Lab cheat sheet (1)

iris.data.shape

shape is an attribute, it returns the rows and columns of a dataset (rows, columns), which could be (150, 4)

.data and .target

`iris.data[12, :]`	Prints the 13th sample's feature values	`[4.3 3. 1.1 0.1]`
`iris.target[12]`	Prints the class label of the 13th sample	`0` (Setosa)

A **class label** is the category or group that a given sample (data point) belongs to in a **classification problem**. It is the output variable that a machine learning model is trying to predict.

`.target` stores the class labels (output values) in the form of a NumPy array.

`.target_names` is an **attribute** in scikit-learn datasets that provides **human-readable names** for the class labels stored in `.target`.

Train_Test_Split

`train_test_split` is a way of **dividing a dataset into two parts**:

- **A training set**, which is used to teach the model how to recognise patterns.
- **A test set**, which is used to check how well the model performs on new, unseen data.

Function Call

x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

Function Arguments

- `iris.data` → Features (inputs).
- `iris.target` → Labels (outputs).
- `test_size=0.2` → Specifies that 20% of the data should be used for testing.

Function Outputs

- `x_train` → The **training features** (80% of `iris.data`).
- `x_test` → The **testing features** (20% of `iris.data`).
- `y_train` → The **training labels** (80% of `iris.target`).
- `y_test` → The **testing labels** (20% of `iris.target`).

Support Vector Machine (SVM) classifier

SVM (Support Vector Machine) is a **supervised machine learning algorithm** used for **classification and regression** tasks. It is particularly powerful for **binary classification problems** and works well in high-dimensional spaces.

```
clf = SVC()
clf.fit(x_train, y_train)
```

- `SVC()` stands for **Support Vector Classification** and is a function from the `sklearn.svm` (Support Vector Machine) module in **scikit-learn**. It is used to train an SVM model for **classification tasks**.
  - **SVC stands for**: **Support Vector Classifier**.
  - It **creates an SVM model** that learns to classify data into different categories.
  - It **finds the best decision boundary** (hyperplane) that separates different classes.

The `fit(x_train, y_train)` function **trains the model**, meaning it finds the **best hyperplane** that separates the classes.

- `x_train` **(Features/Input Data)** → The independent variables (what the model uses to make predictions).

- `y_train` **(Labels/Target Data)** → The dependent variable (the actual class labels the model is learning to predict).

Predict() Function

The `predict()` function is a method used in **machine learning models** (like SVM, Decision Trees, and Neural Networks) to **make predictions** based on **new input data**.

After training a model with `.fit(x_train, y_train)`, you use:

```
model.predict(x_test)
```

- Takes in **new data** (`x_test`) as input.
- Uses the **trained model** to make predictions.
- Returns a **NumPy array** of predicted class labels (for classification) or numerical values (for regression).
- Each value in the output corresponds to a prediction for a sample in `x_test`.
- The predicted labels are based on the **decision boundary** the model learned during training.

Confusion Matrix

A **confusion matrix** is a table used to **evaluate the performance of a classification model**. It shows how well the predicted labels (`y_pred`) match the actual labels (`y_test`).

For a **binary classification** problem (e.g., detecting spam: **Spam (1) vs. Not Spam (0)**), the confusion matrix looks like this:

	Predicted: 0	**Predicted: 1**
Actual: 0 (True Negative)	**TN** (Correctly predicted negatives)	**FP** (False Positives)
Actual: 1 (False Negative)	**FN** (False Negatives)	**TP** (Correctly predicted positives)

- **True Positive (TP)** → Model correctly predicted **positive** (e.g., predicted "Spam" when it's actually Spam).
- **True Negative (TN)** → Model correctly predicted **negative** (e.g., predicted "Not Spam" when it's actually Not Spam).
- **False Positive (FP)** → Model **incorrectly** predicted **positive** (e.g., predicted "Spam" when it's Not Spam → Type I Error).
- **False Negative (FN)** → Model **incorrectly** predicted **negative** (e.g., predicted "Not Spam" when it's actually Spam → Type II Error).

```
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
plt.show()
```

Score() Function

The `.score()` function in **scikit-learn** evaluates the performance of a trained model by returning a **single numeric score**.

```
clf.score(x_test, y_test)
```

- `clf` → A trained machine learning model (e.g., `SVC`, `RandomForestClassifier`, etc.).
- `x_test` → The test set's feature values (input data).
- `y_test` → The actual labels (true values).
- **Returns:** A **performance score** (accuracy for classifiers, R^2 score for regressors).

Hyperparameters in training

- Hyperparameters are settings that control how a machine learning model learns from data.
- They are not learned from the data; instead, they must be set before training.
- Different hyperparameter values can improve or degrade performance depending on the dataset.

Hyperparameters Vary Across Datasets

- The optimal hyperparameter values depend on the specific dataset.
- A value that works well for one dataset might not work for another.
- Tuning hyperparameters is necessary to maximize performance.

`C` is a Key Hyperparameter in `SVC`

- The code provided:

```
python
CopyEdit
clf2 = SVC(C=1)
clf2.fit(x_train, y_train)
clf2.score(x_test, y_test)
```

  - Uses `C=1`, which controls how much the model penalises misclassified points.
  - A lower `C` (e.g., `C=0.1`) results in a more flexible decision boundary (higher bias).
  - A higher `C` (e.g., `C=10`) results in a more strict decision boundary (lower bias, higher variance).

MLPClassifier

An **MLPClassifier (Multi-Layer Perceptron Classifier)** is a **neural network-based algorithm** in `scikit-learn` that is used for **classification problems**.

- It belongs to the `sklearn.neural_network` module.
- Uses **backpropagation and gradient descent** to learn from data.
- Suitable for **both linear and non-linear classification** problems.

`clf3 = MLPClassifier(hidden_layer_sizes=(10,), max_iter=400)`

- Creates an MLP model with the following settings:
  - **`hidden_layer_sizes=(10,)`** → Defines the **structure of the neural network**:
    - `(10,)` means **one hidden layer** with **10 neurons**.

        - You can add more layers: `(10, 5)` → two hidden layers (10 neurons & 5 neurons).
    - `max_iter=400` → Sets the **maximum number of iterations** (epochs) for training.
        - If the model doesn't converge, increasing this value may help.

`clf3.fit(x_train, y_train)` **(Training the Model)**

- Trains the **MLP neural network** using:
    - `x_train` → Feature inputs.
    - `y_train` → Target labels (output classes).
- The model **adjusts its weights** using **backpropagation** to minimize error.

Re-Substitution Method

- You are able to train a model using the re-substitution method
- The re-substitution validation method is **where you use all of your data as training data**. Then, you compare the error rate of the machine learning model's output to the actual value from the training data set.
- Unlike the **Holdout** method, which splits the data into a training set and a testing set, the accuracy is overly optimistic as the model is being tested on the same data it has been trained with

```
clf = SVC()
clf.fit(bc.data, bc.target)
clf.score(bc.data, bc.target)
```

Holdout Method

- This is considered the **traditional and most commonly used approach** for training and evaluating machine learning models.
- You split the dataset into training and testing, perhaps 80:20, where 80% of the dataset will be used to train the model, and then 20% of the data from the

dataset will be used to test the accuracy of the model once it has been trained

- You use the tran_test_split() function

Stratified training and testing splits

- Stratified sampling is **a type of sampling method that aims to preserve the proportions of different categories or classes in the original data**. For example, if you have a data set with 60% male and 40% female customers, you would want to maintain the same ratio in your sample.
- **Ensuring class distribution is preserved** → `stratify=bc.target` ensures the proportion of each class in the training and test sets matches the original dataset.
- **It first checks the class distribution in `bc.target` (the original dataset).**
- **It then performs the train-test split while maintaining the same class proportions** in both the training and test sets.
- **Each class is proportionally represented** in both sets, avoiding issues where one class is overrepresented in one set and underrepresented in the other.

```
x_train_str, x_test_str, y_train_str, y_test_str = train_test_split(
bc.data, bc.target, test_size=0.2, stratify=bc.target)
```

- You then use x_train_str etc as you would x_train in a holdout model

When you should/shouldn't use stratify

Cross Validation (CV)

- You should always use CV for evaluation of any models you build

Cross-validation is a **model evaluation technique** used in **machine learning and data mining** to assess how well a model generalizes to unseen data. Instead of

relying on a **single train-test split**, cross-validation **splits the dataset multiple times** to ensure a more **reliable and robust evaluation**.

Prevents Overfitting

→ Tests the model on multiple subsets to ensure it generalizes well.

More Reliable Accuracy

→ Provides more dependable results by using multiple test sets rather than just one.

Efficient Use of Data

→ Maximises available data as every sample serves for both training and testing.

cross_val_score()

- `cross_val_score()` is a **function in Scikit-Learn** used for **cross-validation**.
- It **automatically splits the dataset** multiple times and evaluates the model's performance.
- It **returns an array of accuracy scores** (or another metric) for each split.
- Used instead of train_test_split()

```
# Initialize SVM classifier
clf = SVC()

# Perform 10-Fold Cross-Validation
scores = cross_val_score(clf, bc.data, bc.target, cv=10)

# Print mean accuracy and standard deviation
print(scores.mean(), scores.std())
```

What Does `cv=10` Mean?

- * `cv=10` means **10-Fold Cross-Validation**.
- The dataset is **divided into 10 equal parts (folds)**.
- The model is **trained on 9 folds** and **tested on 1 fold**.

- This process **repeats 10 times**, using a different fold for testing each time.
- The function **returns 10 accuracy scores** (one for each fold).
- The final model accuracy is usually calculated as **the mean of these 10 scores**.

Leave One Out Cross Validation (LOO-CV)

- LOOCV is **a type of K-Fold Cross-Validation where K = total samples (`n`)**.
- Each sample is tested **one at a time**, using all other samples for training.
- **Pros:** Uses all data for training, provides **unbiased estimates**.
- **Cons:** Computationally expensive, can be **slow for large datasets**.

```
import numpy as np
from sklearn.svm import SVC
from sklearn.model_selection import KFold
from sklearn.datasets import load_breast_cancer

# Load the Breast Cancer dataset
bc = load_breast_cancer()

# Initialize KFold with n_splits equal to the total number of samples (LOO-CV)
kf = KFold(n_splits=bc.target.size)

# Create an empty array to store accuracy scores for each iteration
scores2 = np.zeros(bc.target.size)

# Initialize index counter for tracking iterations
i = 0

# Loop over each fold (each fold consists of leaving out one sample for testing)
for train_idx, test_idx in kf.split(bc.data):
```

```
    # Split data: Train on all but one sample, Test on the left-out sample
    x_train_kfold, x_test_kfold = bc.data[train_idx], bc.data[test_idx]
    y_train_kfold, y_test_kfold = bc.target[train_idx], bc.target[test_idx]

    # Initialize and train an SVM classifier
    clf4 = SVC()
    clf4.fit(x_train_kfold, y_train_kfold)

    # Evaluate the model on the left-out sample and store the accuracy
    scores2[i] = clf4.score(x_test_kfold, y_test_kfold)

    # Increment index counter
    i = i + 1

# Print the mean accuracy and standard deviation of the scores
print(scores2.mean(), scores2.std())
```

Code Explained

- **Load the dataset** (`bc.data` for features, `bc.target` for labels).
- **Create** `KFold` **with** `n_splits` **equal to dataset size** → This makes it **Leave-One-Out Cross-Validation**.
- **Initialise** `scores2` **array** to store accuracy scores for each iteration.
- **Loop through each fold**:
  - **Train on** `n-1` **samples**, leaving **1 sample as test data**.
  - **Train an SVM model** on the training data.
  - **Evaluate model performance** on the left-out test sample.
  - **Store the accuracy score** in `scores2[i]` .
- **Compute the mean and standard deviation of accuracy scores** to summarise performance.

np.genfromtxt()

A function in **NumPy** that is used to **load data from text files (like CSV, TSV, or TXT files)** into a NumPy array.

Syntax

```
numpy.genfromtxt(fname, delimiter=',', dtype=None, encoding='utf-8', skip_header=0)
```

Parameter	Description
`fname`	File name or path (string)
`delimiter`	Character that separates values (e.g., `,` for CSV, `\t` for TSV)
`dtype`	Data type of the output array (`None` auto-detects types)
`encoding`	Encoding type (`utf-8` for text files)
`skip_header`	Number of rows to skip (useful for headers)

Slicing

Slicing is used to select what columns of a dataset you'd like to work with

Slicing in NumPy **extracts parts of an array** using:

```
array[start:stop:step]
```

- `start` → Index where slicing starts (default `0`).
- `stop` → Index where slicing stops (exclusive).
- `step` → Interval between elements (default `1`).

```
hotels = hotels[:, 1:]
```

- `:` → **Selects all rows**.

- `1:` → **Selects columns starting from index** `1` **to the end** (removes the first column).

OrdinalEncoder()

- Linear regression models **require numerical data**—they **cannot** directly process categorical (string) values.
- `OrdinalEncoder` converts **categorical features (strings)** into **numerical values** so they can be used in regression models.

```
oec = OrdinalEncoder(categories='auto', dtype=np.float64)
oec.fit(hotels[:, [4]])
```

- `dtype` must be a **NumPy data type** (e.g., `np.float64` or `np.int64`).
- `float` alone is a **Python built-in type**, not a NumPy type.
- `oec.fit(X)` **learns the unique categorical values** from `X` (column 4 in this case).
- It **does NOT transform the data yet**—it just stores the mapping.
- After calling `.fit()`, the encoder **knows how to convert categories into numbers**.

```
oec.categories_
hotels[:, 4] = oec.transform(hotels[:, [4]]).flatten()
```

`oec.categories_`

- **Retrieves the unique categories** that `OrdinalEncoder` learned from `.fit()`.
- This shows **how categories are mapped to numerical values**.

`hotels[:, 4] = oec.transform(hotels[:, [4]]).flatten()`

- This **replaces the original categorical column** (`hotels[:, 4]`) with **numerical values**.
- `oec.transform(hotels[:, [4]])` **converts** categories into numbers.

- `.flatten()` **converts the 2D array to 1D**, making it compatible for assignment.

OneHotEncoder()

- `OneHotEncoder` is a **categorical encoding technique** in **Scikit-Learn** that converts categorical (string) values into **binary columns (0s and 1s)**.

Why Use One-Hot Encoding?

- Machine learning models **cannot process categorical data (strings)** directly.
- One-hot encoding **converts categorical values into numerical form** while **avoiding ordinal relationships**.
- Useful for **nominal categories** (e.g., "Red", "Blue", "Green") where there is **no order**.

Example Dataset (Before Encoding):

```
Color
------
Red
Blue
Green
Red
Green
```

After One-Hot Encoding:

```
Color_Red  Color_Blue  Color_Green
   1         0          0
   0         1          0
   0         0          1
   1         0          0
   0         0          1
```

- **Best for nominal categorical variables** where there is **no meaningful order**.

- **Creates more features** than ordinal encoding but **prevents incorrect numerical relationships**.

Syntax

```
# Initialize OneHotEncoder:
# categories='auto': Automatically detects unique categories.
# dtype=np.float64: Sets the output data type to float.
# sparse=False: Returns a dense NumPy array (not a sparse matrix).
ohc = OneHotEncoder(categories='auto', dtype=np.float64, sparse=False)

# Fit the encoder on the data and transform the data into one-hot encoded forma
encoded_data = ohc.fit_transform(data)
```

Building a Linear Regression Model

Linear Regression is a fundamental **supervised learning algorithm** used for **predicting numerical values** based on input features. It establishes a **linear relationship** between one or more independent variables (**features**) and a dependent variable (**target**).

When to use Linear Regression

- Predicting continuous values
    - (e.g., house prices, salaries, sales revenue).
- Understanding relationships between variables
    - (e.g., how advertising spend affects sales).

Feature selection

- (analysing the impact of variables on predictions).

Syntax

```
lr = LinearRegression()
lr.fit(x_train, y_train)
```

`lr = LinearRegression()`

- **Creates an instance** of the `LinearRegression` class from Scikit-Learn.
- **Initialises** the model with default parameters (unless specified otherwise).
- **Assigns** the new LinearRegression object to the variable `lr`.

`lr.fit(x_train, y_train)`

- **Calls the `fit()` method** on the LinearRegression instance `lr`.
- **Trains the model** using the provided training data:
  - `x_train`: The feature matrix containing the independent variables.
  - `y_train`: The target vector containing the dependent variable.

Calculates the regression coefficients (e.g., slope and intercept) that best fit the training data using a least squares approach.

np.unique()

`np.unique()` is a function in **NumPy** that finds **all unique values** in an array. It is useful for **removing duplicates** and analysing categorical data.

Syntax

```
np.unique(hr[:, -1])
```

- `hr[:, -1]` Selects the class column of the array
- `:` selects all rows
- `-1` selects the last column (negative indexing in NumPy)
- `np.unique(hr[:, -1])` → Finds unique values in that column
  - Removes duplicates.
  - Sorts the values (default behaviour).

When to use

You would typically use `np.unique()` when working with raw NumPy arrays, especially if the data is not downloaded using Scikit-Learn.

r_regression

`r_regression` is a feature selection method in Scikit-Learn that calculates the correlation between each feature and the target variable in a regression problem.

It measures the strength and direction of the linear relationship between each independent variable (feature) and the dependent variable (target) using the Pearson correlation coefficient.

```
hotel_pr = r_regression(hr[:, :-1], hr[:, -1])
```

`hr[:, :-1]` → Selects all columns except the last one (features)

- `:` → Selects **all rows**.
- `:-1` → Selects **all columns except the last one** (assumed to be the target variable).
- This is the **feature matrix (`X`)**.

`hr[:, -1]` → Selects only the last column (target variable)

- `1` → Selects the **last column**.
- This is the **dependent variable (`y`)** that we want to predict.

`r_regression(X, y)` → Computes the **correlation** between each feature and the target variable.

- Returns a NumPy array of **Pearson correlation coefficients (`r`)** for each feature.
- Higher values ($|r| \to 1$) mean **strong correlation**, lower values ($|r| \approx 0$) mean **weak correlation**.

np.argsort()

`np.argsort()` is a **sorting function** in NumPy that **returns the indices** that would **sort an array in ascending order**.

Instead of sorting the actual values, it provides the **index positions** of the sorted elements.

```
imp_feature_index = np.argsort(np.abs(hotel_pr))
```

1️⃣ `np.abs(hotel_pr)` → **Takes the absolute values of** `hotel_pr`

- `hotel_pr` is assumed to be **an array of correlation values** obtained from `r_regression()`.
- Correlation values **range from -1 to 1**, but we are only interested in their **magnitude** (strength of relationship), so we apply `np.abs()`.
- This ensures that **negative correlations are treated the same as positive correlations**.

2️⃣ `np.argsort(np.abs(hotel_pr))` → **Returns the indices that would sort these absolute values in ascending order**

- `argsort()` returns **the indices of the smallest to largest absolute correlation values**.
- The features with **the weakest correlation to the target** will be at the beginning, and those with the **strongest correlation** will be at the end.

3️⃣ `imp_feature_index` **stores these indices**

- This is a **ranking of feature importance based on correlation strength**.

SelectKBest

```
skb = SelectKBest(lambda x, y: np.abs(r_regression(x, y)), k=6)
hr2 = skb.fit_transform(hr[:, :-1], hr[:, -1])
```

1️⃣ SelectKBest(lambda x, y: np.abs(r_regression(x, y)), k=6)

- `SelectKBest` is a **feature selection method** in Scikit-Learn.
- It selects **the top** `k` **features** based on a scoring function.

- The scoring function is defined using a **lambda function**:

  ```
  lambda x, y: np.abs(r_regression(x, y))
  ```

  - `r_regression(x, y)` → Computes the **correlation** between each feature in `x` and the target `y`.
  - `np.abs(...)` → Takes the **absolute values** of the correlation scores to ensure both positive and negative correlations are considered.
- `k=6` → Selects the **top 6 most correlated features**.

2️⃣ `hr2 = skb.fit_transform(hr[:, :-1], hr[:, -1])`

- `hr[:, :-1]` → **Selects all feature columns (excluding the last column).**
- `hr[:, -1]` → **Selects the last column (target variable).**
- `fit_transform()`:
  - **Computes correlation scores** for each feature.
  - **Selects the top 6 features** with the highest absolute correlation.
  - **Transforms `hr` by keeping only these top features**.
- `hr2` now contains only the **6 most relevant features**.

When Would You Use This?

✅ **Feature Selection Before Regression** → Keep only the most relevant features for training.

✅ **Dimensionality Reduction** → Improve model efficiency by removing weakly correlated features.

✅ **Handling Multicollinearity** → Helps eliminate redundant features.

MinMaxScaler

`MinMaxScaler()` is a **feature scaling technique** in Scikit-Learn that **normalises data** to a fixed range. Typically this is between 0 and 1 but can be adjusted.

- **Ensures all features are on the same scale**, preventing some features from dominating others.
- **Improves model performance**, especially for distance-based models like **KNN, SVM, and Neural Networks**
- **Prevents large feature values from causing numerical instability in optimisation algorithms**

Formula:

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}.$$

Where:

- X = Original feature value
- X min = Minimum value in the feature
- X max = Maximum value in the feature
- X scaled = Scaled value between 0 and 1

When to use:

- **When features have different scales** (e.g., age in years vs. salary in dollars).
- **For ML models sensitive to feature scaling**, such as:
  - **Distance-based algorithms** (KNN, K-Means, SVM, PCA).
  - **Gradient-based models** (Neural Networks, Logistic Regression).

When not to use:

- If the dataset has **outliers**, because `MinMaxScaler()` is **sensitive to min/max values**.