Machine Learning in Finance (RSM338)

Week 3: Introduction to Machine Learning

Table of contents

1 Introduction

Last week, we studied the statistical properties of financial returns—how they're distributed, why the normality assumption fails, and why prediction is so difficult. We saw that the signal-to-noise ratio in finance is extremely low, that most predictors fail out-of-sample, and that estimation uncertainty compounds through nonlinear transformations. These are the challenges that any forecasting approach must confront.

This week, we step back to understand the broader framework that provides tools for addressing these challenges: **Machine Learning**. ML is not a magic solution—it faces the same fundamental difficulties we identified in Week 2—but it provides a systematic way to think about prediction problems, measure performance, and combat overfitting.

2 What is Machine Learning?

In traditional programming, you write explicit rules for the computer to follow. Consider building a spam filter. You might write rules like: if the email contains "Nigerian prince," mark it as spam; if the sender is in your contacts, it's probably legitimate; if the message asks for an urgent wire transfer, be suspicious. This approach has fundamental problems. You must anticipate every possible pattern, but spammers are creative and new tactics appear constantly. The rules become unwieldy as edge cases accumulate. Some patterns are too complex for humans to articulate—how would you write rules to recognize a face in a photo? And crucially for our purposes: how would you write explicit rules to predict tomorrow's stock return?

The rule-based approach works well when the domain is well-understood and stable. Chess programs used to be built on explicit rules written by grandmasters. Tax preparation software encodes the tax code. But for pattern recognition in complex, changing environments, we need a different approach.

Machine learning inverts the traditional paradigm. Instead of writing rules, you show the computer examples and let it learn the patterns. The same spam filter, built with ML, works like this: collect thousands of emails labeled "spam" or "not spam," feed them to an algorithm, and let the algorithm learn which patterns distinguish spam from legitimate email. Then apply those learned patterns to new emails. For many problems, it's easier to collect labeled examples than to articulate rules. This is the key insight: **Machine Learning means building models that learn patterns directly from data, rather than being explicitly programmed.**

The relationship between inputs and outputs differs fundamentally between paradigms. In traditional programming, a human writes rules, and the computer applies them to data to produce output. In machine learning, a human provides data and desired outputs, and the computer learns the rules.

ML excels when rules are too complex to articulate (recognizing faces, understanding speech, reading handwriting), when rules change over time (fraud patterns evolving as criminals adapt, market regimes shifting), when rules differ across contexts (what predicts returns varies by asset class, time period, and market conditions), or when you have lots of labeled examples that can reveal patterns humans wouldn't think to look for.

In finance, ML has found applications across many domains: credit scoring (predicting which borrowers will default from millions of loan records), fraud detection (identifying fraudulent transactions from labeled cases), return prediction (identifying which stocks will outperform from historical data and features), and portfolio construction (grouping similar assets from return patterns).

This connects directly to Week 2. We had historical data (S&P 500 returns), wanted to estimate parameters (μ, σ) and make forecasts (expected wealth), and faced the reality that estimation is uncertain and most predictors fail out-of-sample. Machine learning provides a systematic framework for choosing what to estimate (model selection), measuring how wrong we are (loss functions), finding the best estimates (learning algorithms), and testing whether our estimates generalize (out-of-sample evaluation). This framework doesn't solve the low signal-to-noise problem—nothing can—but it gives us disciplined tools for working with it.

3 Types of Learning

Machine learning problems fall into three main categories, distinguished by what kind of feedback the algorithm receives.

3.1 Supervised Learning

In supervised learning, you have **labeled data**: input-output pairs $(\mathbf{x}_i, y_i)$. The feature vector $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})'$ contains p observable inputs for observation i, and y_i is the target or label we want to predict. The training set $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ consists of N labeled examples. The algorithm's goal is to learn a function $f : \mathbb{R}^p \to \mathcal{Y}$ such that $f(\mathbf{x}) \approx y$—given features, we can predict the target.

Supervised learning divides into two main types based on what kind of target we're predicting. In **regression**, the target is continuous (a real number). We might predict next month's return, the price of a house, or the volatility of a stock. The goal is to minimize how far off our predictions are. In **classification**, the target is categorical (a discrete label). We might predict whether an email is spam, whether a borrower will default, or whether to buy, sell, or hold a stock. The goal is to predict the correct category as often as possible.

Finance offers abundant examples of both. Credit scoring is classification: given features like income, debt, credit history, and employment, predict whether a borrower will default or not. Return prediction is regression:

given features like P/E ratio, momentum, size, and earnings surprise, predict next month's return. Fraud detection is classification: given transaction features like amount, time, location, and merchant type, predict whether the transaction is fraudulent or legitimate.

3.2 Unsupervised Learning

In unsupervised learning, you have only input features $\mathbf{x}$—no labels. The algorithm's goal is to discover structure or patterns in the data itself. The key difference from supervised learning is what we're modeling: supervised learning models $P(Y|\mathbf{X})$, the distribution of the target given the features, while unsupervised learning models $P(\mathbf{X})$, the distribution of the features themselves. There's no target variable we're trying to predict; we're trying to understand the structure of the data.

The main unsupervised tasks include clustering (grouping similar observations—for instance, grouping stocks by return patterns), dimensionality reduction (finding low-dimensional representations—reducing 100 features to 5 factors), density estimation (modeling the joint distribution of returns), and anomaly detection (finding unusual observations like outlier transactions).

Finance applications are natural. Clustering can group stocks that move together without using industry labels, identifying "sectors" directly from return data. Factor models and PCA find the dominant factors driving returns, reducing dimensionality from thousands of stocks to a few factors. Anomaly detection can identify unusual trading patterns or flag outlier returns for investigation. We'll study clustering in detail in Week 4.

3.3 Reinforcement Learning

In reinforcement learning, an agent learns by interacting with an environment over time, receiving rewards or penalties for its actions. The agent observes a state s_t (current portfolio, market conditions), takes an action a_t (buy, sell, hold), receives a reward r_t (profit or loss), and aims to learn a policy that maximizes cumulative reward.

This differs fundamentally from supervised and unsupervised learning. In supervised learning, we have a fixed dataset of examples. In reinforcement learning, the agent's actions affect what data it sees next—your trade moves the price, which changes the environment for your next trade. Finance applications include optimal execution (minimizing market impact when trading large orders), dynamic portfolio management, and market making (setting bid-ask spreads). We won't cover reinforcement learning in this course, but it's an active research area in quantitative finance.

3.4 Building Your Toolbox

Think of ML methods as tools in a toolbox. Just as an experienced contractor knows which tool is right for each job, you'll learn which ML method is right for each problem. Linear regression predicts a number from features and works best for simple relationships where interpretability matters. Regularized regression prevents overfitting when you have many features relative to observations. Logistic regression predicts probabilities and classes for binary outcomes like default/no default. Decision trees capture nonlinear patterns and complex interactions between features. Clustering groups similar observations when you have no labels and want to find structure. The goal of this course is to build your intuition so you recognize which tool fits which problem—and understand *why*.

4 The ML Formalism

Every machine learning algorithm, no matter how fancy it sounds, is really just answering three questions. Once you understand these three questions, you'll see the same structure in every method we encounter—from simple linear regression to neural networks.

Let's preview with an example you already know: linear regression.

1. **Model:** We assume the relationship is linear: $f(\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$
2. **Loss:** We measure error with squared differences: $L(y, \hat{y}) = (y - \hat{y})^2$
3. **Algorithm:** We use the OLS formula: $\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$

Don't worry if the formulas look intimidating—we'll unpack each piece. The point is that this same three-part structure applies to every ML method.

4.1 The Model: What Shape Should the Answer Take?

Before we can learn anything, we have to decide what kind of relationship we're looking for between inputs and outputs. This is a choice *you* make—it's not something the data tells you directly.

Think of it like choosing what kind of line to draw through a scatterplot. Should it be a straight line? A curve? Multiple lines for different regions?

Model type	What it assumes	Parameters to learn
Linear regression	Output is a weighted sum of inputs	The weights (β)
Polynomial regression	Output depends on powers of inputs	Coefficients on each power
Decision tree	Different rules for different regions	Where to split, what to predict in each region
Neural network	Layers of simple nonlinear functions	Weights connecting the layers

For linear regression, we're saying: "I believe the output is roughly a weighted combination of the inputs, plus some noise." Mathematically:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \varepsilon$$

The β coefficients are the unknown numbers we need to learn. In matrix form (for all N observations at once): $\mathbf{y} = \mathbf{X}\beta + \varepsilon$.

The fundamental tradeoff is always present: a model that's too simple misses real patterns (**underfitting**), while a model that's too complex fits noise (**overfitting**). Linear regression is simple and interpretable but can only capture straight-line relationships. Neural networks can approximate almost any relationship but need lots of data and can easily overfit.

Here's the key insight: once you pick a model type, you're committing to a specific mathematical form with unknown numbers in it. **Learning means finding the specific values of those numbers that work best for your data.**

4.2 The Loss Function: What Does "Best" Mean?

We need a way to measure how wrong a prediction is, so we can find the parameters that make predictions least wrong.

The notation is simple: $L(y, \hat{y})$ is the loss when the true value is y and our prediction is $\hat{y}$. Lower loss means better prediction.

The intuition: if we predict a stock will return 5% and it actually returns 8%, we were off by 3 percentage points. That's the error. But how should we combine all our errors into a single number?

Common loss functions for regression:

Name	Formula	What it does
Squared error	$L(y, \hat{y}) = (y - \hat{y})^2$	Penalizes large errors heavily
Absolute error	$L(y, \hat{y}) = \|y - \hat{y}\|$	More robust to outliers

The most common choice is **squared error**. Squaring has two effects: it makes all errors positive (so they don't cancel out), and it penalizes big errors more than small ones. Being off by 10 is not just twice as bad as being off by 5—it's four times as bad (because $10^2 = 100$ while $5^2 = 25$).

Common loss functions for classification:

Name	Formula	What it does
0-1 loss	$L(y, \hat{y}) = \mathbf{1}[y \neq \hat{y}]$	1 if wrong, 0 if correct
Cross-entropy	$L(y, p) =$ $-y \log p - (1 - y) \log(1 - p)$	For probabilistic predictions

The choice of loss function matters—different loss functions lead to different "best" parameters. You should pick one that reflects what you actually care about.

Once we've chosen a loss function, we compute the **average loss** across all training data. This is called the **empirical risk**:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f_\theta(\mathbf{x}_i))$$

In words: for each observation, compute the loss between the true value y_i and what the model predicts $f_\theta(\mathbf{x}_i)$, then average. For squared error, this is the Mean Squared Error (MSE).

The learning problem becomes: find the parameters θ^* that minimize this average loss.

4.3 The Learning Algorithm: How Do We Find the Best Parameters?

We've defined what we're looking for (a model of a certain shape) and what "best" means (minimizing average loss). Now we need a procedure to actually find the best parameters.

For some simple problems, there's a formula. Linear regression with squared error is one of these lucky cases. The OLS (ordinary least squares) formula gives you the answer directly:

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

This is the formula from your statistics courses. Plug in the data matrix $\mathbf{X}$ and the target vector $\mathbf{y}$, do some matrix math, and out pop the optimal coefficients.

For most problems, we use iterative optimization. The math doesn't work out to a nice formula, so instead we search for good parameters step by step.

The most common search method is **gradient descent**. The idea is intuitive: imagine you're lost in fog on a hilly landscape, and you want to find the lowest point (a valley). You can't see the whole landscape, but you can feel which way the ground slopes under your feet. So you take a small step downhill. Feel the slope again. Take another step downhill. Keep going until you reach a point where it's flat in every direction—that's a minimum.

Mathematically, the "slope" is the gradient $\nabla_\theta \mathcal{L}$—the vector of partial derivatives telling us how the loss changes when we nudge each parameter. The gradient points in the direction of steepest *increase*, so we move in the *opposite* direction (steepest decrease):

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta \mathcal{L}(\theta^{(t)})$$

The symbol η (eta) is the **learning rate**—how big a step we take each time. We keep iterating until the gradient is approximately zero, meaning we've reached a minimum.

This connects back to Week 1: at a minimum, all partial derivatives are zero. We want to find where $\nabla\mathcal{L} = 0$. For complex models we can't solve this equation directly, so gradient descent is a procedure for getting there step by step.

Gradient descent is the workhorse of modern ML—it's how neural networks with millions of parameters are trained.

4.4 The Recipe

Every ML method follows the same workflow:

1. **Choose a model** (what shape of relationship are we looking for?)
2. **Choose a loss function** (what does "good prediction" mean?)
3. **Run the learning algorithm** (find the parameters that minimize loss on the training data)
4. **Evaluate on new data** (because training performance is misleadingly optimistic)

That last step is crucial—it's what we emphasized in Week 2. A model that fits the training data perfectly might be memorizing noise rather than learning real patterns. The only way to know if you've learned something useful is to test on data the model hasn't seen.

5 Python for Machine Learning

You don't need to be an expert programmer to use ML. Most of the hard work is already done—you just need to know which tools to use and how to interpret results.

The main Python packages for ML are `numpy` (fast math on arrays), `pandas` (data tables, like Excel with programming), `matplotlib` (plotting), and `scikit-learn` (the ML algorithms themselves). These are pre-installed in most Python environments.

The algorithms behind ML are genuinely complex—gradient descent, matrix decompositions, optimization routines. A production implementation of random forests is thousands of lines of carefully optimized code. But Python is a language built on **packages**. Someone else has already written the complex algorithms, debugged edge cases, and optimized for speed. So our code stays high-level: load data with pandas, prepare features with numpy, split data with sklearn's `train_test_split`, fit a model with `model.fit()`, make predictions with `model.predict()`, and evaluate by comparing predictions to truth. Every ML project follows this pattern. The hard work is understanding *which* model to use and *how* to interpret results—that's what this course teaches.

Almost every ML model in scikit-learn uses the same interface:

Python

```
from sklearn.some_module import SomeModel

model = SomeModel()            # Create the model
model.fit(X_train, y_train)    # Learn from training data
predictions = model.predict(X_test)  # Apply to new data
```

This consistency is powerful. Whether you're using linear regression, ridge regression, or random forests, the code structure is identical—you just import a different model. You learn one interface, you can use dozens of models.

Here's a complete example that estimates a stock's beta using the CAPM framework:

Python

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# 1. Create some fake stock data
np.random.seed(42)
data = pd.DataFrame({
    'market_return': np.random.randn(100),
})
data['stock_return'] = 0.5 + 1.2 * data['market_return'] + 0.3 * np.random.randn(100)

# 2. Split into training and test sets
X = data[['market_return']]  # Features (what we observe)
y = data['stock_return']      # Target (what we predict)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 3. Fit model
model = LinearRegression()
model.fit(X_train, y_train)

print(f"Estimated beta: {model.coef_[0]:.2f}")
print(f"Estimated alpha: {model.intercept_:.2f}")

# 4. Predict on test data
predictions = model.predict(X_test)

# 5. Evaluate: how close are predictions to actual values?
mse = np.mean((predictions - y_test)**2)
print(f"Test MSE: {mse:.4f}")
```

Output

```
Estimated beta: 1.16
Estimated alpha: 0.50
Test MSE: 0.0786
```

This is CAPM in 15 lines: we're estimating the stock's beta (sensitivity to the market) and alpha (intercept), then testing on held-out data. When you see code in this course, focus on three things: what data goes in (features and target), what model we're using (the import tells you), and what comes out (learned parameters and predictions). You don't need to memorize syntax; you need to understand what the code is *doing*.

6 Limitations of Machine Learning

ML is not magic. It fails when its assumptions are violated.

Dependence on historical data. ML learns patterns from the past. If the future differs systematically from the past, predictions fail. A model trained on bull market data may fail in a crash. A model trained on pre-COVID behavior may fail when consumption patterns change overnight.

The stationarity assumption. Most ML methods assume the data-generating process is stable over time. If relationships change—if what predicted returns in the 1990s no longer predicts returns in the 2020s—models become stale. In finance, factor returns vary across market regimes, making this assumption particularly suspect.

Regime changes. Major structural breaks invalidate learned patterns. The 2008 financial crisis, the COVID-19 pandemic, major regulatory changes—these represent fundamentally different environments than the training data. The patterns that worked before may not just fail to work; they may actively mislead.

> ℹ Reality Check
>
> "All models are wrong, but some are useful." — George Box
> The question is never whether your model is perfect (it isn't). The question is whether it's useful enough to inform decisions, and whether you understand its limitations well enough to avoid being blindsided.

6.1 Overfitting: The Central Challenge

Week 2 previewed this: most return predictors fail out-of-sample (Goyal-Welch 2008). Why? **Overfitting.** The model learns patterns in the training data that don't generalize to new data.

Some patterns in historical data are real—they reflect genuine, persistent relationships. Other patterns are coincidence—they're artifacts of the specific sample that won't repeat. A model fit to historical data captures both, and can't tell them apart. The symptoms are excellent performance on training data alongside poor performance on new data. This is where the low signal-to-noise ratio from Week 2 bites hardest. When signal is weak relative to noise, it's easy for a model to latch onto noise patterns that happen to correlate with the target in the training sample. The more flexible the model, the more capacity it has to fit noise.

Much of this course is about avoiding overfitting through train/test splits (which we've already seen), cross-validation (Week 5), regularization (Week 5), and ensemble methods (Week 9).

Overfitting is one side of a fundamental tradeoff. A model that's too simple will **underfit**—it can't capture the true relationship, so it makes systematic errors. A model that's too complex will **overfit**—it captures noise, so it fails on new data. The sweet spot is somewhere in between: complex enough to capture the real pattern, simple enough to ignore the noise. Finding this balance is the art of machine learning.

7 Summary

Machine learning is about learning patterns from data rather than explicitly programming rules. Instead of a human writing rules and a computer applying them, the human provides data and desired outputs and the computer figures out the rules. This approach works well when patterns are too complex to articulate, when they change over time, or when you have lots of examples that can reveal relationships humans wouldn't think to look for.

The three main types of learning are distinguished by what feedback the algorithm receives. In supervised learning, you have labeled examples and want to learn to predict the label from features—either a number (regression) or a category (classification). In unsupervised learning, you have only features and want to discover structure, like grouping similar observations together. Reinforcement learning involves an agent learning through trial and error in an environment, but we won't cover it in this course.

Every ML algorithm answers three questions. First, what shape of relationship are we looking for? This is the model choice—linear, tree-based, neural network, and so on. Second, what does "best" mean? This is the loss function, measuring how wrong predictions are. Third, how do we find the best parameters? Sometimes there's a formula; usually we search step-by-step using gradient descent. The workflow is always the same: choose a model, choose a loss, fit to training data, and evaluate on new data.

Python makes all of this practical. The scikit-learn library provides a consistent interface where you create a model, fit it to training data, and use it to make predictions. The algorithms behind the scenes are complex, but your code stays simple.

The central limitation is overfitting: models can memorize noise in the training data rather than learning real patterns, performing well in-sample but poorly out-of-sample. This is where the low signal-to-noise ratio from Week 2 bites hardest. Much of this course is about techniques to combat overfitting—train/test splits, cross-validation, regularization, and ensemble methods.

The framework introduced today structures everything that follows. Each week adds new tools to your toolbox, but you'll always be choosing a model, defining a loss function, running a learning algorithm, and evaluating on held-out data.