Contents

[Tracks & Github Answer 2](#_Toc174955901)

[03/26/24 【**Supervised Learning with scikit-learn】**- Classification 2](#_Toc174955902)

[Machine Learning 2](#_Toc174955903)

[Supervised Learning 3](#_Toc174955904)

[KNN - k-Nearest Neighbors 3](#_Toc174955905)

[04/16/24 Regression 6](#_Toc174955906)

[R-squared 7](#_Toc174955907)

[Cross-validation 8](#_Toc174955908)

[Regularized regression 9](#_Toc174955909)

[Ridge 9](#_Toc174955910)

[Lasso 10](#_Toc174955911)

[04/21/24 Fine-Tuning Your Model 10](#_Toc174955912)

[Classification report和Confusion matrix 10](#_Toc174955913)

[ROC Curve 11](#_Toc174955914)

[ROC AUC 12](#_Toc174955915)

[Hyperparameter Tuning 12](#_Toc174955916)

[GridSearchCV 13](#_Toc174955917)

[RandomizedSearchCV 13](#_Toc174955918)

[04/23/24 Preprocessing and Pipelines 14](#_Toc174955919)

[Categorical features 14](#_Toc174955920)

[Handling missing data - isna(), dropna() 15](#_Toc174955921)

[Imputing within a pipeline – Pipeline + SimpleImputer 15](#_Toc174955922)

[Imputing within a scikit-learn – SimpleImputer 17](#_Toc174955923)

[Centering and scaling – Regression – StandardScaler 17](#_Toc174955924)

[Centering and scaling – Classification 18](#_Toc174955925)

[Evaluating models 18](#_Toc174955926)

[04/28/24 【**Predictive Modeling for Agriculture】**Project 21](#_Toc174955927)

[04/25/24 【**Unsupervised Learning in Python】-** Clustering for Dataset Exploration 22](#_Toc174955928)

[K-means clustering – Kmeans() 22](#_Toc174955929)

[Measure clustering quality 24](#_Toc174955930)

[Transforming features for better clusterings – StandardScaler() 25](#_Toc174955931)

[Normalizer () 27](#_Toc174955932)

[04/28/24 Visualization with Hierarchical Clustering and t-SNE 27](#_Toc174955933)

[Hierarchical clustering - SciPy linkage() 28](#_Toc174955934)

[t-SNE = t-distributed stochastic neighbor embedding 30](#_Toc174955935)

[05/07/24 Decorrelating Your Data and Dimension Reduction 31](#_Toc174955936)

[PCA – Principal Component Analysis 31](#_Toc174955937)

[tf-idf word-frequency array – TfidfVectorizer() 34](#_Toc174955938)

[05/16/24 Discovering Interpretable Features 36](#_Toc174955939)

[Non-negative matrix factorization" ("NMF") 36](#_Toc174955940)

[NMF learns interpretable parts 39](#_Toc174955941)

[Building recommender systems using NMF 40](#_Toc174955942)

[06/04/24 【**Clustering Antarctic Penguin Species】**Project 42](#_Toc174955943)

[06/06/24 【**Linear Classification in Python】-** Applying logistic regression and SVM 43](#_Toc174955944)

[06//24 Loss functions 43](#_Toc174955945)

[06//24 Logistic regression 43](#_Toc174955946)

[06//24 Support Vector Machines 43](#_Toc174955947)

[06/18/24 【**Introduction to Deep Learning with PyTorch】-** Introduction to PyTorch, a Deep Learning Library 43](#_Toc174955948)

# Tracks & Github Answer

<https://app.datacamp.com/learn/skill-tracks/machine-learning-fundamentals-with-python>

<https://github.com/Ashleshk/DataScience-DataCamp-Exercises/blob/63014e891a69328aafe44f08ddb1d8ffc5236cc2/Data%20Scientist%20Python/21.%20Supervised%20Learning%20with%20scikit-learn/Notes.md?plain=1#L11>

<https://github.com/Ashleshk/DataScience-DataCamp-Exercises/blob/63014e891a69328aafe44f08ddb1d8ffc5236cc2/Data%20Scientist%20Python/22.%20UnSupervised%20Learning%20in%20Python/Notes.md?plain=1#L41>

<https://github.com/insafhamdi/Predictive-Modeling-for-Agriculture/blob/main/notebook.ipynb>

# 03/26/24 【**Supervised Learning with scikit-learn】**- Classification

## Machine Learning

* Machine learning is the process whereby computers learn to make decisions from data without being explicitly programmed.
* Unsupervised learning is the process of uncovering hidden patterns and structures from unlabeled data. For example, a business may wish to group its customers into distinct categories based on their purchasing behavior without knowing in advance what these categories are. This is known as clustering, one branch of unsupervised learning.
* Supervised learning is a type of machine learning where the values to be predicted are already known, and a model is built with the aim of accurately predicting values of previously unseen data. Supervised learning uses features to predict the value of a target variable, such as predicting a basketball player's position based on their points per game. This course will exclusively focus on supervised learning.

## Supervised Learning

* There are two types of supervised learning.
* Classification is used to predict the label, or category, of an observation. For example, we can predict whether a bank transaction is fraudulent or not. As there are two outcomes here - a fraudulent transaction, or non-fraudulent transaction, this is known as binary classification.
* Regression is used to predict continuous values. For example, a model can use features such as number of bedrooms, and the size of a property, to predict the target variable, price of the property.
* There are some requirements to satisfy before performing supervised learning.
  + Our data must not have missing values,
  + must be in numeric format, and stored as pandas DataFrames or Series, or NumPy arrays.
  + This requires some exploratory data analysis (EDA) first to ensure data is in the correct format.

A white paper with blue text

Description automatically generated

## KNN - k-Nearest Neighbors

* k-Nearest Neighbors, which is popular for classification problems.
* The idea of k-Nearest Neighbors, or KNN, is to predict the label of any data point by looking at the k, for example, three, closest labeled data points and getting them to vote on what label the unlabeled observation should have.
* KNN uses majority voting, which makes predictions based on what label the majority of nearest neighbors have.
* 例子，注意使用n\_neighbors定义用周边多少个邻居来分类。预测的时候使用knn.predict

# Import KNeighborsClassifier

from sklearn.neighbors import KNeighborsClassifier

y = churn\_df["churn"].values

X = churn\_df[["account\_length", "customer\_service\_calls"]].values

# Create a KNN classifier with 6 neighbors

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the classifier to the data

knn.fit(X, y)

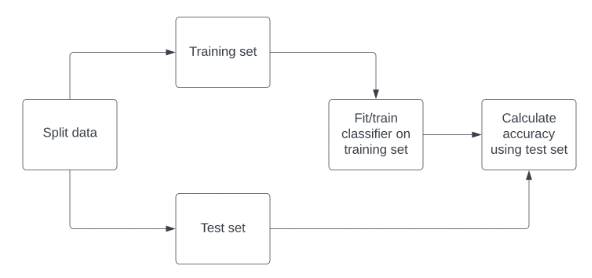
# Predict the labels for the X\_new

y\_pred = knn.predict(X\_new)

# Print the predictions

print("Predictions: {}".format(y\_pred))

* Accuracy 经常用来衡量模型的准确性How do we measure accuracy? We could compute accuracy on the data used to fit the classifier.



* we import train\_test\_split from sklearn.model\_selection. We call train\_test\_split, passing our features and targets.
* We commonly use 20-30% of our data as the test set.
* The random\_state argument sets a seed for a random number generator that splits the data. Using the same number when repeating this step allows us to reproduce the exact split and our downstream results.
* It is best practice to ensure our split reflects the proportion of labels in our data. So if churn occurs in 10% of observations, we want 10% of labels in our training and test sets to represent churn. We achieve this by setting stratify equal to y. train\_test\_split returns four arrays: the training data, the test data, the training labels, and the test labels. We unpack these into X\_train, X\_test, y\_train, and y\_test, respectively.
* We then instantiate a KNN model and fit it to the training data using the dot-fit method. To check the accuracy, we use the dot-score method, passing X test and y test. The accuracy of our model is 88%, which is low given our labels have a 9 to 1 ratio.



* Accuracy的例子

# Import the module

from sklearn.model\_selection import train\_test\_split

X = churn\_df.drop("churn", axis=1).values

y = churn\_df["churn"].values

# Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

knn = KNeighborsClassifier(n\_neighbors=5)

# Fit the classifier to the training data

knn.fit(X\_train, y\_train)

# Print the accuracy

print(knn.score(X\_test, y\_test))

* 例子。注意的是这里要包含12！！！所以括号右边边界是13！！！
* training accuracy decreases as the number of neighbors initially gets larger, and vice versa
* training accuracy decreases and test accuracy increases as the number of neighbors gets larger.

# Create neighbors 这一部分建立了一个循环去得到不同n\_neighbors对应的准确性指标

neighbors = np.arange(1, 13)

train\_accuracies = {}

test\_accuracies = {}

for neighbor in neighbors:

    # Set up a KNN Classifier

    knn = KNeighborsClassifier(n\_neighbors=neighbor)

    # Fit the model

    knn.fit(X\_train, y\_train)

    # Compute accuracy

    train\_accuracies[neighbor] = knn.score(X\_train, y\_train)

    test\_accuracies[neighbor] = knn.score(X\_test, y\_test)

print(neighbors, '\n', train\_accuracies, '\n', test\_accuracies)

# Add a title 这一部分是把前面的出来的neighbors和对应的准确性指标作图

plt.title("KNN: Varying Number of Neighbors")

# Plot training accuracies

plt.plot(neighbors, train\_accuracies.values(), label="Training Accuracy")

# Plot test accuracies

plt.plot(neighbors, test\_accuracies.values(), label="Testing Accuracy")

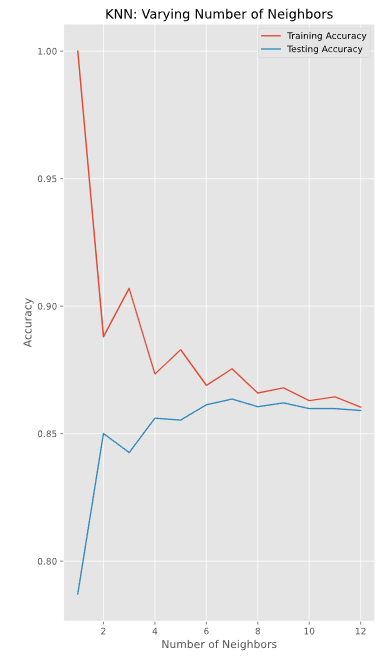
plt.legend()

plt.xlabel("Number of Neighbors")

plt.ylabel("Accuracy")

# Display the plot

plt.show()



# 04/16/24 Regression

* our features must be formatted as a two-dimensional array to be accepted by scikit-learn.
* 使用.reshape()调
* 整one-dimensional arrays到two-dimensional

A screenshot of a computer program

Description automatically generated

* 例子。注意使用的package是sklearn里面的。画图用的是matplotlib.pyplot

# Import LinearRegression

from sklearn.linear\_model import LinearRegression

# Create the model

reg = LinearRegression()

# Fit the model to the data

reg.fit(X,y)

# Make predictions

predictions = reg.predict(X)

print(predictions[:5])

# Import matplotlib.pyplot

import matplotlib.pyplot as plt

# Create line plot

plt.plot(X, y, color="red")

plt.xlabel("Radio Expenditure ($)")

plt.ylabel("Sales ($)")

# Display the plot

plt.show()

* Error function = loss function = cost function

## R-squared

* The default metric for linear regression is R-squared, which quantifies the amount of variance in the target variable that is explained by the features. Values can range from 0 to 1, with one meaning the features completely explain the target's variance. R越高越好说明越能解释

A graph of a person with blue dots

Description automatically generated with medium confidence

* 例子。注意这里定义X的时候用的drop方法以及axis。默认是0，指删除行，如果删除列的话要指明axis=1

# Create X and y arrays

X = sales\_df.drop("sales", axis=1).values

y = sales\_df["sales"].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Instantiate the model

reg = LinearRegression()

# Fit the model to the data

reg.fit(X\_train,y\_train)

# Make predictions

y\_pred = reg.predict(X\_test)

print("Predictions: {}, Actual Values: {}".format(y\_pred[:2], y\_test[:2]))

* 例子。注意MSE用的package是sklearn.metrics里面的。
* R squared计算用的.score()方法
* Mean squared error计算用的是sklearn.metrics里面的mean\_squared\_error

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

# Compute R-squared

r\_squared = reg.score(X\_test, y\_test)

# Compute RMSE

rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)

# Print the metrics

print("R^2: {}".format(r\_squared))

print("RMSE: {}".format(rmse))

## Cross-validation

* Cross-validation is a vital approach to evaluating a model. It maximizes the amount of data that is available to the model, as the model is not only trained but also tested on all of the available data.
* 逻辑是每一次都会计算出一个R squared酱紫

A table of text with yellow squares

Description automatically generated with medium confidence

* sklearn.model\_selection.cross\_val\_score(estimator, X, y=None, \*, groups=None, scoring=None, cv=None, n\_jobs=None, verbose=0, fit\_params=None, params=None, pre\_dispatch='2\*n\_jobs', error\_score=nan)[source]¶
* 6-fold cross-validation R-squared例子

# Import the necessary modules

from sklearn.model\_selection import cross\_val\_score, KFold

# Create a KFold object

kf = KFold(n\_splits=6, shuffle=True, random\_state=5)

reg = LinearRegression()

# Compute 6-fold cross-validation scores

cv\_scores = cross\_val\_score(reg, X, y, cv=kf)

# Print scores

print(cv\_scores)

* 注意np.mean! 还有怎么写出95%置信空间的

# Print the mean

print(np.mean(cv\_results))

# Print the standard deviation

print(np.std(cv\_results))

# Print the 95% confidence interval

print(np.quantile(cv\_results, [0.025, 0.975]))

## Regularized regression

* Recall that fitting a linear regression model minimizes a loss function to choose a coefficient, a, for each feature, and the intercept, b. If we allow these coefficients to be very large, we can get overfitting. Therefore, it is common practice to alter the loss function so that it penalizes large coefficients. This is called regularization.
* Ridge Regression

A screenshot of a computer

Description automatically generated A screenshot of a computer code

Description automatically generated

* Lasso Regression

A close-up of a white background

Description automatically generated A screenshot of a computer code

Description automatically generated

A close-up of a text

Description automatically generated A screenshot of a computer code

Description automatically generated

## Ridge

* Ridge regression performs regularization by computing the squared values of the model parameters multiplied by alpha and adding them to the loss function.
* 例子

# Import Ridge

from sklearn.linear\_model import Ridge

alphas = [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]

ridge\_scores = []

for alpha in alphas:

   # Create a Ridge regression model

   ridge = Ridge(alpha=alpha)

   # Fit the data

   ridge.fit(X\_train,y\_train)

   # Obtain R-squared

   score = ridge.score(X\_test,y\_test)

   ridge\_scores.append(score)

print(ridge\_scores)

## Lasso

* 例子，注意这里直接用lasso.coef\_来计算

# Import Lasso

from sklearn.linear\_model import Lasso

# Instantiate a lasso regression model

lasso = Lasso(alpha=0.3)

# Fit the model to the data

lasso.fit(X,y)

# Compute and print the coefficients

lasso\_coef = lasso.coef\_

print(lasso\_coef)

plt.bar(sales\_columns, lasso\_coef)

plt.xticks(rotation=45)

plt.show()

# 04/21/24 Fine-Tuning Your Model

## Classification report和Confusion matrix

* 注意classification report和confusion matrix两者都是从sklearn.metrics里面import的。并且生成的时候使用的都是target y的值

# Import confusion matrix

from sklearn.metrics import confusion\_matrix, classification\_report

knn = KNeighborsClassifier(n\_neighbors=6)

# Fit the model to the training data

knn.fit(X\_train,y\_train)

# Predict the labels of the test data: y\_pred

y\_pred = knn.predict(X\_test)

# Generate the confusion matrix and classification report

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

## ROC Curve

* LogisticRegression例子。注意安装包。另外最后那部分的目的

# Import LogisticRegression

from sklearn.linear\_model import LogisticRegression

# Instantiate the model

logreg = LogisticRegression()

# Fit the model

logreg.fit(X\_train,y\_train)

# Predict probabilities

# Predict the probabilities of each individual in the test set having a diabetes diagnosis, storing the array of positive probabilities as y\_pred\_probs.

y\_pred\_probs = logreg.predict\_proba(X\_test)[:,1]

print(y\_pred\_probs[:10])

* ROC 例子

# Import roc\_curve

from sklearn.metrics import roc\_curve

# Generate ROC curve values: fpr, tpr, thresholds

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred\_probs)

plt.plot([0, 1], [0, 1], 'k--')

# Plot tpr against fpr

plt.plot(fpr, tpr)

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve for Diabetes Prediction')

plt.show()

* The ROC curve is above the dotted line, so the model performs better than randomly guessing the class of each observation.

A graph of a curve

Description automatically generated

## ROC AUC

* A ROC AUC score of 0.8002 means this model is 60% better than a chance model at correctly predicting labels!
* 例子。注意这里ROC AUC Curve用的y\_pred\_probs是positive 概率

# Import roc\_auc\_score

from sklearn.metrics import roc\_auc\_score

# Calculate roc\_auc\_score

print(roc\_auc\_score(y\_test, y\_pred\_probs))

# Calculate the confusion matrix

print(confusion\_matrix(y\_test, y\_pred))

# Calculate the classification report

print(classification\_report(y\_test, y\_pred))

## Hyperparameter Tuning

* Recall that we had to choose a value for alpha in ridge and lasso regression before fitting it. Likewise, before fitting and predicting KNN, we choose n\_neighbors. Parameters that we specify before fitting a model, like alpha and n\_neighbors, are called hyperparameters. So, a fundamental step for building a successful model
* 过程

A close-up of a text

Description automatically generated

## GridSearchCV

* 过程

A screenshot of a computer code

Description automatically generated

* 局限性：However, the number of fits is equal to the number of hyperparameters multiplied by the number of values multiplied by the number of folds. Therefore, it doesn't scale well! So, performing 3-fold cross-validation for one hyperparameter with 10 values each means 30 fits, while 10-fold cross-validation on 3 hyperparameters with 10 values each equals 900 fits!
* 例子

# Import GridSearchCV

from sklearn.model\_selection import GridSearchCV

# Set up the parameter grid

param\_grid = {"alpha": np.linspace(0.00001, 1, 20)} #取0.00001到1之间20个等间距的数值

# Instantiate lasso\_cv

lasso\_cv = GridSearchCV(lasso, param\_grid, cv=kf)

# Fit to the training data

lasso\_cv.fit(X\_train,y\_train)

print("Tuned lasso paramaters: {}".format(lasso\_cv.best\_params\_))

print("Tuned lasso score: {}".format(lasso\_cv.best\_score\_))

## RandomizedSearchCV

* GridSearchCV can be computationally expensive, especially if you are searching over a large hyperparameter space. In this case, you can use RandomizedSearchCV, which tests a fixed number of hyperparameter settings from specified probability distributions.
* 过程

A screenshot of a computer code

Description automatically generated

* 也是从sklearn.model\_selection
* 例子

# Create the parameter space

params = {"penalty": ["l1", "l2"],

          "tol": np.linspace(0.0001, 1.0, 50),

          "C": np.linspace(0.1, 1.0, 50),

          "class\_weight": ["balanced", {0:0.8, 1:0.2}]}

# Instantiate the RandomizedSearchCV object

logreg\_cv = RandomizedSearchCV(logreg, params, cv=kf)

# Fit the data to the model

logreg\_cv.fit(X\_train, y\_train)

# Print the tuned parameters and score

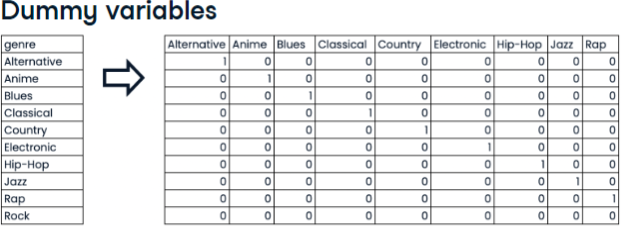
print("Tuned Logistic Regression Parameters: {}".format(logreg\_cv.best\_params\_))

print("Tuned Logistic Regression Best Accuracy Score: {}".format(logreg\_cv.best\_score\_))

# 04/23/24 Preprocessing and Pipelines

## Categorical features

* scikit-learn: OneHotEncoder()
* pandas: get\_dummies()
* 重点是将非binary变量转换成binary的(create dummy variables)



* 例子

# Create music\_dummies

music\_dummies = pd.get\_dummies(music\_df["genre"],drop\_first=True)

# Print the new DataFrame's shape

print("Shape of music\_dummies: {}".format(music\_dummies.shape))

* 例子。注意这里是在做cross validation所以用的是cross\_val\_score()。“Perform cross-validation on X and y using the ridge model, setting cv equal to kf, and using negative mean squared error as the scoring metric. “
* Scoring变量第一次见
* 这里计算RMSE是根据要求直接算squared root。“Print the RMSE values by converting negative scores to positive and taking the square root.“

# Create X and y

X = music\_dummies.drop("popularity",axis=1)

y = music\_dummies["popularity"]

# Instantiate a ridge model

ridge = Ridge(alpha=0.2)

# Perform cross-validation

scores = cross\_val\_score(ridge,X, y, cv=kf, scoring="neg\_mean\_squared\_error")

# Calculate RMSE

rmse = np.sqrt(-scores)

print("Average RMSE: {}".format(np.mean(rmse)))

print("Standard Deviation of the target array: {}".format(np.std(y)))

## Handling missing data - isna(), dropna()

* Dropna
* Inpute values: SimpleImputer, Pipeline
* 例子

# Print the number of missing values for each column in the music\_df dataset, sorted in ascending order

print(music\_df.isna().sum().sort\_values())

# Remove values where less than 5% are missing

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

# Convert genre to a binary feature

# Convert music\_df["genre"] to values of 1 if the row contains "Rock", otherwise change the value to 0.

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1, 0)

print(music\_df.isna().sum().sort\_values())

print("Shape of the `music\_df`: {}".format(music\_df.shape))

## Imputing within a pipeline – Pipeline + SimpleImputer

* 在机器学习中，Pipeline是一个用于处理数据和训练模型的流水线式工作流程。它将多个步骤按照一定的顺序连接起来，形成一个完整的数据处理和模型训练过程。通过Pipeline，我们可以将数据清洗、特征工程、模型训练和评估等多个环节整合在一起，实现自动化和高效的数据处理和模型训练。
* Pipeline可以看作是一条流水线，每个步骤都是流水线上的一个环节。这些环节按照一定的顺序排列，前一个环节的输出作为后一个环节的输入，直到最终输出我们需要的模型或结果。在机器学习中，Pipeline通常包括数据清洗、特征工程、模型训练和评估等环节。
* Imputers are known as transformers
* For categorical values, we typically use the most frequent value - the mode
* Must split our data first, to avoid data leakage
* 例子

# Import modules

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

# Instantiate an imputer

imputer = SimpleImputer()

# Instantiate a KNN classifier with three neighbors

knn = KNeighborsClassifier(n\_neighbors=3)

# Build steps for the pipeline, a list of tuples containing the imputer variable you created

steps = [("imputer", imputer), ("knn", knn)]

# Create the pipeline

pipeline = Pipeline(steps)

# Fit the pipeline to the training data

# X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y)

pipeline.fit(X\_train,y\_train)

# Make predictions on the test set

y\_pred = pipeline.predict(X\_test)

# Print the confusion matrix

print(confusion\_matrix(y\_test, y\_pred))

* 例子

# Import modules

from sklearn.pipeline import Pipeline

from sklearn.impute import SimpleImputer

# 清理数据，创建binary变量和定义X,y

music\_df = music\_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])

music\_df["genre"] = np.where(music\_df["genre"] == "Rock", 1, 0)

X = music\_df.drop("genre", axis=1).values

y = music\_df["genre"].values

# Build steps for the pipeline, a list of tuples containing the imputer variable you created

steps = [("imputation", SimpleImputer()),("logistic\_regression", LogisticRegression())]

# Create the pipeline

pipeline = Pipeline(steps)

# Split the dataset first

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.3,random\_state=42))

# Fit the pipeline to the training data

pipeline.fit(X\_train,y\_train)

# Make predictions on the test set

y\_pred = pipeline.predict(X\_test)

# Print the score

pipeline.score(X\_test, y\_test)

## Imputing within a scikit-learn – SimpleImputer

* 例子

from sklearn.impute import SimpleImputer

X\_cat = music\_df["genre"].values.reshape(-1, 1)

X\_num = music\_df.drop(["genre", "popularity"], axis=1).values

y = music\_df["popularity"].values

X\_train\_cat, X\_test\_cat, y\_train, y\_test = train\_test\_split(X\_cat, y, test\_size=0.2,random\_state=12)

X\_train\_num, X\_test\_num, y\_train, y\_test = train\_test\_split(X\_num, y, test\_size=0.2,random\_state=12)

imp\_cat = SimpleImputer(strategy="most\_frequent")

X\_train\_cat = imp\_cat.fit\_transform(X\_train\_cat)

X\_test\_cat = imp\_cat.transform(X\_test\_cat)

imp\_num = SimpleImputer()

X\_train\_num = imp\_num.fit\_transform(X\_train\_num)

X\_test\_num = imp\_num.transform(X\_test\_num)

X\_train = np.append(X\_train\_num, X\_train\_cat, axis=1)

X\_test = np.append(X\_test\_num, X\_test\_cat, axis=1)

## Centering and scaling – Regression – StandardScaler

* 例子。注意这里计算R-squared的时候用的是pipeline.score()

# Import StandardScaler

from sklearn.preprocessing import StandardScaler

# Create pipeline steps

steps = [("scaler", StandardScaler()),

         ("lasso", Lasso(alpha=0.5))]

# Instantiate the pipeline

pipeline = Pipeline(steps)

pipeline.fit(X\_train, y\_train)

# Calculate and print R-squared

print(pipeline.score(X\_test, y\_test))

## Centering and scaling – Classification

* 例子。注意在指定某个model里面的parameter在一定范围内的时候要用两个下划线！！！

# Build the steps

steps = [("scaler", StandardScaler()),

         ("logreg", LogisticRegression())]

pipeline = Pipeline(steps)

# Create the parameter space -  searching 20 equally spaced float values ranging from 0.001 to 1.0 for the logistic regression model's C hyperparameter within the pipeline

parameters = {"logreg\_\_C": np.linspace(0.001, 1.0, 20)}

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,random\_state=21)

# Instantiate the grid search object

cv = GridSearchCV(pipeline, param\_grid=parameters)

# Fit to the training data

cv.fit(X\_train, y\_train)

print(cv.best\_score\_, "\n", cv.best\_params\_)

* 例子

from sklearn.model\_selection import GridSearchCV

# Build the steps

steps = [('scaler', StandardScaler()),('knn', KNeighborsClassifier())]

# Instantiate the pipeline

pipeline = Pipeline(steps)

# Create the parameter space

parameters = {"knn\_\_n\_neighbors": np.arange(1, 50)}

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2,random\_state=21)

# Instantiate the grid search object

cv = GridSearchCV(pipeline, param\_grid=parameters)

cv.fit(X\_train, y\_train)

# Predict the model

y\_pred = cv.predict(X\_test)

# Checking model parameters

print(cv.best\_score\_)

print(cv.best\_params\_)

## Evaluating models

* Regression model常用：RMSE, R-squared
* Classification model常用：Accuracy, Confusion matrix, Precision/Recall/F-1 Score, ROC AUC
* 会受到Scaling影响的模型：KNN, Linear Regression, Ridge, Lasso, Logistic Regression, Atrifical Neural Network
* 例子。“Lasso regression is not a good model for this problem, while linear regression and ridge perform fairly equally.“

models = {"Linear Regression": LinearRegression(), "Ridge": Ridge(alpha=0.1), "Lasso": Lasso(alpha=0.1)}

results = []

# Loop through the models' values

for model in models.values():

   kf = KFold(n\_splits=6, random\_state=42, shuffle=True)

   # Perform cross-validation

   cv\_scores = cross\_val\_score(model, X\_train, y\_train, cv=kf)

   # Append the results

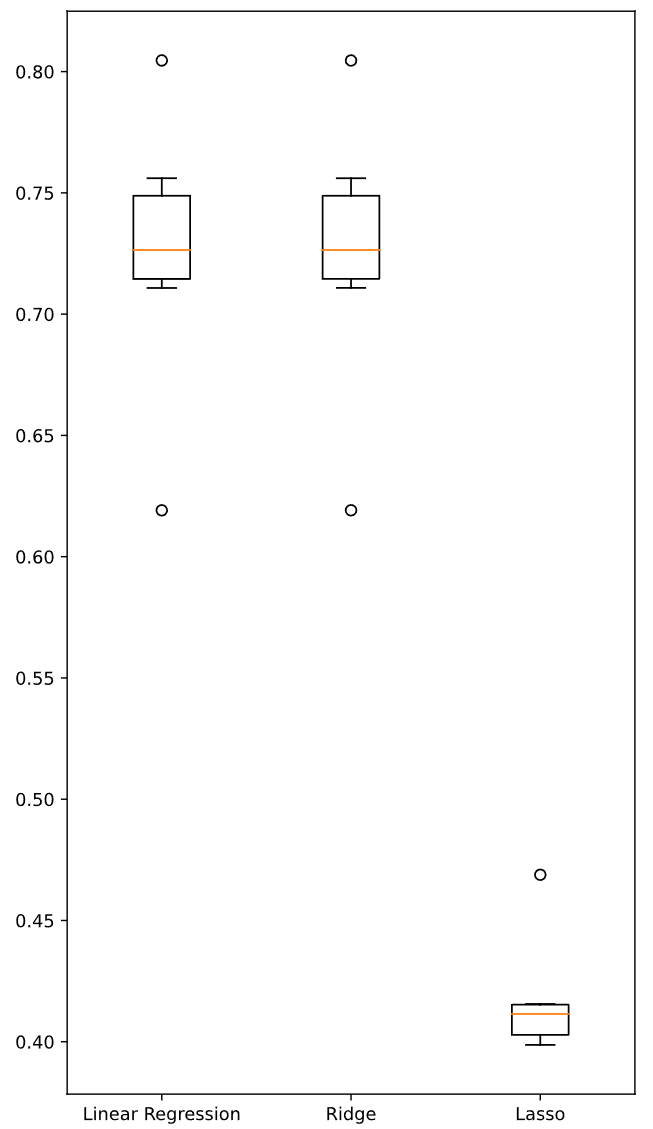
   results.append(cv\_scores)

# Create a box plot of the results

# x-axis labels as the names of the models 使用keys

plt.boxplot(results, labels=models.keys())

plt.show()



* 例子。注意计算RMSE的时候，在mean\_squared\_error里面计算的对象是y\_test和y\_pred，和X没有关系！
* Squared=false参数：If True returns MSE value, if False returns RMSE value.

# Import mean\_squared\_error

from sklearn.metrics import mean\_squared\_error

# Calculate RMSE

for name, model in models.items():

   # Fit the model to the training data

   model.fit(X\_train\_scaled, y\_train)

   # Make predictions on the test set

   y\_pred = model.predict(X\_test\_scaled)

   # Calculate the test\_rmse

   test\_rmse = mean\_squared\_error(y\_test,y\_pred,squared=False)

   print("{} Test Set RMSE: {}".format(name, test\_rmse))

* 例子。注意在loop里面的时候用的是model.values()。cross\_val\_score你老是不记得。

# Create models dictionary

models = {"Logistic Regression": LogisticRegression(), "KNN": KNeighborsClassifier(), "Decision Tree Classifier": DecisionTreeClassifier()}

results = []

# Loop through the models' values

for model in models.values():

   # Instantiate a KFold object

   kf = KFold(n\_splits=6, random\_state=12, shuffle=True)

   # Perform cross-validation

   cv\_results = cross\_val\_score(model, X\_train\_scaled, y\_train, cv=kf)

   results.append(cv\_results)

plt.boxplot(results, labels=models.keys())

plt.show()

* 例子。注意在最后计算accuracy的时候用的是cv.score(X\_test, y\_test)不是cv.best\_score\_
* “For the final exercise, you will build a pipeline to impute missing values, scale features, and perform hyperparameter tuning of a logistic regression model. The aim is to find the best parameters and accuracy when predicting song genre!“

# Create the steps for the pipeline by calling a simple imputer, a standard scaler, and a logistic regression model

steps = [("imp\_mean", SimpleImputer()), ("scaler", StandardScaler()), ("logreg", LogisticRegression())]

# Set up pipeline object, and pass the steps variable

pipeline = Pipeline(steps)

params = {"logreg\_\_solver": ["newton-cg", "saga", "lbfgs"], "logreg\_\_C": np.linspace(0.001, 1.0, 10)}

# Create the GridSearchCV object to perform cross-validation using the pipeline and the parameters

tuning = GridSearchCV(pipeline, param\_grid=params)

tuning.fit(X\_train, y\_train)

y\_pred = tuning.predict(X\_test)

# Print the best parameters and compute and print the test set accuracy score for the grid search object

print("Tuned Logistic Regression Parameters: {}, Accuracy: {}".format(tuning.best\_params\_, tuning.score(X\_test,y\_test)))

# 04/28/24 【**Predictive Modeling for Agriculture】**Project

# All required libraries are imported here for you.

import pandas as pd

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

from sklearn.metrics import accuracy\_score #自己加的

# Read the data into a pandas DataFrame and perform exploratory data analysis

crops = pd.read\_csv("soil\_measures.csv")

# Print out crops

crops.head()

A screenshot of a computer

Description automatically generated

# Check for missing values

print(crops.isna().sum().sort\_values())

A white background with black text

Description automatically generated

# Check for crop types

crops.crop.unique()

A white background with black text

Description automatically generated

# Define features and target variables

X = crops.drop("crop",axis=1)

y = crops["crop"]

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a dictionary to store each features predictive performance

features\_performance = {}

# Create the best\_predictive\_feature variable

best\_predictive\_feature = {}

best\_score = 0

# Loop through the features

for feature in ["N",'P','K','ph']:

    # Instantiate the model

    model = LogisticRegression()

    # Fit the model

    model.fit(X\_train[[feature]],y\_train)

    # Predicting target values using the test set

    y\_pred = model.predict(X\_test[[feature]])

    # Evaluating the performance of each feature

    accuracy = accuracy\_score(y\_test, y\_pred)

    # Append the feature performance list

    features\_performance[feature] = accuracy

    # Find the best feature accuracy

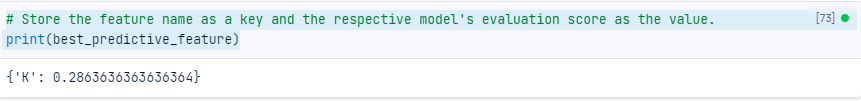
    if accuracy > best\_score:

        best\_score = accuracy

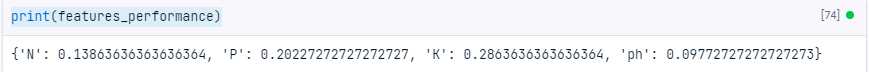
        best\_predictive\_feature = {feature:accuracy}

# Store the feature name as a key and the respective model's evaluation score as the value.

print(best\_predictive\_feature)



print(features\_performance)



# 04/25/24 【**Unsupervised Learning in Python】-** Clustering for Dataset Exploration

* Unsupervised learning is a class of machine learning techniques for discovering patterns in data. For instance, finding the natural "clusters" of customers based on their purchase histories, or searching for patterns and correlations among these purchases, and using these patterns to express the data in a compressed form. These are examples of unsupervised learning techniques called "clustering" and "dimension reduction".
* Unsupervised learning, in contrast, is learning without labels. It is pure pattern discovery, unguided by a prediction task.
* 一般可以先用scatter plot（使用的是PyPlot (matplotlib.pyplot)包）看看有多少個cluster，然後再做clustering
* from sklearn.cluster import KMeans

## K-means clustering – Kmeans()

* In this chapter, we'll cluster these samples using k-means clustering. k-means finds a specified number of clusters in the samples. It's implemented in the scikit-learn or "sklearn" library.
* If someone comes along with some new iris samples, k-means can determine to which clusters they belong without starting over. k-means does this by remembering the mean (or average) of the samples in each cluster. These are called the "centroids". New samples are assigned to the cluster whose centroid is closest.
* 例子

# Import KMeans from sklearn.cluster

from sklearn.cluster import KMeans

# Create a KMeans instance with 3 clusters: model

# Using KMeans(), create a KMeans instance called model to find 3 clusters. To specify the number of clusters, use the n\_clusters keyword argument

model = KMeans(n\_clusters=3)

# Fit model to points

# Use the .fit() method of model to fit the model to the array of points points

model.fit(points)

# Determine the cluster labels of new\_points: labels

# Use the .predict() method of model to predict the cluster labels of new\_points, assigning the result to labels

labels = model.predict(new\_points)

# Print cluster labels of new\_points

print(labels)

* 例子

# Import pyplot

import matplotlib.pyplot as plt

# Assign the columns of new\_points: xs and ys

# Assign column 0 of new\_points to xs, and column 1 of new\_points to ys.

xs = new\_points[:,0]

ys = new\_points[:,1]

# Make a scatter plot of xs and ys, using labels to define the colors

# Make a scatter plot of xs and ys, specifying the c=labels keyword arguments to color the points by their cluster label. Also specify alpha=0.5.

plt.scatter(xs,ys,c=labels,alpha=0.5)

# Assign the cluster centers: centroids

# Compute the coordinates of the centroids using the .cluster\_centers\_ attribute of model.

centroids = model.cluster\_centers\_

# Assign the columns of centroids: centroids\_x, centroids\_y

centroids\_x = centroids[:,0]

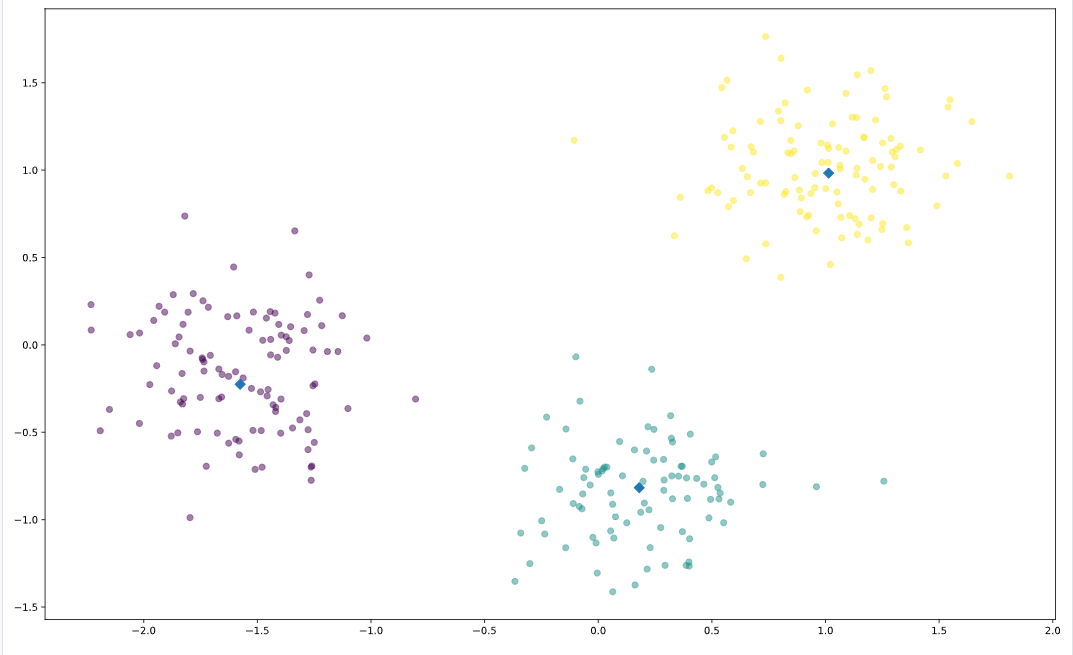
centroids\_y = centroids[:,1]

# Make a scatter plot of centroids\_x and centroids\_y

# Make a scatter plot of centroids\_x and centroids\_y, using 'D' (a diamond) as a marker by specifying the marker parameter. Set the size of the markers to be 50 using s=50.

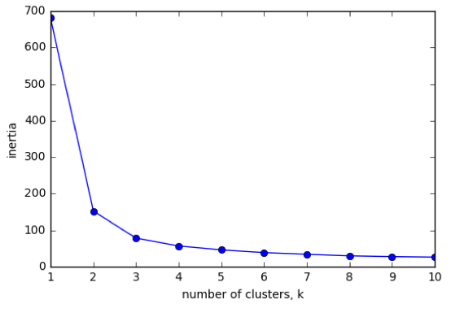
plt.scatter(centroids\_x,centroids\_y,marker='D',s=50)

plt.show()



## Measure clustering quality

* A good clustering has tight clusters, meaning that the samples in each cluster are bunched together, not spread out.
* How spread out the samples within each cluster are can be measured by the "inertia". Intuitively, inertia measures how far samples are from their centroids. You can find the precise definition in the scikit-learn documentation.
* We want clusters that are not spread out, so lower values of the inertia are better. The inertia of a kmeans model is measured automatically when any of the fit methods are called, and is available afterwards as the inertia attribute.
* In fact, kmeans aims to place the clusters in a way that minimizes the inertia.
* Ultimately, this is a trade-off. A good clustering has tight clusters (meaning low inertia). But it also doesn't have too many clusters. A good rule of thumb is to choose an elbow in the inertia plot, that is, a point where the inertia begins to decrease more slowly. For example, by this criterion, 3 is a good number of clusters for the iris dataset.



* 例子。“The inertia decreases very slowly from 3 clusters to 4, so it looks like 3 clusters would be a good choice for this data.”

ks = range(1, 6)

inertias = []

for k in ks:

    # Create a KMeans instance with k clusters: model

    model = KMeans(n\_clusters=k)

    # Fit model to samples

    model.fit(samples)

    # Append the inertia to the list of inertias

# Append the value of the inertia\_ attribute of model to the list inertias.

    inertias.append(model.inertia\_)

# Plot ks vs inertias

plt.plot(ks, inertias, '-o')

plt.xlabel('number of clusters, k')

plt.ylabel('inertia')

plt.xticks(ks)

plt.show()

* 例子。In the previous exercise, you observed from the inertia plot that 3 is a good number of clusters for the grain data. In fact, the grain samples come from a mix of 3 different grain varieties: "Kama", "Rosa" and "Canadian". In this exercise, cluster the grain samples into three clusters, and compare the clusters to the grain varieties using a cross-tabulation.
* Using .fit\_predict() is the same as using .fit() followed by .predict().
* The cross-tabulation shows that the 3 varieties of grain separate really well into 3 clusters.

# Create a KMeans model with 3 clusters: model

model = KMeans(n\_clusters=3)

# Use fit\_predict to fit model and obtain cluster labels: labels

# Use the .fit\_predict() method of model to fit it to samples and derive the cluster labels.

# Using .fit\_predict() is the same as using .fit() followed by .predict().

labels = model.fit\_predict(samples)

# Create a DataFrame with labels and varieties as columns: df

# Create a DataFrame df with two columns named 'labels' and 'varieties', using labels and varieties

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

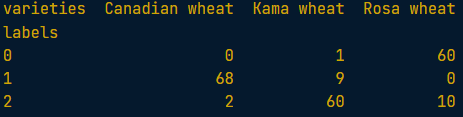
# Create crosstab: ct

# Use the pd.crosstab() function on df['labels'] and df['varieties'] to count the number of times each grain variety coincides with each cluster label. Assign the result to ct.

ct = pd.crosstab(df['labels'],df['varieties'])

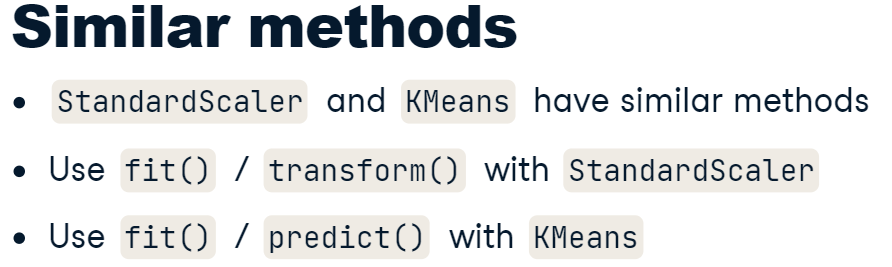
# Display ct

print(ct)



## Transforming features for better clusterings – StandardScaler()

* But depending on the type of data you are working with, the clustering may not always be this good. Is there anything you can do in such situations to improve your clustering?
* In KMeans clustering, the variance of a feature corresponds to its influence on the clustering algorithm. To give every feature a chance, the data needs to be transformed so that features have equal variance.
* This can be achieved with the StandardScaler from scikit-learn. It transforms every feature to have mean 0 and variance 1. The resulting "standardized" features can be very informative.
* import StandardScaler from sklearn.preprocessing
* The APIs of StandardScaler and KMeans are similar, but there is an important difference. StandardScaler transforms data, and so has a transform method. KMeans, in contrast, assigns cluster labels to samples, and this done using the predict method.



* 先用StandardScaler把數據標準化，然後用KMeans來做cluster。兩個步驟可以用scikit-learn pipeline 結合
* 還有其他的處理數據的方式StandardScaler is an example of a "preprocessing" step. There are several of these available in scikit-learn, for example MaxAbsScaler and Normalizer.

# Perform the necessary imports

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

# Create an instance of StandardScaler called scaler.

scaler = StandardScaler()

# Create an instance of KMeans with 4 clusters called kmeans.

kmeans = KMeans(n\_clusters=4)

# Create a pipeline called pipeline that chains scaler and kmeans. To do this, you just need to pass them in as arguments to make\_pipeline().

pipeline = make\_pipeline(scaler,kmeans)

* 例子。建立cross-tabulation來比較cluster labels結果和原來的數據看看聚類效果。

# Import pandas

import pandas as pd

# Fit the pipeline to samples

pipeline.fit(samples)

# Calculate the cluster labels: labels

# Obtain the cluster labels for samples by using the .predict() method of pipeline.

labels = pipeline.predict(samples)

# Create a DataFrame with labels and species as columns: df

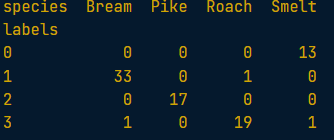
df = pd.DataFrame({'labels':labels,'species':species})

# Create crosstab: ct

ct = pd.crosstab(df['labels'],df['species'])

# Display ct

print(ct)



## Normalizer ()

* 例子
* Note that Normalizer() is different to StandardScaler(), which you used in the previous exercise. While StandardScaler() standardizes features by removing the mean and scaling to unit variance, Normalizer() rescales each sample - here, each company's stock price - independently of the other.

# Import Normalizer

from sklearn.preprocessing import Normalizer

# Create a normalizer: normalizer

normalizer = Normalizer()

# Create a KMeans model with 10 clusters: kmeans

kmeans = KMeans(n\_clusters=10)

# Make a pipeline chaining normalizer and kmeans: pipeline

pipeline = make\_pipeline(normalizer,kmeans)

# Fit pipeline to the daily price movements

pipeline.fit(movements)

# Import pandas

import pandas as pd

# Predict the cluster labels: labels

labels = pipeline.predict(movements)

# Create a DataFrame aligning labels and companies: df

df = pd.DataFrame({'labels': labels, 'companies': companies})

# Display df sorted by cluster label

print(df.sort\_values('labels'))

# 04/28/24 Visualization with Hierarchical Clustering and t-SNE

* Hierarchical clustering merges the data samples into ever-coarser clusters, yielding a tree visualization of the resulting cluster hierarchy.
* t-SNE maps the data samples into 2d space so that the proximity of the samples to one another can be visualized.

## Hierarchical clustering - SciPy linkage()

* In the video, you learned that the SciPy linkage() function performs hierarchical clustering on an array of samples. Use the linkage() function to obtain a hierarchical clustering of the grain samples, and use dendrogram() to visualize the result.
* 例子

# Perform the necessary imports

from scipy.cluster.hierarchy import linkage, dendrogram

import matplotlib.pyplot as plt

# Calculate the linkage: mergings

mergings = linkage(samples,method='complete')

# Plot the dendrogram, using varieties as labels

dendrogram(mergings, labels=varieties, leaf\_rotation=90, leaf\_font\_size=6,)

plt.show()

A diagram of a city

Description automatically generated

* 例子。leaf\_rotation=90是底部标签的角度。leaf\_font\_size=6是底部标签的大小

# Import normalize

from sklearn.preprocessing import normalize

# Normalize the movements: normalized\_movements

normalized\_movements = normalize(movements)

# Calculate the linkage: mergings

mergings = linkage(normalized\_movements,method='complete')

# Plot the dendrogram of the hierarchical clustering, using the list companies of company names as the labels. In addition, specify the leaf\_rotation=90, and leaf\_font\_size=6 keyword arguments as you did in the previous exercise.

dendrogram(mergings,labels=companies,leaf\_rotation=90,leaf\_font\_size=6)

plt.show()

A diagram of a city

Description automatically generated with medium confidence

* In complete linkage, the distance between clusters is the distance between the furthest points of the clusters. In single linkage, the distance between clusters is the distance between the closest points of the clusters.
* Single linkage例子

# Perform the necessary imports

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import linkage, dendrogram

# Calculate the linkage: mergings

mergings = linkage(samples,method='single')

# Plot the dendrogram

dendrogram(mergings,labels=country\_names,leaf\_rotation=90,leaf\_font\_size=6)

plt.show()

* 例子Now, use the fcluster() function to extract the cluster labels for this intermediate clustering, and compare the labels with the grain varieties using a cross-tabulation.

# Perform the necessary imports

import pandas as pd

from scipy.cluster.hierarchy import fcluster

# Use fcluster to extract labels: labels

# Perform a flat hierarchical clustering by using the fcluster() function on mergings. Specify a maximum height of 6 and the keyword argument criterion='distance'.

labels = fcluster(mergings,6,criterion='distance')

# Create a DataFrame df with two columns named 'labels' and 'varieties', using labels and varieties, respectively, for the column values.

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create a cross-tabulation ct between df['labels'] and df['varieties'] to count the number of times each grain variety coincides with each cluster label.

ct = pd.crosstab(df['labels'],df['varieties'])

# Display ct

print(ct)

## t-SNE = t-distributed stochastic neighbor embedding

* It maps samples from their high-dimensional space into a 2- or 3-dimensional space so they can visualized. While some distortion is inevitable, t-SNE does a great job of approximately representing the distances between the samples.
* t-SNE only has a fit\_transform method. As you might expect, the fit\_transform method simultaneously fits the model and transforms the data. However, t-SNE does not have separate fit and transform methods. This means that you can't extend a t-SNE map to include new samples. Instead, you have to start over each time.
* 例子

# Import TSNE

from sklearn.manifold import TSNE

# Create a TSNE instance: model

model = TSNE(learning\_rate=200)

# Apply the .fit\_transform() method of model to samples. Assign the result to tsne\_features.

tsne\_features = model.fit\_transform(samples)

# Select the 0th feature: xs

# column 0 of tsne\_features

xs = tsne\_features[:,0]

# Select the 1st feature: ys

# column 1 of tsne\_features

ys = tsne\_features[:,1]

# Scatter plot, coloring by variety\_numbers

plt.scatter(xs,ys,c=variety\_numbers)

plt.show()

* 例子。注意最后标注公司名字的用法

# Import TSNE

from sklearn.manifold import TSNE

# Create a TSNE instance: model

model = TSNE(learning\_rate=50)

# Apply fit\_transform to normalized\_movements: tsne\_features

tsne\_features = model.fit\_transform(normalized\_movements)

# Select the 0th feature: xs

xs = tsne\_features[:,0]

# Select the 1th feature: ys

ys = tsne\_features[:,1]

# Scatter plot

plt.scatter(xs,ys,alpha=0.5)

# Annotate the points - Code to label each point with its company name has been written for you using plt.annotate(), so just hit submit to see the visualization!for x, y, company in zip(xs, ys, companies):

    plt.annotate(company, (x, y), fontsize=5, alpha=0.75)

plt.show()

A diagram of a number of cities

Description automatically generated with medium confidence

# 05/07/24 Decorrelating Your Data and Dimension Reduction

* Dimension reduction finds patterns in data, and uses these patterns to re-express it in a compressed form.
* Code to label each point with its company name has been written for you using plt.annotate(), so just hit submit to see the visualization!

## PCA – Principal Component Analysis

* PCA performs dimension reduction in two steps
  + 1st called "de-correlation", doesn't change the dimension of the data at all. In this first step, PCA rotates the samples so that they are aligned with the coordinate axes. In fact, it does more than this: PCA also shifts the samples so that they have mean zero. These scatter plots show the effect of PCA applied to two features of the wine dataset. Notice that no information is lost - this is true no matter how many features your dataset has.
* 例子 Pearson Correlation – 可以看到两组数列的相关性

# Perform the necessary imports

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

# Assign the 0th column of grains: width

width = grains[:,0]

# Assign the 1st column of grains: length

length = grains[:,1]

# Scatter plot width vs length

plt.scatter(width, length)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation

correlation, pvalue = pearsonr(width,length)

# Display the correlation

print(correlation)

* 例子 – 使用PCA取decorrelate

# Import PCA

from sklearn.decomposition import PCA

# Create PCA instance: model

model = PCA()

# Apply the fit\_transform method of model to grains: pca\_features

pca\_features = model.fit\_transform(grains)

# Assign 0th column of pca\_features: xs

xs = pca\_features[:,0]

# Assign 1st column of pca\_features: ys

ys = pca\_features[:,1]

# Scatter plot xs vs ys

plt.scatter(xs, ys)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation of xs and ys

correlation, pvalue = pearsonr(xs, ys)

# Display the correlation

print(correlation)

* 图一和图三都可以代表principal components of the point cloud

A screenshot of a computer

Description automatically generated

* The intrinsic dimension of a dataset is the number of features required to approximate it. The intrinsic dimension informs dimension reduction, because it tells us how much a dataset can be compressed.
* scatter plots are only possible if there are 3 features or less.
* So how can the intrinsic dimension be identified, even if there are many features? This is where PCA is really helpful. The intrinsic dimension can be identified by counting the PCA features that have high variance.
* The intrinsic dimension is the number of PCA features that have significant variance.
* PCA discards the low variance features, and assumes that the higher variance features are informative.
* 例子first principal component

# Make a scatter plot of the untransformed points

plt.scatter(grains[:,0], grains[:,1])

# Create a PCA instance: model

model = PCA()

# Fit model to points

model.fit(grains)

# Get the mean of the grain samples: mean

mean = model.mean\_

# Get the first principal component: first\_pc

first\_pc = model.components\_[0,:]

# Plot first\_pc as an arrow, starting at mean

plt.arrow(mean[0], mean[1], first\_pc[0], first\_pc[1], color='red', width=0.01)

# Keep axes on same scale

plt.axis('equal')

plt.show()

* 例子。画出PCA feature的variance图

# Perform the necessary imports

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import make\_pipeline

import matplotlib.pyplot as plt

# Create scaler: scaler

scaler = StandardScaler()

# Create a PCA instance: pca

pca = PCA()

# Create pipeline: pipeline

pipeline = make\_pipeline(scaler,pca)

# Fit the pipeline to 'samples'

pipeline.fit(samples)

# Extract the number of components used using the .n\_components\_ attribute of pca. Place this inside a range() function and store the result as features.

features = range(pca.n\_components\_)

# Use the plt.bar() function to plot the explained variances, with features on the x-axis and pca.explained\_variance\_ on the y-axis.

plt.bar(features, pca.explained\_variance\_)

plt.xlabel('PCA feature')

plt.ylabel('variance')

plt.xticks(features)

plt.show()

* Dimension reduction with PCA

# Import PCA

from sklearn.decomposition import PCA

# Create a PCA model with 2 components: pca

pca = PCA(n\_components=2)

# Fit the PCA instance to the scaled samples

pca.fit(scaled\_samples)

# Transform the scaled samples: pca\_features

pca\_features = pca.transform(scaled\_samples)

# Print the shape of pca\_features

print(pca\_features.shape)

## tf-idf word-frequency array – TfidfVectorizer()

* use the TfidfVectorizer from sklearn. It transforms a list of documents into a word frequency array, which it outputs as a csr\_matrix. It has fit() and transform() methods like other sklearn objects.
* "tf" is the frequency of the word in the document. So if 10% of the words in the document are "datacamp", then the tf of "datacamp" for that document is 0.1.
* "idf" is a weighting scheme that reduces the influence of frequent words like "the".

# Import TfidfVectorizer

from sklearn.feature\_extraction.text import TfidfVectorizer

# Create a TfidfVectorizer: tfidf

tfidf = TfidfVectorizer()

# Apply .fit\_transform() method of tfidf to documents and assign the result to csr\_mat.

# This is a word-frequency array in csr\_matrix format.

csr\_mat = tfidf.fit\_transform(documents)

# Print result of toarray() method

print(csr\_mat.toarray())

# Get the words: words

words = tfidf.get\_feature\_names()

# Print words

print(words)

* 例子

# Perform the necessary imports

from sklearn.decomposition import TruncatedSVD

from sklearn.cluster import KMeans

from sklearn.pipeline import make\_pipeline

# Create a TruncatedSVD instance: svd

svd = TruncatedSVD(n\_components=50)

# Create a KMeans instance: kmeans

kmeans = KMeans(n\_clusters=6)

# Create a pipeline: pipeline

pipeline = make\_pipeline(svd,kmeans)

# Import pandas

import pandas as pd

# Fit the pipeline to articles

pipeline.fit(articles)

# Calculate the cluster labels: labels

# Predict the cluster labels.

labels = pipeline.predict(articles)

# Create a DataFrame aligning labels and titles: df

df = pd.DataFrame({'label': labels, 'article': titles})

# Display df sorted by cluster label

print(df.sort\_values('label'))

# 05/16/24 Discovering Interpretable Features

* In this chapter, you'll learn about a dimension reduction technique called "Non-negative matrix factorization" ("NMF") that expresses samples as combinations of interpretable parts. For example, it expresses documents as combinations of topics, and images in terms of commonly occurring visual patterns.

## Non-negative matrix factorization" ("NMF")

* In contract to PCA, NMF models are interpretable. This means an NMF models are easier to understand yourself, and much easier for you to explain to others.
* NMF cannot be applied to every dataset, however. It is required that the sample features be "non-negative", so greater than or equal to 0.
* NMF is available in scikit learn, and follows the same fit/transform pattern as PCA. >> “NMF(n\_components=2)”
* However, unlike PCA, the desired number of components must always be specified. NMF works both with numpy arrays and sparse arrays in the csr\_matrix format.
* The NMF feature values are non-negative, as well. As we saw with PCA
* 非负矩阵分解，顾名思义就是，将非负的大矩阵分解成两个非负的小矩阵。
* 例子

# Import NMF

from sklearn.decomposition import NMF

# Create an NMF instance: model

model = NMF(n\_components=6)

# Fit the model to to the word count data "articles"

model.fit(articles)

# Use the .transform() method of model to transform articles, and assign the result to nmf\_features.

nmf\_features = model.transform(articles)

# Print nmf\_features to get a first idea what it looks like (.round(2) rounds the entries to 2 decimal places.)

print(nmf\_features.round(2))

* 例子。注意這裏如何設置index的
* 結論：Notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component.

# Import pandas

import pandas as pd

# Create a DataFrame df from nmf\_features using pd.DataFrame(). Set the index to titles using index=titles.

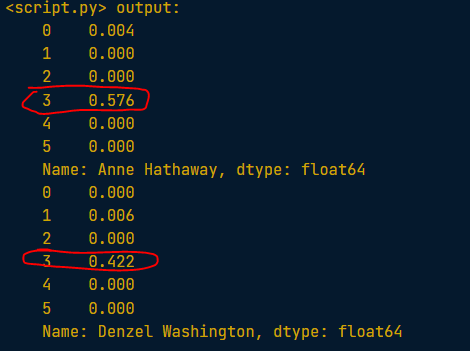
df = pd.DataFrame(nmf\_features,index = titles)

# Use the .loc[] accessor of df to select the row with title 'Anne Hathaway', and print the result. These are the NMF features for the article about the actress Anne Hathaway.

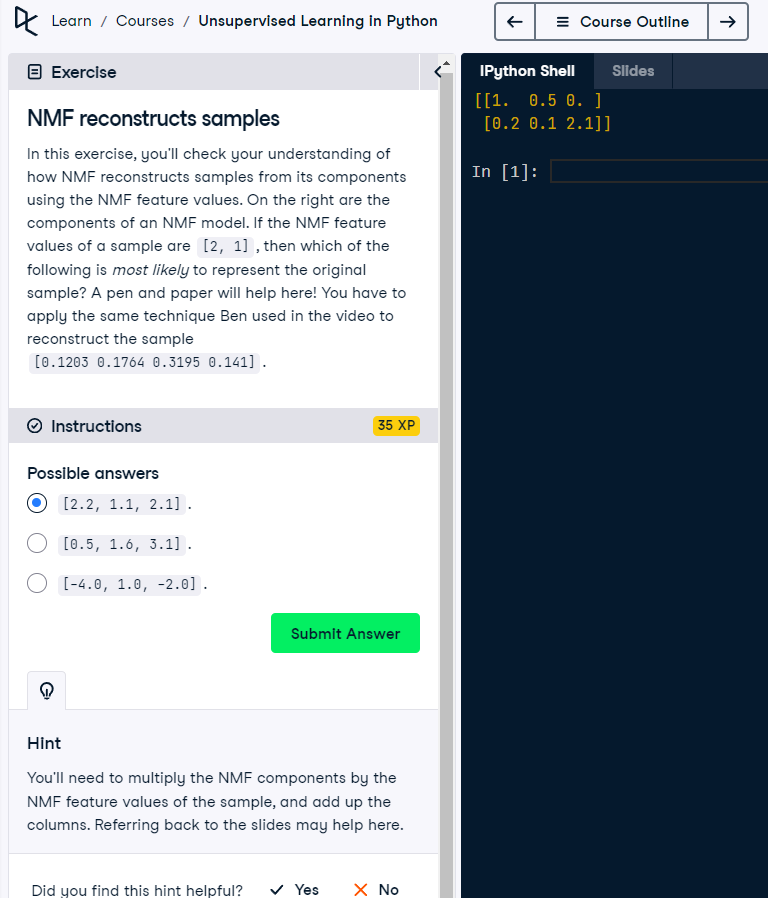
print(df.loc['Anne Hathaway',:])

# Print the row for 'Denzel Washington'

print(df.loc['Denzel Washington',:])



* 簡單例子理解NMF。
* 2\*1 + 1\*0.2
* 2\*0.5 + 1\*0.1
* 2\*0 + 1\*2.1



* 例子

# Import pandas

import pandas as pd

# Create a DataFrame components\_df from model.components\_, setting columns=words so that columns are labeled by the words.

components\_df = pd.DataFrame(model.components\_, columns=words)

# Print the shape of the DataFrame

print(components\_df.shape)

# Select row 3: component

component = components\_df.loc[3,:]

# Print result of nlargest.  This gives the five words with the highest values for that component.

print(component.nlargest())

* 例子

# Import pyplot

from matplotlib import pyplot as plt

# Select row 0 of samples and assign the result to digit. Remember that since samples is a NumPy array, you can't use the .loc[] or iloc[] accessors to select specific rows or columns.

digit = samples[0,:]

# Print digit

print(digit)

# Reshape digit to a 13x8 array: bitmap

bitmap = digit.reshape(13,8)

# Print bitmap

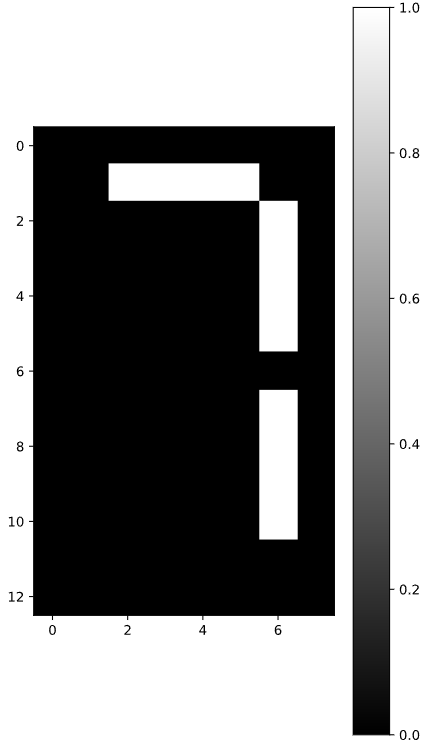
print(bitmap)

# Use plt.imshow to display bitmap

plt.imshow(bitmap, cmap='gray', interpolation='nearest')

plt.colorbar()

plt.show()



* 例子

def show\_as\_image(sample):

    bitmap = sample.reshape((13, 8))

    plt.figure()

    plt.imshow(bitmap, cmap='gray', interpolation='nearest')

    plt.colorbar()

    plt.show()

# Import NMF

from sklearn.decomposition import NMF

# Create an NMF model: model.  7 is the number of cells in an LED display

model = NMF(n\_components=7)

# Apply the .fit\_transform() method of model to samples. Assign the result to features.

features = model.fit\_transform(samples)

# To each component of the model (accessed via model.components\_), apply the show\_as\_image() function to that component inside the loop.

for component in model.components\_:

    show\_as\_image(component)

# Select the 0th row of features: digit\_features

digit\_features = features[0,:]

# Print digit\_features

print(digit\_features)

## NMF learns interpretable parts

* Unlike NMF, PCA doesn't learn the parts of things. Its components do not correspond to topics (in the case of documents) or to parts of images, when trained on images.
* 例子。可以看到Notice that the components of PCA do not represent meaningful parts of images of LED digits!

# Import PCA

from sklearn.decomposition import PCA

# Create a PCA instance: model

model = PCA(n\_components=7)

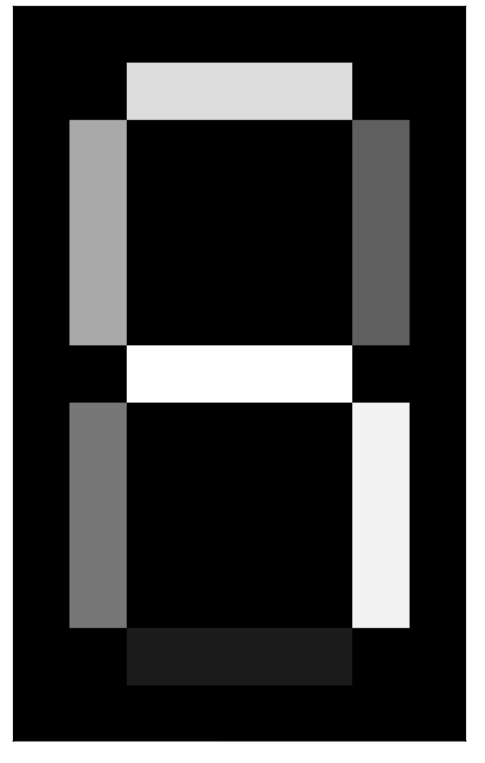
# Apply fit\_transform to samples: features

features = model.fit\_transform(samples)

# Call show\_as\_image on each component

for component in model.components\_:

    show\_as\_image(component)

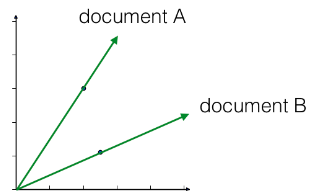


## Building recommender systems using NMF

* On a scatter plot of the NMF features, all these versions lie on a single line passing through the origin.



* For this reason, when comparing two documents, it's a good idea to compare these lines. We'll compare them using what is known as the cosine similarity 餘弦相似度, which uses the angle between the two lines.
* Higher values indicate greater similarity.



* 例子

# Perform the necessary imports

import pandas as pd

from sklearn.preprocessing import normalize

# Normalize the NMF features: norm\_features

norm\_features = normalize(nmf\_features)

# Create a DataFrame df from norm\_features, using titles as an index.

df = pd.DataFrame(norm\_features,index=titles)

# Use the .loc[] accessor of df to select the row of 'Cristiano Ronaldo'. Assign the result to article.

article = df.loc['Cristiano Ronaldo',:]

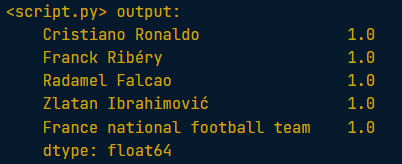
# Compute the dot products: similarities

# Apply the .dot() method of df to article to calculate the cosine similarity of every row with article.

similarities = df.dot(article)

# Display those with the largest cosine similarity

print(similarities.nlargest())



* 例子
* build a pipeline and transform the array into normalized NMF features. The first step in the pipeline, MaxAbsScaler, transforms the data so that all users have the same influence on the model, regardless of how many different artists they've listened to.
* 這一步驟的結果norm\_features得到的是normalized NMF features as rows

# Perform the necessary imports

from sklearn.decomposition import NMF

from sklearn.preprocessing import Normalizer, MaxAbsScaler

from sklearn.pipeline import make\_pipeline

# Create a MaxAbsScaler: scaler

scaler = MaxAbsScaler()

# Create an NMF model: nmf

nmf = NMF(n\_components=20)

# Create a Normalizer: normalizer

normalizer = Normalizer()

# Create a pipeline: pipeline

pipeline = make\_pipeline(scaler,nmf,normalizer)

# Apply fit\_transform to artists: norm\_features

norm\_features = pipeline.fit\_transform(artists)

* 這一步驟是利用餘弦相似性找到類似于Bruce Springsteen的其他音樂

# Import pandas

import pandas as pd

# Create a DataFrame: df

df = pd.DataFrame(norm\_features,index=artist\_names)

# Select row of 'Bruce Springsteen': artist

artist = df.loc["Bruce Springsteen",:]

# Compute cosine similarities: similarities

# Apply the .dot() method of df to artist to calculate the dot product of every row with artist. Save the result as similarities.

similarities = df.dot(artist)

# Display those with highest cosine similarity

print(similarities.nlargest())

# 06/04/24 【**Clustering Antarctic Penguin Species】**Project

<https://www.kaggle.com/code/muhammadanas0716/penguins-data-analysis>

答案參考：<https://www.kaggle.com/code/mustafagerme/arctic-penguin-exploration-kmeans-clustering>

未來回顧問題：

1. 我沒有用PCA部分，爲什麽kmeans之前要用PCA？
2. 最後一步的nunique用法

Takeaways:

主成分分析（Principal Components Analysis），简称PCA，是一种数据降维技术，用于数据预处理。PCA是一种将高维坐标系映射到低维坐标系中的特征提取技术。如果想要确保处理后的变量相互独立，识别可以完全移除的变量，并且接受变化后的自变量可以不那么容易解释，那么PCA是一个很合适的选择。

<https://365datascience.com/tutorials/python-tutorials/pca-k-means/>

However, when we employ PCA prior to using K-means we can visually separate almost the entire data set. That was one of the biggest goals of PCA - to reduce the number of variables by combining them into bigger, more meaningful features.

Not only that, but they are ‘orthogonal’ to each other. This means that the difference between components is as big as possible.

<https://stats.stackexchange.com/questions/235946/pca-before-cluster-analysis#:~:text=By%20doing%20PCA%20you%20are,they%20are%20most%20spread%20out>).

By doing PCA you are retaining all the important information. If your data exhibits clustering, this will be generally revealed after your PCA analysis: by retaining only the components with the highest variance, the clusters will be likely more visibile (as they are most spread out).

# 06/06/24 【**Linear Classification in Python】-** Applying logistic regression and SVM

* 這章也是會用到supervised learning裏面的scikit-learn的說
* 例子。記得這裏fit用X\_train和y\_train。然後預測時候用X\_test即可

from sklearn.neighbors import KNeighborsClassifier

# Create and fit the model

knn = KNeighborsClassifier()

knn.fit(X\_train,y\_train)

# Predict on the test features, print the results

pred = knn.predict(X\_test)[0]

print("Prediction for test example 0:", pred)

# 06//24 Loss functions

# 06//24 Logistic regression

# 06//24 Support Vector Machines

# 06/18/24 【**Introduction to Deep Learning with PyTorch】-** Introduction to PyTorch, a Deep Learning Library

* Deep learning outperforms traditional machine learning by extracting intricate patterns from "big" and unstructured data, such as image pixels, large corpora of text, and audio signals. Traditional machine learning relies on hand-crafted feature engineering. Deep learning models, on the other hand, using a "layered" network structure, are able to discover features from raw data, giving them that edge over traditional machine learning. This is known as feature learning or representation learning.