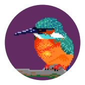
Machine Learning for Trading

Notes for *CS 7646*



George Kudrayvtsev

george.k@gatech.edu

Last Updated: June 4, 2019

Listings

Part I

Manipulating Financial Data

Python for Finance

Python has emerged as a dominant language in financial analysis in recent years due to its accessibility and the development powerful libraries such as **`pandas`** and **`numpy`** which enable highly-optimized manipulation of stock market data. Many of the statistics we'll talk about here have some accompanying snippets that describe how calculating that statistic would look in **`pandas`**. This is especially useful for various projects in *CS 7646* as there are many equally-correct ways to calculate certain values, but a particular calculation is expected.

A convenient way to compare price data for stocks is to normalize them to all start at 1.0. In **`pandas`**, this is very simple: given a dataframe `df`, just divide by its first price:

$$df = \frac{df}{df[0]}$$

1.1 Global Statistics

Given a selection of stock market data, there are a number of global statistics we can calculate that can give us insights to trends in a particular stock over time and allow us to compare performance of various stocks in a portfolio.

These include things like the **rolling mean**, the **rolling standard deviation**, and more.

Rolling Statistics in `pandas`

There are a number of functions in the **`pandas`** library that give us global statistics. These aren't members of the **`DataFrame`** class; rather, we pass in a **`DataFrame`** and some additional parameters that specify a few details about the computation we want.

- `stats.moments.rolling_mean` — notable parameters include the window size and the number of minimum necessary observations

1.1.1 Bollinger Bands®

Invented by John Bollinger in the 1980s, **Bollinger bands** are a measurement of a stock's volatility. By leveraging the rolling standard deviation—specifically, 2σ above and below the global mean—we can get a sense of the volatility based on when the current stock intersects the bands.

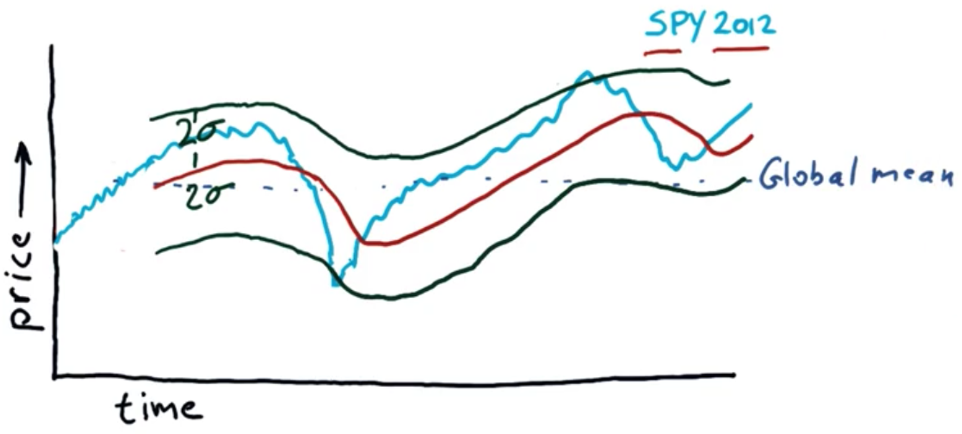


Figure 1.1: A visual demonstration of Bollinger bands® applied to $SPY. In this particular example, buying at the dip and selling at the peak would've rendered great profits, but there are plenty of examples in which this isn't an effective strategy.

A point of intersection, such as the dip in Figure 1.1, can indicate a trading opportunity. A double intersection (as in, it dips through the lower band and back up through it) with -2σ suggests a buy, whereas a double intersection with $+2\sigma$ suggest a sell. We won't discuss the effectiveness of this method as an actual trading strategy until a later chapter, but it's an important demonstration of the power of these simple global statistics.

1.1.2 Daily Returns

One of the most important statistics used in financial analysis is **daily returns**. It's very straightforward: how much did the price go *up* or *down* on a particular day? The calculation is very simple: the daily return at time t is based on the price that day divided by the prices on the previous day.

$$\text{return}_t = \frac{\text{price}_t}{\text{price}_{t-1}} - 1 \tag{1.1}$$

Listing 1.1: Given the portfolio value (either as a `DataFrame`, `Series`, or other `pandas`data structure) in `port_val`, we can find the daily returns (as a percentage) rather easily by dividing the portfolio value by itself, offset by one day.

```
daily = (port_val[1:] / port_val[:-1].values) - 1
```

For example, if a stock was at \$100 on Monday and \$110 on Tuesday, the daily return would simply be: $^{110}/_{100} - 1 = 10\%$. We can plot the daily return for a stock over time to get a sense of its growth trend; it will generally zig-zag around zero, though its mean will be above zero if the stock price is increasing over time and below zero otherwise.

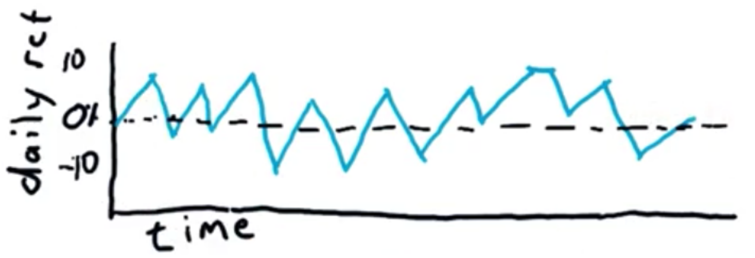


Figure 1.2: A potential daily return graph for an arbitrary stock.

Daily returns aren't very useful on their own relative to other global statistics like the rolling mean. Instead, they shine when *comparing* stocks: we can compare the daily returns of, for example, \$AAPL[1] against the S&P 500.

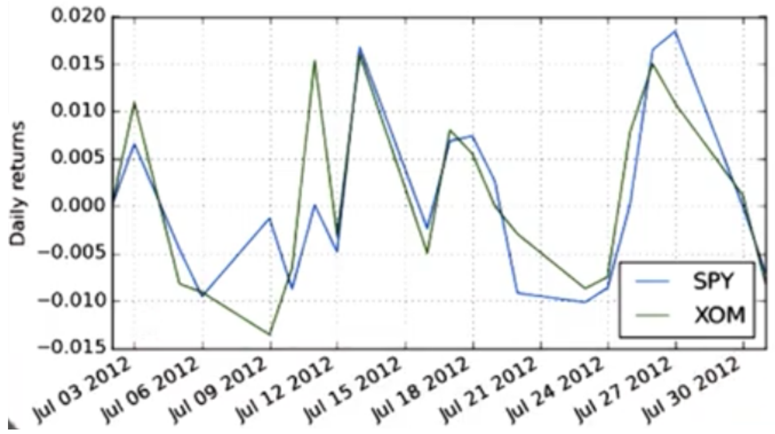


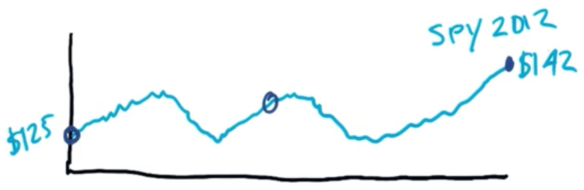
Figure 1.3: A comparison of the daily returns of Exxon Mobile (\$XOM) to the daily returns of the S&P 500 (\$SPY). We can see that it matches the ups and downs of the stock market in general quite closely.

1.1.3 Cumulative Returns

Another important statistic is **cumulative returns**. This is a measure of the stock price over a large period of time starting from some t_0. For example, you might say that the S&P has grown 10% in 2012; that would be its cumulative return (for t_0 = January 1st 2012). The calculation is almost identical to (1.1), except it uses a fixed initial time:

$$\text{return}_t = \frac{\text{price}_t}{\text{price}_{t_0}} - 1 \tag{1.2}$$

[1] Generally speaking, we use the notation \$SYM to indicate a stock symbol.



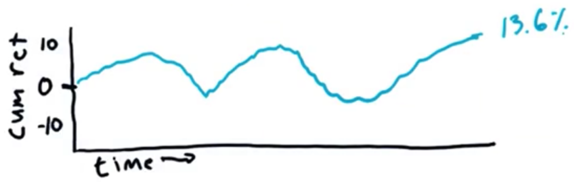


Figure 1.4: An example of calculating and subsequently charting cumulative returns for $SPY in 2012.

Listing 1.2: As usual, we assume an existing portfolio value calculation in `port_val`. Note that the syntax `.iloc[idx]` is a way of directly accessing a particular row in a **pandas DataFrame** or **Series** object.

```
cumulative = (port_val.iloc[-1] / port_val_value.iloc[0]) - 1
```

1.2 Fixing Bad Data

We need to work around a number of (invalid) assumptions about financial data in order to manipulate it successfully. Many people assume stock ticker data is perfectly recorded minute-by-minute and has no gaps or missing data. The harsh reality is that the data is actually an amalgamation from multiple sources. A particular stock can trade at different (though similar) prices on different exchanges (like the NYSE or the NASDAQ).

Naturally, data can also be missing. Low-volume stocks that don't have a lot of trading activity might not have any data for a particular day; similarly, stocks come in and out of existence as the company (d)evolves.

The best way to work around gaps in a stock's price data is to **fill forward** then **fill backward**. To fill forward is to fill the price data from the last known price until trading activity resumes. This is far better than **interpolation** because interpolation is a "prediction" about the stock's price in the future which is a big no-no when it comes to analysis. After filling forward, there may still be an initial gap; for this, we fill backwards.

Fixing Bad Data in pandas

The key to fixing bad data in **pandas** is **DataFrame.fillna**, typically called with the parameter `method='ffill'` (or `'bfill'` for backwards filling).

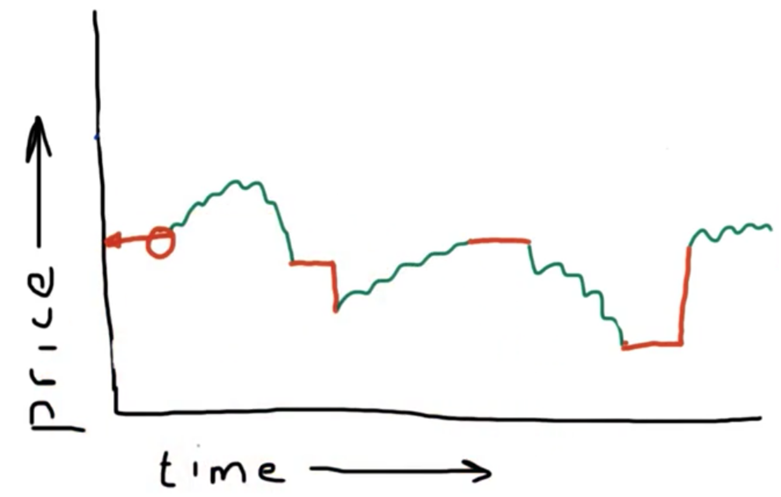


Figure 1.5

1.3 Graphing Financial Data

We'll be discussing **histograms** and **scatterplots** in this section, and we'll start by taking a closer look at daily returns.

1.3.1 Histograms

On their own, daily return graphs don't offer us much. What we can do to get a better insight on a stock's behavior is to turn it into a histogram, which is essentially a bar graph that relates values to the frequency of their occurrences. For example, if there were 3 days in which you had 1% daily returns, you would have a 3-unit tall bar for $x = 1\%$. An important factor is choosing the granularity of the histogram (i.e. how many "buckets" of values we want to track).

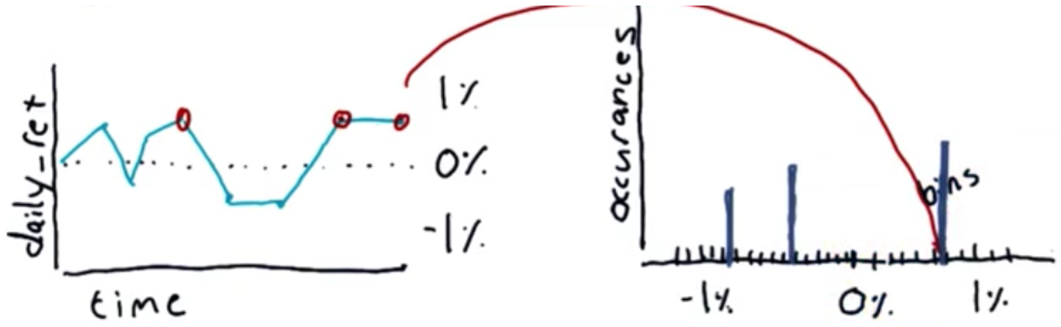
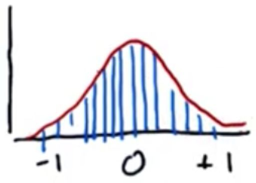


Figure 1.6: An example demonstrating how a daily return graph can be turned into a discretized histogram.

Suppose we looked at $SPY over many years and created a histogram, normalizing it to $[-1, 1]$. What would it look like? Interestingly enough, it results in a **normal distribution** (also called a Gaussian distribution).



With our neat little histogram, we can now compute interesting values like the mean (in the case of a normal distribution, this is its peak) and the standard deviation (which gives us a sense of how often and how far things deviate from the mean). Another important thing to note is the measure of **kurtosis**. Kurtosis shows us how much our histogram would deviate from an actual perfect normal distribution.

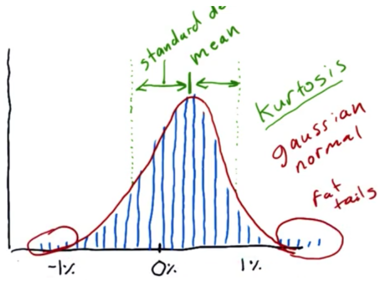
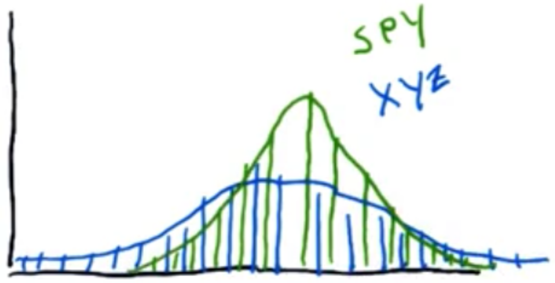


Figure 1.7

This is especially notable at the tails of the distribution; we can see "fat tails" in the histogram in Figure 1.7. This tells us there are frequently large outliers relative to a perfect normal distribution, indicated by a positive kurtosis.

Kurtosis has real-world consequences. If we assume that our models are perfect normal distributions, we mathematically disregard inconsistencies in the data. This is a big mistake that is amplified with the amount of outliers actually present in the data. Kurtosis contributed to the Great Recession of 2008: banks built bonds based on mortgages whose returns they assumed were normally distributed. Thus, they claimed that the bonds had a very low probability of default. This proved to be false as massive numbers of homeowners defaulted on their loans, precipitating the economic collapse.

Comparing the histograms of stocks is an important part of financial analysis. For example, consider the two histograms for $SPY and $XYZ:



We can see that $XYZ has a lower return and a higher volatility than $SPY: its mean is lower and the standard deviation is larger. There is a higher spread of data, so there is more variation in daily returns, which is the definition of volatility.

1.3.2 Scatter Plots

Scatter plots let us visualize differences between stocks at particular points in time by plotting them against each other. For example, in Figure 1.8 we can see the darkly circled area, which corresponds to both $SPY and $XYZ having a positive daily return, but $XYZ's is higher.

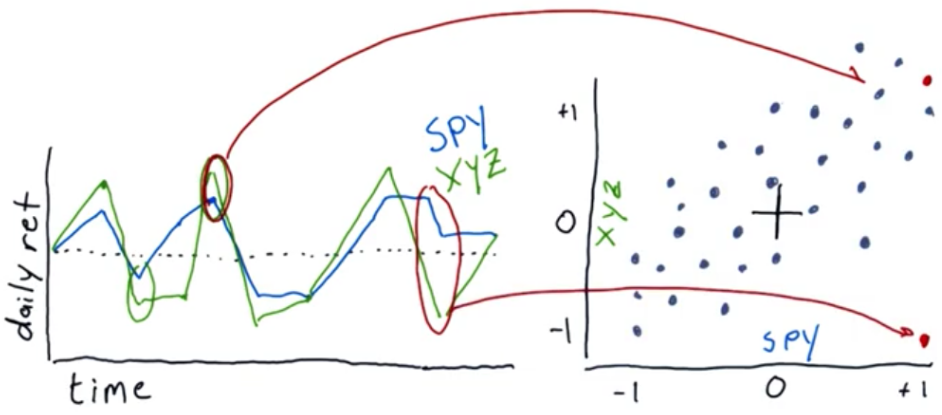


Figure 1.8: Turning the daily return plots of two stocks into a scatter plot.

Given enough data, we can typically see a trend. In Figure 1.8, we can see that there appears to be a linear relationship, however its quite loose as the dots are relatively spread apart. We can use linear regression[2] to fit a line to the plot. The **slope** of that line—referred to in the financial world as β when comparing to a general market-tracking stock like $SPY—describes how the stock reacts to the market. If $\beta = 1$, it means that when the market goes up 1%, that stock will also go up 1%.

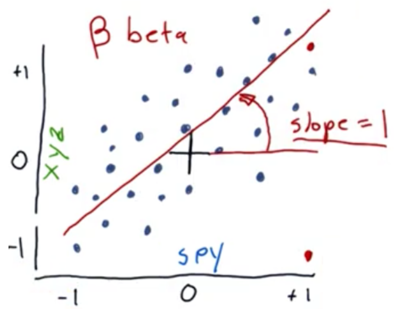


Figure 1.9: The slope of a line indicates a relationship between market trends and a specific stock (when comparing to $SPY).

Notice that the line of best fit in Figure 1.9 is *above* the origin. This value (the y-intercept of the line) known as α describes how much better a stock performs relative to the market on average. In this case, $XYZ often outperforms the market slightly.

Another important metric on the scatterplot is **correlation**. This is a measure of how tight the points are to the line of best fit, in the range $[0, 1]$. In Figure 1.9, the dots are typically

[2] We won't cover linear regression for now. It should be familiar to most of us, but if not, you can think of it as a mathematical way of finding the "line of best fit" that we could likely draw fairly intuitively.

fairly far from the line,[3] which means there is a **low** correlation. It's critical to keep in mind that slope $\neq$ correlation: it's just a measure of how tightly the dots fit a line of a particular slope.

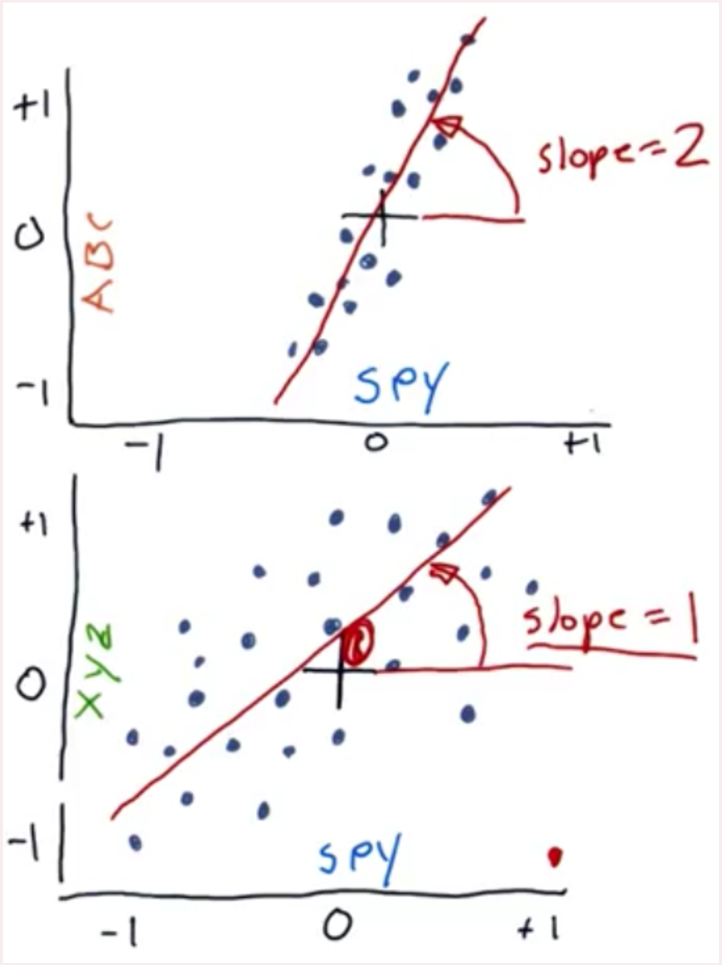
Understanding Check: Correlation vs. Slope

Below are scatter plots for two stocks in relation to the S&P 500: \$ABC and \$XYZ. Which of the following is true about \$ABC?

(a) \$ABC has a higher β (slope) and a lower correlation.

(b) \$ABC has a lower β and a higher correlation.

(c) \$ABC has a higher β and a higher correlation.



Answer: (c)

[3] Mathematically speaking, this would be a measure of the total error using something like the distance.

1.4 Portfolios

Let's expand beyond the realm of single-stock-analysis. We'll start out with a "buy-and-hold" strategy for our portfolio in which we have some initial principal investment spread out over a number of stocks; we will observe its behavior and performance over time.

Suppose our portfolio has the following setup:

- Principal starting value: $1,000,000 *(I wish. . .)*
- Investment period: Jan. 1st 2009 to Dec. 31st 2011
- Stocks: $SPY, $XOM, $GOOG, $GLD
- Allocation ratios:[4] $[0.4, 0.4, 0.1, 0.1]$

To calculate our daily portfolio value, we can use the following formula (assuming an initial data frame `prices` which has daily prices for our stocks):

$$\text{portfolio} = \sum \underbrace{\underbrace{\text{ratio} * \text{start}}_{\text{per-stock principal}} * \underbrace{\frac{\text{prices}}{\text{prices}[0]}}_{\text{normalized prices}}}_{\text{daily position}} \tag{1.3}$$

Let's assume now we have a data frame `port_val` which is the list of portfolio values over time and the data frame `daily_rets` which is its daily returns. We can now calculate some statistics that everyone wants to know about a portfolio: the cumulative and average daily return, the standard deviation of the daily return, and the **Sharpe ratio**.

The cumulative return is simply the percentage of total portfolio growth, so its final value minus the initial value. The next two we have already discussed. The Sharpe ratio gives us an insight about our returns in the context of risk. Finance folk often associate risk with the standard deviation as that represents a stock's volatility (hence why cryptocurrency investment is risky: your coin could be worth $100 tomorrow or $1000 or $3).

Listing 1.3: In this snippet, we assume the existence of a `DataFrame` of stock data called `prices` that's formed as we've discussed: each row is labeled by a date and each column corresponds to a $SYMBOL. We'll also assume that `allocs` is our ratio of allocations and that `principal` is our initial (total) investment.

```
normalized = prices / prices.iloc[0]     # normalized prices
allocated = normalized_prices * allocs   # prices scaled by allocation ratio
positions = allocated * principal        # per-stock investment
portfolio = positions.sum(axis=1)        # portfolio value every day
```

Naturally, given two stocks with similar returns, we'd choose the one with less volatility; similarly, given similar volatility, we'd choose higher returns. What about a stock that is

[4] This means that 10% of our funds are invested in Google stock at the start.

more volatile but has higher returns than another? This typically involves a bit of a "gut feeling," but we want to do better than that.

That's where the **Sharpe ratio** comes in: it gives us a **risk-adjusted** return. All else being equal, it considers low levels of risk and high returns to be good. The Sharpe ratio additionally considers the risk-free rate of return which is critical when comparing financial strategies. A risk-free investment is something like an FDIC-insured bank account or a short-term treasury, which are (more-or-less) guaranteed to grow at a particular percentage. If your investment strategy doesn't outperform a risk-free return rate, you might want to reconsider it.

Given the portfolio return R_p, the risk-free return rate R_f, and the standard deviation of R_p (i.e. the volatility or risk), we can formulate the Sharpe ratio as such:

$$\frac{R_p - R_f}{\sigma_p}$$

Notice that as volatility goes up, the ratio shrinks. Similarly, as the return goes up (sans the risk-free return), the ratio grows. The actual formula is a little more complicated, but describes the same pattern (here E is the expectation):

$$S = \frac{\mathrm{E}[R_p - R_f]}{\mathrm{std}[R_p - R_f]} \tag{1.4}$$

The "risk-free rate" can be taken from LIBOR, the 3-month treasury bill, or just set as 0% which has been an accurate approximation lately. Instead of having to constantly check and update the aforementioned values every day, we can approximate the daily risk-free rate by taking the 252th root of the annual rate (remember, there are 252 trading days in a year): $R_f = \sqrt[252]{1.0 + R_{f_{\text{annual}}}} - 1$.

The Sharpe ratio can vary wildly depending on how frequently its sampled. It was originally envisioned as an **annual** measure. If we want to sample more frequently, we need to add an adjustment factor k to make it all work out, where k is the square root of the number of samples per year. For an R_p that represents daily portfolio returns, then, $k = \sqrt{252}$. Similarly, $k = \sqrt{52}$ for weekly samples, etc.

Example 1.1: Sharpe ratio

Suppose we've had our portfolio for 60 trading days. Our average daily return is 1/10% per day (0.001%, or 10 **basis points**[a]). Our daily risk-free return is 2 basis points, and our volatility (σ) is 10 basis points.

What's the Sharpe ratio of this strategy?

Since we are operating in days, we know $k = \sqrt{252}$. Then, we just use (1.4) directly:

$$S = \sqrt{252} \cdot \frac{10 - 2}{10} = 12.7$$

[a] Basis points are often called “bips,” and their unit is bps.

1.5 Optimizers

The engineers or mathematicians among us may already be familiar with the notion of **optimization problems**, which are a class of mathematical problems that require us to solve for a value that maximizes (or minimizes) an equation. In this section, we'll be discussing optimizers in code that can solve these types of problems in a financial context.

An optimizer can:

- find the minimum values of a function,
- build parameterized models from data, and
- refine stock allocations in a portfolio

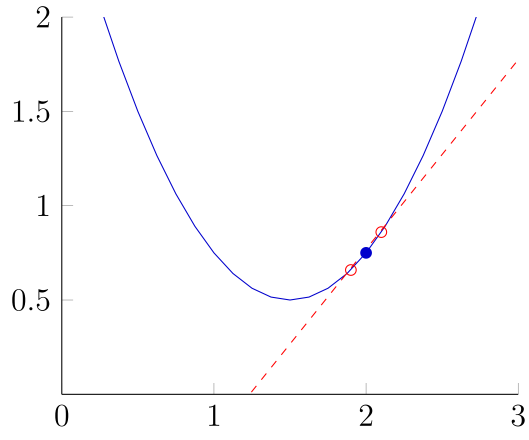
Using a pre-built optimizer is fairly simple. All we need to do is provide a function to minimize (such as $f(x) = x^2 + 5$) and an initial guess.

1.5.1 Using an Optimizer

Let's walk through a simple minimzation example. Suppose we're given the function (plotted below):

$$f(x) = (x - 1.5)^2 + 0.5$$

How does the minimizer work? Suppose our initial guess was $x = 2$. At that value, it uses the neighbors (like $x = 1.9$ and $x = 2.1$) to approximate a local slope and “marches downhill” (this is gradient descent). It repeats the process with another value that is down this slope, eventually converging to the minimum value $x = 1.5$.



Gradient descent is just one of a myriad of ways to minimize a function; it's an easy configurable parameter to `scipy.optimize.minimize`.

Unfortunately, this isn't a bulletproof method. Functions with many **local** minima and flat areas are tough for minimizers to solve. Specifically, **convex functions** are the easiest for optimizers to solve. Formally-speaking, a convex function is one in which you can draw a line segment between *any* two points and that line would not intersect the function. A convex function must have just one minimum and not have any problematic flat areas.

1.5.2 Building Parameterized Models

Our goal in this section is to build a parameterized model from data.

Those of you who have taken a course on computer vision may have covered a similar topic: instead of treating a line as an equation $y = mx + b$ in which m and b are known, and we can plug in arbitrary x values to get their corresponding ys, we instead already *have* a set of $\{(x_0, y_0), (x_1, y_1), \ldots\}$ data points and wish to solve for m and b.

For convenience and generalization in code, we'll use c_i to refer to our parameters, so $c_1 = m$ and $c_2 = b$ in our aforementioned parameterized model of a line.

Error Metrics

If the data fit our model perfectly, solving the parameters would be trivial. It's unlikely, though, that all of our points would be perfectly colinear, for example. Thus, we need a way to evaluate the quality of a "candidate model."

Given our understanding of minimizers, we want to reframe the "model of best fit" problem into something a minimizer can process. A common go-to error metric is simply the total vertical error from the points to the line.

Our error function in this case is the vertical distance between point (x_i, y_i) and the line, for all points. $mx_i + b$ is the "real y" at that x value via the true equation of the line:

$$E = \sum_{i=1}^{n} (y_i - (mx_i + b))^2$$

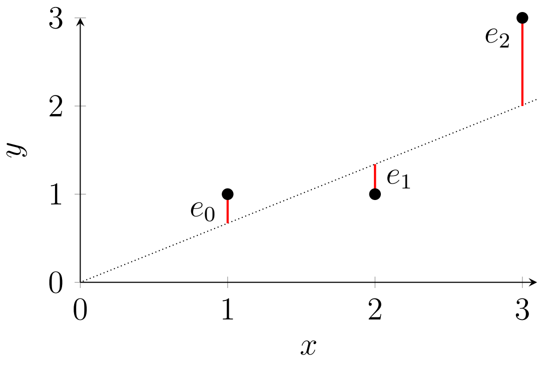


Figure 1.10: Fitting a line to a set of points, with the errors in red.

We can feed the minimizer the above error function applied to our simple line function: $f(x) = c_1 x + c_2$, resulting in the parameters that best fit our data.

This error function isn't perfect: it over-emphasizes the error of steep lines because it only considers the y component, as seen in Figure 1.11. A better function would be distance, or *perpendicular* error, but we won't get into that here because apparently it's too advanced of a topic for a graduate-level course on financial algorithms.

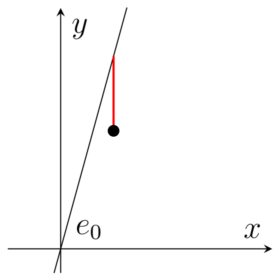


Figure 1.11: Measuring vertical error overemphasizes errors in steep lines, but don't let that distract you from the fact that the Warriors blew a 3-1 lead in the 2016 Finals.

Listing 1.4: We can find the best-fitting model to a set of observed data by feeding the model function into SciPy's optimization toolkit, specifically `scipy.optimize.minimize`.

```
import numpy as np

def error_function(line, data):
  """ Computes the error between a line (m, b) and observed data.

  line:   a 2-tuple (c0, c1) where c0 is the slope and c1 is the y-intercept
  data:   a 2D array where each row is an (x, y) point
  """
  return np.sum((data[:, 1] - (line[0] * data[:, 0] + line[1])) ** 2)
```

n Dimensions

This process is extensible to n-dimensional data fairly simply. It does rely on the key assumption that the function can be modeled as an n^{th}-degree polynomial. That's okay for our current purposes, though, because we'll be applying this technique to optimize a portfolio which can be viewed as a simple function of allocation ratios applied to prices.

Listing 1.5: We can find the best-fitting polynomial function to a set of observed data much like in Listing 1.4.

```
import numpy as np

def error_function(C, data):
  """ Computes the error between a line (m, b) and observed data.

  C:      np.poly1d object (or 1D array) representing polynomial coefficients
  data:   2D array where each row is an (x, y) point
  """
  return np.sum((data[:, 1] - np.polyval(C, data[:, 0] + line[1])) ** 2)
```

1.5.3 Portfolio Optimization

Given a set of funds (i.e. stock symbols), assets (i.e. an initial amount of money to invest), and a time period (e.g. 1 year), find the ratio of assets to funds that maximizes performance. "Performance" here is the variable that depends on your investment strategy. If all you care about is growth, for example, you'd want to maximize cumulative returns (see Listing 1.2).

In our running example, we'll optimize the portfolio's Sharpe ratio instead. The set of variables we wish to optimize is the allocation ratios across our chosen stock symbols. For example, we might want 20% of our funds to $AAPL, 10% to $GLD, and the rest to $XOM for a nice, diversified portfolio. Of course, this might not be the optimal ratio. Framing this problem is a straightforward process:

- Provide a function for **`minimize()`** to work with.

  We're trying to find the optimal allocations across a fixed set of stocks. Thus, in this function we'd need to calculate the Sharpe ratio for a given "best allocation" guess. Because we're using a *minimizer*, though, we need to multiply our result by -1.

- Provide an initial guess for our allocation ratios.

  For this parameter, many values are likely to result in the same resulting optimal ratio. It might make sense to pick your true best guess, your ideal resulting ratio, or just a uniform distribution (like 33% across our above 3 stocks).

- Set up our ranges and constraints.

  Obviously, we don't want to try any input values that result in allocations beyond 100% of our funds. In other words, our constraint must require that the sum of the ratios is 1. Similarly, we can restrict the possible inputs to our function to fall in $[0, 1]$.

- Call the optimizer. *(I don't think this bears more explanation.)*

Part II

Computational Investing

ETFs Mutual Funds Hedge Funds

Part III

Learning Algorithms for Trading

The third part of this topic will cover applying machine learning algorithms to financial data in order to create models that give us insights about stock behavior. We can use these insights to (try to) predict future price data and make trading decisions based on those predictions. Financial models have existed for a long time, but using machine learning lets us create "data-centric models" in which the evidence speaks for itself; there is no longer a need to subjectively interpret the data.

Let's talk about machine learning in broad terms at first, then get into specific techniques and algorithms. What problem does machine learning solve? Well, given an observation input X (like the last year's daily returns of a stock, for example), we can feed it into a **model** and get some output prediction Y (like the *next* set of daily returns). The model, of course, is created from massive amounts of data fed into a machine learning algorithm.

$$X \longrightarrow \boxed{\text{model}} \longrightarrow Y$$

Supervised Regression Learning

The first part of this chapter will focus on **supervised regression learning**: **supervised** by a sequence of example predictions $(\mathbf{x_i}, y_i)$, we will **learn** a numerical prediction (the **regression**, a poor naming choice). Some examples that fall under this category of machine learning algorithms that we'll be discussing soon include linear regression, k-nearest neighbor (or kNN), decision trees, and decision forests.

In terms of stock data, the most pertinent "prediction" is a future price. This allows us to make decisions on top of that price data rather than restricting our model to making strategy-specific decisions (like "Buy!") directly.

Remember, we can have many feature vectors $\mathbf{x}_i$ for a particular stock. Typically, we have a **`pandas DataFrame`** for each feature in which each column is the data for a particular stock and each row is the data at a specific point in time. When we extend this to many features, we can think of it as a third dimension to our data in which the "depth" is the feature number. What is our $\mathbf{y}$, then? Well, as we've already established, it will be price. Specifically, we can create an association between a particular feature and a "future" price. We say "future" because we actually have historical price data. So we could look at the Sharpe ratio for $IBM on January 1st (this is our x_i) and correlate it with its price on January 5th (this is our y_i).

The "art" of model creation is picking these predictive factors—the more accurate your resulting predictions, the more someone is probably willing to pay to use your model.

Backtesting How accurate can these models *really* be? The stock market is a fickle beast and some argue it runs more on emotion than reason. We need a way to test the effectiveness of our model without losing tons of money in the process. This is where **backtesting** comes in: we apply our model on a different period of time than the one it was trained on and see how accurately its predictions match the actual stock behavior.

This is somewhat akin to splitting the training data into a "training" and "test" set that we'll dive into further soon. Given a model based on a set of training data, we can place a set of orders for a future point in time. We then use the historical data to see how those orders would've performed, plotting the performance of our fake portfolio over time (in terms of any of the statistics we learned about in chapter 1). We can repeat this for as long as we have historical data for, seeing how our model would've "really" performed over that period of time.

Figure 2.1: Our machine learning model associates a 3-dimensional array of feature values for a variety of stocks with their (historically) “future” price data.

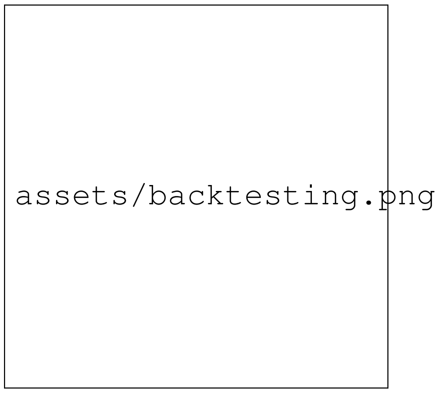


Figure 2.2: Describing the process of backtesting our model over historical price data, plotting the results over time.

With a cursory overview out of the way, let’s dive into our first set of machine learning methods that will make us teh big bux.

2.1 Regression

Don’t let the spooky term “regression” scare you: it’s just a numerical model. It outputs a number (its prediction) based on a bunch of input numbers.

2.1.1 Linear Regression

Let’s take a look at a *parametric* method first. We’ve seen the concept of a parametric model before: in Building Parameterized Models, we chose parameters to our line (m, the slope, and b, the y-intercept from the familiar equation of a line, $y = mx + b$) that best fit our data set. We defined "best fit" as minimizing the total vertical error between the line and the points. This is called **linear regression**.

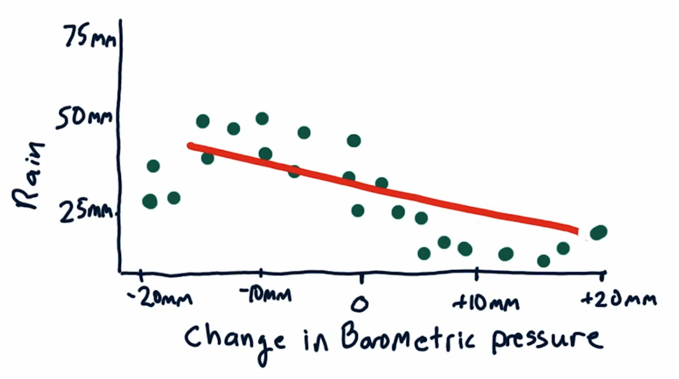


Figure 2.3: Using linear regression to fit a line to a data set mapping the correlation between changes in barometric pressure and rainfall. Notice that a higher-dimensional polynomial might have fit this data better, but this is the best we could do with our model, which was a line.

Once we've found our best-fitting model parameters, calculating predictions is just a matter of plugging our new input values into the model. We can throw the initial data away. The problem, as you can see in Figure 2.3, is that the real world seldom follows a linear model. Furthermore, it's hard to say what degree of a polynomial you'd need to best fit a model. It might be better to let the data speak for itself, which would lead us to. . .

2.1.2 k-Nearest Neighbor

This data-centric approach (called an *instance* method, in contrast with the *parametric* method earlier) uses instances of the data points to create an approximate prediction based on the distance of a novel input to 1, 2, or k of its nearest neighbors in the data. In **k-nearest neighbor**, we just take the mean of the y values. This method actually works surprisingly well for $k > 1$, but isn't always ideal because it requires storing the data which might have many dimensions and take up a lot of space.

If we actually take the *weighted* mean of the y values based on the distance of the neighbors to the new input, that's actually called **kernel regression**.

Example 2.1: Choosing an Approach

How should we decide when a parametric approach like linear regression should be preferred to a situation over an instance or data-centric approach like kNN?

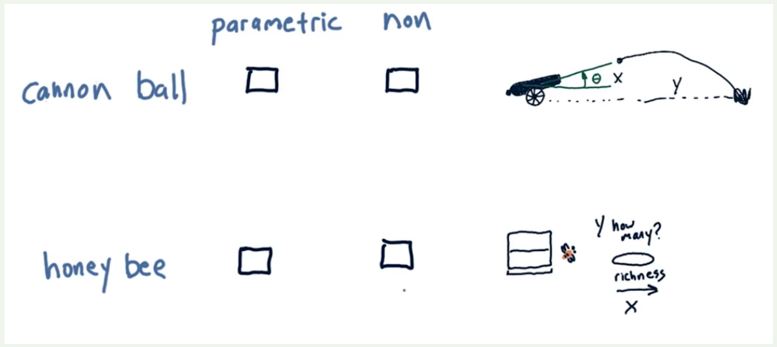
Suppose we have two situations we'd like to have a model for:

- A projectile is fired from a cannonball, and we'd like to predict where it lands.

Our observable is θ, the angle of the cannon, and its landing point.

- A hive of honey bees is attracted to a food source; we'd like to predict how many bees will go to a particular food source given its richness. Our observable data is n, the number of bees, and a food "richness" metric.

To which of these should we apply a parametric model? A non-parametric model?



Answer: The cannon would probably use a parametric model, whereas the bees would be better modeled in an instance-based way.

Generally speaking, the more well-defined a problem is, the more applicable a mathematical model might be, meaning a parametric approach is more suitable. We don't really have much "mathematical bee theory," so crafting a model based on the things we see is much more applicable.

2.1.3 Training vs. Testing

We have a limited amount of historical stock data to work with, right? But we need to evaluate our model without risking real money. Thus, we need to split our data into **training data**—which is fed into the algorithm to create the model—and **testing data**—which we use to evaluate the quality of our model before taking out a 2^{nd} mortgage on our house.

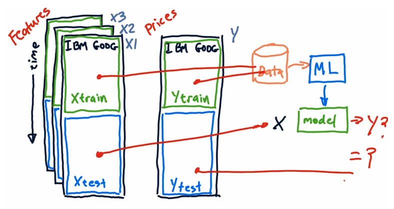


Figure 2.4

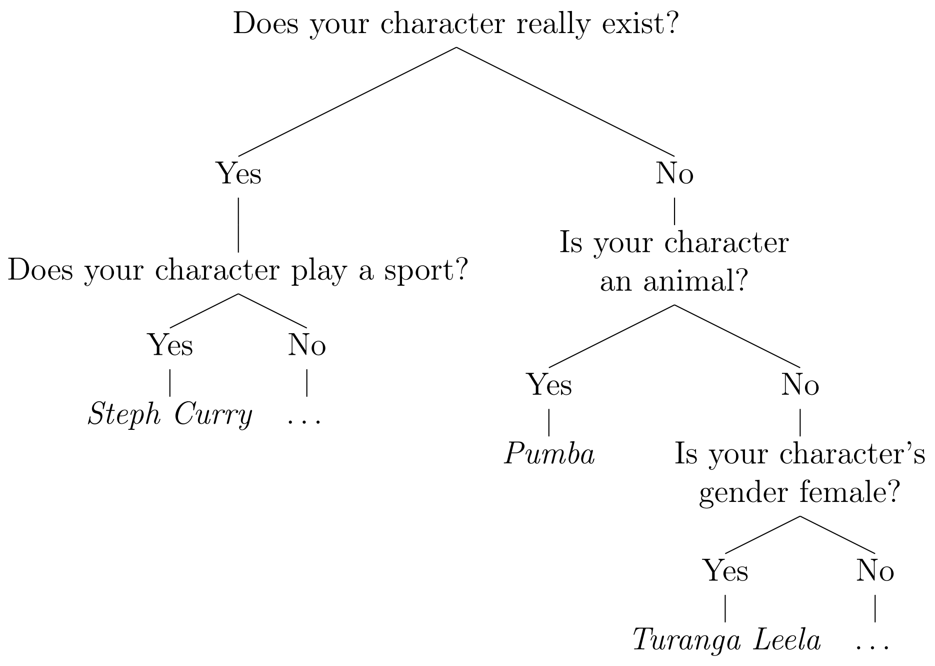
2.1.4 Problems with Regression

Obviously if this was a perfect solution we could call it here and become millionaires. Unfortunately, reality is often disappointing. Regression is a noisy and uncertain method. Furthermore, it's challenging to estimate confidence in a model and reflect that to the user in an understandable way (just showing the standard deviation is not very reliable nor userfriendly). There are also challenges nested in the problem that are specific to stocks: how long do you hold for?[1] How do you optimize allocation ratios?

2.2 Decision Trees

We'll open our discussion of machine learning algorithms with **decision trees**.[2] What is a decision tree? It's fairly self-explanatory: it's a tree that maps various choices to diverging paths that end with some decision. For example, one can imagine the "intelligence" behind the famous "character guessing" AI Akinator: for each yes-or-no question it asks, there are branches the answers that lead down a tree of further questions until it can make a confident guess.

One could imagine the following (incredibly oversimplified) tree in Akinator's "brain:"

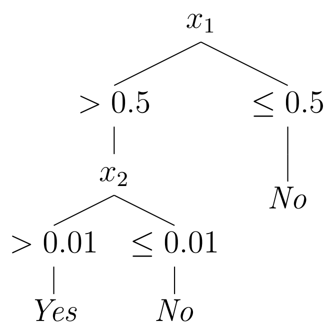

Does your character really exist?
Yes
No
Does your character play a sport?
Is your character an animal?
Yes
No
Steph Curry
...
Yes
No
Pumba
Is your character's gender female?
Yes
No
Turanga Leela
...

Except Akinator has played (and thus improved upon and expanded its decision tree) about 400 ***million*** games as of this writing.

In most applications of decision trees—especially with respect to financial algorithms—the branches are based on *numeric* boundaries in a particular **feature**. For example, we might decide to invest (or not invest) in a particular stock if its Sharpe ratio is above 0.5. Or, to branch further, we might only invest if $S > 0.5$ *and* its daily returns are above 1%. We'd say that $\mathbf{x}_1$ is the "feature vector" for Sharpe ratio (one element for each stock we're choosing from) and $\mathbf{x}_2$ is the feature vector for its daily returns:

[1] Remember, you pay lower taxes on your capital gains (i.e. profits from stocks) if you hold them for over a year! This kind of calculation is important to keep in mind when analyzing a strategy or model.

[2] This section is based in-part on the founding paper by J.R. Quinlan on decision trees.



Note that a feature can (and likely will) appear more than once in a decision tree. Similarly, some features may be completely disregarded when making a decision for a particular input value depending on the path it takes. Consider the table:

	x_1	x_2
\$JPM	1	0.10
\$AAPL	0.2	0.07
\$SPY	0.7	-0.04

Which stocks would we invest in based on our decision tree above? Probably just \$JPM.

Coming to a conclusion given a decision tree seems pretty straightforward. The hard part is *creating* the tree. If we're *given* a set of data like the table above with an additional column describing what our choice *should* be, how do we find the best tree? How do we choose the feature to split our tree along? What values do we split a particular feature along? Etc.

2.2.1 Representing a Decision Tree

Before we dive into this topic, let's discuss the data structure itself. There are two main approaches: an object-oriented approach that many of you are probably familiar with in which there are **`Node`** objects that point to each other, and a matrix-based approach in which each row represents a node.

The matrix representation is more compact and more efficient to process, so let's go over it first. Let's continue with our running example of the buy/don't-buy decision tree and the feature vectors in the table above. Each row represents a node in the tree; the columns are the factor, the splitting value, and the left and right node that follow. For "special" (i.e. leaf) nodes, the splitting value is actually the decision value, and the factor is -1 to indicate that it's a leaf. For the above, it'd look something like:

Index	**Factor**	**Split Value**	**Left**	**Right**
0	1	0.5	1	4
1	2	0.01	2	2
2	-1	Yes		
3	-1	No		
4	-1	No		

For a given input, say $\mathbf{x} = \begin{bmatrix}1, 0.10\end{bmatrix}$ (corresponding to \$JPM), we'd first check the root (i.e. index = 0) and see that it corresponds to the left value since we "pass" the split value[3]. The next node is at index = 1; the input likewise "passes" and we move on to index = 2, which is a leaf node that says "Yes, buy!"

2.2.2 Learning a Decision Tree

Let's talk about "learning" a decision tree now.

[3] We arbitrarily decide that ($x_i >$ split) corresponds to the left index; this is just a matter of convention.

Reinforcement Learning

To be continued...

Index of Terms