1.2 - Financial Data

Sources, Structure, and Meaning

Giovanni Della Lunga
giovanni.dellalunga@unibo.it

Introduction to Machine Learning for Finance

Bologna - February-March, 2026

Contents

Lesson 1.2 — Motivation

- Machine Learning models do not operate on abstract data.
- They operate on **real financial datasets**, generated by markets, institutions, and economic processes.
- Data quality, structure, and availability strongly constrain what ML can realistically achieve in finance.

Key idea: before modeling, we must understand *where data come from* and *how they are structured*.

Lesson 1.2 — Learning Objectives

At the end of this lecture, you should be able to:

- Understand the role of **Pandas** as the core data-handling tool for financial ML.
- Gather financial and macroeconomic data from **public sources** (Yahoo Finance, FRED, World Bank).
- Recognize the main **financial data structures** (prices, returns, OHLC bars).
- Apply basic transformations consistent with financial meaning (prices vs returns, resampling).

Lesson 1.2 — Didactic Targets

This lecture is designed to develop:

- **Operational literacy**: knowing how to acquire and inspect financial data.
- **Financial awareness**: understanding how market conventions shape data.
- **Critical thinking**: recognizing limitations and pitfalls of public datasets.

You are not learning "how to download data". You are learning how to **reason about data sources**.

Introduction to Pandas

What is Pandas?

Definition

Pandas is a powerful, open-source Python library for data analysis and manipulation.

Key Features:

- Fast and efficient DataFrame object
- Tools for reading/writing data between in-memory structures and various file formats
- Data alignment and integrated handling of missing data
- Reshaping and pivoting of datasets
- Label-based slicing, indexing, and subsetting
- Time series functionality

Why Pandas?

The name comes from **"Panel Data"** - an econometrics term for multidimensional structured datasets.

Core Data Structures in Pandas

Series

1-Dimensional labeled array

- Can hold any data type
- Like a column in a spreadsheet
- Has an index

DataFrame

2-Dimensional labeled data structure

- Like a spreadsheet or SQL table
- Columns can have different types
- Most commonly used structure

Example

```python
import pandas as pd
s = pd.Series([1, 3, 5, 7])
```

Example

```python
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})
```

Essential Pandas Operations

Reading Data:

```
df = pd.read_csv('file.csv')       # From CSV
df = pd.read_excel('file.xlsx')    # From Excel
df = pd.read_sql(query, connection) # From database
```

Basic Operations:

```
df.head()              # First 5 rows
df.info()              # Data types and memory usage
df.describe()          # Statistical summary
df['column']           # Select a column
df[df['col'] > 5]      # Filter rows
```

Data Manipulation:

```
df.groupby('col').mean()   # Group and aggregate
df.merge(df2, on='key')    # Join dataframes
df.pivot_table(...)        # Create pivot tables
```

Pandas for Financial Data

Why Pandas for Finance?

Pandas excels at handling financial time series data!

Financial Applications:

- **Time Series Analysis** - Stock prices, returns, volatility
- **Portfolio Management** - Asset allocation, performance tracking
- **Risk Management** - VaR calculations, correlation analysis
- **Economic Data** - GDP, inflation, interest rates
- **Data Integration** - Combining data from multiple sources

Key Strength

Pandas makes it easy to download, clean, analyze, and visualize financial data from remote sources!

Library Management

Library Conflicts and Versioning

Common Challenge

Different libraries may depend on incompatible versions of the same underlying packages.

Consequences:

- Unexpected errors
- Code breaking after updates
- Non-portable code across machines

Solution: Version Control

Explicitly declare library versions to ensure reproducibility!

Checking and Managing Versions

Check library versions:

```
import pandas as pd
import numpy as np
import matplotlib

print(pd.__version__)
print(np.__version__)
print(matplotlib.__version__)
```

Create requirements file:

```
pip freeze > requirements.txt
```

Install from requirements:

```
pip install -r requirements.txt
```

Python Virtual Environments

Best Practice

Use virtual environments to isolate project dependencies.

Benefits:

- Separate dependencies for different projects
- Avoid conflicts between library versions
- Easier to reproduce environments
- Better project organization

Tools for Virtual Environments:

- `venv` - Built into Python
- `conda` - Part of Anaconda distribution
- `virtualenv` - Third-party tool

Pandas-DataReader

What is pandas-datareader?

Definition

An extension to pandas designed to simplify accessing data from various financial and economic sources.

Key Features:

1. **Multiple Data Sources** - Yahoo Finance, FRED, World Bank, Alpha Vantage
2. **Seamless Integration** - Returns pandas DataFrames/Series
3. **Ease of Use** - Simple API with minimal setup
4. **Customizable Queries** - Specific time periods, tickers, categories

Installation

```
pip install pandas-datareader
```

Supported Data Sources

Source	Data Type
Yahoo Finance	Stock prices, indices, forex
FRED	Economic indicators (GDP, inflation, rates)
World Bank	Global economic indicators
Alpha Vantage	Stock market and forex (API key required)
Quandl	Various economic datasets

Focus of This Presentation

We will focus on **Yahoo Finance**, **World Bank** and **FRED** as they are:

- Completely free
- No API key required
- No registration needed

Instability Issues

Be aware that some data sources may occasionally change or become unavailable; this is a common issue when working with free public APIs.

Financial Data Structures

What are OHLC Bars?

Definition

A **BAR** is a fundamental unit of time-based price data used in technical analysis.

Four Key Components (OHLC):

1. **Open Price** - First trade in the period
2. **High Price** - Highest price in the period
3. **Low Price** - Lowest price in the period
4. **Close Price** - Last trade in the period

Additional Component:

- **Volume** - Total quantity traded

Types of Bars

Time-Based Bars:

- Fixed time intervals
- 1 minute, 1 hour, 1 day
- Most common type

Volume-Based Bars:

- New bar after specified units traded
- Adapts to market activity

Range Bars:

- Created after price moves a specified range
- Example: 10 points

Tick Bars:

- Bar after certain number of trades
- Example: 100 trades

Applications

Bars are used for charting, technical indicators, trading strategies, and data aggregation.

Types of Bars

Connection with ML for Finance (very important!)

The choice of bar type directly affects the statistical properties of the dataset and therefore the behavior of ML models.

Financial Data Are Not Generated for Machine Learning

A fundamental perspective

- Do not forget that financial data are generated by markets, institutions, and economic processes.
- They are **not** designed for statistical learning or prediction tasks.

Key consequences

- Non-stationarity and regime changes are the norm.
- Missing values, irregular frequencies, and revisions are common.
- Data availability often changes over time.

Implication for ML

- Data preparation and validation are often more important than model complexity.

Prices vs Returns: Why Transformations Matter

Raw prices

- Price levels are typically non-stationary.
- Trends and scale effects dominate statistical properties.

Returns

- Returns measure relative changes over time.
- Often closer to stationarity than prices.
- More suitable for statistical modeling and ML.

Typical transformation

$$r_t = \frac{P_t - P_{t-1}}{P_{t-1}} \quad \text{or} \quad r_t = \log P_t - \log P_{t-1}$$

Key message

- Feature engineering starts with appropriate data transformations.

Derived Features for ML

1. **Returns:**

$$r_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1 \tag{1}$$

2. **Log Returns:**

$$r_t^{log} = \log\left(\frac{P_t}{P_{t-1}}\right) = \log(P_t) - \log(P_{t-1}) \tag{2}$$

3. **Realized Volatility:**

$$\sigma_t = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(r_{t-i} - \bar{r})^2} \tag{3}$$

4. **Range-based Volatility (Parkinson, 1980):**

$$\sigma_P = \sqrt{\frac{1}{4\ln(2)}\left(\log\frac{H_t}{L_t}\right)^2} \tag{4}$$

Technical Indicators

Moving Averages:

- Simple Moving Average (SMA): $SMA_t(n) = \frac{1}{n}\sum_{i=0}^{n-1} P_{t-i}$
- Exponential Moving Average (EMA): weighted average with exponential decay

Momentum Indicators:

- Relative Strength Index (RSI)
- Moving Average Convergence Divergence (MACD)
- Rate of Change (ROC)

Volatility Indicators:

- Bollinger Bands
- Average True Range (ATR)

What Can Go Wrong with Public Financial Data

Common issues

- Changes in data definitions or methodologies.
- Missing observations and backfilled values.
- Survivorship bias (especially for equities and indices).
- Limited historical depth.

API-related issues

- Temporary outages or silent data changes.
- Inconsistent ticker conventions.
- Revisions without explicit versioning.

Practical lesson

- Always inspect, validate, and sanity-check financial data before modeling.

FRED Economic Data

FRED - Federal Reserve Economic Data

What is FRED?

A comprehensive online database managed by the Federal Reserve Bank of St. Louis providing free access to economic and financial data.

Key Features:

- Over 800,000 economic time series
- Historical data across various sectors and geographies
- Widely used by economists, researchers, and analysts
- Free and publicly accessible

Common Data Series:

- Interest rates (Treasury yields)
- Unemployment rates
- Inflation (CPI)
- GDP and economic indicators

Accessing FRED Data: Basic Example

Retrieve US unemployment rate:

```
from pandas_datareader import data as pdr

# Fetch unemployment data
unemployment = pdr.get_data_fred('UNRATE',
                                  start='2020-01-01',
                                  end='2023-01-01')
print(unemployment.head())
```

Explanation

- 'UNRATE' - FRED series ID for unemployment rate
- start, end - Define time range
- Returns a pandas DataFrame

Fetch 10-Year Treasury Yield:

```
import pandas_datareader as pdr
import datetime

start_date = datetime.datetime(2010, 1, 1)
end_date = datetime.datetime(2024, 2, 1)

# DGS10 = 10-Year Treasury Constant Maturity Rate
interest_rates = pdr.data.DataReader("DGS10", "fred",
                                     start_date, end_date)
```

Multiple series at once:

```
series_ids = ["DGS10", "DGS2", "DGS3MO"]  # 10Y, 2Y, 3-Month
rates = pdr.data.DataReader(series_ids, "fred",
                            start_date, end_date)
```

Analyzing Yield Spreads

Calculate the yield curve spread:

```python
# Calculate 10Y - 2Y spread
rates["Spread_10Y_2Y"] = rates["DGS10"] - rates["DGS2"]
# Plot the spread
import matplotlib.pyplot as plt
rates["Spread_10Y_2Y"].plot(figsize=(10, 3))
plt.title("10-Year Minus 2-Year Spread")
plt.axhline(0, color="red", linestyle="--")
plt.grid()
plt.show()
```

Economic Significance

A **negative spread** (inverted yield curve) has historically been associated with increased **recession risks**!

VIX Index - The Fear Gauge

What is VIX?

The VIX Index represents market's expectation of volatility over the next 30 days, derived from S&P 500 options prices.

Interpreting VIX Values:

- **Low VIX (¡15)** - Stability and investor confidence
- **Moderate VIX (15-25)** - Average market uncertainty
- **High VIX (¿25)** - Elevated uncertainty, fear

```
# Download VIX data from FRED
vix = pdr.DataReader('VIXCLS', 'fred',
                     '1990-01-01', '2023-01-23')
```

CPIAUCSL Dataset

Consumer Price Index for All Urban Consumers: measures inflation by tracking price changes of goods and services.

Fetch and transform CPI data:

```
# Download CPI data
cpi = pdr.DataReader('CPIAUCSL', 'fred',
                     '1950-01-01', '2023-01-23')
# Transform to year-on-year percentage change
cpi = cpi.pct_change(periods=12) * 100
# Drop NaN values
cpi = cpi.dropna()
```

Retrieve global GDP data:

```
from pandas_datareader import wb

# Download GDP data for United States
gdp_data = wb.download(indicator='NY.GDP.MKTP.CD',
                       country=['US'],
                       start=2010,
                       end=2022)

# Format for readability
gdp_data['NY.GDP.MKTP.CD'] = gdp_data['NY.GDP.MKTP.CD'].apply(
    lambda x: f"{x:,.0f}"
)
```

Yahoo Finance Data

What is yfinance?

A Python library to access financial market data from Yahoo Finance programmatically.

Installation:

```
pip install yfinance
```

Import:

```
import yfinance as yf
```

Available Data:

- Stock prices (historical and real-time)
- Company financials
- Market indices
- Forex data
- Options data

Downloading Stock Data

Single ticker:

```
# Download Apple stock data
apple = yf.download("AAPL", start="2020-01-01",
                    end="2022-01-01")
apple.head()
```

Multiple tickers:

```
tickers = ["AAPL", "MSFT", "GOOGL"]
data = yf.download(tickers, start="2020-01-01",
                   end="2023-12-31")
```

Returns a DataFrame with:

- Open, High, Low, Close prices
- Volume
- Adjusted Close

Data Analysis Examples

Extract closing prices:

```
close_prices = data["Close"]
```

Calculate daily returns:

```
daily_returns = apple["Close"].pct_change()
```

Resample to monthly data:

```
monthly_data = apple.resample("M").mean()
```

Plot closing prices:

```
import matplotlib.pyplot as plt
apple["Close"].plot(title="Apple Closing Prices")
plt.show()
```

Market Indices

Yahoo Finance ticker symbols for indices:

Index Name	Ticker Symbol
NASDAQ 100	^NDX
NASDAQ Composite	^IXIC
S&P 500	^GSPC

```
tickers = ["^NDX", "^IXIC", "^GSPC"]
indices = yf.download(tickers, start="2014-01-01",
                      end="2022-12-31")
```

Note

Index symbols are prefixed with caret (^) and are case-sensitive!

Forex Data

Download foreign exchange rates:

```
# Major Forex pairs
forex_pairs = ["EURUSD=X", "GBPUSD=X",
               "AUDUSD=X", "USDCAD=X"]
# Download last year of data
forex_data = yf.download(forex_pairs, period="1y",
                         interval="1d")
# Extract closing prices
forex_close = forex_data['Close']
```

Forex pair notation:

- `EURUSD=X` - Euro to US Dollar
- `GBPUSD=X` - British Pound to US Dollar
- Add `=X` suffix for forex pairs

Company Financials

Access financial statements:

```
apple = yf.Ticker("AAPL")
# Income Statement
income_stmt = apple.financials
# Balance Sheet
balance_sheet = apple.balance_sheet
# Cash Flow Statement
cash_flow = apple.cashflow
```

Available Data

- Quarterly and annual financial statements
- Key metrics and ratios
- Company information
- Analyst recommendations

Subsection 1

Options Data

Access option chains:

```
ticker = yf.Ticker("AAPL")
# Get available expiration dates
expiration_dates = ticker.options
# Fetch options for specific date
options = ticker.option_chain(expiration_dates[0])
# Separate calls and puts
calls = options.calls
puts = options.puts
```

Limitation

yfinance only provides **current** options data, not historical options data.

Subsection 2

Cryptocurrency Data

Yahoo Finance Ticker Symbols

Common Cryptocurrency Tickers:

- Bitcoin: `BTC-USD`
- Ethereum: `ETH-USD`
- Binance Coin: `BNB-USD`
- Cardano: `ADA-USD`
- Solana: `SOL-USD`
- Ripple: `XRP-USD`

Format: `[SYMBOL]-USD`

The `-USD` suffix indicates the quote currency (US Dollar)

Single Cryptocurrency:

```
# Download Bitcoin data
btc = yf.download('BTC-USD',
                  start='2020-01-01',
                  end='2024-01-01')

# Display first few rows
print(btc.head())

# Basic information
print(btc.info())
```

Returns a pandas DataFrame with columns: Open, High, Low, Close, Adj Close, Volume

Multiple Cryptocurrencies

Download Multiple Assets:

```python
# List of tickers
tickers = ['BTC-USD', 'ETH-USD', 'BNB-USD']
# Download data
data = yf.download(tickers,
                   start='2020-01-01',
                   end='2024-01-01')
# Access specific cryptocurrency
btc_close = data['Close']['BTC-USD']
# Or download separately
crypto_dict = {}
for ticker in tickers:
    crypto_dict[ticker] = yf.download(ticker,
                                       start='2020-01-01',
                                       end='2024-01-01')
```

Different Time Intervals

Available Intervals:

```
# Daily data (default)
btc_daily = yf.download('BTC-USD',
                        start='2023-01-01',
                        interval='1d')

# Hourly data
btc_hourly = yf.download('BTC-USD',
                         start='2023-01-01',
                         interval='1h')

# Other intervals: 1m, 2m, 5m, 15m, 30m, 60m, 90m,
# 1h, 1d, 5d, 1wk, 1mo, 3mo
```

Note: Shorter intervals have limited historical availability (typically 7-60 days)

Using Ticker Object

More Control with Ticker Object:

```
# Create ticker object
btc_ticker = yf.Ticker('BTC-USD')
# Get historical data
hist = btc_ticker.history(start='2020-01-01',
                          end='2024-01-01')
# Get metadata
info = btc_ticker.info
print(f"Market Cap: {info.get('marketCap')}")
print(f"24h Volume: {info.get('volume24Hr')}")
# Get all historical data
all_data = btc_ticker.history(period='max')
```

Handle Missing Data:

```
# Check for missing values
print(btc.isnull().sum())
# Forward fill missing values
btc_filled = btc.fillna(method='ffill')
# Or drop missing values
btc_clean = btc.dropna()
# Check for duplicates
print(btc.index.duplicated().sum())
# Remove duplicates
btc_clean = btc[~btc.index.duplicated(keep='first')]
```

Missing data in crypto is rare but can occur during system maintenance or extreme volatility

Calculate Different Types of Returns:

```
# Simple returns
btc['Returns'] = btc['Close'].pct_change()
# Log returns
btc['Log_Returns'] = np.log(btc['Close'] /
                            btc['Close'].shift(1))
# Multi-period returns
btc['Returns_7d'] = btc['Close'].pct_change(periods=7)
# Remove first row (NaN)
btc = btc.dropna()
# Statistics
print(f"Mean return: {btc['Returns'].mean():.4f}")
print(f"Volatility: {btc['Returns'].std():.4f}")
```

Export to Different Formats:

```
# Save to CSV
btc.to_csv('btc_data.csv')
# Save to Excel
btc.to_excel('btc_data.xlsx')
# Save to HDF5 (efficient for large datasets)
btc.to_hdf('crypto_data.h5', key='btc', mode='w')
# Save to pickle (preserves data types)
btc.to_pickle('btc_data.pkl')
# Read back
btc_loaded = pd.read_csv('btc_data.csv',
                         index_col=0,
                         parse_dates=True)
```

Common ML Tasks with Crypto Data

1. **Price Direction Prediction (Classification):**
   - Binary: Up/Down
   - Multi-class: Strong Up / Weak Up / Neutral / Weak Down / Strong Down
2. **Price Level Prediction (Regression):**
   - Next period price
   - Multi-step ahead forecasting
3. **Volatility Forecasting:**
   - GARCH family models
   - Neural network approaches
4. **Anomaly Detection:**
   - Flash crashes
   - Market manipulation
   - Unusual trading patterns

Practical Applications

Stock Correlation Analysis

Calculate correlation between stocks:

```
# Download data for multiple stocks
tickers = ["AAPL", "MSFT", "GOOGL"]
data = yf.download(tickers, start="2020-01-01",
                   end="2023-12-31")
# Calculate returns
returns = data["Close"].pct_change()
# Compute correlation matrix
correlation = returns.corr()
print(correlation)
```

Use Case

Understanding correlations helps in portfolio diversification and risk management.

Calculate weighted portfolio returns:

```
# Define allocation weights
weights = [0.4, 0.4, 0.2]  # AAPL, MSFT, GOOGL
# Calculate portfolio returns
portfolio_returns = (returns * weights).sum(axis=1)
# Calculate cumulative returns
portfolio_cumulative = (1 + portfolio_returns).cumprod()
# Plot performance
portfolio_cumulative.plot(title="Portfolio Performance")
plt.show()
```

Application

Track how a weighted portfolio performs over time. This is a simplified example ignoring transaction costs, rebalancing constraints, and market frictions.

Saving Data to Files

Export data to CSV:

```
import os
# Define path and filename
path_name = '.'
file_name = 'stock_data.csv'
file_path = os.path.join(path_name, file_name)
# Save DataFrame to CSV
data.to_csv(file_path)
print(f"File saved to {file_path}")
```

Other formats:

```
data.to_excel('data.xlsx')  # Excel
data.to_json('data.json')   # JSON
data.to_parquet('data.parquet')  # Parquet (efficient)
```

Conclusions

Lesson 1.2 — Key Concepts Recap

- Pandas provides a unified framework for:
  - loading data from heterogeneous sources,
  - aligning time series,
  - handling missing values and metadata.
- Financial data come in structured forms: prices, returns, OHLC bars, volumes, macro series.
- The choice of data representation already affects the statistical properties of the problem.

Lesson 1.2 — Financial Takeaways

- Financial data are **not designed for machine learning**.
- Public data sources can suffer from:
  - missing observations,
  - revisions and backfilling,
  - survivorship bias,
  - changing definitions over time.
- Data inspection and sanity checks are mandatory, not optional.

Key lesson: bad data pipelines invalidate even the best models.

Lesson 1.2 — What You Should Know Now

After this lecture, you should be comfortable with:

- downloading and handling financial time series in Pandas,
- distinguishing prices from returns and understanding why it matters,
- identifying structural issues in real-world financial datasets,
- preparing raw data for the next stages of the ML pipeline.

From Data Gathering to Data Pre-Processing

- Lesson 1.2 focused on **where data come from** and how they are collected.
- However, raw financial data are rarely ready for modeling.
- Seemingly innocent choices made after data collection can dramatically alter:
  - risk estimates,
  - tail behavior,
  - out-of-sample performance.

Looking Ahead — Lesson 1.3

In the next lecture (Lesson 1.3 – Data Pre-Processing), we will see that:

- preprocessing is **part of the model**, not a technical fix,
- missing data can be informative in finance,
- encoding and scaling choices embed economic assumptions,
- careless preprocessing is a primary source of backtest failure.

Transition message: Data, preprocessing, and modeling form a **single financial pipeline**.