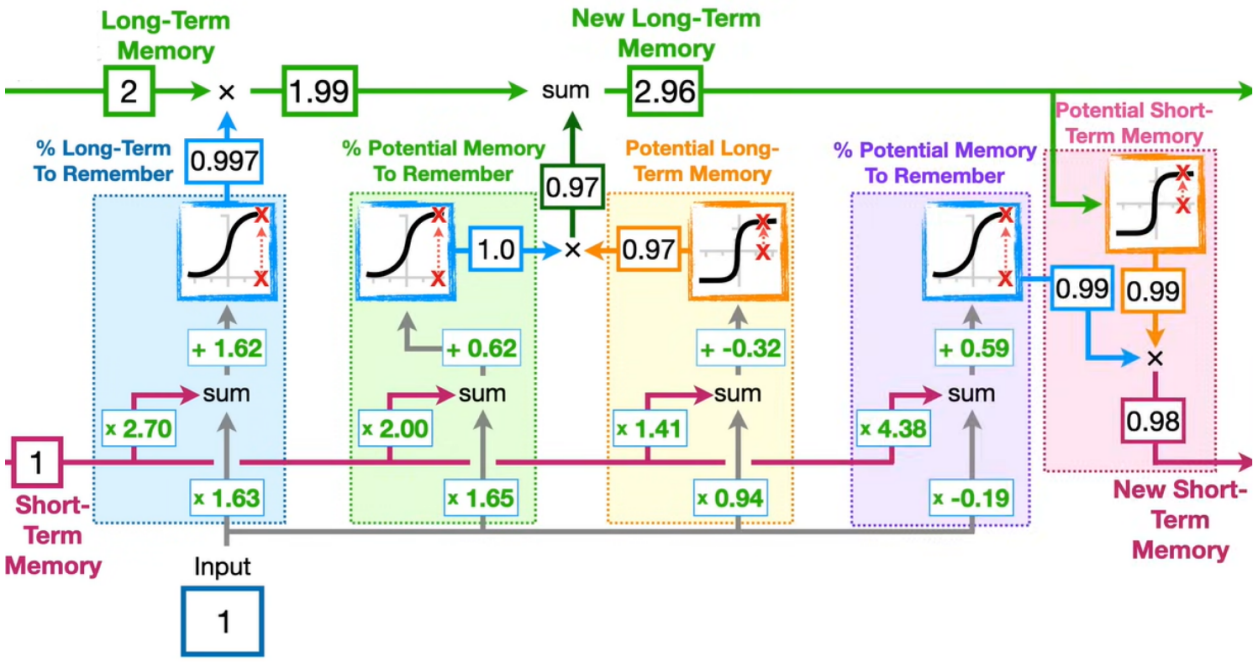
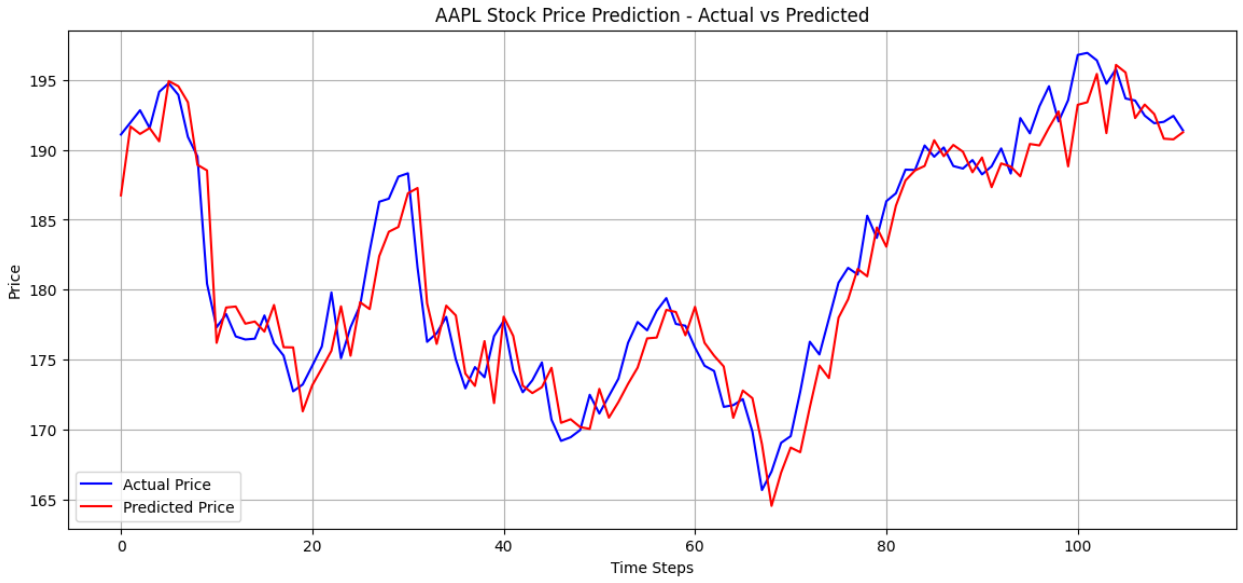
AI-Driven Portfolio Optimization and Budget Planner

[Long Short-Term Memory (LSTM), A* Star and CSP]





Problem Statement

"Develop an AI-driven portfolio optimization and budget planning system that leverages A* search for exploring investment allocations and constraint satisfaction techniques for enforcing budget and risk limits. The system should integrate stock price prediction models as heuristics to enhance strategic decision-making under uncertainty, thereby providing personalized, data-driven financial recommendations."

In today's fast-paced financial environment, individuals often struggle with effectively managing their personal finances. Investment decisions are typically influenced by limited knowledge, emotional bias, and lack of access to adaptive financial tools. While many budgeting apps and investment calculators exist, they are often static, offering generic advice that does not adapt to a user's dynamic financial situation, risk profile, or changing market trends.

Furthermore, novice investors face several challenges:

1) Uncertainty in financial markets: Predicting future stock movements and returns is inherently complex and risky. Many users do not have access to reliable forecasting tools.

2) Complex trade-offs: Users must balance short-term spending needs, long-term savings goals, and varying investment options — all under fixed income constraints.

3) Lack of optimization: Most users follow trial-and-error approaches without considering systematic methods to maximize returns while minimizing risk.

4) Constraint violations: Investment plans often ignore essential personal finance constraints like required monthly savings, emergency buffers, and expense thresholds.

Thus, there is a clear need for a personalized, intelligent, and constraint-aware financial planning system that can suggest optimal investment strategies based on a user's specific financial situation, while accounting for risk, return, and feasibility.

Objectives

The main goal of this project is to develop an AI-driven financial assistant that:

1. Recommends optimal investment allocations using the A* search algorithm, tailored to user constraints and goals.

2. Ensures financial feasibility through a Constraint Satisfaction Problem (CSP) solver that respects income, savings, and expense thresholds.

3. Integrates stock price prediction model (LSTM) to forecast market trends and guide investment decisions.

Why LSTM for This Project

LSTM (Long Short-Term Memory) networks are particularly well-suited for modeling sequential financial data due to their ability to capture temporal dependencies and non-linear relationships over time. In this project, we aimed to predict next-day log returns of stocks based on a series of technical indicators, moving averages, and momentum features derived from historical stock data. These inputs exhibit strong temporal dynamics, where patterns in recent days significantly influence future returns. Traditional feedforward networks fail to retain sequential context, whereas LSTM networks maintain memory of prior states, enabling them to recognize trends, volatility shifts, and cyclical patterns that affect stock behavior. Additionally, by incorporating ticker and sector embeddings, the LSTM architecture allowed the model to learn generalized yet stock- and sector-specific temporal behaviors. This was critical in building a global yet context-aware predictor, making LSTM an ideal choice for time-series forecasting in stock price movements.

Workflow

1. **Data Pipeline Phase**: The workflow starts with the data pipeline, which collects and processes raw stock data (e.g., prices, technical indicators) into a structured format (combined_stock_features.csv).
2. **LSTM Processing Phase**: This feeds into the LSTM code, where Part 1 preprocesses the data (scaling, sequencing) and Part 2 trains sector-specific LSTM models to predict log returns and compute volatility, outputting predictions to CSV files.
3. **Heuristic Analysis Phase**: The heuristic code then analyzes these predictions, calculating financial metrics (Sharpe, Kelly, Sortino, Custom) for each stock and normalizing scores to identify the best heuristic per stock.
4. **Constraint Satisfaction Phase**: Next, the CSP code uses these heuristics in the BudgetCSP class to allocate budget discretely across stocks, respecting budget, risk, and diversification constraints, while the PortfolioOptimizer class optimizes continuous weights for maximum Sharpe ratio.
5. **Final Optimization Phase**: Finally, the A* code builds on BudgetCSP, using A* search to optimize portfolio allocation based on heuristic scores, ensuring constraints are met, and falling back to a greedy solution if needed, producing the final portfolio.

Stock Data Pipeline

Code Overview

This is a data pipeline designed to fetch, process, and store stock market data for a predefined list of tickers. It uses the yfinance library to download historical stock data, calculates technical indicators (e.g., moving averages, RSI, Bollinger Bands), and saves the processed data in CSV files organized by ticker and sector. The pipeline also merges smaller sectors into a single category ("Physical Assets & Resources") and generates both individual and combined datasets.

Key Functionalities:

1. **Sector Mapping**: Dynamically retrieves sector information for each ticker and merges specified sectors.
2. **Data Retrieval**: Downloads historical stock data for a list of tickers from Yahoo Finance.
3. **Data Processing**:
   - Cleans data by ensuring numeric types and handling missing values.
   - Computes technical indicators: moving averages (10, 50, 100, 200 days), RSI (14-day), normalized volume, Bollinger Bands, log returns, and percentage change.

Purpose:

The pipeline is designed for financial data analysis, likely for use in quantitative finance, algorithmic trading, or portfolio management. It provides a structured way to preprocess stock data and generate features suitable for machine learning or statistical analysis.

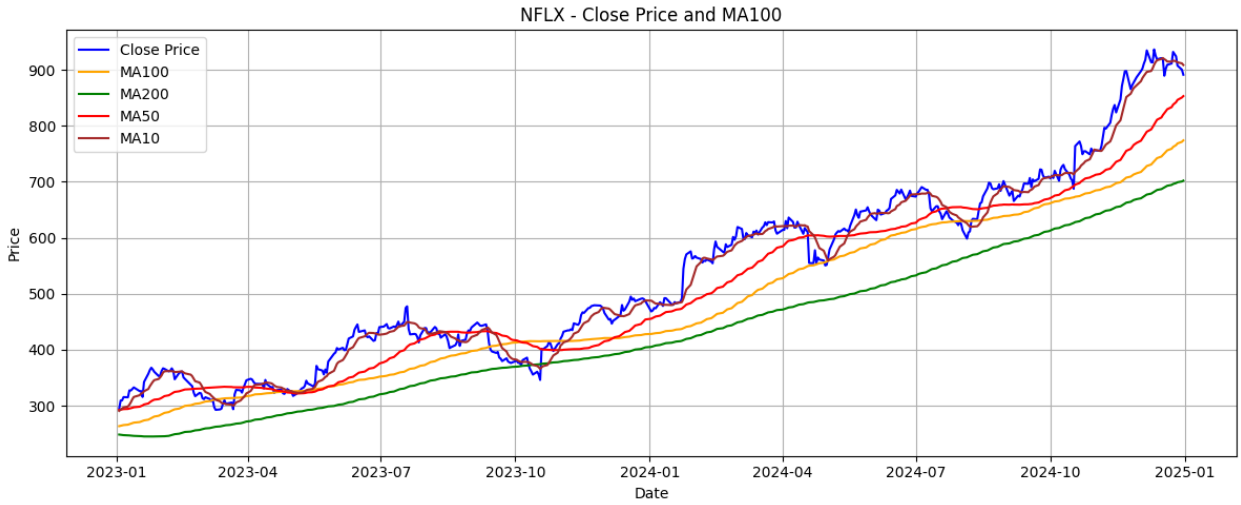
Explanation

Data Processing Functions:

1. calculate_moving_averages(df):

```
def calculate_moving_averages(df):
    df['MA10'] = df['Close'].rolling(window=10).mean()
    df['MA50'] = df['Close'].rolling(window=50).mean()
    df['MA100'] = df['Close'].rolling(window=100).mean()
    df['MA200'] = df['Close'].rolling(window=200).mean()
    return df
```

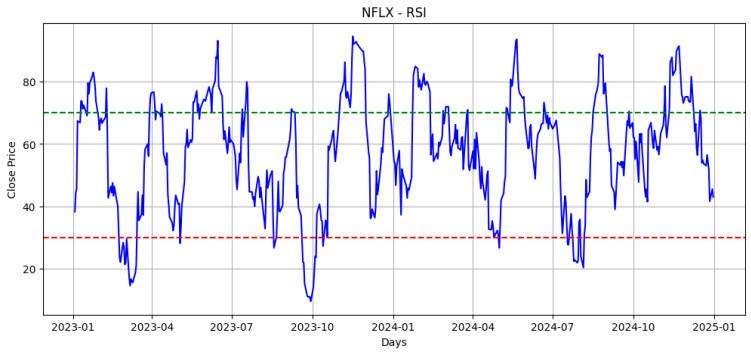
- Calculates simple moving averages (SMAs) over 10, 50, 100, and 200 days for the Close price.
- Uses pandas' rolling function for efficient computation.



2. calculate_rsi(df, period=14):

```
def calculate_rsi(df, period=14):
    delta = df['Close'].diff()
    gain = delta.where(delta > 0, 0).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
    rs = gain / loss
    df['RSI'] = 100 - (100 / (1 + rs))
    return df
```

- Computes the 14-day RSI, a momentum indicator.
- Calculates price changes (delta), separates gains and losses, and computes the relative strength (RS).
- RSI formula: 100 - (100 / (1 + RS)).



3. normalize_volume(df, window=50):

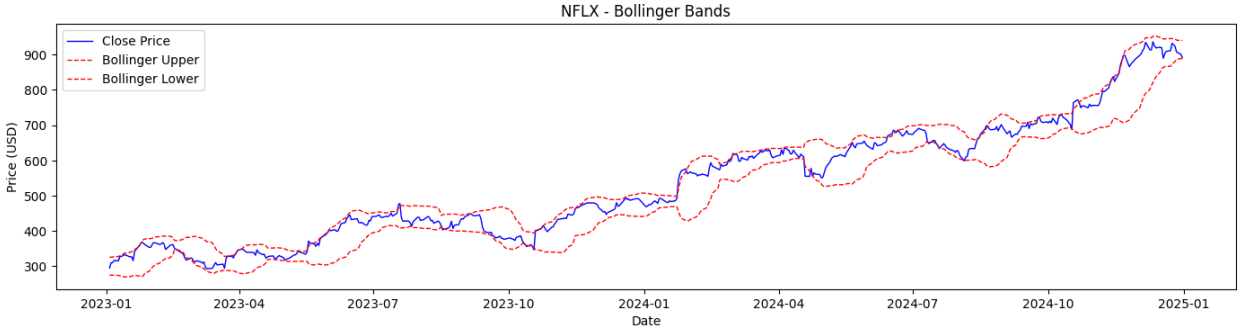
```
def normalize_volume(df, window=50):
    df['Norm_Volume'] = df['Volume'] / df['Volume'].rolling(window=window).mean()
    return df
```

- Normalizes trading volume by dividing by its 50-day moving average, useful for comparing volume across stocks.

4. calculate_bollinger_bands(df, window=20, num_std=2):

```
def calculate_bollinger_bands(df, window=20, num_std=2):
    rolling_mean = df['Close'].rolling(window=window).mean()
    rolling_std = df['Close'].rolling(window=window).std()
    df['Bollinger_Upper'] = rolling_mean + (rolling_std * num_std)
    df['Bollinger_Lower'] = rolling_mean - (rolling_std * num_std)
    return df
```

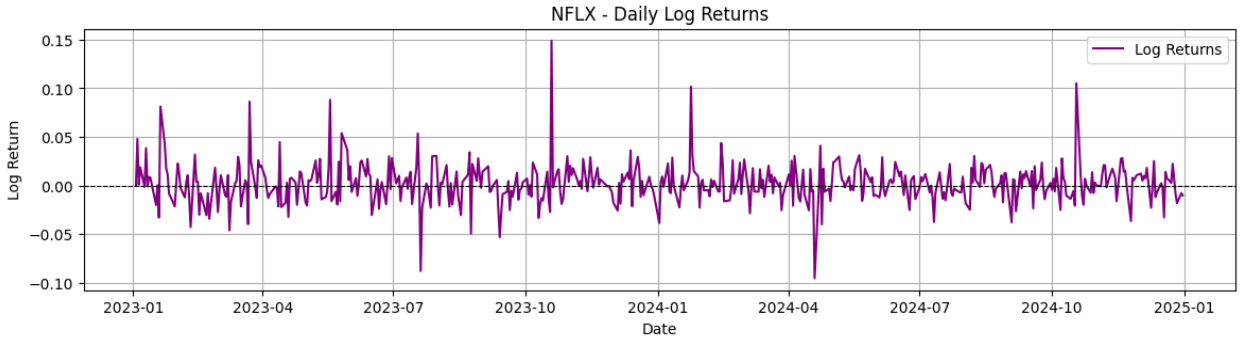
- Computes 20-day Bollinger Bands, with upper and lower bands set at 2 standard deviations from the moving average.



5. calculate_additional_returns(df):

```
def calculate_additional_returns(df):
    df['Log_Returns'] = np.log(df['Close'] / df['Close'].shift(1))
    df['Pct_Change'] = df['Close'].pct_change()
    return df
```

- Calculates daily log returns and percentage change for the Close price, useful for volatility and performance analysis.



Output:

Saves processed data per ticker, combines all data into combined_stock_features.csv, and creates sector-specific CSVs.

This combined_stock_features.csv will be used as input to our LSTM model.

LSTM Model

Code Overview

LSTM pipeline system designed to preprocess stock market data and train a custom Long Short-Term Memory (LSTM) model to predict stock log returns. The pipeline handles data loading, feature selection, scaling, and sequence creation, and implements the LSTM model, sector-wise training, prediction, volatility calculation, and walk-forward validation. The pipeline processes data from combined_stock_features.csv (generated by a prior data pipeline) and focuses on sector-specific modeling, with an example execution for the Technology sector.

Key Functionalities:

1. **LSTM Model Implementation**:
   - Implements a custom LSTM with ticker and sector embeddings, forward/backward propagation, and Adam optimization.
   - Trains sector-specific LSTM models, supporting fine-tuning of existing models.
2. **Prediction and Volatility**:
   - Predicts log returns and inverse-scales them using saved scalers.
   - Calculates rolling volatility (10-day standard deviation) of predictions.
3. **Walk-Forward Validation**:
   - Performs sliding-window validation (365-day training, 30-day testing) to evaluate model performance.
   - Computes metrics (MAE, MSE, RMSE) and plots actual vs. predicted log returns.

Purpose:

The pipeline is designed for financial time-series forecasting, specifically predicting stock log returns for trading strategies or portfolio management. The sector-wise approach captures sector-specific patterns, and walk-forward validation ensures realistic evaluation over time.

Explanation

create_sequences_multi Function:

```
def create_sequences_multi(data, ticker_ids, sector_ids, seq_length=50, target_col='
    Log_Returns'):
    """Create sequences from multiple features (without including Ticker_ID) for sector-wise
        training"""
    X, y, ticker_seq_list, sector_seq_list = [], [], [], []
    data_values = data.drop(['Ticker_ID', 'Sector_ID'], axis=1).values

    for sector_id in np.unique(sector_ids):
        sector_data = data[data['Sector_ID'] == sector_id]
        sector_ticker_ids = sector_data['Ticker_ID'].values

        for ticker_id in np.unique(sector_ticker_ids):
            ticker_data = sector_data[sector_data['Ticker_ID'] == ticker_id]
            features = ticker_data.drop(['Log_Returns', 'Ticker_ID', 'Sector_ID'], axis=1).
                values
            targets = ticker_data['Log_Returns'].values

            for i in range(len(features) - seq_length):
                X.append(features[i:i+seq_length, :])
                ticker_seq_list.append(ticker_id)
                sector_seq_list.append(sector_id)
                y.append(targets[i + seq_length])

    return np.array(X), np.array(y), np.array(ticker_seq_list), np.array(sector_seq_list)
```

- Generates sequences of length 50 for LSTM input, grouped by sector and ticker.
- Excludes Ticker_ID and Sector_ID from input features but tracks them for each sequence.
- For each sector and ticker, creates sequences of features (excluding Log_Returns) and corresponding target values (Log_Returns at t+seq_length).
- Returns NumPy arrays for features (X), targets (y), ticker IDs, and sector IDs.

LSTM Class

```
class LSTM:
    def __init__(self, input_dim, hidden_dim, output_dim, ticker_dim, embedding_dim,
        sector_dim, learning_rate=0.01, beta1=0.9, beta2=0.999, epsilon=1e-8):
        # Initialization of dimensions, hyperparameters, weights, biases, and embeddings
```

- **Initialization**:
  - Parameters: input_dim (number of features), hidden_dim (LSTM hidden state size), output_dim (1 for log returns), ticker_dim (number of unique tickers), embedding_dim (size of ticker/sector embeddings), sector_dim (number of unique sectors), learning_rate, beta1, beta2, epsilon (Adam optimizer parameters).
  - Weights: Initializes gate weights (Wf, Wi, Wc, Wo) and biases (bf, bi, bc, bo) for LSTM gates, plus output layer weights (Wy, by).
  - Embeddings: Creates ticker_embedding and sector_embedding matrices for ticker and sector IDs.
  - Adam: Initializes moment estimates (m, v) for Adam optimization.

```
    def _init_adam_params(self):
        # Initializes moment estimates for Adam optimizer
        self.m = {}
        self.v = {}
        for param_name in ['Wf', 'Wi', 'Wc', 'Wo', 'Wy', 'bf', 'bi', 'bc', 'bo', 'by']:
            param = getattr(self, param_name)
            self.m[param_name] = np.zeros_like(param)
            self.v[param_name] = np.zeros_like(param)
```

- Creates zero-initialized dictionaries for first (m) and second (v) moments for all parameters.

```
    def sigmoid(self, x): return 1 / (1 + np.exp(-x))
    def dsigmoid(self, x): return x * (1 - x)
    def tanh(self, x): return np.tanh(x)
    def dtanh(self, x): return 1 - x ** 2
```

- Implements activation functions (sigmoid, tanh) and their derivatives for backpropagation.

```
    def forward(self, x_seq, ticker_id, sector_id, h=None, c=None):
        # Forward pass through LSTM
        ft = self.sigmoid(np.dot(self.Wf, concat) + self.bf)
        it = self.sigmoid(np.dot(self.Wi, concat) + self.bi)
        c_tilde = self.tanh(np.dot(self.Wc, concat) + self.bc)
        c = ft * c + it * c_tilde
        ot = self.sigmoid(np.dot(self.Wo, concat) + self.bo)
        h = ot * self.tanh(c)
        self.caches.append((h, c, ft, it, c_tilde, ot, concat))
        y_hat = np.dot(self.Wy, h) + self.by
        return y_hat, h, c
```

- For each timestep, concatenates the previous hidden state (h), input (x), ticker embedding, and sector embedding.
- Computes gate activations (forget ft, input it, cell candidate c_tilde, output ot) and updates cell state (c) and hidden state (h).
- Returns the final prediction (y_hat), hidden state, and cell state.

```
    def backward(self, x_seq, y_hat, y_true):
        # Backward pass for gradient computation
        dh_next = np.zeros((self.hidden_dim, 1))
        dc_next = np.zeros((self.hidden_dim, 1))
        grads = {
            'Wf': np.zeros_like(self.Wf),
            'Wi': np.zeros_like(self.Wi),
            'Wc': np.zeros_like(self.Wc),
            'Wo': np.zeros_like(self.Wo),
            'Wy': np.zeros_like(self.Wy),
            'bf': np.zeros_like(self.bf),
            'bi': np.zeros_like(self.bi),
            'bc': np.zeros_like(self.bc),
            'bo': np.zeros_like(self.bo),
            'by': np.zeros_like(self.by)
        }
        dy = y_hat - y_true
        grads['Wy'] += np.dot(dy, self.caches[-1][0].T)
        grads['by'] += dy
        dh = np.dot(self.Wy.T, dy) + dh_next

        for t in reversed(range(len(x_seq))):
            h, c, ft, it, c_tilde, ot, concat = self.caches[t]
```

```
            c_prev = self.caches[t - 1][1] if t > 0 else np.zeros_like(c)
            do = dh * self.tanh(c)
            do_raw = do * self.dsigmoid(ot)
            dc = dh * ot * self.dtanh(self.tanh(c)) + dc_next
            dc_tilde = dc * it
            dc_tilde_raw = dc_tilde * self.dtanh(c_tilde)
            di = dc * c_tilde
            di_raw = di * self.dsigmoid(it)
            df = dc * c_prev
            df_raw = df * self.dsigmoid(ft)
            grads['Wf'] += np.dot(df_raw, concat.T)
            grads['Wi'] += np.dot(di_raw, concat.T)
            grads['Wc'] += np.dot(dc_tilde_raw, concat.T)
            grads['Wo'] += np.dot(do_raw, concat.T)
            grads['bf'] += df_raw
            grads['bi'] += di_raw
            grads['bc'] += dc_tilde_raw
            grads['bo'] += do_raw
            dconcat = (np.dot(self.Wf.T, df_raw) + np.dot(self.Wi.T, di_raw) + np.dot(self.
                Wc.T, dc_tilde_raw) + np.dot(self.Wo.T, do_raw))
            dh = dconcat[:self.hidden_dim, :]
            dc_next = dc * ft
        self._apply_adam(grads)
        self.t += 1  # Increment timestep
```

- Computes gradients for all weights and biases using backpropagation through time (BPTT).
- Propagates error from the output layer backward through the sequence, updating gradients for each gate and the output layer.
- Calls _apply_adam to update parameters.

```
    def _apply_adam(self, grads):
        # Applies Adam optimization to update parameters
        for param_name in grads:
            grad = grads[param_name]
            self.m[param_name] = self.beta1 * self.m[param_name] + (1 - self.beta1) * grad
            self.v[param_name] = self.beta2 * self.v[param_name] + (1 - self.beta2) * (grad
                ** 2)
            m_hat = self.m[param_name] / (1 - self.beta1 ** self.t)
            v_hat = self.v[param_name] / (1 - self.beta2 ** self.t)
            param = getattr(self, param_name)
            param -= self.lr * m_hat / (np.sqrt(v_hat) + self.epsilon)
            setattr(self, param_name, param)
```

- Updates weights and biases using Adam optimization, incorporating first and second moments with bias correction.

walk_forward_validation Function:

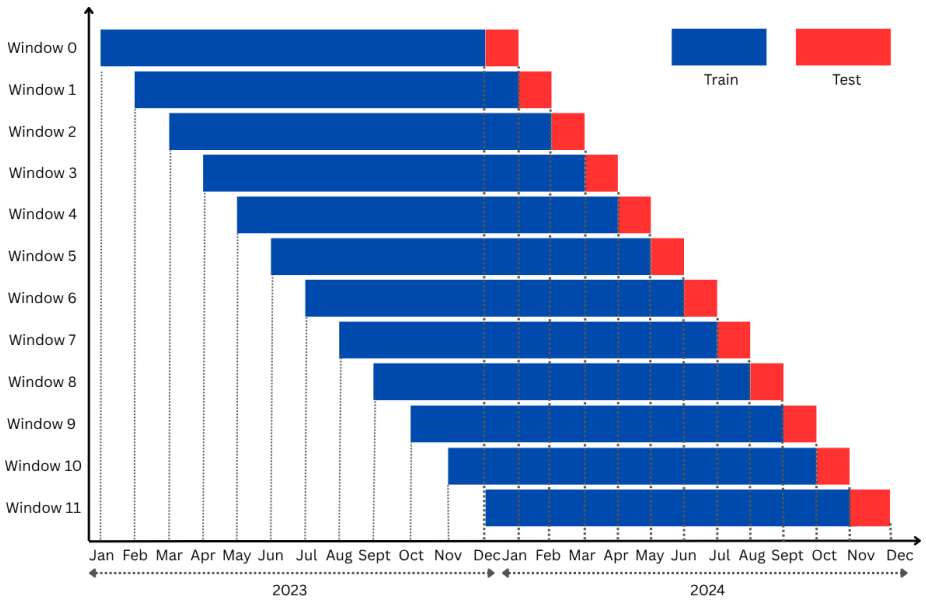
```
def walk_forward_validation(data_df, sector_id, params, start_date, end_date,
    train_window_days=365, test_window_days=30):
    # Performs walk-forward validation
    while current_date + timedelta(days=train_window_days + test_window_days) <= end:
        train_start = current_date
        train_end = train_start + timedelta(days=train_window_days)
        test_start = train_end
        test_end = test_start + timedelta(days=test_window_days)
        print(f"\n=== Walk {walk_id}: {train_start.date()} to {test_end.date()} ===")
        # Filter training data
        train_data = data_df[(data_df['Date'] >= train_start) & (data_df['Date'] < train_end
            ) & (data_df['sector_id'] == sector_id)]
```

```
        test_data = data_df[(data_df['Date'] >= test_start) & (data_df['Date'] < test_end) &
             (data_df['sector_id'] == sector_id)]
        if len(train_data) == 0 or len(test_data) == 0:
            print("Skipping due to lack of data.")
            current_date += timedelta(days=test_window_days)
            continue

        # Train model on current window
        sector_models = train_lstm_sectorwise(X_train, y_train, train_tickers, train_sectors
            , sector_id, params)
        # Predict log_returns = predict_logreturns(sector_models[sector_id], X_test,
            test_tickers, sector_id, scalers)
        # Inverse scale the true values (y_test) and predictions
        min_target, max_target = scalers['target']
        y_test_true = y_test * (max_target - min_target) + min_target
        y_pred = np.array(log_returns)

        print(f"Walk {walk_id} - MAE: {mae:.5f}, RMSE: {rmse:.5f}, MSE: {mse:.5f}")
        # Volatility
        volatility = calculate_volatility(log_returns)
```

- Iterates over time windows (365-day training, 30-day testing).
- Filters data by date and sector, trains an LSTM model, and makes predictions.

Why Walk-Forward Validation

Walk-forward validation was essential for this project because it mirrors the real-world scenario of deploying models in a financial environment, where future data is never known and predictions must be made on unseen, forward-moving time windows. Financial markets are non-stationary, meaning their statistical properties (e.g., volatility, return distributions) evolve over time. Traditional cross-validation would risk data leakage by mixing future data into the training process, leading to overfitting and overly optimistic results. Walk-forward validation addresses this by training the model on a fixed-length historical window (e.g., 1 year) and testing it on the immediate future window (e.g., 1 month), then rolling the window forward. This allowed us to evaluate the model's robustness over multiple time periods, adapt to changing market conditions, and simulate how the model would behave when deployed live. It also enabled periodic recalibration, ensuring our predictions were always based on the most recent and relevant data trends.



Heuristic Calculation

Code Overview

The provided Python script processes a DataFrame containing stock data (likely from the LSTM pipeline's output) to compute four financial heuristics—Sharpe Ratio, Kelly Criterion, Sortino Ratio, and a custom metric—for each stock. It handles numerical issues (e.g., zero volatility, infinite values), normalizes the heuristic scores, and identifies the best heuristic and corresponding equation for each stock. The output includes normalized scores and a summary of the best heuristic per stock, along with sector information.

Key Functionalities:

1. **Data Cleaning**:
   - Replaces zero volatility with a small epsilon value to avoid division-by-zero.
   - Replaces infinite log returns and volatility with NaN and fills with zero or epsilon.
2. **Heuristic Calculation**:
   - Computes Sharpe Ratio (Log_Returns / Volatility), Kelly Criterion ((Log_Returns - variance) / Volatility), Sortino Ratio (Log_Returns / downside standard deviation), and a custom metric (product of Sharpe and Sortino).
   - Uses median values to summarize scores per stock.

Purpose:

The script evaluates stock performance using financial heuristics, likely for portfolio optimization, risk assessment, or trading strategy development. It complements the LSTM pipeline by analyzing predicted log returns and volatility, enabling comparison of stocks within and across sectors.

Explanation

Heuristic Calculation:

```
for stock in df["Stock"].unique():
    stock_df = df[df["Stock"] == stock]
    r = stock_df["Log_Returns"]
    v = stock_df["Volatility"].replace(0, epsilon)
    downside = r[r < 0]
    downside_std = downside.std() if downside.std() != 0 else epsilon

    sharpe_scores = r / v
    kelly_scores = (r - r.var()) / v
    sortino_scores = r / downside_std
    custom_scores = sharpe_scores * sortino_scores

    results["Sharpe"][stock] = np.median(sharpe_scores)
    results["Kelly"][stock] = np.median(kelly_scores)
    results["Sortino"][stock] = np.median(sortino_scores)
    results["Custom"][stock] = np.median(custom_scores)
```

- Iterates over unique stocks in the DataFrame.
- Filters data for each stock (stock_df).
- Extracts Log_Returns (r) and Volatility (v), replacing zero volatility with epsilon.
- Computes downside returns (r ¡ 0) and their standard deviation, using epsilon if zero.

- Calculates:
  - **Sharpe Ratio**: Log_Returns / Volatility, measuring return per unit of risk.
  - **Kelly Criterion**: (Log_Returns - variance) / Volatility, approximating optimal bet sizing.
  - **Sortino Ratio**: Log_Returns / downside_std, focusing on downside risk.
  - **Custom Metric**: Product of Sharpe and Sortino ratios, combining total and downside risk.
- Stores the median score for each heuristic to reduce sensitivity to outliers.

CSP Implementation

Code Overview

This Python script implements two portfolio optimization approaches: a discrete BudgetCSP class for budget-constrained stock allocation and a continuous PortfolioOptimizer class for maximizing the Sharpe ratio. It includes a helper function (fetch_data_cached) for fetching stock data with caching and retries, and test cases to validate both modules. The code is designed for financial planning, optimizing stock portfolios within budget, risk, and diversification constraints.

Key Functionalities:

1. **BudgetCSP**: Allocates budget to stocks using backtracking, enforcing budget, risk, and diversification constraints.
2. **PortfolioOptimizer**: Optimizes portfolio weights to maximize Sharpe ratio using SLSQP.

Purpose:

The script optimizes stock portfolios for investment strategies, balancing budget constraints, risk tolerance, and diversification, integrating with heuristic outputs from prior analysis.

Explanation

BudgetCSP Class

```
class BudgetCSP:
    def __init__(self, income, expenses, required_savings, risk_tolerance, curated_options=
        None, current_portfolio=None, step=50, min_stocks=2):
        def risk_tolerance_percentage(self): return {"low": 0.25, "medium": 0.50, "high":
            0.75}.get(self.risk_tolerance, 0.50)

        def update_investment_options_with_yfinance(self):
            risk_pct = self.risk_tolerance_percentage()
            base_max = int(self._available_budget * risk_pct)
            for stock in self.curated_options:
                step_max = (base_max // self.step) * self.step
                final_max = min(self.investment_options[stock]['max'], step_max)
                self.investment_options[stock]['max'] = final_max
```

- Initializes income, expenses, required_savings, risk_tolerance (low, medium, high), curated_options (stock list, defaults to 100+ tickers), current_portfolio (defaults to empty), step (allocation granularity, $50), and min_stocks (minimum non-zero allocations, 2). Sets default max allocation ($1000 per stock).
- (backtracking_search_optimal): Uses backtracking to maximize budget utilization; updates best solution if higher total.

```
def backtracking_search_optimal(self, assignment=None):
    assignment = assignment or {}
    if self.best_total == self._available_budget:
        return
    unassigned = [v for v in self.variables if v not in assignment]
    if not unassigned:
        current_total = sum(assignment.values())
        if current_total > self.best_total:
            self.best_total = current_total
            self.best_solution = assignment.copy()
            print(f"Found ${current_total}/{self._available_budget}")
        return
    next_var = unassigned[0]
    allocated = sum(assignment.values())
    for value in self.variables[next_var]:
        if (allocated + value) > self._available_budget:
            continue
        if self.is_consistent(assignment, next_var, value):
            assignment[next_var] = value
            self.backtracking_search_optimal(assignment)
            del assignment[next_var]
    if self.best_total == self._available_budget:
        return
```

PortfolioOptimizer Class

```
class PortfolioOptimizer:
    def __init__(self, tickers, available_budget, risk_free_rate=0.01, lookback_days=252):
```

- (compute_statistics): Computes annualized returns and covariance from daily returns.

```
def compute_statistics(self, prices):
    returns = prices.pct_change().dropna()
    self.returns = returns
    mu = returns.mean() * 252
    sigma = returns.cov() * 252
    self.expected_returns = mu
    self.cov_matrix = sigma
```

- **Explanation**: Annualizes mean returns and covariance for portfolio calculations.
- (sharpe_ratio): Calculates Sharpe ratio: (portfolio return - risk_free_rate) / volatility; returns -inf for zero volatility.

```
def sharpe_ratio(self, weights):
    port_return = np.dot(weights, self.expected_returns)
    port_vol = np.sqrt(np.dot(weights.T, np.dot(self.cov_matrix, weights)))
    if port_vol == 0:
        return -np.inf
    sharpe = (port_return - self.risk_free_rate) / port_vol
    return sharpe
```

- (optimize_portfolio): Uses SLSQP to maximize Sharpe ratio with equal-weight initial guess and sum-to-1 constraint.

```
def optimize_portfolio(self):
    n = len(self.tickers)
    x0 = np.ones(n) / n
    constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})
```

```
    bounds = tuple((0, 1) for _ in range(n))
    result = minimize(self.negative_sharpe, x0, method='SLSQP', bounds=bounds, constraints=
        constraints)
    if result.success:
        self.weights = result.x
    else:
        raise ValueError("Optimization did not converge")
    return self.weights

result = minimize(lambda w: -self.sharpe_ratio(w), x0, method='SLSQP', bounds=bounds,
    constraints=constraints)
```

- **Explanation**: Optimizes weights to maximize Sharpe ratio; raises error if optimization fails.
- (get_allocations): Converts weights to dollar allocations: weight * available_budget.

```
def get_allocations(self):
    if self.weights is None:
        raise ValueError("You must run optimize_portfolio() first.")
    allocations = {ticker: weight * self.available_budget for ticker, weight in zip(self.
        tickers, self.weights)}
    return allocations
```

A* Portfolio Optimization

Code Overview

This Python script optimizes a stock portfolio using A* search, leveraging heuristic scores from Stocks - Heuristics.csv and constraints managed by the BudgetCSP class (imported from the CSP code). It loads heuristics for selected stocks, initializes a diversified starting portfolio, and uses A* to allocate budget while maximizing heuristic scores, respecting budget, risk, and diversification constraints.

Key Functionalities:

1. **A* Optimization**: Allocates budget using A* search, guided by heuristics and BudgetCSP constraints.
2. **Diversified Initial State**: Creates a starting portfolio meeting minimum stock requirements.
3. **Greedy Fallback**: Constructs a greedy solution if A* fails.

Purpose:

The script builds a diversified stock portfolio for investment, maximizing risk-adjusted returns within constraints, integrating with heuristic outputs and BudgetCSP for financial planning.

Explanation

AStarPortfolioOptimizer Class

```
class AStarPortfolioOptimizer:
    def __init__(self, initial_state, budget_csp, heuristics, step=50):
```

- Initializes empty initial_state, budget_csp (instance of BudgetCSP), heuristics, and step ($50); sets max_iterations (10000) and calls create_diversified_initial_state.
  - **Explanation**: Links to BudgetCSP for constraint checks.

- **(construct_greedy_solution)**: Builds a greedy portfolio if A* fails, starting with minimum allocations.

```
# Sort stocks by their heuristic value (descending)
sorted_stocks = sorted(self.heuristics.items(), key=lambda x: x[1], reverse=True)
# Ensure we have min_stocks with allocation
min_required = self.budget_csp.min_stocks
stocks_to_try = sorted_stocks[:min_required]
# Start with min allocation for each required stock (step size)
remaining_budget = self.budget_csp._available_budget
for stock, _ in stocks_to_try:
    state[stock] = self.step
    remaining_budget -= self.step
    if remaining_budget < 0:
        return None  # Not enough budget for minimum allocation
# Now distribute remaining budget according to heuristics
for idx, (stock, heuristic) in enumerate(sorted_stocks):
    # Only allow additional allocation if budget remains
    if remaining_budget <= 0:
        break
    # Calculate maximum additional allocation
    max_additional = min(
        remaining_budget,
        self.budget_csp.investment_options[stock]['max'] - state.get(stock, 0)
    )
    max_additional = (max_additional // self.step) * self.step
    if max_additional > 0:
        state[stock] = state.get(stock, 0) + max_additional
        remaining_budget -= max_additional
# Check if this satisfies all constraints
if self.is_valid(state):
    return state
# If not valid, try a more conservative approach
state = {t: 0 for t in self.initial_state}
for i in range(min_required):
    if i < len(sorted_stocks):
        stock = sorted_stocks[i][0]
        state[stock] = self.step
return state if self.is_valid(state) else None
```

- **(g)**: Cost function: negative heuristic-weighted allocation plus unused budget penalty.

```
def g(self, state):
    invested = sum(state.values())
    unused_penalty = 0.0001 * (self.budget_csp._available_budget - invested)
    return -sum(self.heuristics.get(stock, 0) * amount for stock, amount in state.items()) +
         unused_penalty
```

- **(h)**: Heuristic function: estimates potential gain from remaining budget.

```
def h(self, state):
    invested = sum(state.values())
    remaining_budget = self.budget_csp._available_budget - invested
    if remaining_budget <= 0:
        return 0
    estimated_gain = 0
    for stock in self.heuristics:
        if stock in state:
            allocated = state[stock]
            max_alloc = self.budget_csp.investment_options[stock]['max']
            if allocated < max_alloc:
                estimated_gain += self.heuristics[stock]
    return -estimated_gain
```

- (search): Runs A* search, exploring states until a valid portfolio is found or max_iterations reached; falls back to greedy solution.

```
def search(self):
    start_state = self.diversified_initial_state
    heapq.heappush(open_set, (self.f(start_state), next(counter), start_state))
    visited = set()
    best_valid_state = None
    best_valid_score = float('inf')

    while open_set and self.iterations < self.max_iterations:
        self.iterations += 1
        current_f, _, current = heapq.heappop(open_set)
        current_key = frozenset(current.items())
        if current_key in visited:
            continue
        visited.add(current_key)

        # Check if this state satisfies all constraints
        if self.is_valid(current):
            current_score = self.f(current)
            if current_score < best_valid_score:
                best_valid_score = current_score
                best_valid_state = current

        neighbors = self.neighbors(current)
        for neighbor in neighbors:
            neighbor_key = frozenset(neighbor.items())
            if neighbor_key not in visited:
                heapq.heappush(open_set, (self.f(neighbor), next(counter), neighbor))

    return best_valid_state
```

- **Explanation**: Uses heapq for priority queue; tracks visited states.