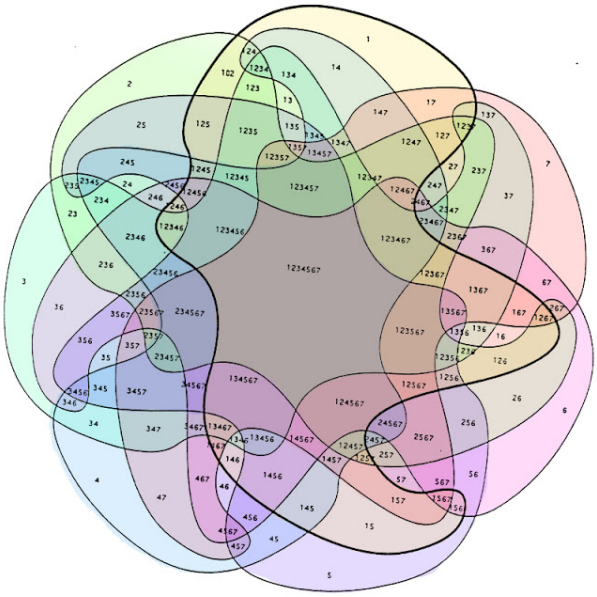



Another version:




Perhaps the reality should be:



David Robinson's Auto-pilot example:

- machine learning: **predict** whether there is a stop sign in the camera

- artificial intelligence: design the **action** of applying brakes (either by rules or from data)
- data science: provide the **insights** why the system does not work well after sunrise

**Peijie's Definition**: Data Science is the science

- *of* the data -- what
- *by* the data -- how
- *for* the data -- why

# Mathematics of Data

## Representation of Data

What data do we have, and how to relate it with math objects?

### Tabular Data

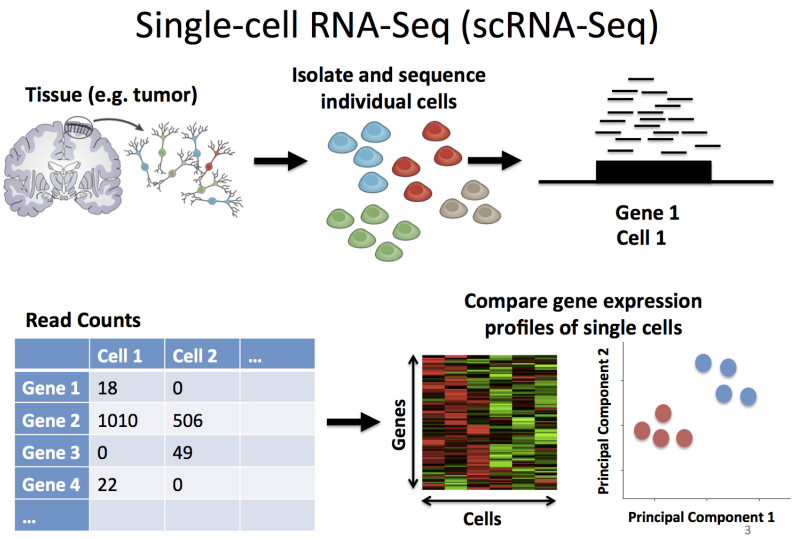
In [ ]:
```
import pandas as pd
import numpy as np
df_house = pd.read_csv('./data/kc_house_data.csv')
print(df_house.shape)
df_house.head()
```

- A structured data table, with $n$ observations and $p$ variables.
- **Mathematical representation**: The data *matrix* $X \in \mathbb{R}^{n\times p}$. For notations we write

$$X = \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \cdots \\ \mathbf{x}^{(n)} \end{pmatrix}, \text{ where the } i\text{-th row vector represents } i\text{-th observation,}$$

$$\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_p^{(i)}) \in \mathbb{R}^p.$$

- Example: Precision Medicine and Single-cell Sequencing.



- *Roughly speaking*, big data -- large $n$, high-dimensional data -- large $p$.

## Time-series Data

In [ ]:

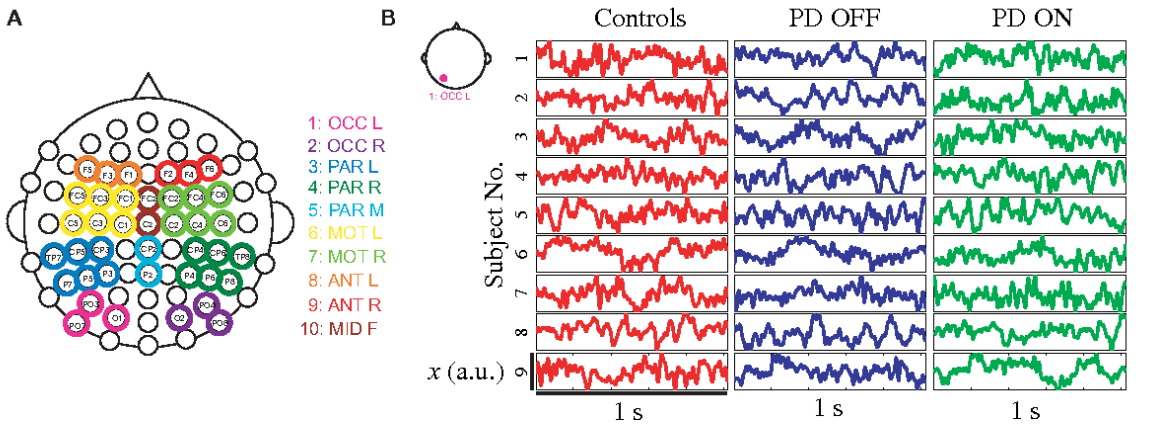
```
import matplotlib.pyplot as plt
ts_tesla = pd.read_csv('./data/Tesla.csv')
print(ts_tesla.head())

ts_tesla['Date'] = pd.to_datetime(ts_tesla['Date'])
ts_tesla.set_index('Date',inplace=True)

# Suppose we only focus on the time-series of close price
plt.figure(dpi=80)
plt.title('Close Price History')
plt.plot(ts_tesla['Close'], color='red')
plt.xlabel('Date', fontsize=18)
plt.ylabel('Close Price USD', fontsize = 18)
plt.show()
# this is only about tesla -- we can also have the time-series of apple,amazon,facebook
```

- Simple case: $N$ one-dimensional trajectories with each sampled at $T$ time points.
- **Mathematical representation I**: Still use the data *matrix* $X \in \mathbb{R}^{N\times T}$. For notations we write

$$X = \begin{pmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \cdots \\ \mathbf{x}^{(N)} \end{pmatrix}, \text{ where the } i\text{-th row vector represents } i\text{-th trajectory,}$$

$$\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_T^{(i)}) \in \mathbb{R}^T.$$

- Question: The difference with tabular data?
- **Mathematical representation II**: Each trajectory is a *function* of time $t$. The whole dataset can be represented as $z = f(\omega, t)$ where $\omega$ represents the sample and $t$ represents the time. In probability theory, this is called *stochastic process*.
  - For fixed $\omega$, we have a trajectory, which is the function of time.
  - For fixed $t$, we obtain an ensemble drawn from random distribution.
- Question: How about $N$ $d$-dimensional trajectories with each sampled at $T$ time points?
- Example: Electroencephalography (EEG) data and Parkinson's disease.




## Images

Example: MNIST handwritten digits data:Each image is 28x28 matrix

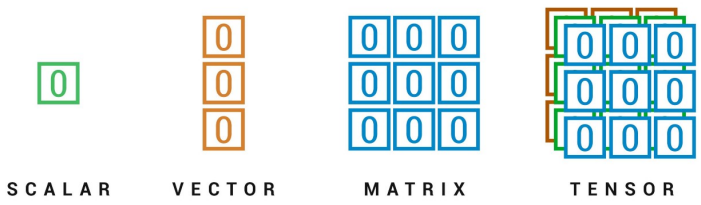
In [ ]:
```
import pandas as pd
mnist = pd.read_csv('./data/train.csv') # stored as data table
mnist.sample(5)
```

In [ ]:
```
mnist.shape
```

In [ ]:
```
target = mnist['label']
mnist = mnist.drop("label",axis=1)

import matplotlib.pyplot as plt
plt.figure(dpi=100)
for i in range(0,70): #plot the first 70 images
    plt.subplot(7,10,i+1)
    grid_data = mnist.iloc[i,:].to_numpy().reshape(28,28)  # reshape from 1d to 2d pixe
    plt.imshow(grid_data,cmap='gray_r', vmin=0, vmax=255)
    plt.xticks([])
    plt.yticks([])
plt.tight_layout()
```

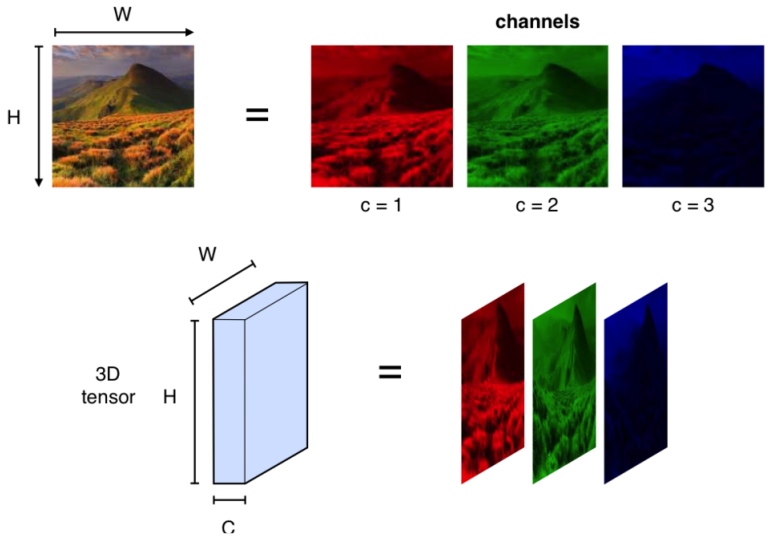
- Simple case: N grayscale images with $m \times n$ pixels each.
- **Mathematical Representation I**: Each image can be represented by a matrix $I \in \mathbb{R}^{m \times n}$, whose elements denotes the intensities of pixels. The whole datasets have $N$ matrices of $m$ by $n$, or represented by a $N \times m \times n$ *tensor*.

Illustrated Introduction to Linear Algebra using NumPy




- **Mathematical representation II**: *Random field model* $z = \mathbf{f}(\omega, x, y)$.
- **Color images**: Decompose into RGB (red,green and blue) channels and
  - use three matrices (or three-dimensional tensor) to represent one image, or
  - build the random field model with vector-valued functions $z = \mathbf{f}(\omega, x, y) \in \mathbb{R}^3$

convolutional neural networks




- Question: Can image datasets also be transformed into tabular data? What are the pros/cons?

In [ ]:
```
mnist.head()
```

## Videos

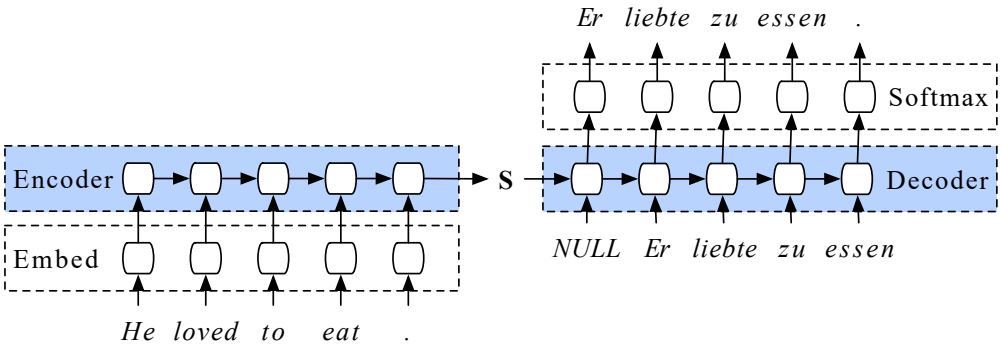
- *Time-series* of images, or *random field* model $z = \mathbf{f}(\omega, x, y, t)$

## Texts

In [ ]:
```
from sklearn.feature_extraction.text import CountVectorizer

corpus = ['He is a good person',
          'He is bad student',
          'He is hardworking']
df = pd.DataFrame(data=corpus, columns=['sentences'])
print(df)
vectorizer = CountVectorizer(vocabulary=['he', 'is', 'a', 'good', 'person', 'bad', 'stu
                             stop_words=frozenset(), token_pattern=r"(?u)\b\w+\b")
X = vectorizer.fit_transform(df['sentences'].values)
result = pd.DataFrame(data=X.toarray(), columns=vectorizer.get_feature_names())
result.head()
```

- **Proposal I**: Tabular data by extracting key words. "Document-Term Matrix"
  - useful in sentiment analysis, document clustering, topic modelling
  - popular algorithms include tf-idf,Word2Vec,bag of words, etc.
- **Proposal II**: Time-series of individual words.
  - useful in machine translation

Recurrent neural network model for machine translations




### Networks

- Concepts: node/edge/weight, directed/undirected
- **Mathematical Representation**: adjacency matrix
- Question: what about the whole datasets of networks, and time-evolving networks?

# Tasks with Data: Machine Learning

The tasks with data can often be transfromed into *machine learning* problems, which can be generally classified as:

- Supervised Learning -- "learning with training";
- Unsupervised Learning -- "learning without training";
- Reinforment Learning -- "learning by doing".

Our course will focus on the first two categories.

# Supervised Learning

- Given the *training dataset* $(x^{(i)}, y^{(i)})$ with $y^{(i)} \in \mathbb{R}^q$ denotes the *labels*, the supervised learning aims to find a mapping $\mathbf{f} : \mathbb{R}^p \to \mathbb{R}^q$ such that $y^{(i)} \approx \mathbf{f}(x^{(i)})$. Then with a new observation $x^{(new)}$, we can predict that $y^{(new)} = \mathbf{f}(x^{(new)})$.
    - when $y \in \mathbb{R}$ is continuous, the problem is also called as *regression*. **Example**: Housing price prediction
    - when $y \in \mathbb{R}$ is discrete, the problem is also called as *classification*. **Example**: Handwritten digit recognization
- **Practical Strategy**: Limit the mapping $\mathbf{f}$ to certain space by parametrization $\mathbf{f(x;\theta)}$. Then define the loss function of $\theta$
$$L(\theta) = \sum_{i=1}^{n} \ell(y^{(i)}, \mathbf{f}(x^{(i)})),$$
where $\ell$ quantifies the "distance" between $y^{(i)}$ and $\mathbf{f}(x^{(i)})$, and a common choice is mean squre error (MSE) for continous data $\ell(y^{(i)}, \mathbf{f}(x^{(i)})) = ||y^{(i)} - \mathbf{f}(x^{(i)})||^2$. We then seek to choose the optimal $\theta$ that minimizes the loss function
$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta),$$

which can be tacked numeracally by optimzation methods (including the popular stochastic gradient descent).

- Difference choice of $\mathbf{f}(\mathbf{x};\theta)$ leads to various supervised learning models:
  - Linear function : Linear Regression (for regression)/Logistic Regression (for classification)
  - Composition of linear + nonlinear functions: Neural Network

- **Important Terms**:
  - **Training Data**: Both X and y are provided. The dataset which we use to fit the function.
  - **Test Data**: In principle, only X is provided (some times $y^{test}$ is also provided as the ground-truth to verify). The dataset which we generate new predictions $y^{pred}$. -- This is the final judgement of your unsupervised ML model!
  - **Validation Data**: A good-fit model on training data does not guarantee the good performance on test data. To gain more confidence before really applying to test data, we "fake" some test data as the "sample exam". To do this, we further split the original training data into new traning data and validation data, and then learn the mapping on new training data, and judge on the validation data. We may make some adjustment if the model does not perform well in the "sample exam".
  - Intuitive Understanding: Training data is like quizzes -- you want to learn the "mapping" between the question and correct answer. Test data is like your exam. Validation is like you take a sample exam before the real exam and make some "clinics" about your weakpoints.
  - See the illustration here

**Example:** The Wisconsin breast cancer dataset) and low-code ML package pycaret.

In [ ]:
```
pip install --upgrade pycaret #install pycaret -- it's a new package, not coming with A
```

In [1]:
```
from sklearn.datasets import load_breast_cancer # load the dataset
X,y = load_breast_cancer(as_frame = True,return_X_y = True)
```

In [3]:
```
X
```

Out[3]:

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dim |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 17.99 | 10.38 | 122.80 | 1001.0 | 0.11840 | 0.27760 | 0.30010 | 0.14710 | 0.2419 | 0 |
| **1** | 20.57 | 17.77 | 132.90 | 1326.0 | 0.08474 | 0.07864 | 0.08690 | 0.07017 | 0.1812 | 0 |
| **2** | 19.69 | 21.25 | 130.00 | 1203.0 | 0.10960 | 0.15990 | 0.19740 | 0.12790 | 0.2069 | 0 |
| **3** | 11.42 | 20.38 | 77.58 | 386.1 | 0.14250 | 0.28390 | 0.24140 | 0.10520 | 0.2597 | 0 |
| **4** | 20.29 | 14.34 | 135.10 | 1297.0 | 0.10030 | 0.13280 | 0.19800 | 0.10430 | 0.1809 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **564** | 21.56 | 22.39 | 142.00 | 1479.0 | 0.11100 | 0.11590 | 0.24390 | 0.13890 | 0.1726 | 0 |
| **565** | 20.13 | 28.25 | 131.20 | 1261.0 | 0.09780 | 0.10340 | 0.14400 | 0.09791 | 0.1752 | 0 |

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dim |
|---|---|---|---|---|---|---|---|---|---|---|
| **566** | 16.60 | 28.08 | 108.30 | 858.1 | 0.08455 | 0.10230 | 0.09251 | 0.05302 | 0.1590 | 0 |
| **567** | 20.60 | 29.33 | 140.10 | 1265.0 | 0.11780 | 0.27700 | 0.35140 | 0.15200 | 0.2397 | 0 |
| **568** | 7.76 | 24.54 | 47.92 | 181.0 | 0.05263 | 0.04362 | 0.00000 | 0.00000 | 0.1587 | 0 |

569 rows × 30 columns

In [ ]:
```
y
```

In this dataset, all labels are known. To mimic a real situation, we manully create train and test datasets.

In [4]:
```
from sklearn.model_selection import train_test_split # manually split into train and te
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=
```

In [5]:
```
X_train.shape
```

Out[5]:
```
(381, 30)
```

In [6]:
```
y_test.shape
```

Out[6]:
```
(188,)
```

In [7]:
```
import pandas as pd
data_train = pd.concat([X_train,y_train],axis=1) # the whole data table of training
data_train
```

Out[7]:

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dim |
|---|---|---|---|---|---|---|---|---|---|---|
| **56** | 19.210 | 18.57 | 125.50 | 1152.0 | 0.10530 | 0.12670 | 0.13230 | 0.089940 | 0.1917 | ( |
| **144** | 10.750 | 14.97 | 68.26 | 355.3 | 0.07793 | 0.05139 | 0.02251 | 0.007875 | 0.1399 | ( |
| **60** | 10.170 | 14.88 | 64.55 | 311.9 | 0.11340 | 0.08061 | 0.01084 | 0.012900 | 0.2743 | ( |
| **6** | 18.250 | 19.98 | 119.60 | 1040.0 | 0.09463 | 0.10900 | 0.11270 | 0.074000 | 0.1794 | ( |
| **8** | 13.000 | 21.82 | 87.50 | 519.8 | 0.12730 | 0.19320 | 0.18590 | 0.093530 | 0.2350 | ( |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **277** | 18.810 | 19.98 | 120.90 | 1102.0 | 0.08923 | 0.05884 | 0.08020 | 0.058430 | 0.1550 | ( |
| **9** | 12.460 | 24.04 | 83.97 | 475.9 | 0.11860 | 0.23960 | 0.22730 | 0.085430 | 0.2030 | ( |
| **359** | 9.436 | 18.32 | 59.82 | 278.6 | 0.10090 | 0.05956 | 0.02710 | 0.014060 | 0.1506 | ( |

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dim |
|---|---|---|---|---|---|---|---|---|---|---|
| **192** | 9.720 | 18.22 | 60.73 | 288.1 | 0.06950 | 0.02344 | 0.00000 | 0.000000 | 0.1653 | ( |
| **559** | 11.510 | 23.93 | 74.52 | 403.5 | 0.09261 | 0.10210 | 0.11120 | 0.041050 | 0.1388 | ( |

381 rows × 31 columns

In [25]:
```
from pycaret.classification import setup
from pycaret.classification import compare_models

bc = setup(data=data_train, target='target') # target is the y column name we want to p
```

| | Description | Value |
|---|---|---|
| **0** | session_id | 7232 |
| **1** | Target | target |
| **2** | Target Type | Binary |
| **3** | Label Encoded | 0: 0, 1: 1 |
| **4** | Original Data | (381, 31) |
| **5** | Missing Values | False |
| **6** | Numeric Features | 30 |
| **7** | Categorical Features | 0 |
| **8** | Ordinal Features | False |
| **9** | High Cardinality Features | False |
| **10** | High Cardinality Method | None |
| **11** | Transformed Train Set | (266, 29) |
| **12** | Transformed Test Set | (115, 29) |
| **13** | Shuffle Train-Test | True |
| **14** | Stratify Train-Test | False |
| **15** | Fold Generator | StratifiedKFold |
| **16** | Fold Number | 10 |
| **17** | CPU Jobs | -1 |
| **18** | Use GPU | False |
| **19** | Log Experiment | False |
| **20** | Experiment Name | clf-default-name |
| **21** | USI | 3312 |
| **22** | Imputation Type | simple |

| | Description | Value |
|---|---|---|
| **23** | Iterative Imputation Iteration | None |
| **24** | Numeric Imputer | mean |
| **25** | Iterative Imputation Numeric Model | None |
| **26** | Categorical Imputer | constant |
| **27** | Iterative Imputation Categorical Model | None |
| **28** | Unknown Categoricals Handling | least_frequent |
| **29** | Normalize | False |
| **30** | Normalize Method | None |
| **31** | Transformation | False |
| **32** | Transformation Method | None |
| **33** | PCA | False |
| **34** | PCA Method | None |
| **35** | PCA Components | None |
| **36** | Ignore Low Variance | False |
| **37** | Combine Rare Levels | False |
| **38** | Rare Level Threshold | None |
| **39** | Numeric Binning | False |
| **40** | Remove Outliers | False |
| **41** | Outliers Threshold | None |
| **42** | Remove Multicollinearity | False |
| **43** | Multicollinearity Threshold | None |
| **44** | Clustering | False |
| **45** | Clustering Iteration | None |
| **46** | Polynomial Features | False |
| **47** | Polynomial Degree | None |
| **48** | Trignometry Features | False |
| **49** | Polynomial Threshold | None |
| **50** | Group Features | False |
| **51** | Feature Selection | False |
| **52** | Features Selection Threshold | None |
| **53** | Feature Interaction | False |
| **54** | Feature Ratio | False |
| **55** | Interaction Threshold | None |

| | Description | Value |
|---|---|---|
| **56** | Fix Imbalance | False |
| **57** | Fix Imbalance Method | SMOTE |

In [26]:
```
best = compare_models() # pycaret automatically fit different ML models for you, and co
```

| | Model | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC | TT (Sec) |
|---|---|---|---|---|---|---|---|---|---|
| **et** | Extra Trees Classifier | 0.9655 | 0.9887 | 0.9816 | 0.9678 | 0.9733 | 0.9238 | 0.9291 | 0.2650 |
| **lda** | Linear Discriminant Analysis | 0.9581 | 0.9892 | 0.9938 | 0.9482 | 0.9691 | 0.9038 | 0.9121 | 0.0300 |
| **catboost** | CatBoost Classifier | 0.9581 | 0.9889 | 0.9640 | 0.9722 | 0.9667 | 0.9095 | 0.9137 | 5.3810 |
| **ridge** | Ridge Classifier | 0.9546 | 0.0000 | 0.9938 | 0.9411 | 0.9658 | 0.8979 | 0.9047 | 0.0190 |
| **qda** | Quadratic Discriminant Analysis | 0.9546 | 0.9905 | 0.9585 | 0.9702 | 0.9640 | 0.9024 | 0.9035 | 0.0220 |
| **lightgbm** | Light Gradient Boosting Machine | 0.9543 | 0.9904 | 0.9699 | 0.9617 | 0.9642 | 0.9000 | 0.9052 | 0.4910 |
| **lr** | Logistic Regression | 0.9511 | 0.9878 | 0.9647 | 0.9601 | 0.9615 | 0.8945 | 0.8977 | 1.1400 |
| **rf** | Random Forest Classifier | 0.9507 | 0.9920 | 0.9640 | 0.9611 | 0.9610 | 0.8933 | 0.8981 | 0.2820 |
| **ada** | Ada Boost Classifier | 0.9472 | 0.9868 | 0.9522 | 0.9663 | 0.9566 | 0.8884 | 0.8951 | 0.1080 |
| **xgboost** | Extreme Gradient Boosting | 0.9432 | 0.9878 | 0.9522 | 0.9605 | 0.9546 | 0.8780 | 0.8832 | 0.2610 |
| **nb** | Naive Bayes | 0.9393 | 0.9890 | 0.9643 | 0.9451 | 0.9534 | 0.8658 | 0.8714 | 0.0230 |
| **gbc** | Gradient Boosting Classifier | 0.9356 | 0.9849 | 0.9522 | 0.9517 | 0.9491 | 0.8597 | 0.8688 | 0.1700 |
| **dt** | Decision Tree Classifier | 0.9093 | 0.9031 | 0.9228 | 0.9396 | 0.9277 | 0.8040 | 0.8143 | 0.0260 |
| **knn** | K Neighbors Classifier | 0.8984 | 0.9629 | 0.9404 | 0.9074 | 0.9215 | 0.7768 | 0.7862 | 0.0550 |
| **svm** | SVM - Linear Kernel | 0.8313 | 0.0000 | 0.8632 | 0.8880 | 0.8567 | 0.6414 | 0.6724 | 0.0250 |

In [27]:
```
best # the best model selected by pycaret
```

Out[27]:
```
ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                     criterion='gini', max_depth=None, max_features='auto',
                     max_leaf_nodes=None, max_samples=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=-1,
                     oob_score=False, random_state=7232, verbose=0,
                     warm_start=False)
```

In [28]:
```
from pycaret.classification import predict_model
predict_model(best); # predict on the validation data that pycaret have selected -- sam
```

| | Model | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|---|

| | Model | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|---|
| **0** | Extra Trees Classifier | 0.9565 | 0.9855 | 0.9706 | 0.9565 | 0.9635 | 0.9097 | 0.9099 |

In [29]:
```
from pycaret.classification import finalize_model
best_final = finalize_model(best) # re-train the dataset with whole input training data
```

In [30]:
```
from pycaret.classification import predict_model
predictions = predict_model(best_final, data = X_test) # make new predictions on new-co
predictions
```

Out[30]:

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dime |
|---|---|---|---|---|---|---|---|---|---|---|
| **512** | 13.40 | 20.52 | 88.64 | 556.7 | 0.11060 | 0.14690 | 0.14450 | 0.08172 | 0.2116 | 0. |
| **457** | 13.21 | 25.25 | 84.10 | 537.9 | 0.08791 | 0.05205 | 0.02772 | 0.02068 | 0.1619 | 0. |
| **439** | 14.02 | 15.66 | 89.59 | 606.5 | 0.07966 | 0.05581 | 0.02087 | 0.02652 | 0.1589 | 0. |
| **298** | 14.26 | 18.17 | 91.22 | 633.1 | 0.06576 | 0.05220 | 0.02475 | 0.01374 | 0.1635 | 0. |
| **37** | 13.03 | 18.42 | 82.61 | 523.8 | 0.08983 | 0.03766 | 0.02562 | 0.02923 | 0.1467 | 0. |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **100** | 13.61 | 24.98 | 88.05 | 582.7 | 0.09488 | 0.08511 | 0.08625 | 0.04489 | 0.1609 | 0. |
| **336** | 12.99 | 14.23 | 84.08 | 514.3 | 0.09462 | 0.09965 | 0.03738 | 0.02098 | 0.1652 | 0. |
| **299** | 10.51 | 23.09 | 66.85 | 334.2 | 0.10150 | 0.06797 | 0.02495 | 0.01875 | 0.1695 | 0. |
| **347** | 14.76 | 14.74 | 94.87 | 668.7 | 0.08875 | 0.07780 | 0.04608 | 0.03528 | 0.1521 | 0. |
| **502** | 12.54 | 16.32 | 81.25 | 476.3 | 0.11580 | 0.10850 | 0.05928 | 0.03279 | 0.1943 | 0. |

188 rows × 32 columns

In [31]:
```
df_compare = pd.concat([predictions['Label'],y_test],axis = 1) # compare with the groun
df_compare
```

Out[31]:

| | Label | target |
|---|---|---|
| **512** | 0 | 0 |
| **457** | 1 | 1 |
| **439** | 1 | 1 |
| **298** | 1 | 1 |
| **37** | 1 | 1 |
| **...** | ... | ... |
| **100** | 0 | 0 |

| | Label | target |
|---|---|---|
| **336** | 1 | 1 |
| **299** | 1 | 1 |
| **347** | 1 | 1 |
| **502** | 1 | 1 |

188 rows × 2 columns

In [32]:
```
import numpy as np
np.mean(predictions['Label'].to_numpy() == y_test.to_numpy()) # calculate the percentag
```

Out[32]: 0.9627659574468085

In [33]:
```
from pycaret.classification import create_model
lr = create_model('lr') # what if we only want the logistic regression model?
```

| | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|
| **0** | 0.9630 | 0.9941 | 1.0000 | 0.9444 | 0.9714 | 0.9189 | 0.9220 |
| **1** | 0.9630 | 1.0000 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **2** | 0.9630 | 0.9941 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **3** | 0.8889 | 0.9765 | 0.8824 | 0.9375 | 0.9091 | 0.7666 | 0.7689 |
| **4** | 0.9630 | 0.9824 | 1.0000 | 0.9444 | 0.9714 | 0.9189 | 0.9220 |
| **5** | 0.9630 | 0.9941 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **6** | 0.9231 | 0.9804 | 0.9412 | 0.9412 | 0.9412 | 0.8301 | 0.8301 |
| **7** | 0.9615 | 0.9935 | 1.0000 | 0.9444 | 0.9714 | 0.9128 | 0.9162 |
| **8** | 0.9231 | 0.9625 | 1.0000 | 0.8889 | 0.9412 | 0.8312 | 0.8433 |
| **9** | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| **Mean** | 0.9511 | 0.9878 | 0.9647 | 0.9601 | 0.9615 | 0.8945 | 0.8977 |
| **SD** | 0.0294 | 0.0114 | 0.0390 | 0.0361 | 0.0236 | 0.0628 | 0.0619 |

In [34]:
```
predict_model(lr) # validation dataset -- sample exam!
```

| | Model | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|---|
| **0** | Logistic Regression | 0.9391 | 0.9862 | 0.9706 | 0.9296 | 0.9496 | 0.8728 | 0.8741 |

Out[34]:

| | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dir |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 16.850000 | 94.209999 | 666.000000 | 0.08641 | 0.06698 | 0.051920 | 0.027910 | 0.1409 | |
| **1** | 15.210000 | 78.010002 | 457.899994 | 0.08673 | 0.06545 | 0.019940 | 0.016920 | 0.1638 | |

| | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dir |
|---|---|---|---|---|---|---|---|---|---|
| **2** | 18.049999 | 105.099998 | 813.000000 | 0.09721 | 0.11370 | 0.094470 | 0.059430 | 0.1861 | |
| **3** | 19.620001 | 91.120003 | 599.500000 | 0.10600 | 0.11330 | 0.112600 | 0.064630 | 0.1669 | |
| **4** | 12.350000 | 69.139999 | 363.700012 | 0.08518 | 0.04721 | 0.012360 | 0.013690 | 0.1449 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **110** | 18.889999 | 72.169998 | 396.000000 | 0.08713 | 0.05008 | 0.023990 | 0.021730 | 0.2013 | |
| **111** | 20.250000 | 102.599998 | 761.299988 | 0.10250 | 0.12040 | 0.114700 | 0.064620 | 0.1935 | |
| **112** | 13.840000 | 82.709999 | 530.599976 | 0.08352 | 0.03735 | 0.004559 | 0.008829 | 0.1453 | |
| **113** | 19.660000 | 131.100006 | 1274.000000 | 0.08020 | 0.08564 | 0.115500 | 0.077260 | 0.1928 | |
| **114** | 16.850000 | 79.190002 | 481.600006 | 0.08511 | 0.03834 | 0.004473 | 0.006423 | 0.1215 | |

115 rows × 32 columns

In [35]:
```
final_lr = finalize_model(lr)
```

In [36]:
```
predictions_lr = predict_model(final_lr, data = X_test)
np.mean(predictions_lr['Label'].to_numpy() == y_test.to_numpy())
```

Out[36]: 0.9627659574468085

In [20]:
```
from pycaret.classification import tune_model
tuned_lr = tune_model(lr) # fine-tuning the parameters in logistic regression
```

| | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|
| **0** | 0.8889 | 0.9882 | 0.8235 | 1.0000 | 0.9032 | 0.7756 | 0.7959 |
| **1** | 0.9630 | 0.9824 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **2** | 0.9630 | 1.0000 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **3** | 0.9630 | 1.0000 | 0.9412 | 1.0000 | 0.9697 | 0.9222 | 0.9250 |
| **4** | 0.9259 | 0.9882 | 1.0000 | 0.8947 | 0.9444 | 0.8344 | 0.8460 |
| **5** | 0.9259 | 0.9432 | 1.0000 | 0.8889 | 0.9412 | 0.8421 | 0.8528 |
| **6** | 0.9615 | 1.0000 | 1.0000 | 0.9412 | 0.9697 | 0.9172 | 0.9204 |
| **7** | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| **8** | 0.9615 | 1.0000 | 1.0000 | 0.9412 | 0.9697 | 0.9172 | 0.9204 |
| **9** | 0.8846 | 0.9750 | 0.8750 | 0.9333 | 0.9032 | 0.7607 | 0.7632 |
| **Mean** | 0.9437 | 0.9877 | 0.9522 | 0.9599 | 0.9541 | 0.8814 | 0.8874 |
| **SD** | 0.0347 | 0.0171 | 0.0586 | 0.0433 | 0.0296 | 0.0715 | 0.0677 |

In [21]:
```
predict_model(tuned_lr) # still doing the sample exam -- validation dataset
```

| | Model | Accuracy | AUC | Recall | Prec. | F1 | Kappa | MCC |
|---|---|---|---|---|---|---|---|---|
| **0** | Logistic Regression | 0.9739 | 0.9968 | 0.9718 | 0.9857 | 0.9787 | 0.9450 | 0.9452 |

Out[21]:

| | mean texture | mean perimeter | mean area | mean smoothness | mean compactness | mean concavity | mean concave points | mean symmetry | dir |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 33.810001 | 70.790001 | 386.799988 | 0.07780 | 0.03574 | 0.004967 | 0.006434 | 0.1845 | |
| **1** | 15.690000 | 92.680000 | 664.900024 | 0.07618 | 0.03515 | 0.014470 | 0.018770 | 0.1632 | |
| **2** | 19.040001 | 71.800003 | 394.100006 | 0.08139 | 0.04701 | 0.037090 | 0.022300 | 0.1516 | |
| **3** | 12.220000 | 65.750000 | 321.600006 | 0.09996 | 0.07542 | 0.019230 | 0.019680 | 0.1800 | |
| **4** | 16.850000 | 79.190002 | 481.600006 | 0.08511 | 0.03834 | 0.004473 | 0.006423 | 0.1215 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **110** | 17.120001 | 121.400002 | 1077.000000 | 0.10540 | 0.11000 | 0.145700 | 0.086650 | 0.1966 | |
| **111** | 16.950001 | 96.220001 | 685.900024 | 0.09855 | 0.07885 | 0.026020 | 0.037810 | 0.1780 | |
| **112** | 18.870001 | 118.699997 | 1027.000000 | 0.09746 | 0.11170 | 0.113000 | 0.079500 | 0.1807 | |
| **113** | 12.840000 | 74.339996 | 412.600006 | 0.08983 | 0.07525 | 0.041960 | 0.033500 | 0.1620 | |
| **114** | 18.600000 | 81.089996 | 481.899994 | 0.09965 | 0.10580 | 0.080050 | 0.038210 | 0.1925 | |

115 rows × 32 columns

In [22]:
```
final_tuned_lr = finalize_model(tuned_lr) #retrain with the whole dataset
```

In [23]:
```
predictions_tuned_lr = predict_model(final_tuned_lr, data = X_test)
np.mean(predictions_tuned_lr['Label'].to_numpy() == y_test.to_numpy())
```

Out[23]:
```
0.9521276595744681
```

Let's recap the workflow above (or about general supervised learning)

- The **minimum requirement** is that we have a training dataset with both $X$ and $y$ (also called labels, targets...). We want to **fit the mapping** between $x$ and $y$ with **training dataset** (the process is indeed called training), and making predictions about the new $y$ given new $X$ in the test dataset.
  - *Remark 1*: The true y in test dataset sometimes can also be known, so that we can know the performance the model immediately. But in general, we won't expect this.
  - *Remark 2*: In our course, just to mimic a real-world situation, sometimes we manually create (split) the train or test data.

- (Optional) We may train multiple models or one model with multiple parameters. How can we compare them and gain more confidence about the final test? Sometimes we further split the training dataset into (real) training dataset and **validation dataset** (imagine it as the sample exam), so that we can get instant feedback because we know the true label in validation dataset.

- (Optional) During training, to be more cautious, sometimes we even make more "quizzes" -- that is called **cross-validation** (will talk about the details in the next lecture)

- (Optional) With 10 "quizzes" (10-fold cross-validation) and "one sample exam" (validation data), for instance, we finally pick up the best candidate model. Before applying to the real test dataset, we don't want to waste any sample. Therefore we **finalize** training by picking up the winner model, while updating it with all the samples (including the validation data) in the training dataset.

- Finally, applying the model to test data -- wait and see!

Of course, as a math course, we are not satisfied with merely calling functions in pycaret. In the rest of lectures this quarter, we are going to dig into details of some algorihms and learn more underlying math -- turn the black box of ML into white (at least gray) one!

# Unsupervised Learning

It is still challenging to give a general and rigorous definition for unsupervised learning mathematically. Let's focus on more specific tasks.

- Dimension Reducion

  Given $X \in \mathbb{R}^{n\times p}$, finding a mapping function $\mathbf{f}: \mathbb{R}^p \to \mathbb{R}^q (q \ll p)$ such that the low-dimensional coordinates $z^{(i)} = \mathbf{f}(x^{(i)})$ "preserve the information" about $x^{(i)}$.

  - Question: Difference with supervised learning?
  - Linear mapping: Principle Component Analysis (PCA)
  - Nonlinear mapping: Manifold Learning, Autoencoder

In [37]:
```
from sklearn.datasets import load_iris
X,y = load_iris(return_X_y = True) # Note that in the hw this week, it's not allowed to
X
```

Out[37]:
```
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],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
```

```
[4.3, 3. , 1.1, 0.1],
[5.8, 4. , 1.2, 0.2],
[5.7, 4.4, 1.5, 0.4],
[5.4, 3.9, 1.3, 0.4],
[5.1, 3.5, 1.4, 0.3],
[5.7, 3.8, 1.7, 0.3],
[5.1, 3.8, 1.5, 0.3],
[5.4, 3.4, 1.7, 0.2],
[5.1, 3.7, 1.5, 0.4],
[4.6, 3.6, 1. , 0.2],
[5.1, 3.3, 1.7, 0.5],
[4.8, 3.4, 1.9, 0.2],
[5. , 3. , 1.6, 0.2],
[5. , 3.4, 1.6, 0.4],
[5.2, 3.5, 1.5, 0.2],
[5.2, 3.4, 1.4, 0.2],
[4.7, 3.2, 1.6, 0.2],
[4.8, 3.1, 1.6, 0.2],
[5.4, 3.4, 1.5, 0.4],
[5.2, 4.1, 1.5, 0.1],
[5.5, 4.2, 1.4, 0.2],
[4.9, 3.1, 1.5, 0.2],
[5. , 3.2, 1.2, 0.2],
[5.5, 3.5, 1.3, 0.2],
[4.9, 3.6, 1.4, 0.1],
[4.4, 3. , 1.3, 0.2],
[5.1, 3.4, 1.5, 0.2],
[5. , 3.5, 1.3, 0.3],
[4.5, 2.3, 1.3, 0.3],
[4.4, 3.2, 1.3, 0.2],
[5. , 3.5, 1.6, 0.6],
[5.1, 3.8, 1.9, 0.4],
[4.8, 3. , 1.4, 0.3],
[5.1, 3.8, 1.6, 0.2],
[4.6, 3.2, 1.4, 0.2],
[5.3, 3.7, 1.5, 0.2],
[5. , 3.3, 1.4, 0.2],
[7. , 3.2, 4.7, 1.4],
[6.4, 3.2, 4.5, 1.5],
[6.9, 3.1, 4.9, 1.5],
[5.5, 2.3, 4. , 1.3],
[6.5, 2.8, 4.6, 1.5],
[5.7, 2.8, 4.5, 1.3],
[6.3, 3.3, 4.7, 1.6],
[4.9, 2.4, 3.3, 1. ],
[6.6, 2.9, 4.6, 1.3],
[5.2, 2.7, 3.9, 1.4],
[5. , 2. , 3.5, 1. ],
[5.9, 3. , 4.2, 1.5],
[6. , 2.2, 4. , 1. ],
[6.1, 2.9, 4.7, 1.4],
[5.6, 2.9, 3.6, 1.3],
[6.7, 3.1, 4.4, 1.4],
[5.6, 3. , 4.5, 1.5],
[5.8, 2.7, 4.1, 1. ],
[6.2, 2.2, 4.5, 1.5],
[5.6, 2.5, 3.9, 1.1],
[5.9, 3.2, 4.8, 1.8],
[6.1, 2.8, 4. , 1.3],
[6.3, 2.5, 4.9, 1.5],
[6.1, 2.8, 4.7, 1.2],
[6.4, 2.9, 4.3, 1.3],
[6.6, 3. , 4.4, 1.4],
[6.8, 2.8, 4.8, 1.4],
[6.7, 3. , 5. , 1.7],
```

```
[6. , 2.9, 4.5, 1.5],
[5.7, 2.6, 3.5, 1. ],
[5.5, 2.4, 3.8, 1.1],
[5.5, 2.4, 3.7, 1. ],
[5.8, 2.7, 3.9, 1.2],
[6. , 2.7, 5.1, 1.6],
[5.4, 3. , 4.5, 1.5],
[6. , 3.4, 4.5, 1.6],
[6.7, 3.1, 4.7, 1.5],
[6.3, 2.3, 4.4, 1.3],
[5.6, 3. , 4.1, 1.3],
[5.5, 2.5, 4. , 1.3],
[5.5, 2.6, 4.4, 1.2],
[6.1, 3. , 4.6, 1.4],
[5.8, 2.6, 4. , 1.2],
[5. , 2.3, 3.3, 1. ],
[5.6, 2.7, 4.2, 1.3],
[5.7, 3. , 4.2, 1.2],
[5.7, 2.9, 4.2, 1.3],
[6.2, 2.9, 4.3, 1.3],
[5.1, 2.5, 3. , 1.1],
[5.7, 2.8, 4.1, 1.3],
[6.3, 3.3, 6. , 2.5],
[5.8, 2.7, 5.1, 1.9],
[7.1, 3. , 5.9, 2.1],
[6.3, 2.9, 5.6, 1.8],
[6.5, 3. , 5.8, 2.2],
[7.6, 3. , 6.6, 2.1],
[4.9, 2.5, 4.5, 1.7],
[7.3, 2.9, 6.3, 1.8],
[6.7, 2.5, 5.8, 1.8],
[7.2, 3.6, 6.1, 2.5],
[6.5, 3.2, 5.1, 2. ],
[6.4, 2.7, 5.3, 1.9],
[6.8, 3. , 5.5, 2.1],
[5.7, 2.5, 5. , 2. ],
[5.8, 2.8, 5.1, 2.4],
[6.4, 3.2, 5.3, 2.3],
[6.5, 3. , 5.5, 1.8],
[7.7, 3.8, 6.7, 2.2],
[7.7, 2.6, 6.9, 2.3],
[6. , 2.2, 5. , 1.5],
[6.9, 3.2, 5.7, 2.3],
[5.6, 2.8, 4.9, 2. ],
[7.7, 2.8, 6.7, 2. ],
[6.3, 2.7, 4.9, 1.8],
[6.7, 3.3, 5.7, 2.1],
[7.2, 3.2, 6. , 1.8],
[6.2, 2.8, 4.8, 1.8],
[6.1, 3. , 4.9, 1.8],
[6.4, 2.8, 5.6, 2.1],
[7.2, 3. , 5.8, 1.6],
[7.4, 2.8, 6.1, 1.9],
[7.9, 3.8, 6.4, 2. ],
[6.4, 2.8, 5.6, 2.2],
[6.3, 2.8, 5.1, 1.5],
[6.1, 2.6, 5.6, 1.4],
[7.7, 3. , 6.1, 2.3],
[6.3, 3.4, 5.6, 2.4],
[6.4, 3.1, 5.5, 1.8],
[6. , 3. , 4.8, 1.8],
[6.9, 3.1, 5.4, 2.1],
[6.7, 3.1, 5.6, 2.4],
[6.9, 3.1, 5.1, 2.3],
[5.8, 2.7, 5.1, 1.9],
```

```
       [6.8, 3.2, 5.9, 2.3],
       [6.7, 3.3, 5.7, 2.5],
       [6.7, 3. , 5.2, 2.3],
       [6.3, 2.5, 5. , 1.9],
       [6.5, 3. , 5.2, 2. ],
       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]])
```

In [38]:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # principle component analysis, ruduce 4-dimenional data to 2
X_pca = pca.fit_transform(X)
X_pca
```

Out[38]:

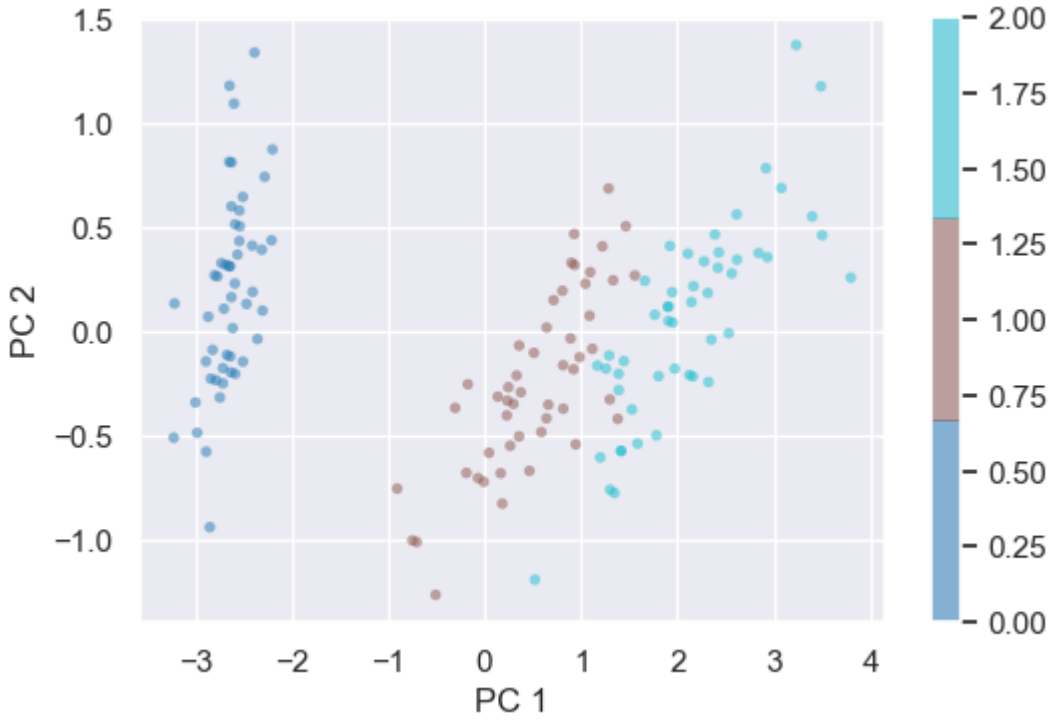
```
array([[-2.68412563,  0.31939725],
       [-2.71414169, -0.17700123],
       [-2.88899057, -0.14494943],
       [-2.74534286, -0.31829898],
       [-2.72871654,  0.32675451],
       [-2.28085963,  0.74133045],
       [-2.82053775, -0.08946138],
       [-2.62614497,  0.16338496],
       [-2.88638273, -0.57831175],
       [-2.6727558 , -0.11377425],
       [-2.50694709,  0.6450689 ],
       [-2.61275523,  0.01472994],
       [-2.78610927, -0.235112  ],
       [-3.22380374, -0.51139459],
       [-2.64475039,  1.17876464],
       [-2.38603903,  1.33806233],
       [-2.62352788,  0.81067951],
       [-2.64829671,  0.31184914],
       [-2.19982032,  0.87283904],
       [-2.5879864 ,  0.51356031],
       [-2.31025622,  0.39134594],
       [-2.54370523,  0.43299606],
       [-3.21593942,  0.13346807],
       [-2.30273318,  0.09870885],
       [-2.35575405, -0.03728186],
       [-2.50666891, -0.14601688],
       [-2.46882007,  0.13095149],
       [-2.56231991,  0.36771886],
       [-2.63953472,  0.31203998],
       [-2.63198939, -0.19696122],
       [-2.58739848, -0.20431849],
       [-2.4099325 ,  0.41092426],
       [-2.64886233,  0.81336382],
       [-2.59873675,  1.09314576],
       [-2.63692688, -0.12132235],
       [-2.86624165,  0.06936447],
       [-2.62523805,  0.59937002],
       [-2.80068412,  0.26864374],
       [-2.98050204, -0.48795834],
       [-2.59000631,  0.22904384],
       [-2.77010243,  0.26352753],
       [-2.84936871, -0.94096057],
       [-2.99740655, -0.34192606],
       [-2.40561449,  0.18887143],
       [-2.20948924,  0.43666314],
       [-2.71445143, -0.2502082 ],
       [-2.53814826,  0.50377114],
       [-2.83946217, -0.22794557],
       [-2.54308575,  0.57941002],
       [-2.70335978,  0.10770608],
```

```
[ 1.28482569,  0.68516047],
[ 0.93248853,  0.31833364],
[ 1.46430232,  0.50426282],
[ 0.18331772, -0.82795901],
[ 1.08810326,  0.07459068],
[ 0.64166908, -0.41824687],
[ 1.09506066,  0.28346827],
[-0.74912267, -1.00489096],
[ 1.04413183,  0.2283619 ],
[-0.0087454 , -0.72308191],
[-0.50784088, -1.26597119],
[ 0.51169856, -0.10398124],
[ 0.26497651, -0.55003646],
[ 0.98493451, -0.12481785],
[-0.17392537, -0.25485421],
[ 0.92786078,  0.46717949],
[ 0.66028376, -0.35296967],
[ 0.23610499, -0.33361077],
[ 0.94473373, -0.54314555],
[ 0.04522698, -0.58383438],
[ 1.11628318, -0.08461685],
[ 0.35788842, -0.06892503],
[ 1.29818388, -0.32778731],
[ 0.92172892, -0.18273779],
[ 0.71485333,  0.14905594],
[ 0.90017437,  0.32850447],
[ 1.33202444,  0.24444088],
[ 1.55780216,  0.26749545],
[ 0.81329065, -0.1633503 ],
[-0.30558378, -0.36826219],
[-0.06812649, -0.70517213],
[-0.18962247, -0.68028676],
[ 0.13642871, -0.31403244],
[ 1.38002644, -0.42095429],
[ 0.58800644, -0.48428742],
[ 0.80685831,  0.19418231],
[ 1.22069088,  0.40761959],
[ 0.81509524, -0.37203706],
[ 0.24595768, -0.2685244 ],
[ 0.16641322, -0.68192672],
[ 0.46480029, -0.67071154],
[ 0.8908152 , -0.03446444],
[ 0.23054802, -0.40438585],
[-0.70453176, -1.01224823],
[ 0.35698149, -0.50491009],
[ 0.33193448, -0.21265468],
[ 0.37621565, -0.29321893],
[ 0.64257601,  0.01773819],
[-0.90646986, -0.75609337],
[ 0.29900084, -0.34889781],
[ 2.53119273, -0.00984911],
[ 1.41523588, -0.57491635],
[ 2.61667602,  0.34390315],
[ 1.97153105, -0.1797279 ],
[ 2.35000592, -0.04026095],
[ 3.39703874,  0.55083667],
[ 0.52123224, -1.19275873],
[ 2.93258707,  0.3555    ],
[ 2.32122882, -0.2438315 ],
[ 2.91675097,  0.78279195],
[ 1.66177415,  0.24222841],
[ 1.80340195, -0.21563762],
[ 2.1655918 ,  0.21627559],
[ 1.34616358, -0.77681835],
[ 1.58592822, -0.53964071],
```

```
       [ 1.90445637,  0.11925069],
       [ 1.94968906,  0.04194326],
       [ 3.48705536,  1.17573933],
       [ 3.79564542,  0.25732297],
       [ 1.30079171, -0.76114964],
       [ 2.42781791,  0.37819601],
       [ 1.19900111, -0.60609153],
       [ 3.49992004,  0.4606741 ],
       [ 1.38876613, -0.20439933],
       [ 2.2754305 ,  0.33499061],
       [ 2.61409047,  0.56090136],
       [ 1.25850816, -0.17970479],
       [ 1.29113206, -0.11666865],
       [ 2.12360872, -0.20972948],
       [ 2.38800302,  0.4646398 ],
       [ 2.84167278,  0.37526917],
       [ 3.23067366,  1.37416509],
       [ 2.15943764, -0.21727758],
       [ 1.44416124, -0.14341341],
       [ 1.78129481, -0.49990168],
       [ 3.07649993,  0.68808568],
       [ 2.14424331,  0.1400642 ],
       [ 1.90509815,  0.04930053],
       [ 1.16932634, -0.16499026],
       [ 2.10761114,  0.37228787],
       [ 2.31415471,  0.18365128],
       [ 1.9222678 ,  0.40920347],
       [ 1.41523588, -0.57491635],
       [ 2.56301338,  0.2778626 ],
       [ 2.41874618,  0.3047982 ],
       [ 1.94410979,  0.1875323 ],
       [ 1.52716661, -0.37531698],
       [ 1.76434572,  0.07885885],
       [ 1.90094161,  0.11662796],
       [ 1.39018886, -0.28266094]])
```

In [39]:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set() # set the seaborn theme style
figure = plt.figure(dpi=100)
plt.scatter(X_pca[:, 0], X_pca[:, 1],c=y, s=15, edgecolor='none', alpha=0.5,cmap=plt.cm
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.colorbar();
```




- Clustering

  Given $X \in \mathbb{R}^{n \times p}$, finding a partition of the dataset into $K$ groups such that

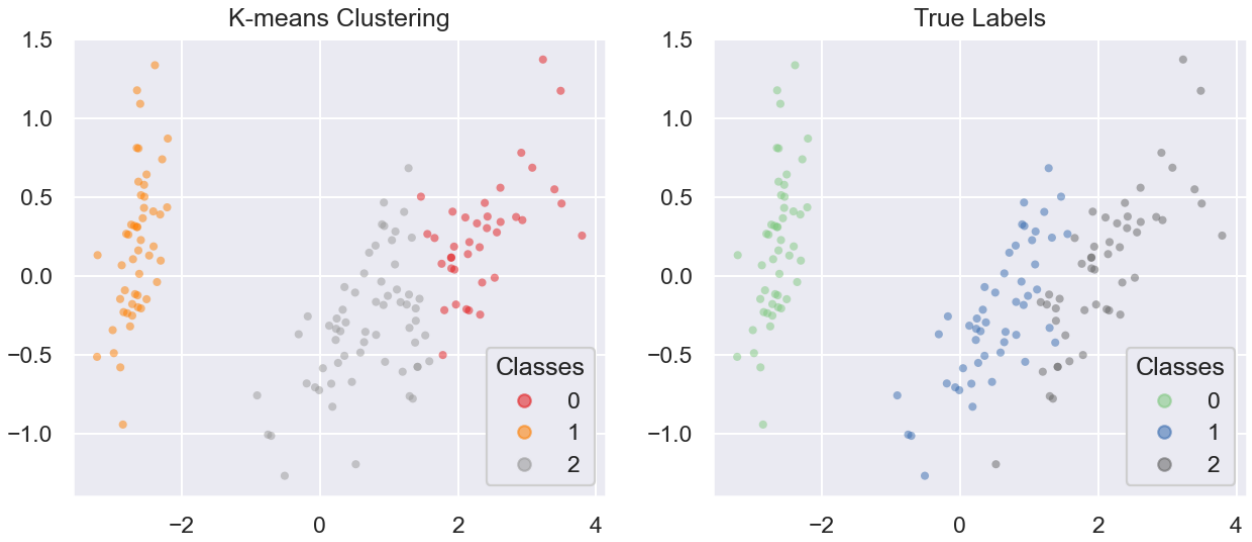
  - data within the same group are similiar;
  - data from different groups are dissimiliar.

In [40]:
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=0) #call k-means clustering algorithm
y_km = kmeans.fit_predict(X)
y_km # the groups assigned by algorithm
```

Out[40]:
```
array([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, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 2, 0, 0, 0, 0,
       0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2], dtype=int32)
```

In [41]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
fig, (ax1, ax2) = plt.subplots(1, 2,dpi=150, figsize=(10,4))

fig1 = ax1.scatter(X_pca[:, 0], X_pca[:, 1],c=y_km, s=15, edgecolor='none', alpha=0.5,c
fig2 = ax2.scatter(X_pca[:, 0], X_pca[:, 1],c=y, s=15, edgecolor='none', alpha=0.5,cmap
ax1.set_title('K-means Clustering')
legend1 = ax1.legend(*fig1.legend_elements(), loc="best", title="Classes")
ax1.add_artist(legend1)
ax2.set_title('True Labels')
legend2 = ax2.legend(*fig2.legend_elements(), loc="best", title="Classes")
ax2.add_artist(legend2)
```

Out[41]: <matplotlib.legend.Legend at 0x7fb5d4ae2810>




Question: What is the difference between clustering and classification? Can you try classification on Iris data with pycaret right now?

In [ ]:
```
# try classification with pycaret for Iris data by yourself!
```