Predicting data over time

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf
Fellow, Berkeley Institute for Data Science

DataCamp

Classification vs. Regression

CLASSIFICATION

```
classification_model.predict(X_test)
```

```
array([0, 1, 1, 0])
```

REGRESSION

```
regression_model.predict(X_test)
```

```
array([0.2, 1.4, 3.6, 0.6])
```

Correlation and regression

- Regression is similar to calculating correlation, with some key differences
    - **Regression**: A process that results in a formal model of the data

    - **Correlation**: A statistic that describes the data. Less information than regression model.

Correlation between variables often changes over time

- Timeseries often have patterns that change over time

- Two timeseries that seem correlated at one moment may not remain so over time

Visualizing relationships between timeseries

```python
fig, axs = plt.subplots(1, 2)

# Make a line plot for each timeseries
axs[0].plot(x, c='k', lw=3, alpha=.2)
axs[0].plot(y)
axs[0].set(xlabel='time', title='X values = time')


# Encode time as color in a scatterplot
axs[1].scatter(x_long, y_long, c=np.arange(len(x_long)), cmap='viridis')
axs[1].set(xlabel='x', ylabel='y', title='Color = time')
```

Visualizing two timeseries
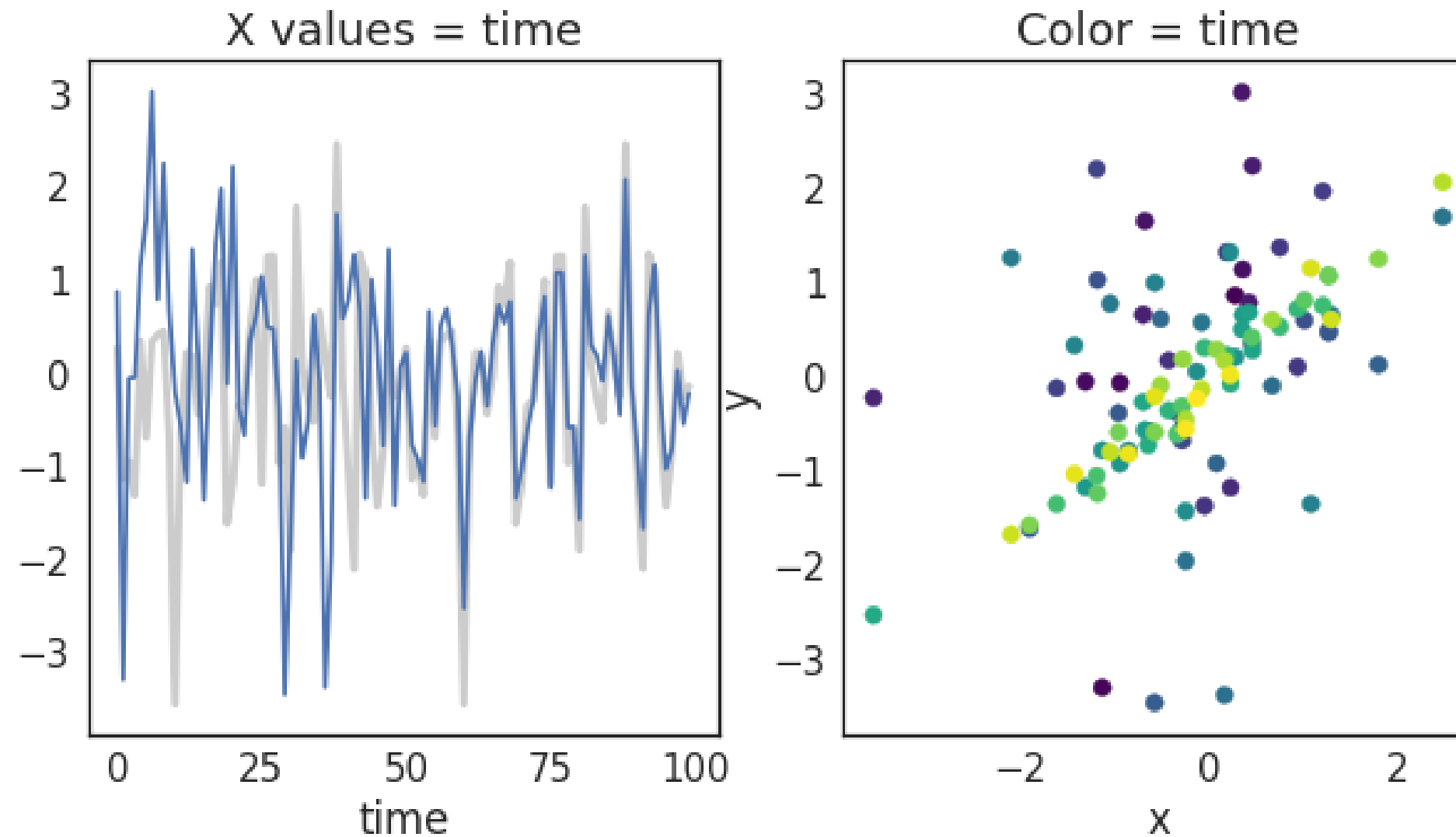
Regression models with scikit-learn

```python
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X, y)
model.predict(X)
```

Visualize predictions with scikit-learn

```python
alphas = [.1, 1e2, 1e3]
ax.plot(y_test, color='k', alpha=.3, lw=3)
for ii, alpha in enumerate(alphas):
    y_predicted = Ridge(alpha=alpha).fit(X_train, y_train).predict(X_test)
    ax.plot(y_predicted, c=cmap(ii / len(alphas)))
ax.legend(['True values', 'Model 1', 'Model 2', 'Model 3'])
ax.set(xlabel="Time")
```

Visualize predictions with scikit-learn
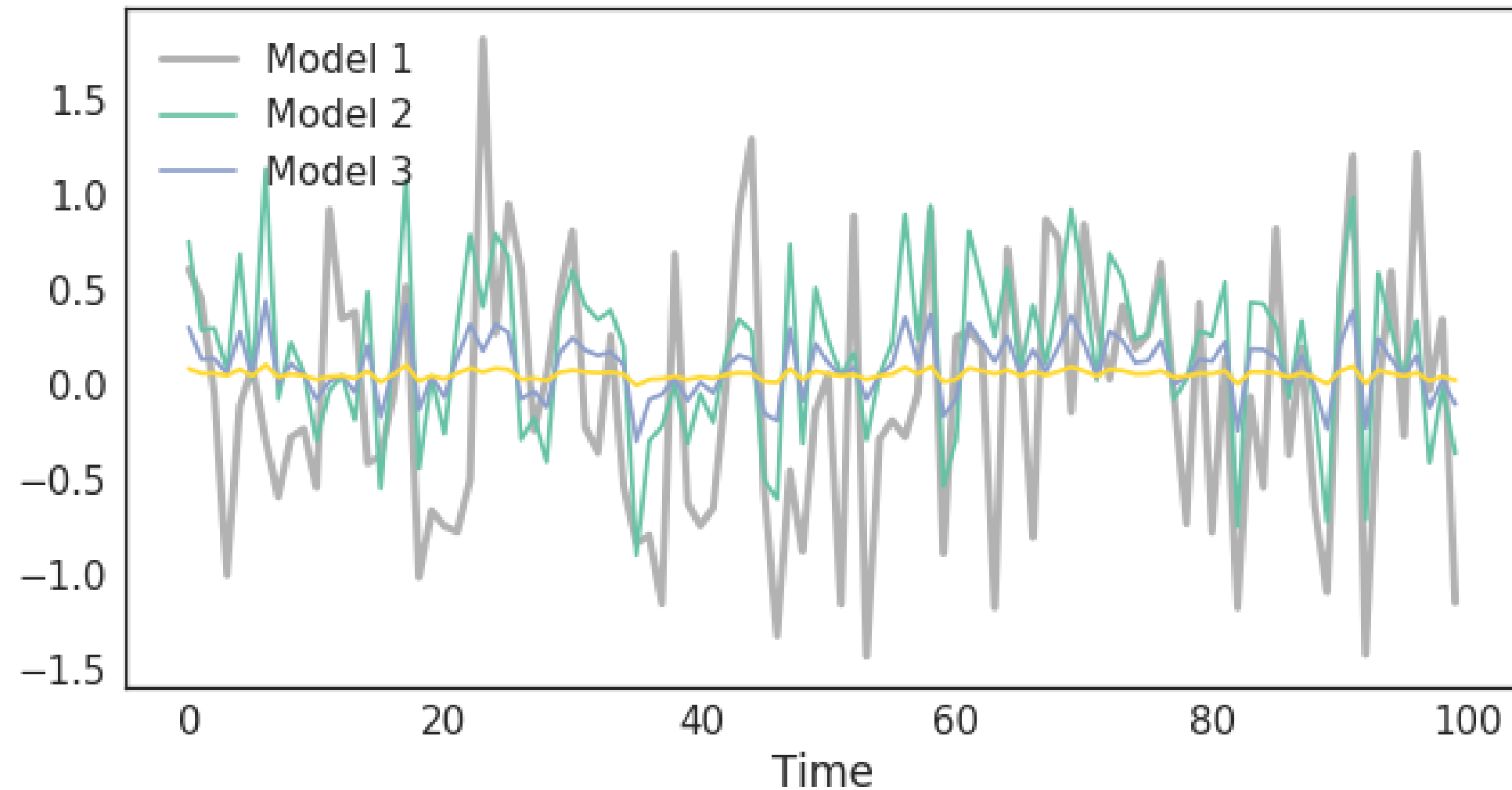
Scoring regression models

- Two most common methods:
  - Correlation (r)

  - Coefficient of Determination (R^2)

Coefficient of Determination (R^2)

- The value of R^2 is bounded on the top by 1, and can be infinitely low

- Values closer to 1 mean the model does a better job of predicting outputs

$$1 - \frac{error(model)}{variance(testdata)}$$

R^2 in scikit-learn

```python
from sklearn.metrics import r2_score
print(r2_score(y_predicted, y_test))
```

```
0.08
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Cleaning and improving your data

## MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Chris Holdgraf
Fellow, Berkeley Institute for Data Science

DataCamp

Data is messy

- Real-world data is often messy

- The two most common problems are *missing data* and *outliers*

- This often happens because of human error, machine sensor malfunction, database failures, etc

- Visualizing your raw data makes it easier to spot these problems

What messy data looks like
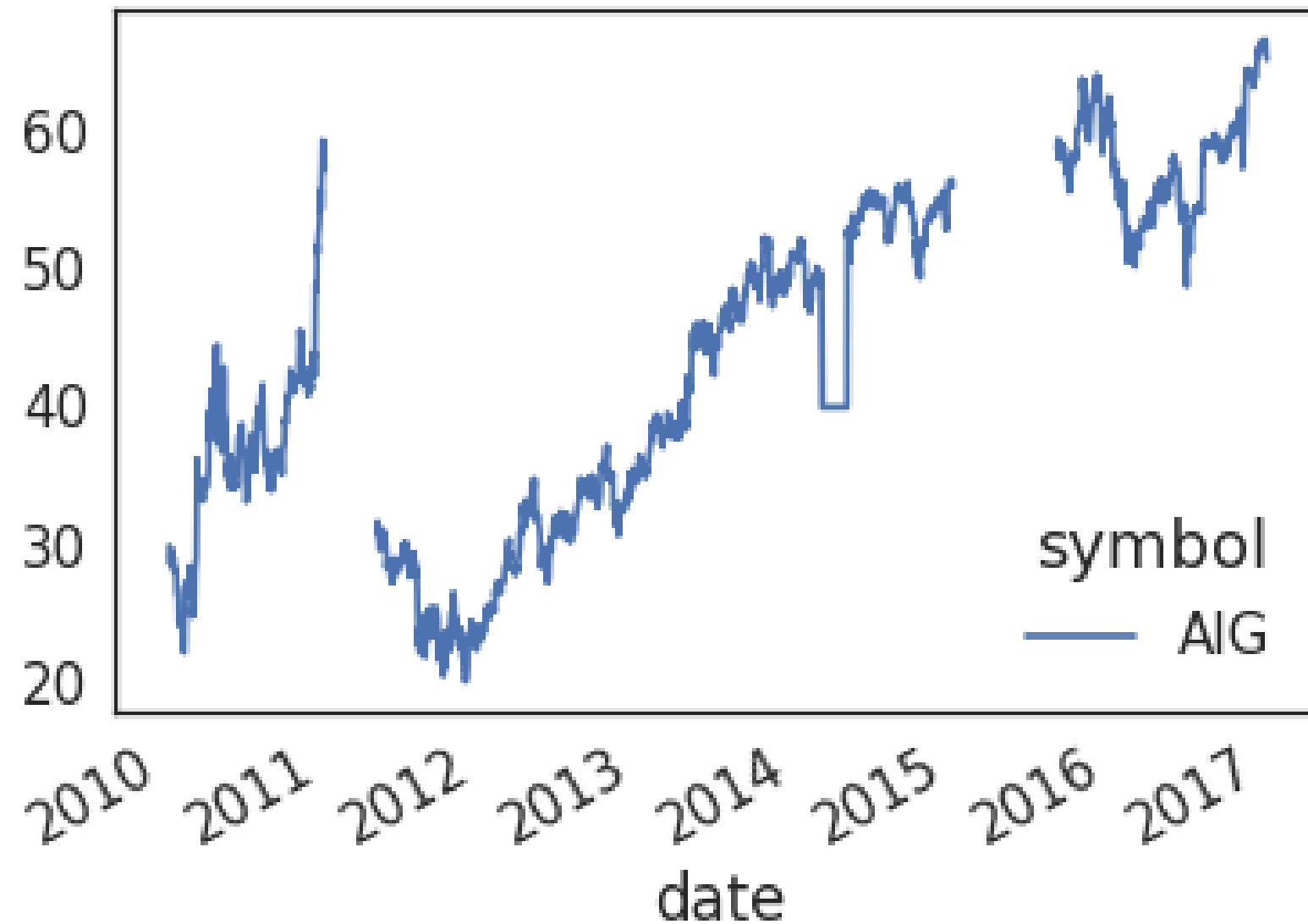
Interpolation: using time to fill in missing data

- A common way to deal with missing data is to *interpolate* missing values

- With timeseries data, you can use time to assist in interpolation.

- In this case, **interpolation** means using using the *known* values on either side of a gap in the data to make assumptions about what's missing.
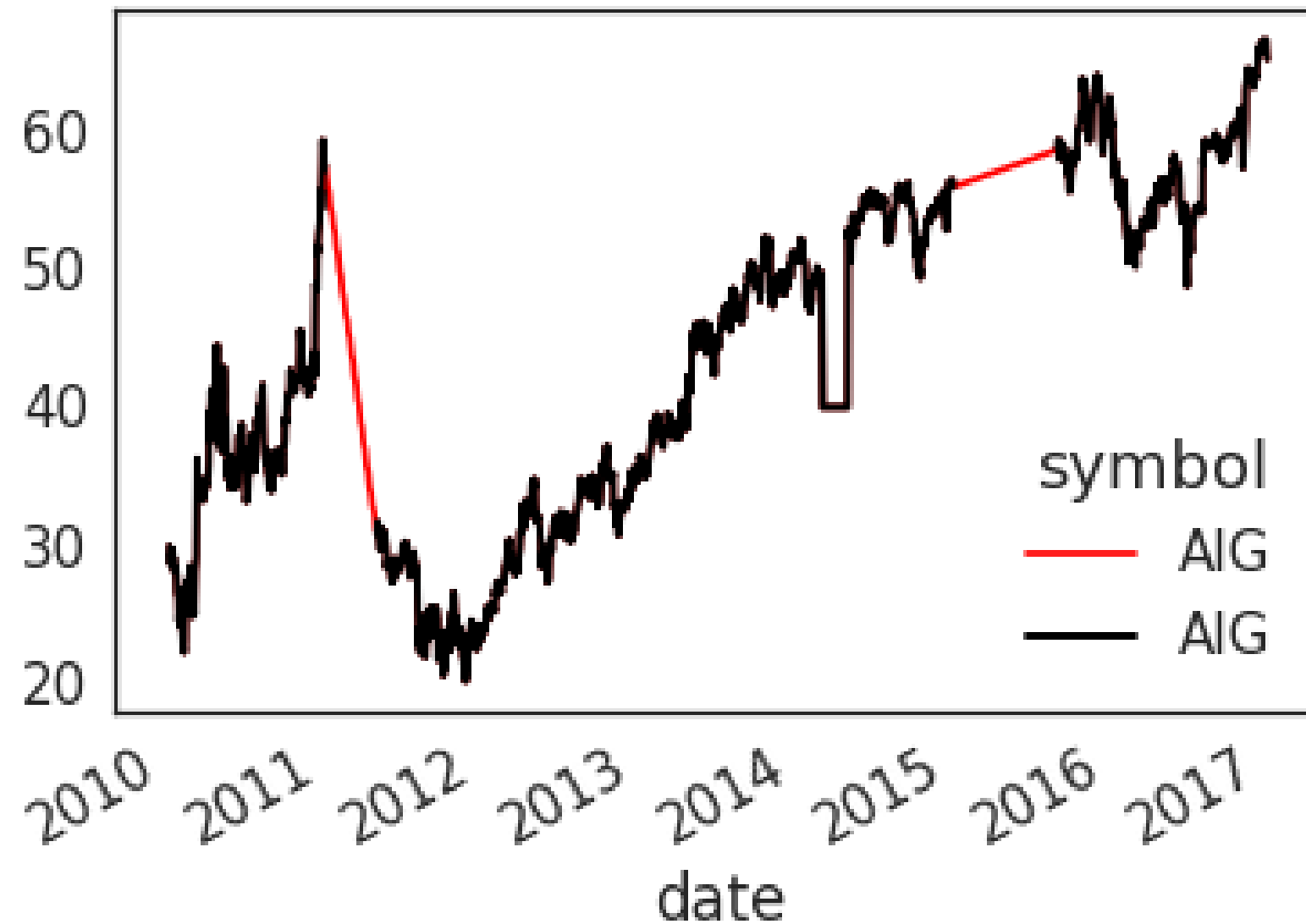
Interpolation in Pandas

```python
# Return a boolean that notes where missing values are
missing = prices.isna()


# Interpolate linearly within missing windows
prices_interp = prices.interpolate('linear')


# Plot the interpolated data in red and the data w/ missing values in black
ax = prices_interp.plot(c='r')
prices.plot(c='k', ax=ax, lw=2)
```

Visualizing the interpolated data

Using a rolling window to transform data

- Another common use of rolling windows is to transform the data

- We've already done this once, in order to *smooth* the data

- However, we can also use this to do more complex transformations

Transforming data to standardize variance

- A common transformation to apply to data is to standardize its mean and variance over time. There are many ways to do this.

- Here, we'll show how to convert your dataset so that each point represents the *% change over a previous window*.

- This makes timepoints more comparable to one another if the absolute values of data change a lot

Transforming to percent change with Pandas

```python
def percent_change(values):
    """Calculates the % change between the last value
    and the mean of previous values"""
    # Separate the last value and all previous values into variables
    previous_values = values[:-1]
    last_value = values[-1]

    # Calculate the % difference between the last value
    # and the mean of earlier values
    percent_change = (last_value - np.mean(previous_values)) \
    / np.mean(previous_values)
    return percent_change
```

Applying this to our data

```python
# Plot the raw data
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
ax = prices.plot(ax=axs[0])

# Calculate % change and plot
ax = prices.rolling(window=20).aggregate(percent_change).plot(ax=axs[1])
ax.legend_.set_visible(False)
```
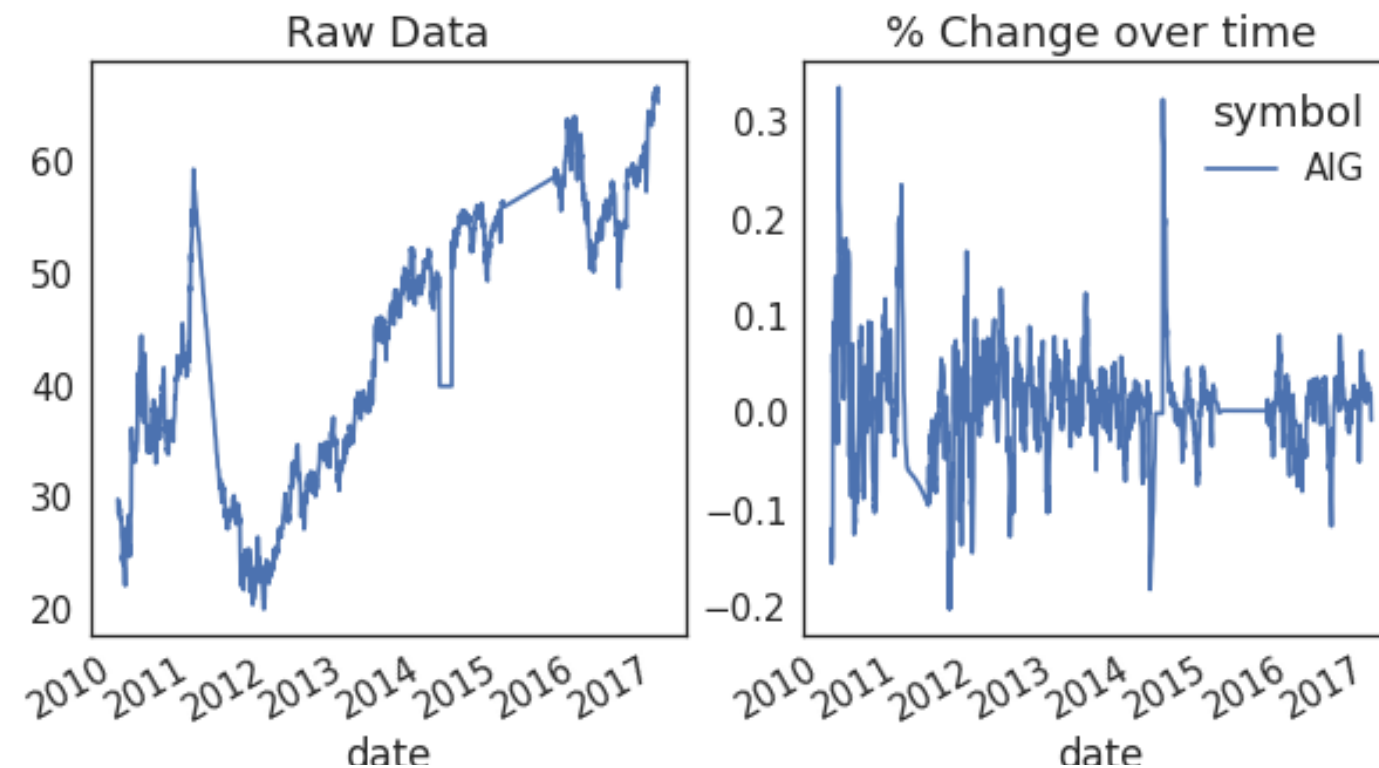
Finding outliers in your data

- Outliers are datapoints that are significantly statistically different from the dataset.

- They can have negative effects on the predictive power of your model, biasing it away from its "true" value

- One solution is to *remove* or *replace* outliers with a more representative value

  Be very careful about doing this - often it is difficult to determine what is a legitimately extreme value vs an abberation

Plotting a threshold on our data

```python
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
for data, ax in zip([prices, prices_perc_change], axs):
    # Calculate the mean / standard deviation for the data
    this_mean = data.mean()
    this_std = data.std()

    # Plot the data, with a window that is 3 standard deviations
    # around the mean
    data.plot(ax=ax)
    ax.axhline(this_mean + this_std * 3, ls='--', c='r')
    ax.axhline(this_mean - this_std * 3, ls='--', c='r')
```

Visualizing outlier thresholds
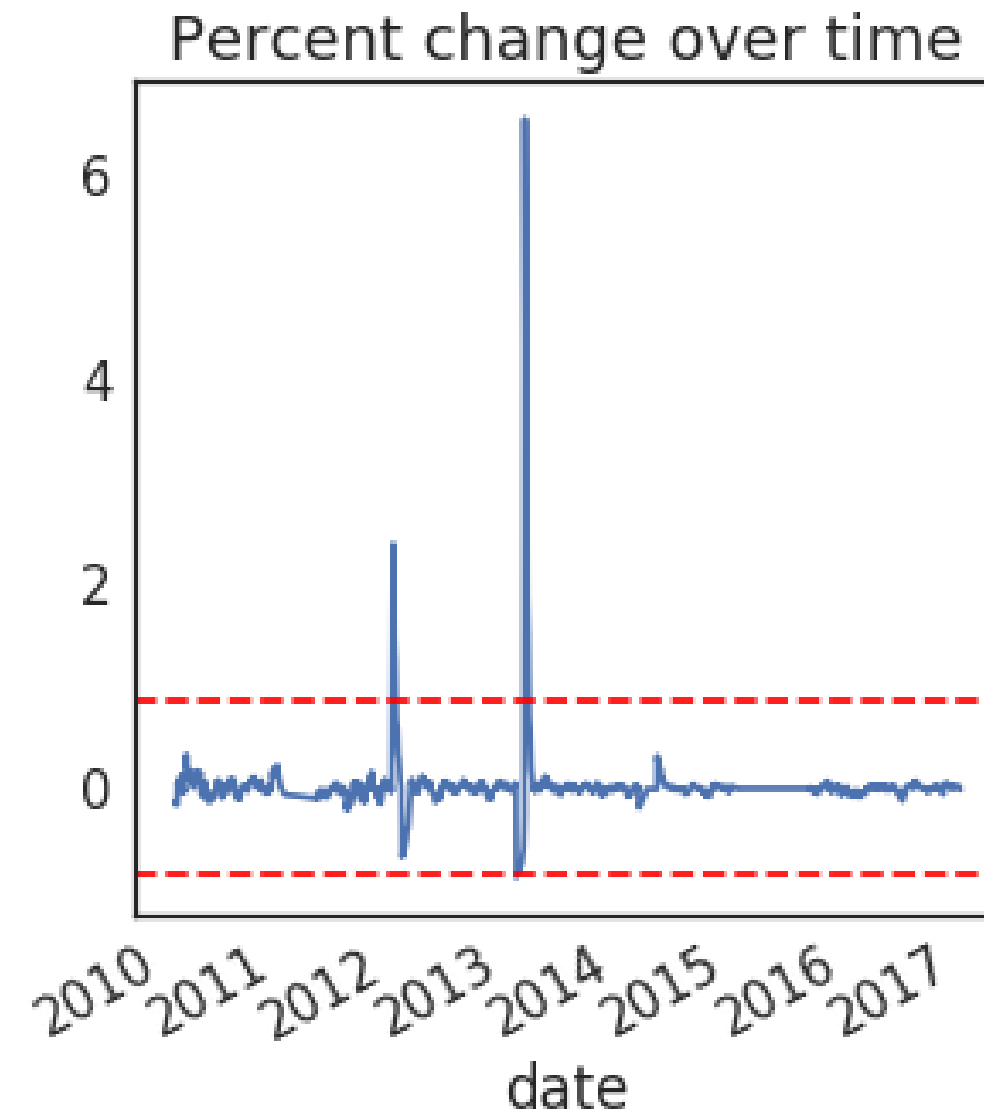
Replacing outliers using the threshold

```python
# Center the data so the mean is 0
prices_outlier_centered = prices_outlier_perc - prices_outlier_perc.mean()

# Calculate standard deviation
std = prices_outlier_perc.std()

# Use the absolute value of each datapoint
# to make it easier to find outliers
outliers = np.abs(prices_outlier_centered) > (std * 3)

# Replace outliers with the median value
# We'll use np.nanmean since there may be nans around the outliers
prices_outlier_fixed = prices_outlier_centered.copy()
prices_outlier_fixed[outliers] = np.nanmedian(prices_outlier_fixed)
```
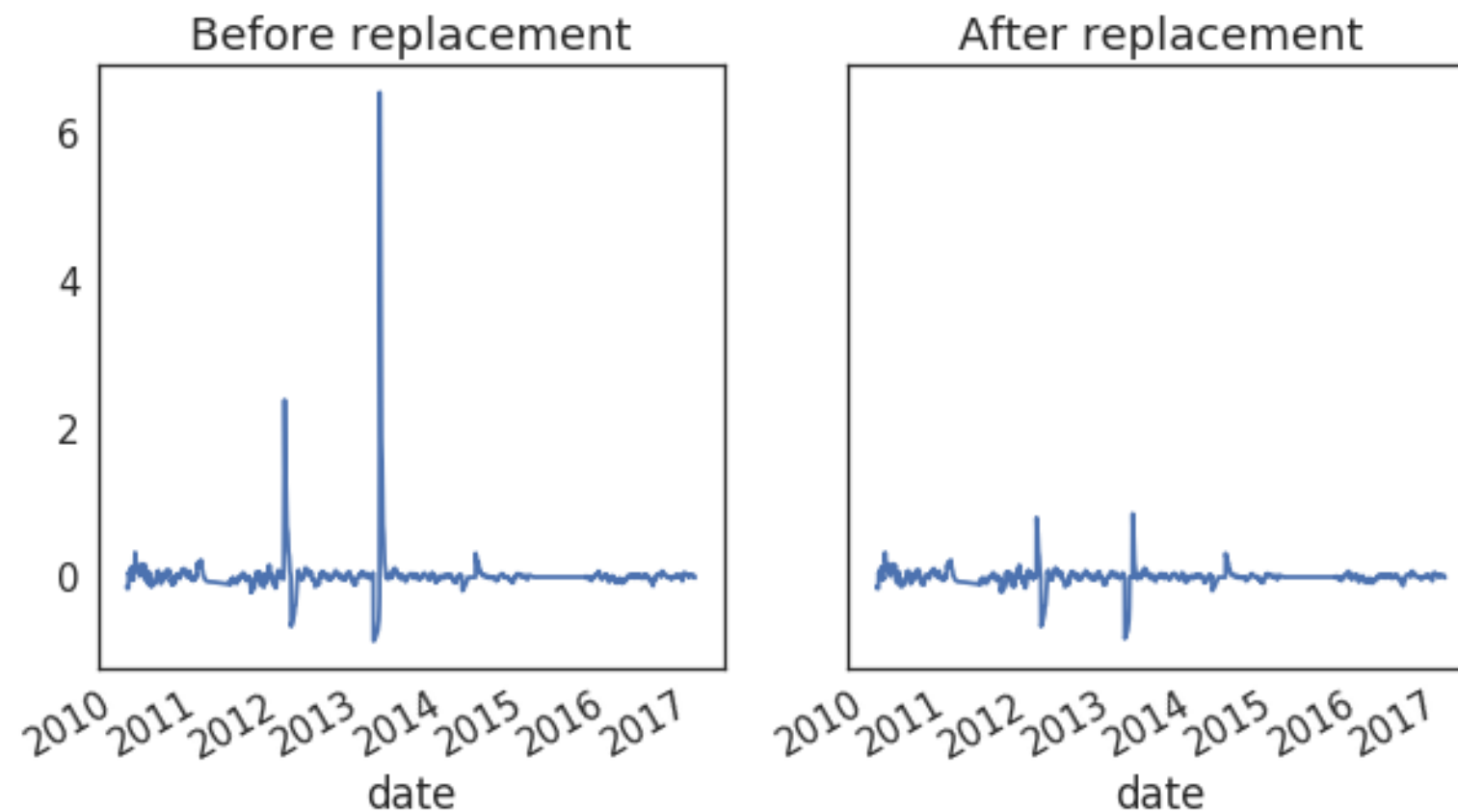
Visualize the results

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
prices_outlier_centered.plot(ax=axs[0])
prices_outlier_fixed.plot(ax=axs[1])
```
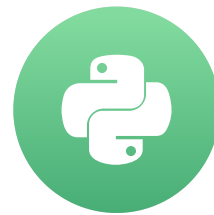
Let's practice!

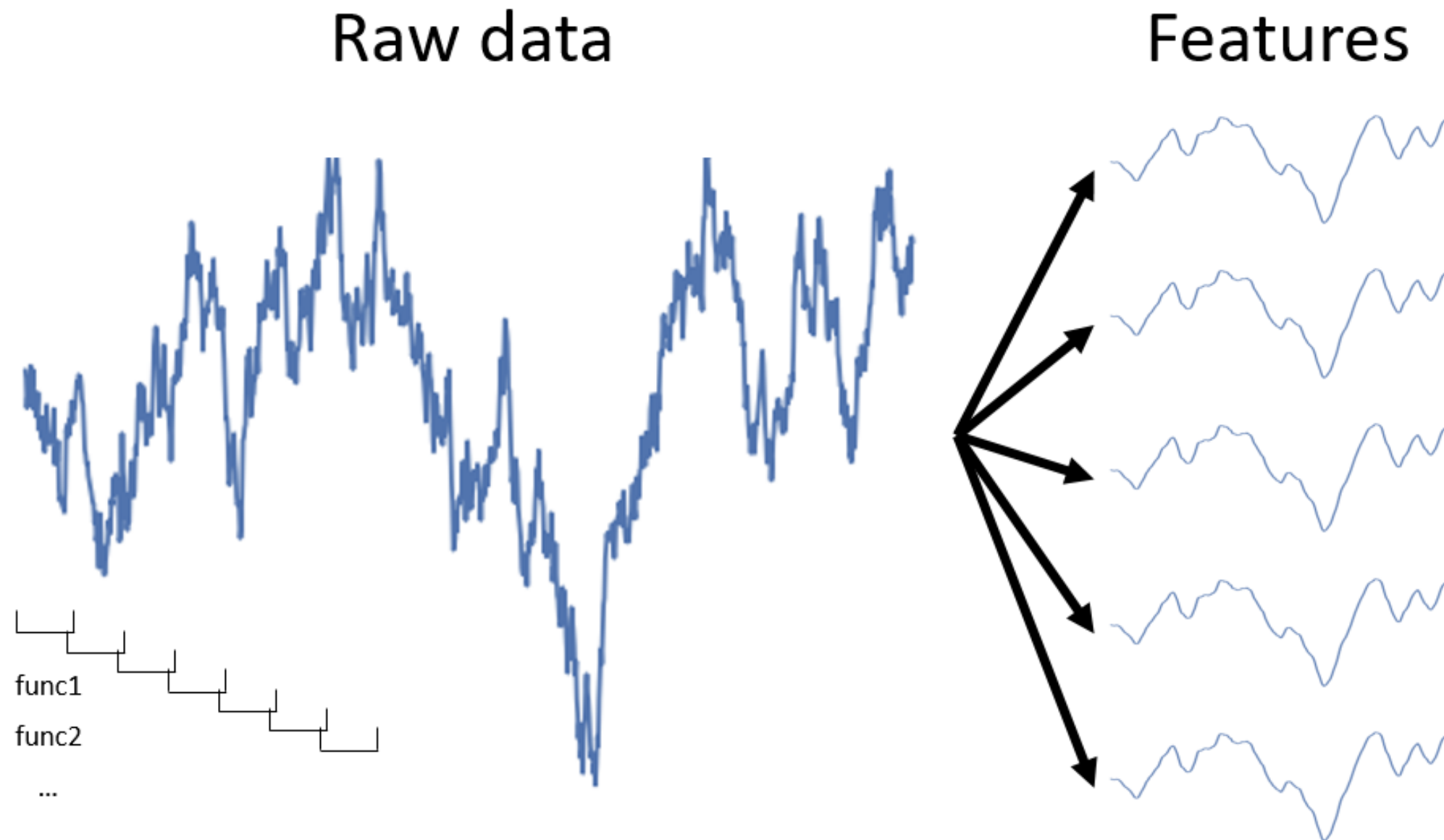MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON

Extracting features with windows

Raw data

Features

func1

func2

...

Using .aggregate for feature extraction

```python
# Visualize the raw data
print(prices.head(3))
```

```
symbol            AIG         ABT
date
2010-01-04  29.889999   54.459951
2010-01-05  29.330000   54.019953
2010-01-06  29.139999   54.319953
```

```python
# Calculate a rolling window, then extract two features
feats = prices.rolling(20).aggregate([np.std, np.max]).dropna()
print(feats.head(3))
```

```
               AIG                  ABT
            std        amax      std        amax
date
2010-02-01  2.051966   29.889999   0.868830   56.239949
2010-02-02  2.101032   29.629999   0.869197   56.239949
2010-02-03  2.157249   29.629999   0.852509   56.239949
```

Check the properties of your features!
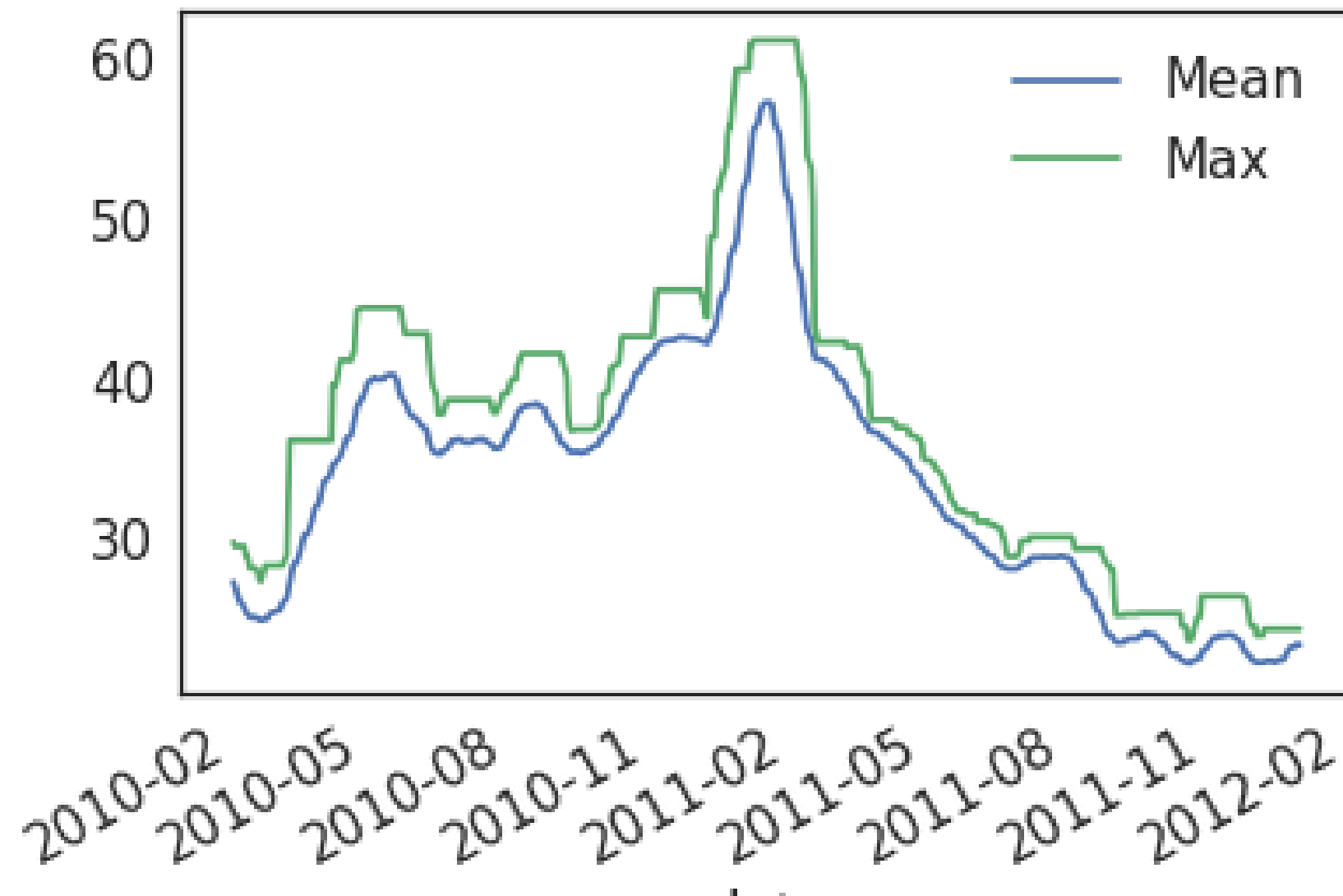
Using partial() in Python

```python
# If we just take the mean, it returns a single value
a = np.array([[0, 1, 2], [0, 1, 2], [0, 1, 2]])
print(np.mean(a))
```

```
1.0
```

```python
# We can use the partial function to initialize np.mean
# with an axis parameter
from functools import partial
mean_over_first_axis = partial(np.mean, axis=0)


print(mean_over_first_axis(a))
```

```
[0. 1. 2.]
```

Percentiles summarize your data

- Percentiles are a useful way to get more fine-grained summaries of your data (as opposed to using `np.mean`)

- For a given dataset, the Nth percentile is the value where N% of the data is below that datapoint, and 100-N% of the data is above that datapoint.

```python
print(np.percentile(np.linspace(0, 200), q=20))
```

```
40.0
```

Combining np.percentile() with partial functions to calculate a range of percentiles

```python
data = np.linspace(0, 100)

# Create a list of functions using a list comprehension
percentile_funcs = [partial(np.percentile, q=ii) for ii in [20, 40, 60]]

# Calculate the output of each function in the same way
percentiles = [i_func(data) for i_func in percentile_funcs]
print(percentiles)
```

```
[20.0, 40.00000000000001, 60.0]
```

```python
# Calculate multiple percentiles of a rolling window
data.rolling(20).aggregate(percentiles)
```

Calculating "date-based" features

- Thus far we've focused on calculating "statistical" features - these are features that correspond statistical properties of the data, like "mean", "standard deviation", etc

- However, don't forget that timeseries data often has more "human" features associated with it, like days of the week, holidays, etc.

- These features are often useful when dealing with timeseries data that spans multiple years (such as stock value over time)

datetime features using Pandas

```python
# Ensure our index is datetime
prices.index = pd.to_datetime(prices.index)

# Extract datetime features
day_of_week_num = prices.index.weekday
print(day_of_week_num[:10])
```

```
Index([0 1 2 3 4 0 1 2 3 4], dtype='object')
```

```python
day_of_week = prices.index.weekday_name
print(day_of_week[:10])
```

```
Index(['Monday' 'Tuesday' 'Wednesday' 'Thursday' 'Friday' 'Monday' 'Tuesday'
  'Wednesday' 'Thursday' 'Friday'], dtype='object')
```

Let's practice!

MACHINE LEARNING FOR TIME SERIES DATA IN PYTHON