sMp 500

A Data-Driven Approach to Market Prediction

Christian Hernandez
Huan Zhang
Mutaz Ismael
Nina Lin

01 —

PROJECT OVERVIEW

a look into our project.

Project Goal: Build a statistical model, which, given past market price data for a stock, predicts the price of this future stock.

Aiming For: ~51% accuracy

What We Did:

- Pulled real stock data using the Yahoo Finance API (yfinance)
- Engineered features such as news signals/technical indicators
- Trained and tested machine learning models to predict future stock prices
- Visualized results through graphs and a webpage/website

© 2025 sMP 500

Our Models

Geometric Brownian Motion

- Traditional financial model for simulating price movements
- Assumes stock prices follow a random walk with a general upward or downward trend
- A future price ($)

Neural Network

- A data-driven machine learning model that learns complex patterns from historical data
- Takes in historical stock data to predict if the stock will rise or fall
- A directional signal (up or down)

© 2025 sMP 500

02 —

THE GBM

geometric brownian motion.

Geometric Brownian Motion (GBM)

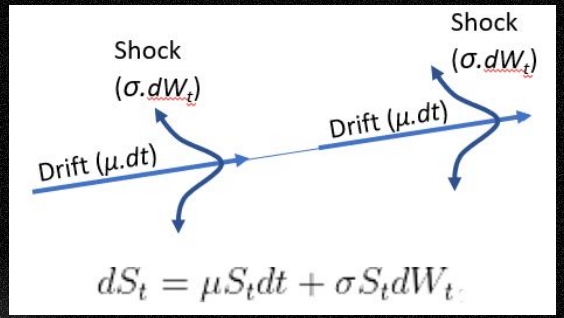
Geometric Brownian Motion is a stochastic process used to model paths of variables that evolve over time, particularly those that cannot become negative and exhibit continuous, random movement.

Properties of GBM

- Continuous Sample Paths
- Markov Property
- Log-normal Distribution of S_t
- Non-stationary

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

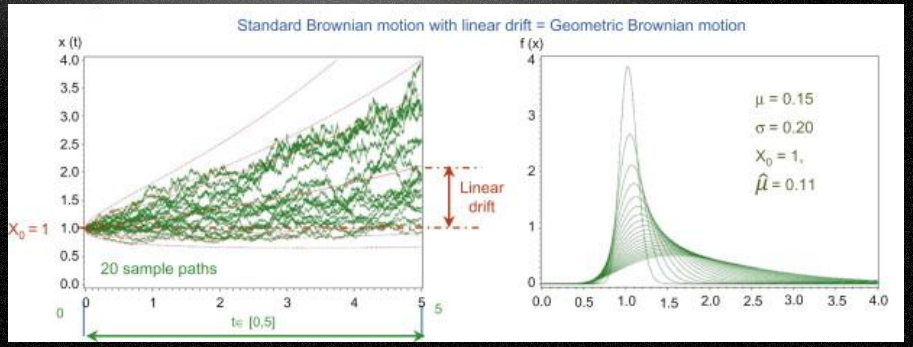
S_t is the asset price at time t
μ is the drift — mean
σ is the volatility — standard deviation
dW_t is the increment of a Wiener process (aka Brownian motion)

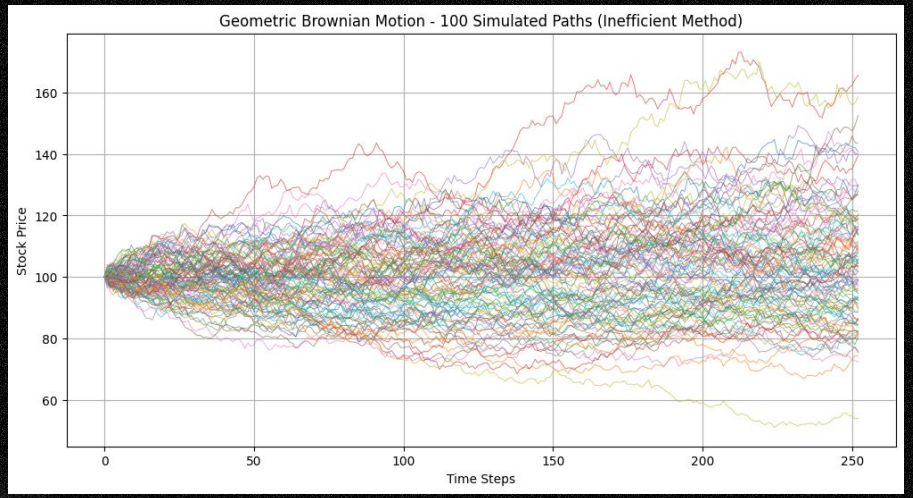


© 2025 sMP 500

Why GBM?

1. Captures Realistic Price Evolution
2. Example of a Monte Carlo Simulation
3. Flexible Input Parameters
4. Time-Scalable





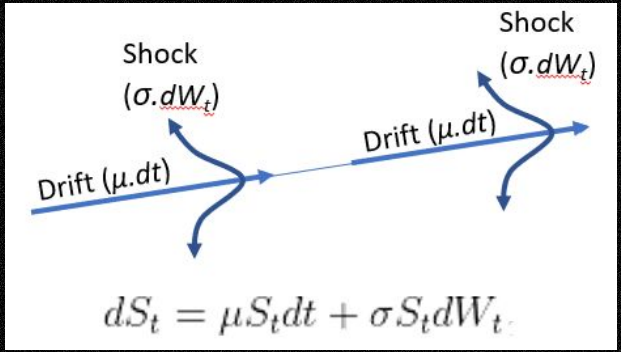
© 2025 sMP 500

```python
import numpy as np
import matplotlib.pyplot as plt
import time
# Parameters for GBM
np.random.seed(42)  # For reproducibility
S0 = 100            # Initial stock price
mu = 0.05           # Drift (annualized return)
sigma = 0.2         # Volatility (annualized standard deviation)
T = 1               # Time horizon in years
steps = 252         # Number of steps (daily steps for a year)
paths = 1000         # Number of paths to simulate

# Time increments
dt = T / steps

# Initialize paths array
gbm_paths = np.zeros((steps + 1, paths))
gbm_paths[0] = S0  # Initial price for all paths

current_time = time.time()
# Simulate paths using a loop for each path
for p in range(paths):
    for t in range(1, steps + 1):
        z = np.random.normal(0, 1)  # Generate a single random noise for this step
        gbm_paths[t, p] = gbm_paths[t - 1, p] * np.exp((mu - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * z)

end_time = time.time()
execution_time = end_time - current_time
print(f"Execution time: {execution_time} seconds")
```

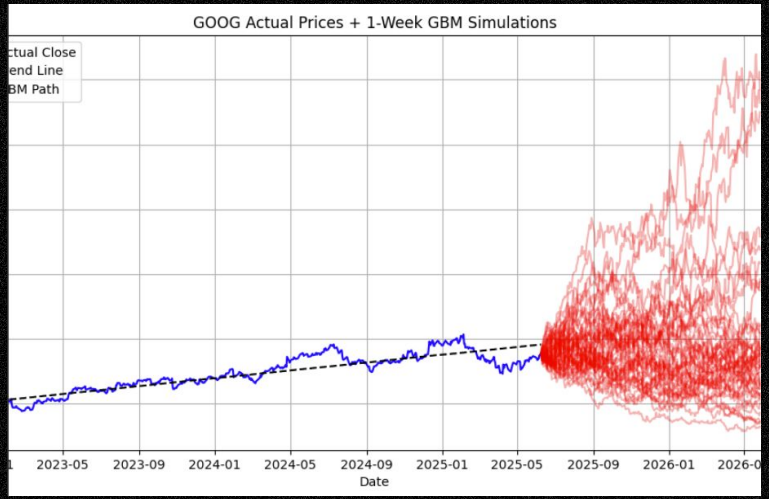
```
Execution time: 1.1035194396972656 seconds
```

```python
# Plot the GBM paths
plt.figure(figsize=(12, 6))
for i in range(100):
    plt.plot(gbm_paths[:, i], lw=0.7, alpha=0.6)
plt.title('Geometric Brownian Motion - 100 Simulated Paths (Inefficient Method)')
plt.xlabel('Time Steps')
plt.ylabel('Stock Price')
plt.grid(True)
plt.show()
```

1. Starts with an initial stock price ($100).
2. Defines how long the simulation will run (1 year = 252 trading days).
3. Generates 1,000 different paths, where each day, the stock moves based on both:
    a. Expected growth
    b. Random "market noise"
4. Draws a graph showing how the stock could rise or fall in different scenarios.



© 2025 sMP 500



```python
returns = np.array(returns)
# Annualized mean return
mu = float(np.mean(returns) * 252)
# Annualized volatility
sigma = float(np.std(returns) * np.sqrt(252))
```

© 2025 sMP 500

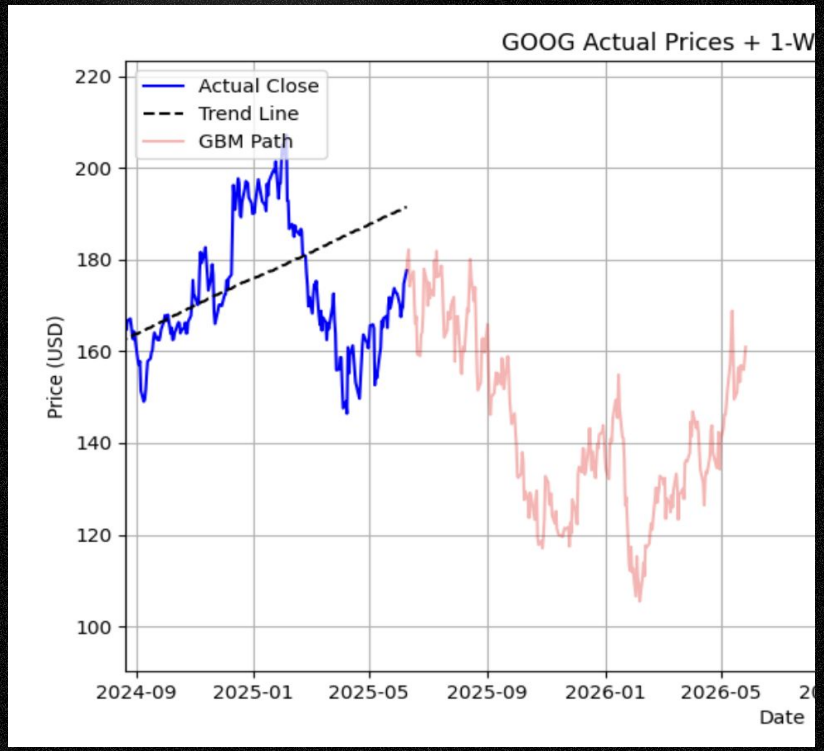
GBM vs. Linear Regression

GBM:

- Simulates how a stock price might move over time with random ups and downs
- Captures both a general trend (drift) and unpredictable fluctuations (random shocks)
- Reflects the natural randomness in markets

Linear Regression:

- Draws a straight line through past data to predict future values (line of best fit)
- Assumes a fixed, steady trend without randomness
- Doesn't account for sudden market changes or volatility



© 2025 sMP 500

Concluding Thoughts on GBMs

Pros:

1. Simplicity
2. Captures Essential Market Features
3. Time-Scalable
4. Efficient for Monte Carlo Simulations

Cons:

1. Ignores Jumps and Discontinuities
2. No Mean Reversion
3. Poor Fit Over Long Time Horizons
4. Doesn't Incorporate External Factors

© 2025 sMP 500

03 —

NEURAL NETWORKS

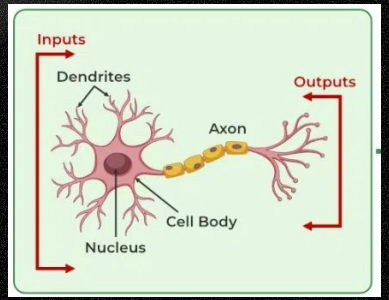
leveraging deep learning to model market behavior.

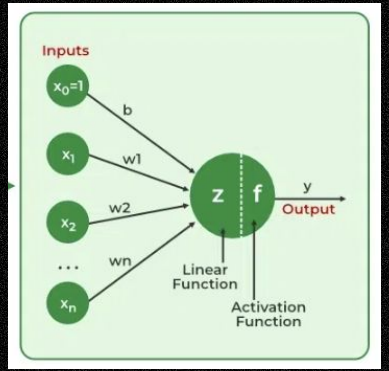
Neural Network Structure

1. *Input Layer*
   - Accepts the input features
   - Each "neuron" in this layer corresponds to a feature in your data set

2. *Hidden Layers*
   - Multiple layers that transforms your output
   - This is where **weights, bias,** and **activation functions** are used

3. *Output Layer*
   - Takes final outputs from the last hidden layer
   - Outputs a prediction in desired format
   - Ex: Regression, Binary Classification, Multi-class Classification, etc





© 2025 sMP 500

Training a Neural Network

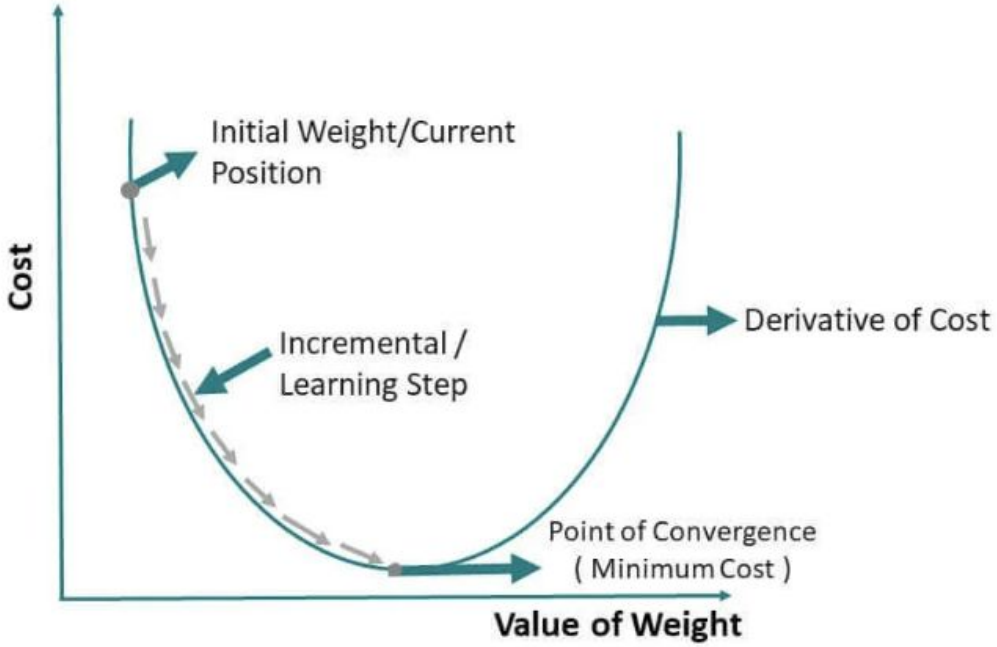
1. *Weights and Biases*
   - Every connection between neurons in the hidden layers have a weight and a bias
   - These are randomly decided by the model (learnable/adjustable)

2. *Loss Function*
   - Measures how well the network's output matches the true value
   - Ex: Mean Squared Error , Mean Absolute Error, Binary Cross-Entropy, etc

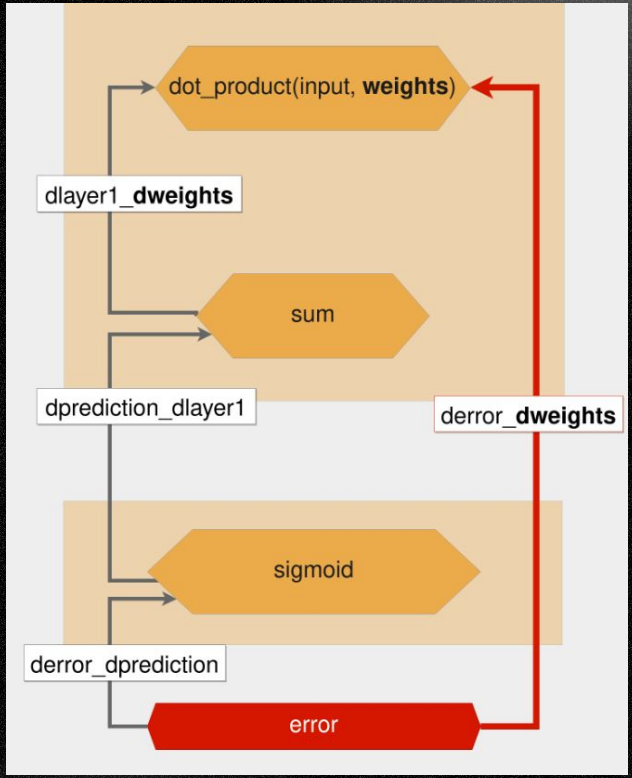
3. *Optimizer (Gradient Descent)*
   - Updates weights and biases to reduce the loss
   - Uses backpropagation to compute gradients of the loss with respect to each weight

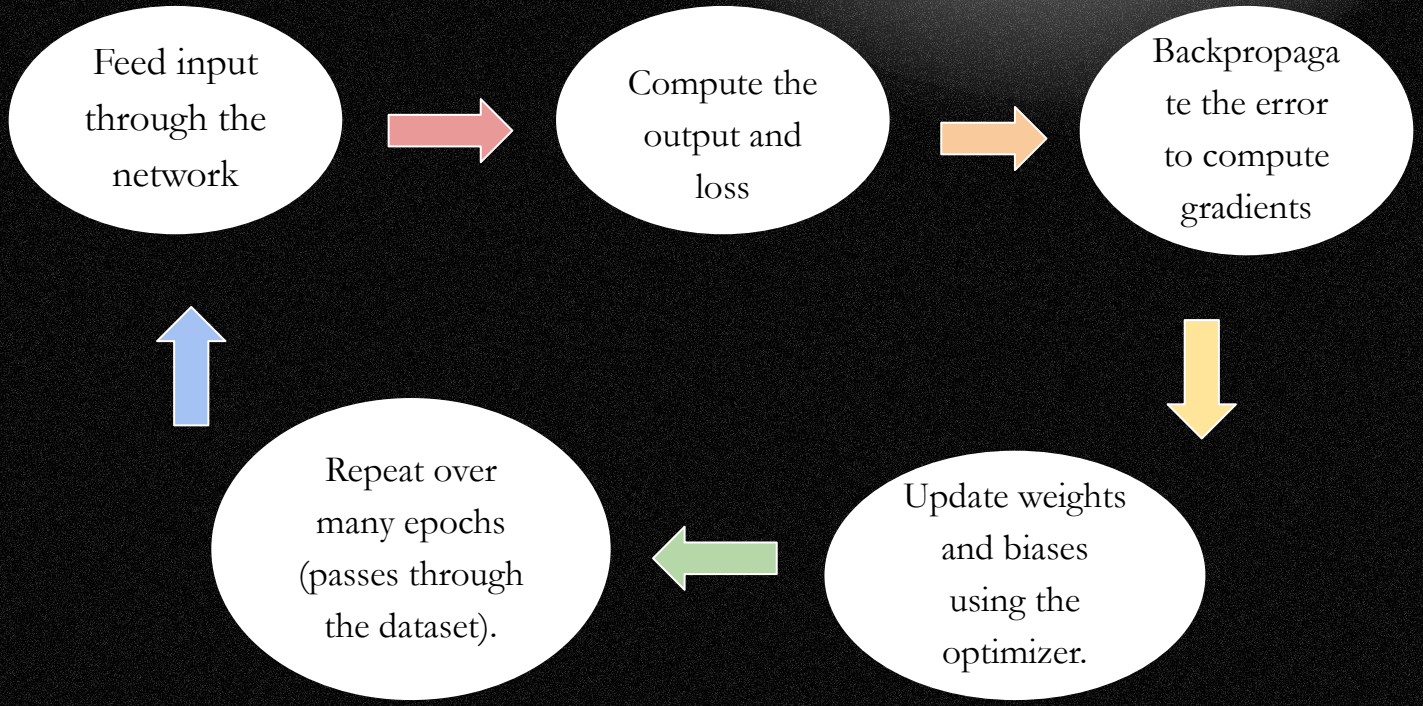
© 2025 sMP 500

Gradient Descent of Machine Learning



© 2025 sMP 500


dot_product(input, **weights**)
dlayer1_**dweights**
sum
dprediction_dlayer1
derror_**dweights**
sigmoid
derror_dprediction
error


Feed input through the network
Compute the output and loss
Backpropagate the error to compute gradients
Update weights and biases using the optimizer.
Repeat over many epochs (passes through the dataset).

© 2025 sMP 500

Pros and Cons

Pros	*Cons*
- Excel at learning complicated patterns - Highly flexible/Adjustable to a lot of problems - Ideal for large scale data - Don't need a lot of manual engineering because the network can learn by itself from the input data	- Requires a LOT of data - $$$ - Hard to interpret why a neural net made a certain decision - Hyperparameter tuning is tedious - Overfitting!!

© 2025 sMP 500

Our Neural Network- *Workflow at a Glance*

Our Goal: To use a Neural Network to predict the short-term direction of stock prices

Process:

1. **Gather Data:** Get historical stock prices.
2. **Prepare Data:** Create meaningful signals and structure the data.
3. **Train Model:** Teach a Neural Network to recognize patterns.
4. **Evaluate Model:** See how well our model performs.
5. **Predict & Save:** Generate a "buy/sell" signal for tomorrow and record our findings.

```python
import yfinance as yf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import joblib
import datetime
import json

from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error,
from sklearn.preprocessing import StandardScaler
from sklearn.utils import resample
```

© 2025 sMP 500

Our Neural Network- *Getting the Data*

Function: *download_data(ticker, period)*
Purpose: Retrieve historical stock information.
What it does: Downloads past stock prices and trading volume (*yfinance*).
Why it's important: Provides the essential raw data for analysis.

```python
def download_data(ticker, period="20y"):
    """
  Downloads historical stock data for a given ticker and period.

  Args:
      ticker (str): The stock ticker symbol (e.g., "AAPL").
      period (str): The time period for which to download data
(e.g., "20y").

  Returns:
      pd.DataFrame: A DataFrame containing the historical stock
data.
  """
    print(f"Downloading data for {ticker} for the last
{period}...")
    stock = yf.Ticker(ticker)
    df = stock.history(period=period)
    print("Data download complete.")
    # df.to_csv(f"{ticker}.csv")
    return df
```

© 2025 sMP 500

Our Neural Network- *Preparing Features*

Function: *add_technical_indicators(df)*

Purpose: Convert raw price data into predictive "signals."

What it does:

- Calculates **Features** (e.g., Moving Averages, Volatility, Momentum, RSI) from price and volume. These are mathematical transforms indicating market trends or conditions.
- Defines the **Target Variable**: *1* if price increases >1% in 3 days, *0* otherwise. This is what the network predicts.

Why it's important: Transforms noisy data into structured input that Neural Networks can learn from effectively.

```python
def add_technical_indicators(df):
    """
    Adds technical indicators and the target variable to the DataFrame.

    Args:
        df (pd.DataFrame): The input DataFrame with stock data.

    Returns:
        pd.DataFrame: The DataFrame with added features and target.
    """
    print("Adding technical indicators...")
    df['Return'] = df['Close'].pct_change()
    df['MA10'] = df['Close'].rolling(window=10).mean()
    df['MA20'] = df['Close'].rolling(window=20).mean()
    df['Volatility'] = df['Return'].rolling(window=5).std()
    df['Momentum'] = df['Close'] - df['Close'].shift(10)
    # Simplified RSI calculation to avoid potential division by zero in edge cases
    delta = df['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
    rs = gain / loss
    df['RSI'] = 100 - (100 / (1 + rs))

    # Create binary target: 1 if the price increases by more than 1% in 3 days, else 0
    df['Target'] = (df['Close'].shift(-3) / df['Close'] - 1 > 0.01).astype(int)
    df.dropna(inplace=True)
    print("Technical indicators added.")
    return df
```

© 2025 sMP 500

Our Neural Network- *Balancing the Dataset*

Function: balance_data(df)

Purpose: Ensure fair learning from both "up" and "down"

What it does:

- Addresses **imbalanced datasets** (e.g., fewer "up" days than "down" days)
- **Downsamples the majority class** to equalize counts, preventing bias
- Shuffles the dataset for randomness

Why it's important: Prevents the model from being biased towards the more frequent outcome, ensuring it learns to predict both scenarios

```
def download_data(ticker, period="20y"):
    """
  Downloads historical stock data for a given ticker and period.

  Args:
      ticker (str): The stock ticker symbol (e.g., "AAPL").
      period (str): The time period for which to download data
(e.g., "20y").

  Returns:
      pd.DataFrame: A DataFrame containing the historical stock
data.
  """
    print(f"Downloading data for {ticker} for the last
{period}...")
    stock = yf.Ticker(ticker)
    df = stock.history(period=period)
    print("Data download complete.")
    # df.to_csv(f"{ticker}.csv")
    return df
```

© 2025 sMP 500

Our Neural Network- *Structuring Data*

Function: prepare_data_for_training(df)

Purpose: Format data for Neural Network input.

What it does:

- **Feature Selection:** Chooses input variables (*X*) and separates the target (*y*).
- **Data Scaling:** Normalizes features (StandardScaler) to a similar range, preventing larger values from dominating.
- **Train/Test Split:** Divides data to ensure unbiased evaluation on unseen data.

Why it's important: Proper formatting and splitting are crucial for effective and honest model training and evaluation

```
print("Preparing data for training...")
features = [
    'Open', 'High', 'Low', 'Close', 'Volume', 'MA10', 'MA20', 'Volatility',
    'Momentum', 'RSI'
]
X = df[features]
y = df['Target'].values

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
print("Data preparation complete.")
return X_train, X_test, y_train, y_test, scaler, features
```

© 2025 sMP 500

Our Neural Network- *Training the NN*

Function: *train_model(X_train, y_train)*

Purpose: Teach the Multi-layer Perceptron (MLP) Neural Network to recognize patterns.

What it does:

- Initializes *MLPClassifier*, setting its architecture (hidden layers, neurons).
- Sets training parameters (e.g., *max_iter*, *solver*, *early_stopping* to prevent overfitting).
- ***model.fit():*** The network learns by iteratively adjusting its internal connections based on training data.

Why it's important: This is the core learning phase where the network builds its predictive intelligence.

```python
def train_model(X_train, y_train):
    """
  Initializes, trains, and returns the MLPClassifier model.

  Args:
      X_train (np.array): Scaled training features.
      y_train (np.array): Training target values.

  Returns:
      MLPClassifier: The trained model.
  """
    print("Training MLPClassifier...")
    model = MLPClassifier(hidden_layer_sizes=(256, 128, 64),
                          max_iter=3000,
                          alpha=0.01,
                          random_state=42,
                          solver='adam',
                          learning_rate_init=0.001,
                          early_stopping=True,
                          validation_fraction=0.1,
                          n_iter_no_change=20)
    model.fit(X_train, y_train)
    print("Model training complete.")
    return model
```

© 2025 sMP 500

Our Neural Network- *Evaluating the NN*

Function: evaluate_model(model, X_test, y_test)

Purpose: Objectively measure the model's prediction accuracy on unseen data.

What it does:

- Generates predictions (y_pred) on the test set.
- Compares y_pred to actual y_test values.
- Calculates metrics: **Accuracy, Mean Squared Error (MSE)**, **Mean Absolute Error (MAE)**, and a detailed **Classification Report** (precision, recall, f1-score).

Why it's important: Essential to know if the model is useful. Metrics provide a quantifiable measure of effectiveness and reveal strengths/weaknesses.

```python
def evaluate_model(model, X_test, y_test):
    print("\n--- Model Evaluation ---")
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True)

    print("MSE:", mse)
    print("MAE:", mae)
    print("Accuracy:", accuracy)
    print(classification_report(y_test, y_pred))

    metrics = {
        "accuracy": accuracy,
        "mse": mse,
        "mae": mae,
        "classification_report": report
    }
    return metrics, y_pred
```

© 2025 sMP 500

Our Neural Network- *Prediction & Results*

Functions: predict_tomorrow_signal, save_results, plot_predictions

Purpose: Generate a real-world trading signal and document the model's run.

What it does:

- **predict_tomorrow_signal:** Applies the trained model to the latest stock data for a "Buy" (1) or "Sell" (0) signal.
- **save_results:** Stores the trained model (.pkl) and a comprehensive log file (.json) with all run details.
- **plot_predictions:** Visualizes actual vs. predicted movements for insight.

Why it's important: Generates actionable output, ensures reproducibility, and provides a clear record of the experiment.

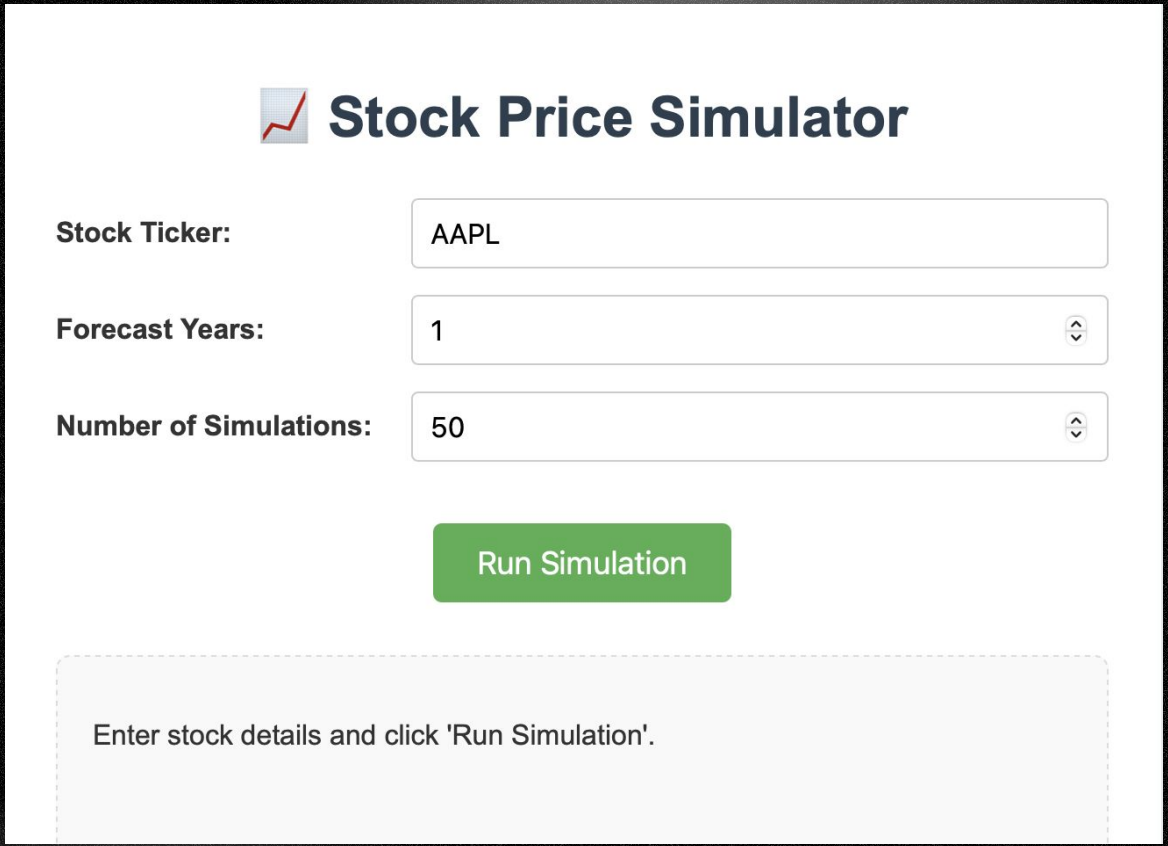
© 2025 sMP 500

04 —

WEBSITE

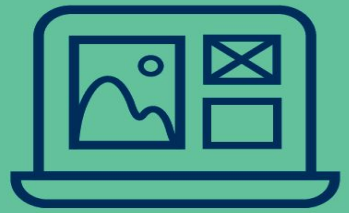
our final product.

sMp 500



© 2025 sMP 500

FRONT END VS BACKEND



Frontend

Focuses on layout, animations, content organization, navigation, graphics.

Programming languages:
JavaScript, HTML, CSS

Backend

Focuses on building code, debugging, database management.

Programming languages:
Node.js, Python, Java

© 2025 sMP 500

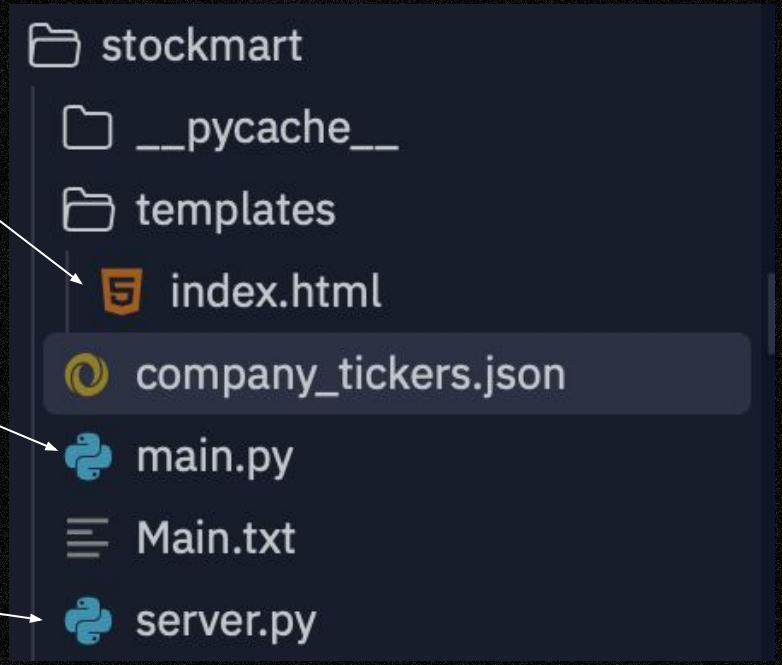
FRONT END

WEBSITE

GBM CODE

BACKEND

SERVER



© 2025 sMP 500

How does a website work?

Client (Your Browser):

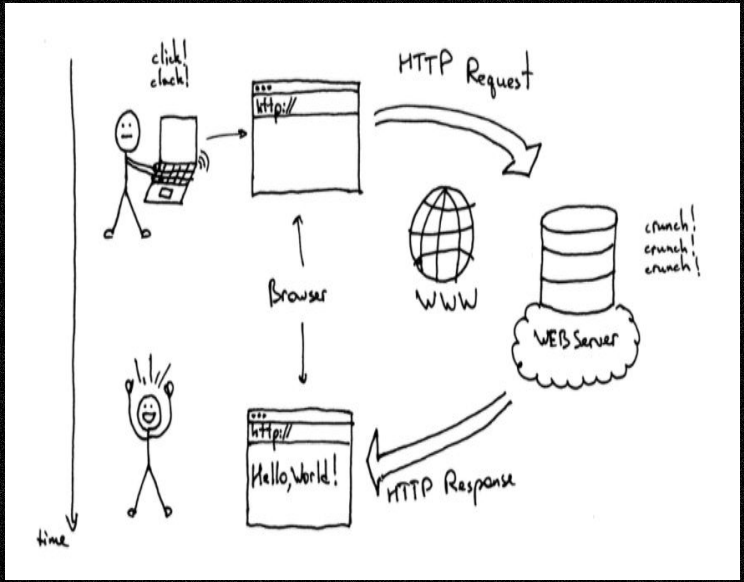
- Your device's web browser (Chrome, Firefox, etc.).
- YOU initiating the action.

Server (The Website's Host):

- A remote computer storing website files & logic.
- THE BRAIN that processes requests.

The Cycle:

- Request: Your Client asks the Server for a page.
- Response: The Server sends back the page's components (HTML, CSS, JS).



© 2025 sMP 500

Python in the Browser: PyScript lets you run Python code directly in web browsers.

WebAssembly (Wasm): It compiles the standard Python interpreter (CPython) into WebAssembly.

Browser Execution: Your browser downloads this Wasm, then executes your Python code embedded in HTML (e.g., <py-script>).

© 2025 sMP 500

PY SCRIPT

What We Used PY Script For:

- Display GBM model interface
- Accept input from user
- Return input
- Display graph

What Professionals Use PY Script For:

- Developing websites
- Software
- Task automation
- Data analysis
- Data visualization.

© 2025 sMP 500

WHAT IS FLASK?

Flask is a LIGHTWEIGHT Python Framework

- Build a website using only a single python file

© 2025 sMP 500

What's the difference between them?

FLASK:

- Runs on Server
- Build real apps, APIs, Websites
- Needs you to install python in server
- Faster Performance
- Host on a server

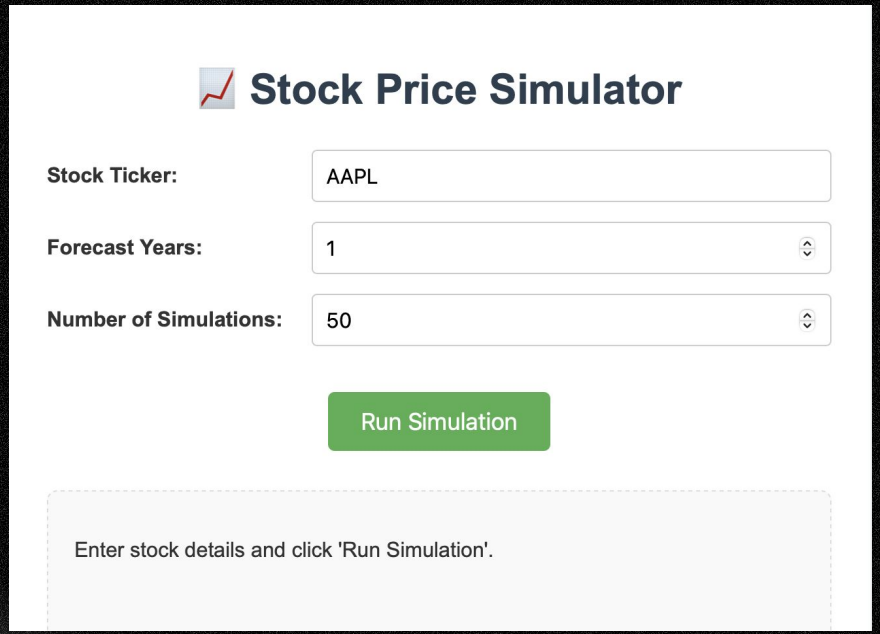
PYTHON:

- Runs in the browser
- Demo APPs or simple tools
- Slower performance
- Only needs a HTML

© 2025 sMP 500

IF WE HAD MORE TIME, WHAT COULD HAVE WE DONE?

- Interactive selection of stocks, forecast, numsimulation
- Modern Website Design
- Use a different PREDICTOR model
- Use NEWS data to affect stock



© 2025 sMP 500

THANKS —

OPEN TO ANY QUESTIONS/FEEDBACK

Christian Hernandez herchris@umich.edu
Huan Zhang huanz@umich.edu
Mutaz Ismael mutaz@umich.edu
Nina Lin lnina@umich.edu

© 2025 sMP 500