ChatGPT

# ML-Based Stock Selection for Swing Trading and Portfolio Optimization

## Introduction

Swing trading involves holding stocks for a short period (a few days to a few weeks) to capture price swings. In this task, we aim to use machine learning (ML) to predict which stocks (from a given list of NSE stocks) are likely to achieve a target return of **x%** within **y** days, and then determine an optimal portfolio allocation for those stocks. All key parameters will be defined as variables in the Python script for easy customization. This includes:

- **Target Return (`target_return`)** – e.g. `0.05` for 5% return.
- **Time Horizon (`time_horizon`)** – e.g. `10` for 10 trading days.
- **Stock List (`stock_list`)** – e.g. a list of NSE stock ticker symbols.
- **Feature Set (`features_to_include`)** – e.g. technical indicators or price/volume features to use in the ML model.
- **Risk Tolerance (`risk_tolerance`)** – e.g. maximum acceptable portfolio volatility or other risk measure.

By treating these inputs as variables, the script can be easily adapted to different targets, timeframes, stock universes, and risk preferences.

## Approach Overview

**1. Data Collection:** We will gather historical price data for the specified `stock_list` (e.g., using Yahoo Finance API via libraries like `yfinance` or `pandas_datareader`). Daily closing prices and volume will be the primary data, from which we can derive technical indicators and returns.

**2. Feature Engineering:** We will compute features to feed into the ML model. Common features include technical indicators (moving averages, RSI, etc.) and recent price/return patterns [1]. For simplicity, one effective approach is using **windowed returns** – e.g. the returns of the past few days – as features [2]. Additional features like moving average ratios or volume changes can also be included. These features help the model recognize short-term momentum or mean-reversion patterns relevant to swing trading.

**3. Defining the Target (Label):** We frame this as a classification problem: for each stock on each date in history, did it achieve at least *x%* return after *y* trading days? If yes, label that instance as `1` (successful swing trade), otherwise `0`. This approach mirrors that of recent research which classifies whether a stock's return over the next period exceeds a threshold (e.g. 2% in 10 days) [3]. For example, if `target_return = 0.05` (5%) and `time_horizon = 10`, and a stock's price today is ₹100, we check if the price 10 trading days later is $\geq$ ₹105. If so, that day's instance is a positive case (`1`); if not, it's a negative case (`0`). We create such labeled examples for all stocks in our list over the historical period.

**4. Model Training:** Using the prepared dataset, we train a machine learning classifier to predict the label. A robust choice is a **Random Forest** or **XGBoost** classifier, as they handle feature interactions well and are less prone to overfitting than plain decision trees. The model will learn patterns from the

features (recent returns, technical indicators, etc.) that precede a ≥x% jump in the next y days. We'll split data into training and testing sets (e.g. train on older data, test on more recent data) to ensure the model generalizes. The performance can be evaluated with metrics like accuracy or precision/recall on the test set, focusing on how well it identifies the positive cases (stocks hitting the target). *For brevity, we won't delve into performance tuning here, but one should be mindful of issues like class imbalance (typically far fewer days achieve large returns than not) and consider techniques like resampling or threshold tuning.*

**5. Prediction:** After training, we use the model to predict the probability or class for each stock's **current** features (most recent day's data). This yields a subset of stocks that the model predicts are likely to achieve the target return in the next y days (our swing trade candidates). For a more conservative approach, one might select the top N stocks by predicted probability or only those above a certain probability threshold, to balance confidence and diversification.

**6. Portfolio Optimization:** Once we have a set of candidate stocks, we determine optimal weights for a portfolio of these stocks. **Portfolio optimization** is the process of allocating capital among assets to maximize returns for a given level of risk (or minimize risk for a given target return) [4] . We will use a simplified **mean-variance optimization** approach (Markowitz Modern Portfolio Theory) to allocate weights. In mean-variance optimization, we supply the **expected returns** of each selected stock and their **risk (covariance matrix)**, and set up an objective to maximize return or Sharpe ratio for a given risk constraint [5] [6] .

- **Estimating Expected Return:** For each selected stock, a conservative estimate can be the target return *x% over y days*, or we could use the model's predicted probability * target return as a rough expected return. For simplicity, we might assume each selected stock is expected to achieve ~x% over the horizon (some may exceed, some may just meet it). If historical average returns or an outright regression model were available, those could refine this estimate.
- **Risk (Volatility/Covariance):** We will compute the historical volatility of each stock (and correlations between stocks) from recent data. This forms the covariance matrix used in optimization. `risk_tolerance` can be interpreted as a maximum allowed portfolio volatility (standard deviation). For example, if `risk_tolerance = 0.2`, that might correspond to ~20% annualized volatility (or an appropriate value for y-day volatility). We can enforce this by adding a constraint in the optimization that the portfolio's volatility ≤ risk_tolerance. Alternatively, we might directly maximize the Sharpe ratio (which inherently balances return vs. risk) and then scale down exposure to meet risk targets.
- **Optimization Solver:** To solve for optimal weights, we can use libraries like **PyPortfolioOpt** (which provides convenient methods for mean-variance optimization) [7] or set up the problem with `cvxopt` / `cvxpy` for custom constraints. In this script, we'll illustrate using PyPortfolioOpt's `EfficientFrontier` to maximize the Sharpe ratio (return/risk) or meet a target return. This will yield weights for each stock that maximize the portfolio's expected return for the given risk profile.

The end result will be: **(a)** a filtered list of stocks predicted to reach the target, and **(b)** an optimal weight allocation for those stocks in a swing trade portfolio.

---

## Implementation Steps and Code

Below is a comprehensive Python script following the above approach. Each section of the code is annotated for clarity. All key parameters are defined upfront as variables for easy customization.

### 1. Setup Variables and Imports

First, we import necessary libraries and define the variables: target return, time horizon, stock list, features to include, and risk tolerance. We assume the use of `yfinance` to get historical data (ensure to install it with `pip install yfinance` if not already installed). We also import `pandas` for data handling and `numpy` for calculations. For the ML model, we import a classifier (e.g., RandomForest). For portfolio optimization, we use `pypfopt` (PyPortfolioOpt) library – this can be installed with `pip install pyportfolioopt`.

```
import pandas as pd
import numpy as np

# Machine learning model (Random Forest Classifier for example)
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Portfolio optimization library
from pypfopt import risk_models, expected_returns
from pypfopt.efficient_frontier import EfficientFrontier

# Data source (yfinance for stock price data)
import yfinance as yf

# --------------------
# Define variables
# --------------------
target_return = 0.05   # e.g., 0.05 for 5% target return
time_horizon = 10      # e.g., 10 days holding period
stock_list = ["RELIANCE.NS", "TCS.NS", "INFY.NS"]  # Example NSE stock
tickers (replace with your list)
features_to_include = ["returns", "moving_averages", "volume"]
# placeholder for feature types to use
risk_tolerance = 0.2
# e.g., 0.2 as maximum acceptable portfolio volatility (20% annualized)
```

**Notes:**
- The stock tickers for NSE stocks on Yahoo Finance typically have the suffix `.NS` (e.g., `RELIANCE.NS` for Reliance Industries on NSE). Ensure your list uses the correct ticker format for `yfinance`.
- `risk_tolerance` here is an abstract number; how it's used (annualized vs. period volatility) will be clarified in the optimization step.

### 2. Data Collection: Fetch Historical Price Data

We will download historical daily price data for the stocks in `stock_list`. Let's get at least enough data to train the model – for example, 3-5 years of data. Using `yfinance`, we fetch the Adjusted Close prices (which account for splits/dividends). We then combine these into a single DataFrame for convenience.

```
# Fetch historical daily data for all stocks in the list
# Define the period for historical data (e.g., last 5 years)
history_period = "5y"

# Dictionary to store each stock's DataFrame
price_data = {}
for ticker in stock_list:
    # Download data using yfinance
    stock_df = yf.download(ticker, period=history_period, interval="1d")
    # Keep only the 'Adj Close' column as the primary price series
    stock_df = stock_df[["Adj Close", "Volume"]].rename(columns={"Adj Close":
"Close"})
    price_data[ticker] = stock_df

# Align all data by date (inner join on dates to ensure all have same index
range)
all_data = pd.concat([price_data[ticker]["Close"] for ticker in stock_list],
axis=1, keys=stock_list).dropna()
print("Data sample:")
print(all_data.head())
```

This code will print a sample of the combined price data (columns are stock tickers, rows indexed by date). We drop dates where any stock is missing data (e.g., an IPO started later) for simplicity.

### 3. Feature Engineering

Now, we generate features for the ML model. We will iterate over each stock's dataframe and create features such as:

- Recent returns: e.g., 1-day return, 2-day return, ... up to `n` days prior. These capture momentum.
- Moving average indicators: e.g., ratio of current price to 5-day or 10-day moving average (to indicate trend position).
- Volume indicators: e.g., relative volume (current volume vs. average volume over last week).

For demonstration, let's compute 3 days of prior returns, a 5-day moving average ratio, and a 5-day volume average ratio for each stock. We'll then merge these features across all stocks into one big training dataset.

```
# Feature generation parameters
n_return_days = 3   # use past 3 days' returns as features
ma_window = 5       # moving average window
vol_window = 5      # volume moving avg window

feature_rows = []   # list to collect feature rows (each with features and
label)
column_names = []   # to store feature column names for reference

for ticker, df in price_data.items():
```

```
    df = df.copy()
    # Calculate daily returns
    df["return_1d"] = df["Close"].pct_change(1)
    df["return_2d"] = df["Close"].pct_change(2)
    df["return_3d"] = df["Close"].pct_change(3)
    # Moving average and price ratio
    df[f"ma{ma_window}"] = df["Close"].rolling(window=ma_window).mean()
    df[f"price_ma_ratio"] = df["Close"] / df[f"ma{ma_window}"]
    # Volume average and ratio
    df[f"vol_ma{vol_window}"] =
df["Volume"].rolling(window=vol_window).mean()
    df[f"vol_ratio"] = df["Volume"] / df[f"vol_ma{vol_window}"]
    # Shift features up by time_horizon to align with the day we make
prediction
    df[[f"return_{i}d" for i in range(1, n_return_days+1)] +
       [f"price_ma_ratio", f"vol_ratio"]] = df[[f"return_{i}d" for i in
range(1, n_return_days+1)] +
                                             [f"price_ma_ratio",
f"vol_ratio"]].shift(1)
    # Define the target label: price change over next 'time_horizon' days
    df["future_return"] = df["Close"].shift(-time_horizon) / df["Close"] -
1.0
    df["target"] = (df["future_return"] >= target_return).astype(int)
    # Drop rows with NaNs (from shifting or rolling windows)
    df = df.dropna()
    # Create feature matrix for this stock
    feature_cols = [f"return_{i}d" for i in range(1, n_return_days+1)] +
["price_ma_ratio", "vol_ratio"]
    for date, row in df.iterrows():
        features = [ticker]  # first element indicates which stock (optional)
        # Append feature values
        features += [row[col] for col in feature_cols]
        # Append target label
        features.append(row["target"])
        feature_rows.append(features)
    # Store column names once
    if not column_names:
        column_names = ["stock"] + feature_cols + ["target"]

# Create DataFrame of all training examples
training_data = pd.DataFrame(feature_rows, columns=column_names)
print("Feature sample:")
print(training_data.head())
print("Number of training examples:", len(training_data))
```

**Explanation:** We computed features for each stock and assembled them into a single `training_data` DataFrame. Each row corresponds to one stock on one date, with features like returns and ratios from **prior days** (shifted by 1 day so that on day *t* we use information up to day *t* to predict future), and a `target` label which looks *time_horizon* days ahead to see if the return exceeds our threshold. This labeling method is akin to Htun *et al.* (2024) who classified whether a stock's 10-day

ahead return beat a threshold (2% above index) [3] – we are directly using an x% threshold on price return.

### 4. Train/Test Split and Model Training

Next, we split the `training_data` into a training set and a test set to train the ML model and evaluate its performance. We can use scikit-learn's `train_test_split` (for example, 70% train, 30% test split). Then we train a RandomForestClassifier on the training data.

```
# Split into features (X) and label (y)
X = training_data.drop(columns=["stock", "target"])  # features (we drop
'stock' identifier and target)
y = training_data["target"]

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

# Initialize and train a Random Forest classifier
clf = RandomForestClassifier(n_estimators=100, random_state=42)
clf.fit(X_train, y_train)

# Optional: Evaluate model on test set
test_pred = clf.predict(X_test)
test_acc = (test_pred == y_test).mean()
print(f"Test Accuracy: {test_acc:.2%}")
```

Here we also print the test accuracy just to have an idea of performance. In practice, one would look at more metrics (precision, recall for the positive class) since this is often an imbalanced problem. The model, once trained, encapsulates the relationship between our features (recent returns, price ratios, etc.) and the outcome of achieving the target return in y days.

### 5. Making Predictions for Today's Candidates

With the model ready, we now prepare the **latest data** for each stock to get predictions for the next *y* days. We will take the most recent day's features for each stock (or we could take the last available row from each stock's feature DataFrame) and run the classifier to predict whether each stock is likely to meet the target.

```
# Prepare latest features for each stock to predict next y-days outcome
latest_features = []
latest_index = []
# to keep track of which stock corresponds to each feature row

for ticker, df in price_data.items():
    df = df.copy()
    # Compute features for the latest day (similar to above, but just the
last row)
    df["return_1d"] = df["Close"].pct_change(1)
```

```
    df["return_2d"] = df["Close"].pct_change(2)
    df["return_3d"] = df["Close"].pct_change(3)
    df[f"ma{ma_window}"] = df["Close"].rolling(window=ma_window).mean()
    df[f"price_ma_ratio"] = df["Close"] / df[f"ma{ma_window}"]
    df[f"vol_ma{vol_window}"] =
df["Volume"].rolling(window=vol_window).mean()
    df[f"vol_ratio"] = df["Volume"] / df[f"vol_ma{vol_window}"]
    df = df.dropna()
    if not df.empty:
        last_row = df.iloc[-1]  # last available day with indicators
        feat = [last_row[f"return_{i}d"] for i in range(1, n_return_days+1)]
+ [last_row["price_ma_ratio"], last_row["vol_ratio"]]
        latest_features.append(feat)
        latest_index.append(ticker)

# Convert to DataFrame for prediction
X_new = pd.DataFrame(latest_features, index=latest_index,
columns=[f"return_{i}d" for i in range(1, n_return_days+1)] +
["price_ma_ratio", "vol_ratio"])
pred_probs = clf.predict_proba(X_new)[:, 1]
# probability of class 1 (target achieved)
pred_labels = clf.predict(X_new)

# Compile prediction results
pred_df = pd.DataFrame({
    "Stock": latest_index,
    "Predicted_Label": pred_labels,
    "Predicted_Prob": pred_probs
}).sort_values("Predicted_Prob", ascending=False)
print("Predictions for next", time_horizon, "days:")
print(pred_df)
```

This snippet gathers the latest feature values for each stock and then uses `clf.predict_proba` to get the probability of reaching the target (`class 1`). It also gets the binary prediction (`clf.predict`). The results are stored in `pred_df` with each stock's predicted label and probability, sorted by probability of success. From here, we can identify which stocks are the top candidates for our swing trade: for example, those with `Predicted_Label = 1` (or even limit to the top N by probability if we want only a few picks).

Suppose `pred_df` shows 2-3 stocks with high probabilities; these will be our list for portfolio allocation. For demonstration, let's filter the DataFrame for `Predicted_Label == 1`:

```
# Filter stocks predicted to achieve target
selected_stocks = pred_df[pred_df["Predicted_Label"] == 1]["Stock"].tolist()
if not selected_stocks:
    selected_stocks = [pred_df.iloc[0]["Stock"]]
# if none predicted 1, take top probability stock as a fallback
print("Selected stocks for portfolio (predicted to reach target):",
selected_stocks)
```

### 6. Portfolio Optimization for Selected Stocks

With `selected_stocks` identified, we perform portfolio optimization to allocate weights to these stocks. We will use a **mean-variance optimization** approach via PyPortfolioOpt's `EfficientFrontier`. The steps are:

- Calculate expected returns for the selected stocks. Here, we can use the historical mean return or simply use the `target_return` as an expected return over the *y*-day horizon (converted to an annualized figure if needed for consistency with volatility). For simplicity, we'll derive **daily expected returns** by dividing `target_return` by `time_horizon` (this assumes a linear return per day which is a rough approximation) and then annualize it. Alternatively, using PyPortfolioOpt's `expected_returns.mean_historical_return` on recent price data is common for getting annual expected returns.
- Calculate the covariance matrix of daily returns for the selected stocks (we can use PyPortfolioOpt's `risk_models.sample_cov` on recent returns). This will be annualized as well.
- Set up the Efficient Frontier optimization. We will aim to maximize the **Sharpe Ratio**, which finds an optimal balance of return vs. risk [6]. We can implicitly incorporate `risk_tolerance` by later checking the portfolio volatility; if it exceeds our tolerance, we might scale down positions or impose a volatility constraint using a different optimization method (like `EfficientFrontier.efficient_risk` which maximizes return for a given risk). For now, let's use the Sharpe-maximizing portfolio.

```
from pypfopt import risk_models, expected_returns

if len(selected_stocks) > 0:
    # Prepare price data for selected stocks
    selected_prices = all_data[selected_stocks]  # subset of the combined
data with selected stocks
    # Use last 1 year of data for return stats (for example)
    if len(selected_prices) > 252:
        recent_prices = selected_prices.iloc[-252:]  # approximately last 1
year of trading days
    else:
        recent_prices = selected_prices

    # Calculate expected annual returns (using historical mean returns)
    mu = expected_returns.mean_historical_return(recent_prices)  # default
returns are annualized
    # Calculate annualized covariance of returns
    S = risk_models.sample_cov(recent_prices)  # sample covariance
(annualized by default)

    # Initialize Efficient Frontier object
    ef = EfficientFrontier(mu, S)
    # Maximize Sharpe ratio (this gives us weights that maximize (return-
risk)/volatility)
    raw_weights = ef.max_sharpe()
    cleaned_weights = ef.clean_weights()  # clean small weights to 0
    print("Optimal weights (max Sharpe):", cleaned_weights)
    # Portfolio performance
```

```
    ret, vol, sharpe = ef.portfolio_performance(verbose=True)
    print(f"Expected annual return: {ret:.2%}, Volatility: {vol:.2%}, Sharpe
Ratio: {sharpe:.2f}")
```

The output will list the optimal weight for each selected stock (in `cleaned_weights`), and print the expected annual return, volatility, and Sharpe ratio of that portfolio. The **weights** tell us how to allocate our capital among the chosen stocks. For example, you might see something like `{ 'RELIANCE.NS': 0.67, 'TCS.NS': 0.33 }` meaning 67% of funds to Reliance and 33% to TCS. This result is based on maximizing Sharpe; if you have a specific risk tolerance, you could instead use `ef.efficient_risk(target_volatility)` or set a minimum return constraint with `ef.efficient_return(target_return, ...)`. The concept remains that we are balancing return and risk: **portfolio optimization selects assets such that the overall return is maximized while risk is minimized** [4].

*(Note: The expected return and volatility printed are annualized figures. If needed, one can convert the target y-day return into an annual expected return by scaling appropriately. In practice, ensure the return estimates align with the horizon or convert everything to the same timeframe.)*

### 7. Final Output Interpretation

Finally, using the above script, we have:

- **Predicted Swing Trade Candidates:** A list of stocks from `stock_list` that the ML model predicts are likely to achieve at least *x%* return in the next *y* days. These are stored in `selected_stocks`.
- **Optimal Portfolio Weights:** The dictionary `cleaned_weights` provides the optimal allocation for the `selected_stocks`. This portfolio is optimized for maximum return-to-risk (Sharpe ratio) given recent performance data. You can adjust the optimization to suit your risk preference (e.g., use `ef.efficient_risk(risk_tolerance)` to target a specific volatility).

Below is how you might present the final results (combining model prediction and portfolio allocation):

```
# Combine selected stocks and weights into a result output
if len(selected_stocks) > 0:
    result = pd.DataFrame({
        "Stock": list(cleaned_weights.keys()),
        "Weight": list(cleaned_weights.values()),
        "Predicted_Prob": [pred_df[pred_df.Stock == s]
["Predicted_Prob"].values[0] for s in cleaned_weights.keys()]
    })
    result = result.sort_values("Weight", ascending=False)
    print("Final recommended portfolio allocation:")
    print(result)
```

This will show each stock, its allocation weight, and (optionally) the model's predicted probability of hitting the target, sorted by weight. You can use this as a guideline for how to allocate capital in your swing trading strategy, focusing on the stocks deemed most promising by the ML model and balanced by the portfolio optimizer.

## Conclusion and Next Steps

This script provides a framework for ML-driven stock selection and portfolio optimization for swing trading. All key parameters (target return, horizon, stock list, features, risk tolerance) are easily adjustable. In practice, you should consider the following before using it live:

- **Data Quality and Freshness:** Ensure you have the latest data for all stocks and that any corporate actions (splits, dividends) are handled (using adjusted prices as we did).
- **Model Validation:** Evaluate the ML model's performance thoroughly. You may try more advanced models (like gradient boosting or neural networks) and feature engineering (including technical indicators or even sentiment data) to improve predictions [8] [1].
- **Risk Management:** The portfolio optimization here is a simplified view (max Sharpe). Real-world constraints like maximum position sizes, transaction costs, or sector diversification can be added. `risk_tolerance` can be more formally integrated by using efficient frontier methods that directly constrain volatility or CVaR. Always consider implementing stop-loss or take-profit levels in swing trades even if the model is confident.

By combining data-driven stock selection with rigorous portfolio optimization, this approach aims to increase the probability of achieving the desired swing trade returns while managing risk. It embodies the principle of maximizing returns and minimizing risk in portfolio construction [4], tailored to short-term trading. Feel free to customize the script further to suit your specific needs and to experiment with different features or optimization criteria for the best results.

**Sources:**


- Htun, H.H., Biehl, M., & Petkov, N. (2024). *Forecasting relative returns for S&P 500 stocks using machine learning*. Financial Innovation, 10(118). (Used for methodology of threshold-based stock return classification [3] [8] and discussion on technical indicator features [1].)
- Pierre, S. (2025). *An Introduction to Portfolio Optimization in Python*. *Built In*. (Referenced for the concept and methods of portfolio optimization, emphasizing maximizing returns while minimizing risk [4] and the use of mean-variance (Markowitz) optimization [5]. Also demonstrated usage of PyPortfolioOpt for efficient frontier optimization [6].)


---


[1] [2] [3] [8] Forecasting relative returns for S&P 500 stocks using machine learning | Financial Innovation | Full Text
https://jfin-swufe.springeropen.com/articles/10.1186/s40854-024-00644-0

[4] [5] [6] [7] An Introduction to Portfolio Optimization in Python | Built In
https://builtin.com/data-science/portfolio-optimization-python