Python Notes

**Lambda** functions in Python are small, anonymous functions defined with the `lambda` keyword. They are useful when you need a simple, one-off function without the overhead of defining it using the `def` keyword. Lambdas are commonly used for short, inline functions where defining a full function would be unnecessary.

### Basic Syntax of Lambda

```python

lambda arguments: expression

```

Here's an example where we might use a lambda instead of a regular function:

```python

# Using lambda for a quick, one-off function

add = lambda x, y: x + y

print(add(3, 5)) # Output: 8

```

### Common Use Cases for Lambda Functions

1. \*\*As arguments to higher-order functions\*\*: They’re often used as arguments to functions like `map`, `filter`, and `sorted`, where you need a short function for a single operation.

```python

# Using lambda with map

numbers = [1, 2, 3, 4]

squares = list(map(lambda x: x \*\* 2, numbers))

print(squares) # Output: [1, 4, 9, 16]

```

2. \*\*For sorting or custom sorting\*\*: Lambdas are also handy for defining small sorting rules directly in a `sorted` call.

```python

# Sorting a list of tuples based on the second element

data = [(1, 'b'), (3, 'a'), (2, 'c')]

sorted\_data = sorted(data, key=lambda x: x[1])

print(sorted\_data) # Output: [(3, 'a'), (1, 'b'), (2, 'c')]

```

3. \*\*In place of simple functions\*\*: Sometimes, defining an entire function with `def` for a short operation can be excessive, so `lambda` provides a quick alternative.

### When to Use `lambda` vs. `def`

- \*\*Use `lambda`\*\* for simple, short functions, especially in places where the function will only be used once.

- \*\*Use `def`\*\* for more complex functions or when reusability and readability are important.

Lambdas are a quick and efficient way to write concise code, but they should be used carefully as they can make the code less readable if overused.

**Apply** method is often used with `lambda` functions in Python, especially with the Pandas library, to perform operations on each element in a Series or each row/column in a DataFrame.

Here’s an example of how to use `apply` with `lambda` on a Pandas DataFrame or Series:

### 1. Applying a Lambda Function to a Series

Suppose you have a Series called `ids` and want to perform an operation on each element. For instance, let’s say you want to add a prefix, like `"ID\_"`, to each element.

```python

import pandas as pd

# Sample Series

ids = pd.Series([101, 102, 103, 104])

# Applying a lambda function to add "ID\_" prefix

ids\_with\_prefix = ids.apply(lambda x: f"ID\_{x}")

print(ids\_with\_prefix)

```

### Output:

```

0 ID\_101

1 ID\_102

2 ID\_103

3 ID\_104

dtype: object

```

### 2. Applying a Lambda Function to a DataFrame Column

If you have a DataFrame and want to apply a lambda function to one of its columns, you can do this by specifying the column.

```python

# Sample DataFrame

data = pd.DataFrame({

'ids': [101, 102, 103, 104],

'names': ['Alice', 'Bob', 'Charlie', 'David']

})

# Adding a prefix to the 'ids' column

data['ids\_with\_prefix'] = data['ids'].apply(lambda x: f"ID\_{x}")

print(data)

```

### Output:

```

ids names ids\_with\_prefix

0 101 Alice ID\_101

1 102 Bob ID\_102

2 103 Charlie ID\_103

3 104 David ID\_104

```

### 3. Applying a Lambda Function to Rows in a DataFrame

If you want to apply a lambda function across rows in a DataFrame, set the `axis=1` parameter. For example, let’s concatenate the `ids` and `names` columns.

```python

# Concatenating 'ids' and 'names' into a single column

data['id\_name'] = data.apply(lambda row: f"{row['ids']}\_{row['names']}", axis=1)

print(data)

```

### Output:

```

ids names ids\_with\_prefix id\_name

0 101 Alice ID\_101 101\_Alice

1 102 Bob ID\_102 102\_Bob

2 103 Charlie ID\_103 103\_Charlie

3 104 David ID\_104 104\_David

```

Using `lambda` with `apply` gives you great flexibility to create custom transformations in a concise way!

**Pickle**

```python

import pickle

# Fetch the data

performance\_data = fetch\_csv() # Ensure this function is defined and returns your data

# Initialize the dictionary and store your data

my\_datasets = {

"performance\_data": performance\_data

}

# Save my\_datasets to a file using pickle

with open('performance\_data.pkl', 'wb') as studydata\_cache:

pickle.dump(my\_datasets, studydata\_cache)

print("Data saved successfully!")

```

### Key Points

- The `my\_datasets` dictionary now contains the `performance\_data` under the key `"performance\_data"`.

- This will allow you to save the actual data you're working with.

### Loading the Data Back

To load the data back later, you can use the following code:

```python

with open('performance\_data.pkl', 'rb') as studydata\_cache:

loaded\_data = pickle.load(studydata\_cache)

# Access the DataFrame

performance\_data = loaded\_data["performance\_data"]

print(performance\_data)

```

This way, you'll be able to save and load your `performance\_data` correctly using `pickle`.

**Pandas cut() function is used to separate the array elements into different bins . The cut function is mainly used to perform statistical analysis on scalar data.**

**import** pandas as pd

**import** numpy as np

df **=** pd.DataFrame({'number': np.random.randint(1, 100, 10)})

df['bins'] **=** pd.cut(x**=**df['number'], bins**=**[1, 20, 40, 60, 80, 100],

                    labels**=**['1 to 20', '21 to 40', '41 to 60',

                            '61 to 80', '81 to 100'])

print(df)

# We can check the frequency of each bin

print(df['bins'].unique())

**CHAPTER 2**

From Chat

The goal is to ensure that the test set remains the same across different runs of your machine learning model, even if the dataset changes (e.g., by adding new data).

Here’s a more detailed breakdown:

### Problem Context

When you're training a machine learning model, you typically split your data into a \*\*training set\*\* and a \*\*test set\*\*. The training set is used to train the model, while the test set is used to evaluate its performance. It’s important that the test set is kept separate from the training data to get an unbiased estimate of the model’s performance.

### Challenge

If you randomly split the data into a training set and a test set every time you run the model, the test set might include different instances each time. This inconsistency can make it hard to compare results across different model runs, especially if you refresh or update the dataset.

### Proposed Solution

To address this, the solution involves using each instance’s unique identifier (e.g., a customer ID, product ID, etc.) to decide whether it should be in the test set or the training set. Here’s how it works:

1. \*\*Unique Identifier\*\*: Each instance in your dataset must have a unique and unchanging identifier.

2. \*\*Hash Function\*\*: Compute a hash of this identifier. A hash function takes an input (in this case, the identifier) and produces a fixed-size string of bytes, which typically appears as a seemingly random string of numbers and letters. Importantly, the same input always produces the same hash.

3. \*\*Test Set Inclusion\*\*: Compare the hash value to a threshold. The threshold is determined based on the percentage of data you want in the test set. For instance, if you want 20% of your data in the test set, you could compare the hash value to 20% of the maximum possible hash value.

- If the hash value of an instance's identifier is less than or equal to 20% of the maximum possible hash value, that instance goes into the test set.

- Otherwise, it goes into the training set.

### Why This Works

- \*\*Consistency\*\*: Since the hash function always produces the same output for the same input, the same instances will end up in the test set each time you split the data, even if the dataset is refreshed or new instances are added. This ensures the test set is consistent across runs.

- \*\*Proportional Distribution\*\*: By setting the threshold based on a percentage (e.g., 20%), you ensure that the test set contains approximately 20% of the instances, even as new data is added.

In summary, this method creates a consistent and stable test set by using a hash of each instance's unique identifier, making it ideal for situations where the dataset might change over time.

In the provided code, the hash function is the `crc32` function used within the `test\_set\_check` function. Here's how it works:

**Code Breakdown**

1. \*\*`crc32` Function\*\*:

- The `crc32` function is a cyclic redundancy check (CRC) hash function from Python’s `zlib` module. It takes an input (in this case, the identifier) and produces a 32-bit hash value.

- The `np.int64(identifier)` ensures that the identifier is treated as a 64-bit integer before being hashed.

2. \*\*Bitwise AND Operation\*\*:

- `& 0xffffffff` is a bitwise operation that ensures the hash value is treated as an unsigned 32-bit integer. This is a safeguard to ensure the value is within the correct range.

3. \*\*Test Set Check\*\*:

- The condition `crc32(np.int64(identifier)) & 0xffffffff < test\_ratio \* 2\*\*32` compares the hash value to the threshold determined by `test\_ratio \* 2\*\*32`.

- `test\_ratio \* 2\*\*32` calculates the threshold value based on the desired test set size (e.g., if `test\_ratio` is 0.2, this represents 20% of the maximum possible 32-bit value).

### Summary

In this code:

- \*\*Hash Function\*\*: `crc32` is the hash function, generating a 32-bit hash value from the identifier.

- \*\*Purpose\*\*: The hash value is used to decide whether an instance (identified by its ID) belongs in the test set or not.

The `split\_train\_test\_by\_id` function then likely uses `test\_set\_check` to separate the dataset into training and test sets based on the hash value of each instance's identifier.

Let’s clarify the concepts of maximum and minimum hash values and how they relate to the 20% threshold:

**Hash Values**

1. \*\*What is a Hash Value?\*\*

- A \*\*hash function\*\* converts data (like a string or number) into a fixed-size value, typically a number. For instance, the `crc32` function produces a 32-bit integer, meaning the output (hash value) can range from 0 to `2^32 - 1` (which is 4,294,967,295).

- \*\*Minimum Hash Value\*\*: The smallest possible output of the hash function, which is `0`.

- \*\*Maximum Hash Value\*\*: The largest possible output of the hash function, which is `2^32 - 1` (or 4,294,967,295).

### Threshold Calculation

2. \*\*Threshold for Test Set Inclusion\*\*

- To include approximately 20% of your data in the test set, you need to determine which hash values fall into the "first 20%" of all possible hash values.

- Since the hash values range from `0` to `4,294,967,295`, 20% of this range corresponds to hash values from `0` to `0.2 \* 4,294,967,295`, which is `858,993,459`.

### The `test\_set\_check` Function

3. \*\*How Does It Work?\*\*

- The function checks whether the hash value of an identifier is within this "first 20%" range. If it is, the corresponding instance is placed in the test set.

- Specifically, the condition `crc32(np.int64(identifier)) & 0xffffffff < test\_ratio \* 2\*\*32` checks if the hash value is less than or equal to `858,993,459` when `test\_ratio` is 0.2.

### Putting It All Together

- \*\*Maximum Hash Value\*\*: `2^32 - 1` (4,294,967,295) is the largest possible output of the `crc32` function.

- \*\*Minimum Hash Value\*\*: `0` is the smallest possible output.

- \*\*20% Threshold\*\*: If you want 20% of the data in the test set, you look at hash values from `0` to `858,993,459` (which is 20% of the maximum hash value).

Thus, for an instance to be in the test set, its hash value (computed by `crc32`) must be in the lower 20% of the possible hash values.

**Understanding Hashing and Data Set Size**

1. \*\*Hashing and Data Distribution\*\*:

- The purpose of hashing is to distribute data across the entire range of possible hash values. The hash function is designed to produce a fairly uniform distribution of hash values, regardless of the actual size of your dataset.

- Even if your dataset is small, the hash function will still map each identifier to a value somewhere in the full range of possible hash values (from `0` to `2^32 - 1` in the case of a 32-bit hash like `crc32`).

2. \*\*Why It Works Even with a Small Dataset\*\*:

- \*\*Uniform Distribution\*\*: The hash values generated for your identifiers will be spread across the entire range, not just clustered in a small portion. This means that, proportionally, you can still divide the hash space into segments (like the first 20%) and expect to capture roughly the desired proportion of your data (e.g., 20%) in the test set.

- \*\*Ratio Calculation\*\*: The ratio (like 20%) is based on the range of possible hash values, not the size of your dataset. Since hash values are uniformly distributed, selecting a portion of this range (e.g., the first 20%) will result in approximately the same proportion of instances, regardless of how many instances you have.

### Example with a Small Dataset

Let’s say you have only 100 instances in your dataset, but the hash function can produce over 4 billion possible hash values.

- \*\*Uniform Spread\*\*: Even though you only have 100 instances, their hash values will still be spread out over the entire range from `0` to `2^32 - 1`.

- \*\*20% Selection\*\*: If you select hash values that are in the lower 20% of this range (from `0` to `858,993,459`), you should end up with approximately 20% of your instances in the test set. That’s because the selection is based on the hash values, not on the total number of instances.

### Conclusion

The method works because it leverages the properties of the hash function to ensure that the test set represents a consistent proportion of the data, regardless of the dataset size. Even with a small dataset, the hash function's uniform distribution across a large range of possible values allows the test ratio to be correctly applied.

Now that you've provided more context with the CSV file, let’s clarify how the `~` operator works in this specific scenario.

**Code Overview on ~:**

1. \*\*`test\_set\_check` function\*\*:

- This function takes an identifier and a test ratio (e.g., 0.2 for 20% test set).

- It uses the `crc32` hash function to compute a hash value for the identifier.

- It checks whether the hash value is less than the threshold based on the `test\_ratio`.

2. \*\*`split\_train\_test\_by\_id` function\*\*:

- This function splits the dataset into a training set and a test set based on the identifiers in a specified `id\_column`.

- The function computes a boolean array `in\_test\_set`, where each element is `True` if that row should go into the test set (based on the `test\_set\_check` function).

- The `apply` method applies the `test\_set\_check` function to each identifier in the `id\_column`.

### The Role of `~` in `data.loc[~in\_test\_set]`

Let’s focus on how the data is split into training and test sets:

- \*\*`in\_test\_set`\*\*: After running the `apply` function, `in\_test\_set` will be a \*\*boolean Series\*\* that corresponds to the rows in the dataset. It will contain `True` for rows that should go into the test set, and `False` for rows that should go into the training set.

- \*\*`data.loc[in\_test\_set]`\*\*: This selects the rows where `in\_test\_set` is `True`, i.e., the rows that belong to the \*\*test set\*\*.

- \*\*`~in\_test\_set`\*\*: This is the \*\*bitwise negation\*\* or \*\*inversion\*\* of the `in\_test\_set` boolean array.

- Where `in\_test\_set` is `True`, `~in\_test\_set` will be `False`.

- Where `in\_test\_set` is `False`, `~in\_test\_set` will be `True`.

So, \*\*`data.loc[~in\_test\_set]`\*\* selects the rows where `in\_test\_set` is `False`, which corresponds to the rows that will go into the \*\*training set\*\*.

### The Overall Purpose

- `data.loc[~in\_test\_set]`: Returns the rows \*\*not in the test set\*\* (i.e., the training set).

- `data.loc[in\_test\_set]`: Returns the rows \*\*in the test set\*\*.

By using `~`, the code efficiently inverts the selection, allowing you to split the dataset into training and test sets based on whether each row’s identifier meets the condition for inclusion in the test set.

**How It Works With a CSV File**

When you load a CSV file into a pandas DataFrame, the `split\_train\_test\_by\_id` function operates on the column specified by `id\_column`. The boolean mask (`in\_test\_set`) tells pandas which rows belong in the test set, and the `~` operator is used to select the rows that belong in the training set.

The line `housing\_with\_id["id"] = housing["longitude"] \* 1000 + housing["latitude"]` is creating a new column in the `housing\_with\_id` DataFrame called `id`. This new column is intended to serve as a unique identifier for each row in the dataset. Here's a breakdown of what it's doing:

### Explanation:

1. \*\*Latitude and Longitude\*\*:

- `housing["longitude"]` and `housing["latitude"]` refer to the longitude and latitude columns in the `housing` DataFrame. These columns represent the geographical coordinates of each housing district.

2. \*\*Multiplication by 1000\*\*:

- The `longitude` values are multiplied by 1000. This is done to give the longitude values a different scale compared to the latitude values, which helps in ensuring that the combination of `longitude` and `latitude` creates a unique identifier.

- By multiplying `longitude` by 1000, you effectively shift the decimal point three places to the right. This makes the `longitude` values larger and helps avoid potential overlap when added to the `latitude` values.

3. \*\*Adding Latitude\*\*:

- The `latitude` value is then added to the scaled `longitude` value. The resulting sum is a unique number that is used as an ID for each row in the dataset.

### Purpose:

The idea behind this is to create a relatively stable and unique identifier for each row based on geographical location. Since latitude and longitude are stable properties (they don't change over time), this approach ensures that each row in the dataset can be uniquely identified, even if new data is added or some rows are removed in the future.

### Why Do This?

- \*\*Consistency in Splitting Data\*\*: When splitting the data into training and test sets, you want to ensure that the same rows consistently end up in the same set across different runs or when new data is added. By using a stable ID, the data can be split deterministically, avoiding the issue where the same data might end up in different sets after the dataset is modified.

This approach is particularly useful in situations where the row index is not stable or might change, making it unreliable as an identifier.

**Loading the Data**

Sure! Let's break down the entire code snippet step by step, so you can fully understand what each part is doing and why.

housing = load\_housing\_data(Path.cwd())

```

- \*\*`load\_housing\_data(Path.cwd())`\*\*: This line calls a function `load\_housing\_data` which is presumably defined somewhere else in your code. The function is likely responsible for loading a dataset (probably a CSV file or similar) that contains housing data.

- \*\*`Path.cwd()`\*\*: This is a method from the `pathlib` module that returns the current working directory. This is passed to `load\_housing\_data` to indicate where the data should be loaded from.

### Step 2: Resetting the Index

```python

housing\_with\_id = housing.reset\_index() # adds an `index` column

```

- \*\*`housing.reset\_index()`\*\*: This method resets the index of the DataFrame `housing`. By default, when you load a dataset into a DataFrame, it has an implicit index (usually starting from 0). `reset\_index()` turns this index into a regular column, adding it to the DataFrame as a new column named `index`.

- \*\*`housing\_with\_id`\*\*: This is the new DataFrame that now includes the original data plus the new `index` column.

### Step 3: Splitting the Dataset

```python

train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "index")

```

- \*\*`split\_train\_test\_by\_id(housing\_with\_id, 0.2, "index")`\*\*: This is a function call that splits the dataset into two parts: a training set and a test set.

- \*\*Parameters\*\*:

- \*\*`housing\_with\_id`\*\*: The DataFrame you want to split.

- \*\*`0.2`\*\*: This indicates that 20% of the data should go into the test set, and the remaining 80% into the training set.

- \*\*`"index"`\*\*: This specifies that the split should be done based on the `index` column, which was created when you reset the index.

- \*\*`train\_set` and `test\_set`\*\*: These are the resulting DataFrames after the split, where `train\_set` contains 80% of the data and `test\_set` contains 20%.

### Important Note on Row Index Stability

- The comment in your code highlights a potential issue with using the `index` as a unique identifier: \*\*If new data is added or existing data is deleted, the row indices might change\*\*, leading to inconsistency in your train/test splits. This is why you need to ensure that the dataset is stable if you rely on row indices.

### Step 4: Creating a Stable Unique Identifier

```python

housing\_with\_id["id"] = housing["longitude"] \* 1000 + housing["latitude"]

```

- \*\*`housing\_with\_id["id"]`\*\*: This creates a new column in the `housing\_with\_id` DataFrame called `id`.

- \*\*`housing["longitude"] \* 1000 + housing["latitude"]`\*\*: This operation creates a unique identifier by combining the longitude and latitude of each housing entry.

- \*\*`housing["longitude"] \* 1000`\*\*: Multiplies the longitude by 1000 to shift the decimal point and give the longitude a larger scale.

- \*\*`+ housing["latitude"]`\*\*: Adds the latitude to the scaled longitude value. The result is a unique ID for each row based on its geographical coordinates.

### Step 5: Splitting the Dataset with a Stable ID

```python

train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "id")

```

- After creating the stable `id`, you split the dataset again, this time using the `id` column instead of the `index`.

- \*\*`"id"`\*\*: This ensures that even if the dataset changes in the future (e.g., new rows are added or existing rows are removed), the train/test split remains consistent because the `id` is based on stable geographical coordinates.

### Summary:

- \*\*Load the Data\*\*: Get the housing data into a DataFrame.

- \*\*Reset Index\*\*: Convert the DataFrame's index into a regular column for potential use in splitting the data.

- \*\*Initial Split\*\*: Split the data into train and test sets using the index. However, this method has limitations if the dataset changes.

- \*\*Create a Stable ID\*\*: Use geographical coordinates to create a stable, unique identifier for each row.

- \*\*Final Split\*\*: Split the data using the stable `id` to ensure consistent results even if the dataset changes in the future.

This process ensures that your train/test splits are consistent and reliable, especially when working with data that might change over time.

Let's break down what this code does, step by step:

### Step 1: Creating Income Categories

```python

housing["income\_cat"] = pd.cut(housing["median\_income"],

bins=[0., 1.5, 3.0, 4.5, 6., np.inf],

labels=[1, 2, 3, 4, 5])

```

- \*\*`pd.cut()`\*\*: This function is used to segment and sort data values into discrete bins. It essentially categorizes a continuous variable (in this case, `median\_income`) into a categorical variable with specified intervals (bins).

- \*\*Parameters\*\*:

- \*\*`housing["median\_income"]`\*\*: This is the column in the DataFrame that contains the median income of households. It's a continuous variable.

- \*\*`bins=[0., 1.5, 3.0, 4.5, 6., np.inf]`\*\*: These are the boundaries of the bins. The `median\_income` values are divided into intervals:

- `0. to 1.5`: Represents very low income.

- `1.5 to 3.0`: Represents low income.

- `3.0 to 4.5`: Represents moderate income.

- `4.5 to 6.0`: Represents high income.

- `6.0 to infinity (`np.inf`)`: Represents very high income.

- \*\*`labels=[1, 2, 3, 4, 5]`\*\*: These are the labels assigned to the bins. Each bin gets a label that corresponds to a category (1 for very low income, 5 for very high income, etc.).

- \*\*Result\*\*: A new column, `income\_cat`, is added to the `housing` DataFrame. Each entry in this column corresponds to one of the five income categories based on the `median\_income`.

**Stratified Shuffle Split**

```python

from sklearn.model\_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n\_splits=1, test\_size=0.2, random\_state=42)

```

- \*\*`StratifiedShuffleSplit`\*\*: This is a utility from the `sklearn` library that splits the data into training and testing sets in such a way that each set maintains the same proportion of samples for each category (in this case, income categories).

- \*\*Parameters\*\*:

- \*\*`n\_splits=1`\*\*: Specifies the number of re-shuffling and splitting iterations. Here, it's set to 1, meaning the data will be split just once.

- \*\*`test\_size=0.2`\*\*: Specifies that 20% of the data should go into the test set, while the remaining 80% goes into the training set.

- \*\*`random\_state=42`\*\*: Ensures that the split is reproducible. The same data will be selected for training and testing if you run the code again, as long as the `random\_state` is fixed.

### Step 3: Applying the Split

```python

for train\_index, test\_index in split.split(housing, housing["income\_cat"]):

strat\_train\_set = housing.loc[train\_index]

strat\_test\_set = housing.loc[test\_index]

```

- \*\*`split.split(housing, housing["income\_cat"])`\*\*: This method splits the `housing` DataFrame into training and testing sets based on the `income\_cat` column. It returns the indices of the samples for the training and testing sets.

- \*\*For Loop\*\*:

- \*\*`train\_index, test\_index`\*\*: These variables hold the indices for the training and testing sets, respectively.

- \*\*`strat\_train\_set = housing.loc[train\_index]`\*\*: This selects the rows corresponding to `train\_index` from the `housing` DataFrame, creating the stratified training set.

- \*\*`strat\_test\_set = housing.loc[test\_index]`\*\*: Similarly, this selects the rows corresponding to `test\_index`, creating the stratified test set.

### Purpose of This Code:

- The main goal is to split the `housing` data into training and testing sets while ensuring that the distribution of the `income\_cat` variable is consistent across both sets. This is important because it prevents any skew in income distribution between the training and test sets, leading to more reliable and generalizable models.

This process of stratification based on income categories helps preserve the representativeness of the data in both the training and test sets.

The error you're encountering, `ValueError: could not convert string to float: 'INLAND'`, indicates that there's a non-numeric column in your `housing` DataFrame that the `corr()` function is trying to process. The `corr()` function calculates the Pearson correlation coefficient between numeric columns, so it can't handle string or categorical data.

### How to Fix This

You need to make sure that your DataFrame only contains numeric columns when you call the `corr()` function. Here’s how you can do it:

1. \*\*Select Only Numeric Columns\*\*: Before calculating the correlation matrix, filter out any non-numeric columns.

```python

housing\_numeric = housing.select\_dtypes(include=[np.number])

corr\_matrix = housing\_numeric.corr()

```

This code will create a new DataFrame `housing\_numeric` that only includes columns with numeric data types.

2. \*\*Proceed with the Correlation Calculation\*\*:

```python

corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

```

This approach ensures that only numeric data is passed to the `corr()` function, which should resolve the error.

### Explanation of the Issue:

- The column `'INLAND'` mentioned in the error message likely belongs to a categorical or string column in your DataFrame. When the `corr()` function tried to process this column, it encountered an error because it couldn't convert the string values to numbers.

- By selecting only numeric columns, you avoid this issue, and the correlation matrix is computed correctly.

These lines of code present three different methods for handling missing data in the `housing` DataFrame, specifically for the `total\_bedrooms` column. Let's go through each method to understand what it does and when you might use it.

### Option 1: Dropping Rows with Missing Values

```python

housing.dropna(subset=["total\_bedrooms"])

```

- \*\*What it does\*\*: This method drops all rows from the `housing` DataFrame where the `total\_bedrooms` column has missing values (`NaN`).

- \*\*When to use\*\*: Use this approach if you have a small amount of missing data and can afford to lose some rows without significantly affecting your analysis. It's the simplest way to handle missing values but can result in loss of data.

### Option 2: Dropping the Entire Column

```python

housing.drop("total\_bedrooms", axis=1)

```

- \*\*What it does\*\*: This method drops the entire `total\_bedrooms` column from the DataFrame.

- \*\*When to use\*\*: Use this approach if the `total\_bedrooms` column is not essential for your analysis. Dropping the column eliminates the missing data issue altogether but at the cost of losing potentially valuable information.

### Option 3: Filling Missing Values with the Median

```python

median = housing["total\_bedrooms"].median()

housing["total\_bedrooms"].fillna(median, inplace=True)

```

- \*\*What it does\*\*:

- \*\*Step 1\*\*: Calculates the median of the `total\_bedrooms` column.

- \*\*Step 2\*\*: Fills all missing values (`NaN`) in the `total\_bedrooms` column with this median value. The `inplace=True` argument modifies the DataFrame directly without needing to create a new one.

- \*\*When to use\*\*: Use this approach when you don't want to lose any data. Filling missing values with the median is a common technique that preserves the overall distribution of the data while replacing the missing values with a central tendency measure (median), which is robust to outliers.

### Summary:

- \*\*Option 1\*\* (`dropna`): Removes rows with missing values, which might be suitable for small datasets with few missing entries.

- \*\*Option 2\*\* (`drop`): Removes the entire column with missing values, which is useful if the column isn't crucial to your analysis.

- \*\*Option 3\*\* (`fillna` with median): Replaces missing values with the median, preserving all rows and mitigating the impact of missing data on the analysis.

Each method has its advantages and is useful in different scenarios depending on how critical the `total\_bedrooms` data is to your analysis and how much missing data you have.

**Why do we cut the data into bins with pd.cut before applying straftifedshuflesplit**

When working with data splitting for stratified sampling (like using StratifiedShuffleSplit in scikit-learn), using pd.cut to create bins can help maintain the distribution of target classes or continuous values across the training and testing datasets.

Here's a breakdown of why this is often done:

Handling Continuous Variables: Stratified sampling typically works best with categorical data. If your target variable is continuous (e.g., house prices, income), you can convert it into categorical "bins" that represent ranges. This allows StratifiedShuffleSplit to split the data while preserving the distribution of the target variable across bins.

Better Distribution Balance: By binning continuous data, you ensure that each split has a balanced distribution of the target variable, maintaining consistency in how different ranges are represented in both training and test sets. Without binning, you might get an uneven split, which can introduce bias and reduce the representativeness of the sample.

Improving Model Generalization: Keeping a balanced distribution in both sets helps the model learn from a representative subset of data, increasing the likelihood that it generalizes well to unseen data.

Here's a simple example to illustrate this:

python

Copy code

import pandas as pdfrom sklearn.model\_selection import StratifiedShuffleSplit

# Example data

data = pd.DataFrame({'feature': range(100), 'target': range(100)})

# Cut target variable into bins for stratification

data['target\_binned'] = pd.cut(data['target'], bins=5, labels=False)

# Initialize StratifiedShuffleSplit with binned target variable

split = StratifiedShuffleSplit(n\_splits=1, test\_size=0.2, random\_state=42)

for train\_idx, test\_idx in split.split(data, data['target\_binned']):

train\_set = data.loc[train\_idx]

test\_set = data.loc[test\_idx]

In this example, pd.cut bins the continuous target variable into 5 categories, enabling stratified sampling. After splitting, you can drop the target\_binned column.

**loc and iloc in python**

The role of defining a custom \*\*index\*\* in a Pandas DataFrame, such as in this line:

```python

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

```

### Purpose of Defining a Custom Index:

In the code above, you're explicitly setting row labels (`'row1'`, `'row2'`, `'row3'`) as the index. This allows you to reference rows by those labels instead of the default integer-based index (`0, 1, 2, ...`).

### Is it Necessary?

- \*\*Not necessary\*\* if you are okay with using the default integer-based index (e.g., `0, 1, 2, ...`).

- \*\*Useful\*\* if you want to label rows with meaningful names (e.g., `'row1'`, `'row2'`, etc.), or if your data naturally has non-integer row identifiers (e.g., timestamps, categories, IDs).

Without providing the `index` parameter, Pandas will automatically assign integer indices to the rows, starting from 0.

### Example Without Custom Index:

```python

data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}

df = pd.DataFrame(data)

print(df)

```

This would yield:

```

A B C

0 1 4 7

1 2 5 8

2 3 6 9

```

Here, the index is automatically assigned as `0, 1, 2`.

### Example With Custom Index:

```python

df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

print(df)

```

This would yield:

```

A B C

row1 1 4 7

row2 2 5 8

row3 3 6 9

```

Now, the rows are labeled as `row1`, `row2`, and `row3`.

### When Custom Indexing is Useful:

- \*\*Meaningful row identifiers\*\*: If your rows represent entities with names, IDs, or other meaningful labels, it makes sense to set the index explicitly.

- \*\*Convenient for selection\*\*: Custom indices make it easier to access specific rows using `loc`:

```python

print(df.loc['row2'])

```

This selects the row with the label `'row2'`.

### Conclusion:

Setting a custom index is \*\*not necessary\*\* but can be helpful if you want more descriptive or meaningful row labels. If you don’t set a custom index, Pandas will automatically use a numeric index starting from 0.

**Create a Custom Transformer**

While Scikit-Learn provides many built-in transformers (like `StandardScaler`, `MinMaxScaler`, etc.), sometimes you’ll need to do something custom. For example, you might want to clean data in a specific way, create new features, or combine certain attributes. In those cases, you’ll need to create your own transformer.

### \*\*Duck Typing and Compatibility with Scikit-Learn:\*\*

Scikit-Learn doesn’t require you to inherit from a specific base class to create a transformer. Instead, it uses \*\*duck typing\*\*, meaning that if your class has the necessary methods (`fit()`, `transform()`, and `fit\_transform()`), Scikit-Learn will treat it like one of its own transformers.

### \*\*Three Methods You Need to Implement:\*\*

1. \*\*`fit(self, X, y=None)`\*\*:

- This method is used to learn from the data (if needed). For many transformers (like scalers), the `fit()` method is where the statistics (e.g., mean, standard deviation) are computed.

- For a custom transformer that doesn't need to learn anything from the data (like a simple column selector), `fit()` can just return `self`.

2. \*\*`transform(self, X)`\*\*:

- This method is where the actual transformation happens. It takes the data (usually `X`), performs some transformation, and returns the modified data.

3. \*\*`fit\_transform(self, X, y=None)`\*\*:

- This is a convenience method that combines `fit()` and `transform()`. When you call `fit\_transform()`, it first calls `fit()` and then `transform()`. You can inherit this method for free by including the `TransformerMixin` class.

### \*\*Using `TransformerMixin` for Convenience:\*\*

By making your custom transformer inherit from `TransformerMixin`, you get the `fit\_transform()` method automatically. You don’t need to write it yourself.

### \*\*Using `BaseEstimator` for Extra Functionality:\*\*

If your class also inherits from `BaseEstimator`, Scikit-Learn will provide additional methods (`get\_params()` and `set\_params()`). These methods are particularly useful when you're performing hyperparameter tuning (e.g., with `GridSearchCV` or `RandomizedSearchCV`), as they allow you to retrieve and set the parameters of your transformer easily.

#### \*\*Example of a Custom Transformer:\*\*

```python

from sklearn.base import BaseEstimator, TransformerMixin

class CustomTransformer(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, some\_param=42): # Avoid \*args and \*\*kwargs

self.some\_param = some\_param

def fit(self, X, y=None):

# Perform any necessary fitting steps here (optional)

return self

def transform(self, X):

# Apply transformation logic

# For example, add some\_param to all elements in X

return X + self.some\_param

```

### \*\*Key Points:\*\*

- \*\*Avoid `\*args` and `\*\*kwargs`\*\* in your constructor: This helps Scikit-Learn automatically track the parameters and makes the transformer compatible with hyperparameter tuning tools.

- \*\*`get\_params()` and `set\_params()`\*\*: These methods come from `BaseEstimator` and are used by Scikit-Learn to retrieve and modify parameters. This is crucial for things like hyperparameter optimization.

### \*\*Why is this important?\*\*

By following this structure, your custom transformer will work with all of Scikit-Learn's features, like being part of a \*\*pipeline\*\* or using \*\*hyperparameter tuning\*\* with `GridSearchCV`.

This code defines a custom transformer called `CombinedAttributesAdder` in Scikit-Learn that adds new features to a dataset. It is designed to work with pipelines and other Scikit-Learn functionality. Let's break it down step by step.

### \*\*Components Overview\*\*

- \*\*`BaseEstimator` and `TransformerMixin`\*\*: These are base classes from Scikit-Learn that allow the transformer to integrate seamlessly with Scikit-Learn's pipeline framework.

- `BaseEstimator`: Provides methods like `get\_params()` and `set\_params()` which are useful for hyperparameter tuning.

- `TransformerMixin`: Provides the `fit\_transform()` method for free if you define `fit()` and `transform()` methods.

### \*\*Index Definitions:\*\*

```python

rooms\_ix, bedrooms\_ix, population\_ix, households\_ix = 3, 4, 5, 6

```

These are indices of specific columns in the dataset. The transformer will use these indices to reference certain attributes (columns):

- `rooms\_ix = 3`: The index for the "total rooms" column.

- `bedrooms\_ix = 4`: The index for the "bedrooms" column.

- `population\_ix = 5`: The index for the "population" column.

- `households\_ix = 6`: The index for the "households" column.

### \*\*Class Definition:\*\*

```python

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, add\_bedrooms\_per\_room=True): # no \*args or \*\*kwargs

self.add\_bedrooms\_per\_room = add\_bedrooms\_per\_room

```

- The `\_\_init\_\_()` method initializes the transformer with a parameter called `add\_bedrooms\_per\_room`. By default, this is set to `True`, but you can set it to `False` when creating an instance.

- The note about "no `\*args` or `\*\*kwargs`" means you should avoid these variable argument lists so that Scikit-Learn can properly track the transformer's parameters, making it compatible with things like hyperparameter tuning.

### \*\*`fit()` Method:\*\*

```python

def fit(self, X, y=None):

return self # nothing else to do

```

- The `fit()` method is where the transformer typically "learns" from the data. In this case, there’s nothing to learn (no internal parameters to compute), so it simply returns `self`. This allows you to chain `fit()` and `transform()` together.

### \*\*`transform()` Method:\*\*

```python

def transform(self, X, y=None):

rooms\_per\_household = X[:, rooms\_ix] / X[:, households\_ix]

population\_per\_household = X[:, population\_ix] / X[:, households\_ix]

if self.add\_bedrooms\_per\_room:

bedrooms\_per\_room = X[:, bedrooms\_ix] / X[:, rooms\_ix]

return np.c\_[X, rooms\_per\_household, population\_per\_household, bedrooms\_per\_room]

else:

return np.c\_[X, rooms\_per\_household, population\_per\_household]

```

- \*\*Input\*\*: The method takes the data matrix `X` as input. The data matrix `X` is assumed to be a 2D NumPy array where each column represents a different feature (like number of rooms, number of households, etc.).

- \*\*Calculating New Features\*\*:

- \*\*`rooms\_per\_household`\*\*: This calculates the number of rooms per household by dividing the values in the "total rooms" column by the values in the "households" column.

- \*\*`population\_per\_household`\*\*: This calculates the number of people per household by dividing the values in the "population" column by the values in the "households" column.

- \*\*Conditional Feature Addition\*\*:

- If `self.add\_bedrooms\_per\_room` is `True`, it also calculates \*\*`bedrooms\_per\_room`\*\* by dividing the values in the "bedrooms" column by the values in the "total rooms" column.

- The method then uses `np.c\_[]` to concatenate the original features (`X`) with the new computed features (`rooms\_per\_household`, `population\_per\_household`, and optionally `bedrooms\_per\_room`) horizontally.

- \*\*Return\*\*: The modified matrix with the additional columns is returned.

### \*\*Creating an Instance:\*\*

```python

attr\_adder = CombinedAttributesAdder(add\_bedrooms\_per\_room=False)

```

Here, an instance of the transformer is created, and we set `add\_bedrooms\_per\_room=False`. This means the `transform()` method will \*\*not\*\* add the `bedrooms\_per\_room` feature, only `rooms\_per\_household` and `population\_per\_household`.

### \*\*Applying the Transformer to a Dataset:\*\*

```python

housing\_extra\_attribs = attr\_adder.transform(housing.values)

```

- `housing.values` converts the `housing` DataFrame into a NumPy array.

- `attr\_adder.transform()` applies the custom transformation, adding the computed features (`rooms\_per\_household` and `population\_per\_household`) to the original data and returning the updated dataset (`housing\_extra\_attribs`).

### \*\*Summary:\*\*

- The class `CombinedAttributesAdder` is a custom transformer that adds new features to a dataset based on existing columns.

- It calculates ratios like `rooms\_per\_household` and `population\_per\_household`, and conditionally adds `bedrooms\_per\_room` based on a parameter.

- It integrates seamlessly with Scikit-Learn’s API, making it useful in pipelines and compatible with hyperparameter tuning.

**Hyper-parameter tuning**

This code snippet uses `GridSearchCV` from scikit-learn to perform hyperparameter tuning on a `RandomForestRegressor` model. Here's a breakdown of how it works:

1. \*\*Parameter Grid\*\*:

- You define a list of dictionaries `param\_grid`, where each dictionary specifies a set of hyperparameters to try out.

- In the first dictionary, `n\_estimators` takes values `[3, 10, 30]`, and `max\_features` takes values `[2, 4, 6, 8]`.

- In the second dictionary, `bootstrap` is set to `False`, `n\_estimators` is limited to `[3, 10]`, and `max\_features` to `[2, 3, 4]`.

- This creates two distinct search spaces, one with bootstrapping enabled and one without it.

2. \*\*RandomForestRegressor Model\*\*:

- An instance of the `RandomForestRegressor` class is created and stored in `forest\_reg`.

- No arguments are passed initially, meaning default settings are used except those specified in the `param\_grid`.

3. \*\*GridSearchCV Setup\*\*:

- `GridSearchCV` is set up with `forest\_reg` and the `param\_grid` to search over specified hyperparameters.

- `cv=5` specifies 5-fold cross-validation, meaning the data is split into five subsets, and the model is trained and validated five times.

- `scoring='neg\_mean\_squared\_error'` tells `GridSearchCV` to use the negative mean squared error as the scoring metric.

- `return\_train\_score=True` enables recording of training scores for each parameter setting.

4. \*\*Fitting the Model\*\*:

- `grid\_search.fit(housing\_prepared, housing\_labels)` fits the model using the prepared dataset `housing\_prepared` and the target variable `housing\_labels`.

- After fitting, `grid\_search` will contain the best parameters found, the corresponding score, and various details about each run.

If you'd like to extract the best parameters or scores, you can use the following commands:

```python

grid\_search.best\_params\_ # Best parameters from the grid search

grid\_search.best\_estimator\_ # Best estimator (model) from the grid search

grid\_search.cv\_results\_ # All cross-validation results

```

The `param\_grid` in your code defines a set of hyperparameters that `GridSearchCV` will try for the `RandomForestRegressor`. Each combination of these hyperparameters is tested during the grid search to find the optimal set. Let's break down each hyperparameter and the values specified:

### 1. \*\*`n\_estimators`\*\*:

- This hyperparameter specifies the number of trees in the forest.

- \*\*Values\*\*: `[3, 10, 30]` in the first grid, `[3, 10]` in the second.

- \*\*3\*\*: A small number of trees, which might lead to an underfitted model.

- \*\*10\*\*: A moderate number of trees, often suitable for smaller datasets.

- \*\*30\*\*: A higher number, which might provide a more accurate model but can also increase computational cost.

- In general, higher values for `n\_estimators` lead to better performance, up to a certain point. However, adding more trees can increase computation time, so there's a trade-off.

### 2. \*\*`max\_features`\*\*:

- This hyperparameter determines the maximum number of features considered for splitting at each node.

- \*\*Values\*\*: `[2, 4, 6, 8]` in the first grid, `[2, 3, 4]` in the second.

- \*\*2, 4\*\*: Using fewer features can help reduce overfitting and make each tree less correlated with others.

- \*\*6, 8\*\*: These values allow more features at each split, which may increase the model’s ability to capture complex patterns but can also increase the risk of overfitting.

- In practice, smaller values for `max\_features` lead to more diverse trees in the forest (less correlated), which may improve generalization. Higher values may provide stronger individual trees but could also increase correlation between them.

### 3. \*\*`bootstrap`\*\*:

- This parameter determines whether bootstrap samples are used when building trees.

- \*\*Values\*\*: `[True]` by default, and `[False]` is specified in the second dictionary in `param\_grid`.

- \*\*True\*\*: (default) Enables sampling with replacement. This is a typical setup for Random Forests and usually helps improve model robustness.

- \*\*False\*\*: Disables bootstrapping, meaning each tree is trained on the entire dataset (without resampling).

- In practice, `bootstrap=True` is recommended because it generally helps with the generalization of the model by adding variance to individual trees. However, in some cases, `bootstrap=False` might give better results, hence the grid search tests both options.

### How `GridSearchCV` Uses `param\_grid`

The grid search will try each combination of these values:

- In the first dictionary:

- Each combination of `n\_estimators` and `max\_features` is tested with `bootstrap=True`.

- In the second dictionary:

- Each combination of `n\_estimators` and `max\_features` is tested with `bootstrap=False`.

For example, with this setup, `GridSearchCV` will test configurations like:

- `{n\_estimators: 3, max\_features: 2, bootstrap: True}`

- `{n\_estimators: 10, max\_features: 4, bootstrap: True}`

- `{n\_estimators: 30, max\_features: 8, bootstrap: True}`

- `{n\_estimators: 3, max\_features: 2, bootstrap: False}`

- `{n\_estimators: 10, max\_features: 3, bootstrap: False}`

And many more combinations. `GridSearchCV` will evaluate each combination using cross-validation to determine which set of parameters yields the best performance according to the scoring metric, in this case, `neg\_mean\_squared\_error`.

Would you like further clarification on any specific hyperparameter or combinations?

Randomized Search

Grid search

Ensemble Methods

Why not use !

The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0).

Inverted 3 becomes -4:

3 = 0000000000000011

-4 = 1111111111111100

Decimal numbers and their binary values:

4 = 0000000000000100

3 = 0000000000000011

2 = 0000000000000010

1 = 0000000000000001

0 = 0000000000000000

-1 = 1111111111111111

-2 = 1111111111111110

-3 = 1111111111111101

-4 = 1111111111111100

**CHAPTER 3**

This code performs a \*\*Stratified K-Fold Cross-Validation\*\* on a classifier (`sgd\_clf`) using the `StratifiedKFold` class from `sklearn.model\_selection`. Here’s a breakdown of each part:

### 1. \*\*Importing the Required Modules\*\*

```python

from sklearn.model\_selection import StratifiedKFold

from sklearn.base import clone

```

- `StratifiedKFold`: This class helps split the dataset into `k` stratified folds. Stratification ensures that each fold has a similar distribution of classes, which is especially useful for imbalanced datasets.

- `clone`: This function creates a copy of an existing model without copying the training data.

### 2. \*\*Initializing `StratifiedKFold`\*\*

```python

skfolds = StratifiedKFold(n\_splits=3, random\_state=42)

```

- `n\_splits=3`: Specifies that the data should be split into 3 folds.

- `random\_state=42`: Ensures reproducibility by controlling the shuffling process.

### 3. \*\*Looping through each Fold\*\*

```python

for train\_index, test\_index in skfolds.split(X\_train, y\_train\_5):

```

- The `split` method takes `X\_train` (features) and `y\_train\_5` (labels) and returns indices for each fold's training and test sets.

- `train\_index` and `test\_index` are the indices for the training and test samples, respectively, in each fold.

### 4. \*\*Cloning the Classifier\*\*

```python

clone\_clf = clone(sgd\_clf)

```

- `sgd\_clf` (not defined in the code snippet but presumably a `SGDClassifier`) is cloned for each fold, creating a fresh copy of the classifier to ensure the training process starts with the same initial settings each time.

### 5. \*\*Preparing Training and Testing Sets for the Fold\*\*

```python

X\_train\_folds = X\_train[train\_index]

y\_train\_folds = y\_train\_5[train\_index]

X\_test\_fold = X\_train[test\_index]

y\_test\_fold = y\_train\_5[test\_index]

```

- The code uses the `train\_index` and `test\_index` to subset the training data and labels for the current fold.

### 6. \*\*Training and Evaluating the Classifier on the Fold\*\*

```python

clone\_clf.fit(X\_train\_folds, y\_train\_folds)

y\_pred = clone\_clf.predict(X\_test\_fold)

n\_correct = sum(y\_pred == y\_test\_fold)

```

- The classifier is trained on `X\_train\_folds` and `y\_train\_folds`.

- Predictions are made on `X\_test\_fold`.

- The number of correct predictions (`n\_correct`) is calculated by comparing `y\_pred` to `y\_test\_fold`.

### 7. \*\*Printing the Accuracy for Each Fold\*\*

```python

print(n\_correct / len(y\_pred))

```

- The accuracy for each fold is computed as the ratio of correct predictions (`n\_correct`) to the total number of predictions (`len(y\_pred)`) and printed out.

- Based on the output example, the accuracy values printed are approximately `0.9502`, `0.96565`, and `0.96495`.

### Summary

This code splits the training data into 3 stratified folds, trains a cloned classifier on two of the folds, tests it on the remaining fold, and prints the accuracy for each fold. It helps evaluate the model's performance while maintaining the class distribution across training and test sets in each split.

Here’s a simulation of the `StratifiedKFold` splitting process on dummy data:

We have a dataset with 12 samples:

- \*\*Features (`X\_train`)\*\*: Numbers from 1 to 12.

- \*\*Labels (`y\_train\_5`)\*\*: Binary targets, with alternating 0's and 1's.

We’re using `StratifiedKFold` with 3 splits, which means the data will be divided into three different training/testing combinations, each with a balanced distribution of classes. Here are the results for each fold:

### Fold 1:

- \*\*Training Set\*\*:

- `X\_train\_folds`: [2, 5, 7, 8, 9, 10, 11, 12]

- `y\_train\_folds`: [1, 0, 0, 1, 0, 1, 0, 1]

- \*\*Testing Set\*\*:

- `X\_test\_fold`: [1, 3, 4, 6]

- `y\_test\_fold`: [0, 0, 1, 1]

### Fold 2:

- \*\*Training Set\*\*:

- `X\_train\_folds`: [1, 3, 4, 5, 6, 9, 10, 12]

- `y\_train\_folds`: [0, 0, 1, 0, 1, 0, 1, 1]

- \*\*Testing Set\*\*:

- `X\_test\_fold`: [2, 7, 8, 11]

- `y\_test\_fold`: [1, 0, 1, 0]

### Fold 3:

- \*\*Training Set\*\*:

- `X\_train\_folds`: [1, 2, 3, 4, 6, 7, 8, 11]

- `y\_train\_folds`: [0, 1, 0, 1, 1, 0, 1, 0]

- \*\*Testing Set\*\*:

- `X\_test\_fold`: [5, 9, 10, 12]

- `y\_test\_fold`: [0, 0, 1, 1]

In each fold, the training and testing sets are made up of \*\*different subsets of the data\*\*, which avoids overlap between training and testing data in each iteration. This demonstrates that `X\_train\_folds` and `X\_test\_fold` (as well as `y\_train\_folds` and `y\_test\_fold`) are indeed different for each fold, even though they all come from the original dataset.

**Recall and Precision**

are two key metrics used in evaluating the performance of classification models, particularly in cases of \*\*binary classification\*\* (e.g., spam vs. not spam). These metrics focus on the performance of the model in terms of how well it identifies positive cases (e.g., "spam" emails) out of all the available data.

### 1. \*\*Recall (Sensitivity or True Positive Rate)\*\*

Recall measures the model’s ability to identify \*\*all relevant instances\*\* (i.e., all true positives) in the data. It answers the question: \*\*Out of all the actual positives, how many did the model correctly identify as positive?\*\*

\[ \text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}} \]

- \*\*True Positives (TP):\*\* Cases where the model correctly predicted the positive class (e.g., predicted "spam" and it was actually spam).

- \*\*False Negatives (FN):\*\* Cases where the model incorrectly predicted the negative class (e.g., predicted "not spam" but it was actually spam).

A \*\*high recall\*\* means that the model identifies most of the relevant positives (i.e., it minimizes false negatives).

### 2. \*\*Precision\*\*

Precision measures the model’s ability to correctly identify \*\*only the relevant instances\*\* (i.e., true positives) among all the cases it labeled as positive. It answers the question: \*\*Out of all the instances the model predicted as positive, how many were actually positive?\*\*

\[ \text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}} \]

- \*\*True Positives (TP):\*\* Same as in recall.

- \*\*False Positives (FP):\*\* Cases where the model incorrectly predicted the positive class (e.g., predicted "spam" but it was actually not spam).

A \*\*high precision\*\* means that the model makes very few false positive predictions (i.e., it is good at avoiding false alarms).

### 3. \*\*Key Differences and Trade-offs\*\*

- \*\*Recall\*\* is focused on minimizing false negatives, i.e., capturing as many true positives as possible. This can be important in scenarios like \*\*disease detection\*\* (you don't want to miss any actual cases).

- \*\*Precision\*\* is focused on minimizing false positives, i.e., ensuring that the positive predictions are accurate. This can be critical in scenarios like \*\*spam detection\*\* (you don’t want to flag too many legitimate emails as spam).

### 4. \*\*Trade-off between Precision and Recall\*\*

There is often a trade-off between precision and recall:

- Increasing recall (trying to capture more positives) may decrease precision (you might capture more false positives).

- Increasing precision (being more selective) may decrease recall (you might miss some true positives).

### 5. \*\*F1 Score\*\*

The \*\*F1 Score\*\* combines precision and recall into a single metric. It's the harmonic mean of precision and recall, useful when you need a balance between the two.

\[ \text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]

The F1 Score is particularly helpful when you need to consider both false positives and false negatives, especially in imbalanced datasets.

In summary:

- \*\*Recall\*\* = How many of the true positives were identified?

- \*\*Precision\*\* = How many of the predicted positives were correct?

**Multi purpose classifiers**

### 1. \*\*One-vs-All (OvA)\*\*:

- \*\*Concept\*\*: Also known as \*\*One-vs-Rest (OvR)\*\*, this approach creates \*\*one binary classifier for each class\*\*. For each classifier, the model is trained to distinguish between one specific class and all the other classes combined.

- \*\*How it works\*\*:

- Suppose you have 3 classes: A, B, and C. Using OvA, the algorithm will create 3 separate classifiers:

- Classifier 1: Distinguishes class A from {B, C}.

- Classifier 2: Distinguishes class B from {A, C}.

- Classifier 3: Distinguishes class C from {A, B}.

- During prediction, the classifier with the highest confidence or score is selected as the final prediction.

- \*\*Advantages\*\*:

- Simple to implement and works well with many algorithms.

- Each classifier is only concerned with distinguishing one class, which can make the training process faster.

- \*\*Disadvantages\*\*:

- Each binary classifier is exposed to potentially unbalanced data (one class vs. the rest).

- The approach can struggle when the classes overlap significantly.

### 2. \*\*One-vs-One (OvO)\*\*:

- \*\*Concept\*\*: This strategy creates a binary classifier for \*\*every possible pair of classes\*\*. For a dataset with \( n \) classes, it creates \( \frac{n(n-1)}{2} \) classifiers. Each classifier is trained to differentiate between two classes only.

- \*\*How it works\*\*:

- Suppose you have 3 classes: A, B, and C. Using OvO, the algorithm will create 3 classifiers:

- Classifier 1: Distinguishes between A and B.

- Classifier 2: Distinguishes between B and C.

- Classifier 3: Distinguishes between A and C.

- During prediction, each classifier casts a vote for one of the two classes, and the class with the most votes wins (known as a \*\*majority voting scheme\*\*).

- \*\*Advantages\*\*:

- Each classifier deals with a smaller, more manageable binary problem, which may result in better accuracy.

- The classifiers focus only on the differences between two specific classes, leading to clearer boundaries.

- \*\*Disadvantages\*\*:

- It requires training many classifiers, which can be computationally expensive.

- There are more models to manage and more predictions to make during inference.

### Comparison:

- \*\*OvA\*\*: Fewer classifiers (just \( n \)), faster to train and easier to manage, but it may face difficulties with class imbalance or overlap.

- \*\*OvO\*\*: Many more classifiers ( \( \frac{n(n-1)}{2} \)), potentially better performance, especially with small or imbalanced datasets, but it is computationally more intensive.

Both strategies are widely used depending on the type of classification problem and the specific characteristics of the dataset.

**SGDClassifier from Scikit-learn, the decision\_function**

method provides raw decision scores for each instance, which are used to determine the class labels. These scores are particularly useful when you want to understand how confident the model is about its predictions or when you need to calculate metrics such as those used in the ROC curve.

Here’s how the SGDClassifier works with the decision\_function and how instances are scored:

How decision\_function Works:

Decision Boundary: In linear classifiers like the SGDClassifier (which uses stochastic gradient descent to optimize a linear model), the model fits a decision boundary in the feature space. For binary classification, this decision boundary is a hyperplane that separates the instances into two classes.

Scoring Instances:

The decision\_function computes the distance of a given instance from this decision boundary (hyperplane). The score reflects the model’s confidence in its prediction.

The sign of the score determines the predicted class:

A positive score means the instance is on the side of the hyperplane assigned to the positive class (class 1).

A negative score means the instance is on the side of the hyperplane assigned to the negative class (class 0).

The magnitude of the score indicates the confidence of the model in its prediction:

A large positive score means the model is highly confident the instance belongs to the positive class.

A large negative score means the model is highly confident the instance belongs to the negative class.

A score close to zero means the instance is near the decision boundary, indicating uncertainty in the classification.

Mathematically: The score for an instance is calculated as:

score=wTx+b\text{score} = \mathbf{w}^T \mathbf{x} + bscore=wTx+b

Where:

w\mathbf{w}w is the weight vector (learned coefficients of the model).

x\mathbf{x}x is the feature vector of the instance.

bbb is the bias term (intercept).

This linear combination of the features and weights gives the raw decision score.

Example of How decision\_function Is Used:

python

Copy code

from sklearn.linear\_model import SGDClassifierfrom sklearn.datasets import make\_classification

# Create a binary classification dataset

X, y = make\_classification(n\_samples=1000, n\_features=10, random\_state=42)

# Train an SGDClassifier

sgd\_clf = SGDClassifier(random\_state=42)

sgd\_clf.fit(X, y)

# Get the decision scores for each instance

decision\_scores = sgd\_clf.decision\_function(X)

# Example decision score for a sampleprint(decision\_scores[:5]) # Outputs raw scores for the first 5 instances

In this example, decision\_scores contains the raw scores for each instance. A positive score indicates that the model is classifying the instance as belonging to the positive class (class 1), and a negative score indicates classification into the negative class (class 0).

Multi-Class Case (One-vs-All):

If SGDClassifier is used for multiclass classification, it typically uses a One-vs-All (OvA) strategy. In this case:

The decision\_function will return one score per class for each instance. The class with the highest score is chosen as the predicted class.

For example, if you have 3 classes, the decision function will return a 3-dimensional array of scores (one for each class), and the class with the highest score will be the prediction.

Summary:

sgd\_clf.decision\_function(X) gives the raw scores (distances to the decision boundary) for each instance.

The sign of the score determines the predicted class, and the magnitude reflects the confidence of the prediction.

This method is useful when you need continuous scores for things like ROC curves, where you're not just interested in the class label but in how confident the model is about its decision.

**· Multilabel classification** allows an instance to belong to multiple labels from the same target.

**· Multioutput** classification provides multiple outputs for different targets, which may be of varying types and unrelated to each other.