Features

Learn about the different feature types that can be part of a dataset.

Chapter Goals:

- Understand the difference between quantitative and categorical features
- Learn methods to manipulate features and add them to a DataFrame
- Write code to add MLB statistics to a DataFrame

A. Quantitative vs. categorical

We often refer to the columns of a DataFrame as the *features* of the dataset that it represents. These features can be quantitative or categorical.

A quantitative feature, e.g. height or weight, is a feature that can be measured numerically. These are features we could calculate the sum, mean, or other numerical metrics for.

A categorical feature, e.g. gender or birthplace, is one where the values are categories that could be used to group the dataset. These are the features we would use with the `groupby` function from the previous chapter.

Some features can be both quantitative or categorical, depending on the context they are used. For example, we could use year of birth as a quantitative feature if we were trying to find statistics such as the average birth year for a particular dataset. On the other hand, we could also use it as a categorical feature and group the data by the different years of birth.

B. Quantitative features

In the previous chapter, we focused on grouping a dataset by its categorical features. We'll now describe methods for dealing with quantitative features.

Two of the most important functions to use with quantitative features are `sum` and `mean`. In the previous chapter we also introduced `sum` and `mean` functions, which were used to aggregate quantitative features for each a group.

However, while the functions from the previous chapter were applied to the output of `groupby`, the ones we use in this chapter are applied to individual DataFrames.

The code below shows example usages of `sum` and `mean`. The `df` DataFrame represents three different speed tests (columns) for three different processors (rows). The data values correspond to the seconds taken for a given speed test and processor.

```
df = pd.DataFrame({
  'T1': [10, 15, 8],
  'T2': [25, 27, 25],
  'T3': [16, 15, 10]})

print('{}\n'.format(df))

print('{}\n'.format(df.sum()))

print('{}\n'.format(df.sum(axis=1)))

print('{}\n'.format(df.mean()))

print('{}\n'.format(df.mean(axis=1)))
```

Neither function takes in a required argument. The most commonly used keyword argument for both functions is `axis`. The `axis` argument specifies whether to aggregate over rows (`axis=0`, the default), or columns (`axis=1`).

In the code example, we used a DataFrame representing speed tests for three different processors (measured in seconds). When we used no argument, equivalent to using `axis=0`, the `sum` and `mean` functions calculated total and average times for each test. When we used `axis=1`, the `sum` and `mean` functions calculated total and average test times (across all three tests) for each processor.

C. Weighted features

Along with aggregating quantitative features, we can also apply weights to them. We do this through the `multiply` function.

The `multiply` function takes in a list of weights or a constant as its required argument. If a constant is used, the constant is multiplied across all the rows or columns (depending on the value of `axis`). If a list is used, then the position of each weight in the list corresponds to which row/column it is multiplied to.

In contrast with `sum` and `mean`, the default `axis` for `multiply` is the columns axis. Therefore, to multiply weights along the rows of a DataFrame, we need to explicitly set `axis=0`.

The code below shows example usages of `multiply`. The `df` DataFrame represents three different speed tests (columns) for two different processors (rows).

```
df = pd.DataFrame({
  'T1': [0.1, 150.],
  'T2': [0.25, 240.],
  'T3': [0.16, 100.]})

print('{}\n'.format(df))

print('{}\n'.format(df.multiply(2)))

df_ms = df.multiply([1000, 1], axis=0)
print('{}\n'.format(df_ms))

df_w = df_ms.multiply([1,0.5,1])
print('{}\n'.format(df_w))
print('{}\n'.format(df_w.sum(axis=1)))
```

In the code above, the test times for processor `'p1'` were measured in seconds, while the times for `'p2'` were in milliseconds. So we made all the times in milliseconds by multiplying the values of `'p1'` by `1000`.

Then we multiplied the values in `'T2'` by `0.5`, since those tests were done with two processors rather than one. This makes the final `sum` a *weighted sum* across the three columns.

Time to Code!

The code exercises for this chapter involves calculating various baseball statistics based on the values of other statistics. The `mlb_df` DataFrame is predefined, and contains all historic MLB hitting statistics.

We also provide a `col_list_sum` function. This is a utility function to calculate the sum of multiple columns across a DataFrame.

```
def col_list_sum(df, col_list, weights=None):
    col_df = df[col_list]
    if weights is not None:
        col_df = col_df.multiply(weights)
    return col_df.sum(axis=1)
```

The `df` argument is the input DataFrame, while the `col_list` argument is a list of column labels representing the columns we want to sum in `df`.

The `weights` keyword argument represents the weight coefficients we use for a weighted column sum. Note that if `weights` is not `None`, it must have the same length as `col_list`.

The `mlb_df` doesn't contain one of the key stats in baseball, batting average. Therefore, we'll calculate the batting average and add it as a column in `mlb_df`.

To calculate the batting average, simply divide a player's hits (`'H'`) by their number of at-bats (`'AB'`).

Set `mlb_df['BA']` equal to `mlb_df['H']` divided by `mlb_df['AB']`.

```
# CODE HERE
```

Though `mlb_df` contains columns for doubles, triples, and home runs (labeled `'2B'`, `'3B'`, `'HR'`), it does not contain a column for singles.

However, we can calculate singles by subtracting doubles, triples, and home runs from the total number of hits. We'll label the singles column as `'1B'`.

Set `other_hits` equal to `col_list_sum` with `mlb_df` as the first argument. The second argument should be a list of the column labels for doubles, triples, and home runs.

Set `mlb_df['1B']` equal to `mlb_df['H']` subtracted by `other_hits`.

```
# CODE HERE
```

Now that `mlb_df` contains columns for all four types of hits, we can calculate slugging percentage (column label `'SLG'`). The formula for slugging percentage is:

$$\text{SLG} = \frac{1\text{B} + 2 \cdot 2\text{B} + 3 \cdot 3\text{B} + 4 \cdot \text{HR}}{\text{AB}}$$

Therefore, the numerator represents a weighted sum with column labels `'1B'`, `'2B'`, `'3B'`, `'HR'`.

Set `weighted_hits` equal to `col_list_sum` with `mlb_df` as the first argument and a list of numerator column labels as the second argument. The `weights` keyword argument should be a list of integers from 1 to 4, inclusive.

```
# CODE HERE
```

We can now calculate the slugging percentage by dividing the weighted sum by the number of at-bats.

Set `mlb_df['SLG']` equal to `weighted_hits` divided by `mlb_df['AB']`.

```
# CODE HERE
```