project

December 17, 2024

0.0.1 Student Name: Chung-Mou Pan

0.0.2 Student Net Id: N14124164

0.0.3 All code in this file is written by myself.

[ ]:
```
import pandas as pd
import numpy as np
import yfinance as yf
import time
import requests
from datetime import timedelta
from scipy.optimize import minimize

from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.metrics import r2_score, accuracy_score, roc_auc_score
from sklearn.model_selection import TimeSeriesSplit, GridSearchCV
from sklearn.preprocessing import StandardScaler
from xgboost import XGBRegressor

import joblib
```

0.1 Data Pre-processing

[ ]:
```
# ====================================
# STEP 1: LOAD AND CLEAN CONGRESSIONAL TRADE DATA
# ====================================
df = pd.read_excel('congress-trading-all.xlsx')
df['Traded'] = pd.to_datetime(df['Traded'])

# Convert 'Trade_Size_USD' from strings to a numeric approximation:
# If it's in ranges like "$1,001 - $15,000", take the midpoint.
def parse_trade_size(trade_str):
    # Remove $, commas and split by '-'
    if pd.isna(trade_str) or trade_str.strip() == '':
        return np.nan
    parts = trade_str.replace('$', '').replace(',', '').split('-')
    if len(parts) == 2:
        # Take midpoint
```

```
        try:
            low = float(parts[0])
            high = float(parts[1])
        except ValueError:
            return np.nan
        return (low + high) / 2
    else:
        # If not a range, try directly converting
        return float(parts[0])

df['Trade_Size_Mid'] = df['Trade_Size_USD'].apply(parse_trade_size)

# Filter out rows without a valid ticker or date if necessary
df = df.dropna(subset=['Ticker', 'Traded'])

print(df.head())
```

[ ]:
```
# ====================================
# STEP 2: EXTRACT TICKERS AND GET HISTORICAL PRICE DATA USING ALPACA
# ====================================

# Alpaca API credentials
API_KEY = 'YOUR_API_KEY'
SECRET_KEY = 'YOUR_SECRET_KEY'

# Base URL for Alpaca's data API
BASE_URL = 'https://data.alpaca.markets/v2/stocks/bars'

def fetch_historical_bars(symbol, timeframe='1Day', start=None, end=None,␣
↪limit=10000):
    headers = {
        'APCA-API-KEY-ID': API_KEY,
        'APCA-API-SECRET-KEY': SECRET_KEY
    }

    params = {
        'symbols': symbol,
        'timeframe': timeframe,
        'start': start,
        'end': end,
        'limit': limit
    }

    response = requests.get(BASE_URL, headers=headers, params=params)
    if response.status_code == 200:
        data = response.json()
        if 'bars' in data and symbol in data['bars']:
```

```
            bars = pd.DataFrame(data['bars'][symbol])
            # Convert timestamp t to datetime
            bars['t'] = pd.to_datetime(bars['t'], utc=True)
            # Set datetime as the index
            bars.set_index('t', inplace=True)
            return bars
        else:
            print(f"No data found for symbol {symbol}")
            return None
    else:
        print(f"Error {response.status_code}: {response.text}")
        return None

# Get unique tickers
tickers = df['Ticker'].dropna().unique()
tickers = [t.strip().upper() for t in tickers if isinstance(t, str) and t.
 ↪strip() != '']

start_dt = df['Traded'].min() - timedelta(days=365)
end_dt = pd.Timestamp.today()
start_str = start_dt.strftime('%Y-%m-%dT00:00:00Z')
end_str = end_dt.strftime('%Y-%m-%dT00:00:00Z')

all_data = []

# Download each ticker's data individually
for ticker in tickers:
    try:
        print(f"Downloading data for {ticker}...")
        bars = fetch_historical_bars(ticker, timeframe='1Day', start=start_str,␣
 ↪end=end_str)
        if bars is not None and not bars.empty:
            # Add a column for the ticker symbol
            bars['Ticker'] = ticker
            all_data.append(bars)
        else:
            print(f"No data found for {ticker}")
    except Exception as e:
        print(f"Error fetching data for {ticker}: {e}")

    time.sleep(1)

# Combine all individual DataFrames into one
if all_data:
    full_price_data = pd.concat(all_data, axis=0)
    # Sort by index (datetime), then forward fill
    full_price_data = full_price_data.sort_index().ffill()
```

```
    print("Historical price data download complete.")
    print(full_price_data.head())
else:
    full_price_data = pd.DataFrame()
    print("No data collected.")
```

[ ]:
```
# Save
full_price_data.to_pickle('price_data.pkl')

# Save to CSV
full_price_data.to_csv('historical_price_data.csv')
```

[ ]:
```
# ====================================
# STEP 3: JOIN CONGRESSIONAL TRADE DATA WITH HISTORICAL PRICE DATA
# ====================================

# 1. Pivot price data to wide format so that rows = dates, columns = tickers,␣
↪and values = close prices (c)
pivoted_close = full_price_data.reset_index().pivot(index='t',␣
↪columns='Ticker', values='c')

# Forward fill missing values
pivoted_close = pivoted_close.ffill()

# save the pivoted data
pivoted_close.to_csv('pivoted_close.csv')

# 2. Ensure df's Ticker is upper case and stripped of whitespace
df['Ticker'] = df['Ticker'].str.upper().str.strip()
df['Traded'] = df['Traded'].dt.tz_localize('UTC')

# 3. For each trade in df, find the last available price on or before the trade␣
↪date
merged_list = []
for idx, row in df.iterrows():
    ticker = row['Ticker']
    trade_date = row['Traded']
    # Check if ticker is in pivoted columns
    if ticker in pivoted_close.columns:
        # Find dates in price data up to trade_date
        available_dates = pivoted_close.index[pivoted_close.index <= trade_date]
        if len(available_dates) > 0:
            # The closest date is the last element
            closest_date = available_dates[-1]
            trade_price = pivoted_close.loc[closest_date, ticker]
        else:
            trade_price = np.nan
```

```
    else:
        trade_price = np.nan
    merged_list.append(trade_price)

df['Trade_Price'] = merged_list

# print the merged data
print(df.head())
```

[ ]:
```
# save the data
df.to_csv('merged_data.csv', index=False)
```

[ ]:
```
# ====================================
# STEP 4: CALCULATE SENTIMENT METRICS
# ====================================

# Ensure 'df' has 'Traded' as datetime
df['Traded'] = pd.to_datetime(df['Traded'])

# Create a Buy/Sell Indicator
# Purchase = +1, Sale = -1. If there is any other type, treat as 0.
df['Buy_Sell_Indicator'] = df['Transaction'].apply(lambda x: 1 if x.lower() ==␣
↪'purchase' else (-1 if x.lower() == 'sale' else 0))

# Weighted Sentiment: multiply by Trade_Size_Mid to emphasize trade size
df['Weighted_Sentiment'] = df['Buy_Sell_Indicator'] * df['Trade_Size_Mid']

# Party-based sentiment:
df['Party_Buy_Sell_D'] = df.apply(lambda row: row['Buy_Sell_Indicator'] if␣
↪row['Party'] == 'D' else 0, axis=1)
df['Party_Buy_Sell_R'] = df.apply(lambda row: row['Buy_Sell_Indicator'] if␣
↪row['Party'] == 'R' else 0, axis=1)

# Aggregate at the daily level per Ticker
daily_sentiment = df.groupby(['Traded', 'Ticker']).agg(
    net_sentiment=('Buy_Sell_Indicator', 'sum'),          # Net number of buys␣
↪minus sells
    weighted_sentiment=('Weighted_Sentiment', 'sum'),     # Sum of weighted␣
↪sentiments
    net_sentiment_D=('Party_Buy_Sell_D', 'sum'),          # Net sentiment from␣
↪Democrats
    net_sentiment_R=('Party_Buy_Sell_R', 'sum'),          # Net sentiment from␣
↪Republicans
    total_trades=('Buy_Sell_Indicator', 'count'),         # How many trades␣
↪were made that day for that ticker
```

```
    avg_trade_size=('Trade_Size_Mid', 'mean'),           # Average trade size␣
  ↪that day
).reset_index()

print(daily_sentiment.head())
```

[ ]:
```
# save the data
daily_sentiment.to_csv('daily_sentiment.csv', index=False)
```

0.2 Feature Engineering and Model Training

[ ]:
```
# Experiment 1 (Technical Indicators Dominant)
# ====================================
# STEP 1: PREPARE DATA
# ====================================

# Future returns: next 5 days horizon
future_horizon = 5
future_returns = pivoted_close.shift(-future_horizon) / pivoted_close - 1.0

# Melt future_returns to long format
future_returns_long = future_returns.stack().reset_index()
future_returns_long.columns = ['Traded', 'Ticker', 'Future_Return']

# Ensure ticker and date formatting matches sentiment data
sentiment_df = daily_sentiment.reset_index() if not all(col in daily_sentiment.
  ↪columns for col in ['Traded', 'Ticker']) else daily_sentiment
sentiment_df['Ticker'] = sentiment_df['Ticker'].str.upper().str.strip()
future_returns_long['Ticker'] = future_returns_long['Ticker'].str.upper().str.
  ↪strip()

# Normalize/truncate dates
sentiment_df['Traded'] = sentiment_df['Traded'].dt.normalize()
future_returns_long['Traded'] = future_returns_long['Traded'].dt.normalize()

# Merge sentiment and future returns
merged = pd.merge(sentiment_df, future_returns_long, on=['Traded', 'Ticker'],␣
  ↪how='inner').dropna(subset=['Future_Return'])

# ====================================
# STEP 2: ADD TECHNICAL INDICATORS
# ====================================

# Copy pivoted_close for feature generation
price_features = pivoted_close.copy()
daily_ret = price_features.pct_change()
```

```python
# Calculate technical indicators
mom_5d = daily_ret.rolling(window=5).mean()  # 5-day momentum
mom_20d = daily_ret.rolling(window=20).mean()  # 20-day momentum
vol_20d = daily_ret.rolling(window=20).std()  # 20-day volatility
ma_20d = price_features.rolling(window=20).mean()  # 20-day moving average

# RSI (Relative Strength Index)
def compute_rsi(series, period=14):
    delta = series.diff()
    gain = np.maximum(delta, 0)
    loss = -np.minimum(delta, 0)
    avg_gain = gain.rolling(window=period).mean()
    avg_loss = loss.rolling(window=period).mean()
    rs = avg_gain / avg_loss
    return 100 - (100 / (1 + rs))

rsi_14 = price_features.apply(compute_rsi, period=14)

# MACD and Signal Line
ema_12 = price_features.ewm(span=12, adjust=False).mean()
ema_26 = price_features.ewm(span=26, adjust=False).mean()
macd = ema_12 - ema_26
signal_line = macd.ewm(span=9, adjust=False).mean()

# Bollinger Bands
bollinger_upper = price_features.rolling(window=20).mean() + 2 * price_features.
 ↪rolling(window=20).std()
bollinger_lower = price_features.rolling(window=20).mean() - 2 * price_features.
 ↪rolling(window=20).std()

# Average True Range (ATR)
def compute_atr(df, window=14):
    high_low = df.diff().abs()
    atr = high_low.rolling(window).mean()
    return atr

atr_14 = compute_atr(price_features)

# Stack features and merge into `merged`
def stack_feature(feat_df, col_name):
    df_long = feat_df.stack().reset_index()
    df_long.columns = ['Traded', 'Ticker', col_name]
    df_long['Traded'] = df_long['Traded'].dt.normalize()
    return df_long

features_to_add = [
```

```
    (mom_5d, 'mom_5d'), (mom_20d, 'mom_20d'), (vol_20d, 'vol_20d'), (ma_20d,␣
↪'ma_20d'),
    (rsi_14, 'rsi_14'), (macd, 'macd'), (signal_line, 'macd_signal'),
    (bollinger_upper, 'bollinger_upper'), (bollinger_lower, 'bollinger_lower'),
    (atr_14, 'atr_14')
]

for feature, name in features_to_add:
    stacked_feature = stack_feature(feature, name)
    merged = pd.merge(merged, stacked_feature, on=['Traded', 'Ticker'],␣
↪how='left')

merged = merged.dropna()

# ====================================
# STEP 3: SELECT FEATURES AND TARGET
# ====================================
feature_cols = [
    'net_sentiment', 'weighted_sentiment', 'net_sentiment_D', 'net_sentiment_R',
    'total_trades', 'avg_trade_size', 'mom_5d', 'mom_20d', 'vol_20d', 'ma_20d',
    'rsi_14', 'macd', 'macd_signal', 'bollinger_upper', 'bollinger_lower',␣
↪'atr_14'
]
target_col = 'Future_Return'

X = merged[feature_cols]
y = merged[target_col]

# ====================================
# STEP 4: TIME SERIES SPLIT AND SCALING
# ====================================
tscv = TimeSeriesSplit(n_splits=5)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# ====================================
# STEP 5: MODEL TRAINING
# ====================================
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 15, None],
    'min_samples_leaf': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'max_features': ['sqrt', 'log2', None],
    'bootstrap': [True, False]
}
model = RandomForestRegressor(random_state=42)
```

```
grid_search = GridSearchCV(model, param_grid, scoring='r2', cv=tscv, n_jobs=-1,␣
↪verbose=1, error_score='raise')
grid_search.fit(X_scaled, y)

print("Best Params:", grid_search.best_params_)
print("Best CV R^2 Score:", grid_search.best_score_)

# ====================================
# STEP 6: FINAL MODEL EVALUATION
# ====================================
cutoff_date = pd.to_datetime("2022-12-31")
merged['Traded'] = merged['Traded'].dt.tz_localize(None)
train_mask = merged['Traded'] < cutoff_date
test_mask = merged['Traded'] >= cutoff_date

X_train, y_train = X[train_mask], y[train_mask]
X_test, y_test = X[test_mask], y[test_mask]

# Scale the train-test data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

final_model = RandomForestRegressor(**grid_search.best_params_, random_state=42)
final_model.fit(X_train_scaled, y_train)
y_pred = final_model.predict(X_test_scaled)

print("Test R^2 Score with improved setup:", r2_score(y_test, y_pred))
print("Feature Importances:", dict(zip(feature_cols, final_model.
↪feature_importances_)))
```

[ ]:
```
# Experiment 2 (Congressional Sentiment Dominant)
# ====================================
# STEP 1: PREPARE DATA
# ====================================
future_horizon = 5
future_returns = pivoted_close.shift(-future_horizon) / pivoted_close - 1.0

# Melt future_returns to long format
future_returns_long = future_returns.stack().reset_index()
future_returns_long.columns = ['Traded', 'Ticker', 'Future_Return']

# Ensure ticker and date formatting matches sentiment data
sentiment_df = daily_sentiment.reset_index() if not all(col in daily_sentiment.
↪columns for col in ['Traded','Ticker']) else daily_sentiment
sentiment_df['Ticker'] = sentiment_df['Ticker'].str.upper().str.strip()
future_returns_long['Ticker'] = future_returns_long['Ticker'].str.upper().str.
↪strip()
```

```
sentiment_df['Traded'] = sentiment_df['Traded'].dt.normalize()
future_returns_long['Traded'] = future_returns_long['Traded'].dt.normalize()

merged = pd.merge(sentiment_df, future_returns_long, on=['Traded', 'Ticker'],␣
 ↪how='inner').dropna(subset=['Future_Return'])

# ====================================
# STEP 2: ADD TECHNICAL INDICATORS (REDUCED)
# ====================================
price_features = pivoted_close.copy()
daily_ret = price_features.pct_change()

# Keep only minimal technical indicators
mom_5d = daily_ret.rolling(window=5).mean()     # 5-day momentum
mom_20d = daily_ret.rolling(window=20).mean()   # 20-day momentum
vol_20d = daily_ret.rolling(window=20).std()    # 20-day volatility
ma_20d = price_features.rolling(window=20).mean() # 20-day MA

def stack_feature(feat_df, col_name):
    df_long = feat_df.stack().reset_index()
    df_long.columns = ['Traded', 'Ticker', col_name]
    df_long['Traded'] = df_long['Traded'].dt.normalize()
    return df_long

technical_features = [
    (mom_5d, 'mom_5d'),
    (mom_20d, 'mom_20d'),
    (vol_20d, 'vol_20d'),
    (ma_20d, 'ma_20d')
]

for feature, name in technical_features:
    stacked_feature = stack_feature(feature, name)
    merged = pd.merge(merged, stacked_feature, on=['Traded','Ticker'],␣
 ↪how='left')

merged = merged.dropna(subset=['mom_5d','mom_20d','vol_20d','ma_20d'])

# ====================================
# STEP 3: ENHANCE SENTIMENT FEATURES
# ====================================
merged = merged.sort_values(['Ticker', 'Traded'])

# Add lagging (3-day shift) for net_sentiment and weighted_sentiment
merged['net_sentiment_lag3'] = merged.groupby('Ticker')['net_sentiment'].
 ↪shift(3)
```

```
merged['weighted_sentiment_lag3'] = merged.
 ↪groupby('Ticker')['weighted_sentiment'].shift(3)

# 30-day rolling means for sentiment
merged['net_sentiment_rolling30'] = merged.groupby('Ticker')['net_sentiment'].
 ↪transform(lambda x: x.rolling(30).mean())
merged['weighted_sentiment_rolling30'] = merged.
 ↪groupby('Ticker')['weighted_sentiment'].transform(lambda x: x.rolling(30).
 ↪mean())
merged['net_sentiment_D_rolling30'] = merged.
 ↪groupby('Ticker')['net_sentiment_D'].transform(lambda x: x.rolling(30).
 ↪mean())
merged['net_sentiment_R_rolling30'] = merged.
 ↪groupby('Ticker')['net_sentiment_R'].transform(lambda x: x.rolling(30).
 ↪mean())

# 30-day rolling sum for total_trades
merged['total_trades_rolling30'] = merged.groupby('Ticker')['total_trades'].
 ↪transform(lambda x: x.rolling(30).sum())

# Drop rows that may now have NaN due to rolling/lagging
merged = merged.dropna(subset=[
    'net_sentiment_lag3', 'weighted_sentiment_lag3',
    'net_sentiment_rolling30', 'weighted_sentiment_rolling30',
    'total_trades_rolling30', 'net_sentiment_D_rolling30',␣
 ↪'net_sentiment_R_rolling30'
])

# ====================================
# STEP 4: SELECT FEATURES AND TARGET
# ====================================
# Focus more on sentiment features now and keep minimal technical
feature_cols = [
    'net_sentiment', 'weighted_sentiment', 'net_sentiment_D', 'net_sentiment_R',
    'total_trades', 'avg_trade_size',
    'mom_5d', 'mom_20d', 'vol_20d', 'ma_20d',
    'net_sentiment_lag3', 'weighted_sentiment_lag3',
    'net_sentiment_rolling30', 'weighted_sentiment_rolling30',
    'total_trades_rolling30', 'net_sentiment_D_rolling30',␣
 ↪'net_sentiment_R_rolling30'
]
target_col = 'Future_Return'

X = merged[feature_cols]
y = merged[target_col]
```

```python
# ====================================
# STEP 5: TIME SERIES SPLIT AND SCALING
# ====================================
tscv = TimeSeriesSplit(n_splits=5)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# ====================================
# STEP 6: MODEL TRAINING
# ====================================
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [5, 10],
    'min_samples_leaf': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'max_features': ['sqrt', None],
    'bootstrap': [True, False]
}
model = RandomForestRegressor(random_state=42)
grid_search = GridSearchCV(model, param_grid, scoring='r2', cv=tscv, n_jobs=-1,
 ↪verbose=1, error_score='raise')
grid_search.fit(X_scaled, y)

print("Best Params:", grid_search.best_params_)
print("Best CV R^2 Score:", grid_search.best_score_)

# ====================================
# STEP 7: FINAL MODEL EVALUATION
# ====================================
cutoff_date = pd.to_datetime("2022-12-31")
merged['Traded'] = merged['Traded'].dt.tz_localize(None)
train_mask = merged['Traded'] < cutoff_date
test_mask = merged['Traded'] >= cutoff_date

X_train, y_train = X[train_mask], y[train_mask]
X_test, y_test = X[test_mask], y[test_mask]

X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

final_model = RandomForestRegressor(**grid_search.best_params_, random_state=42)
final_model.fit(X_train_scaled, y_train)
y_pred = final_model.predict(X_test_scaled)

print("Test R^2 Score with more sentiment:", r2_score(y_test, y_pred))
print("Feature Importances:", dict(zip(feature_cols, final_model.
 ↪feature_importances_)))
```

0.3 Portfolio Construction

[ ]:
```
# =====================================
# STEP 1: GENERATE PREDICTED RETURNS
# =====================================
y_pred = final_model.predict(X_test_scaled)

predicted_returns = pd.DataFrame({
    'Ticker': merged[test_mask]['Ticker'].values,
    'Traded': merged[test_mask]['Traded'].values,
    'Predicted_Return': y_pred
})

mean_predicted_returns = predicted_returns.
 ↪groupby('Ticker')['Predicted_Return'].mean()
mean_predicted_returns = mean_predicted_returns.sort_values(ascending=False)

print("\nTop Predicted Returns (Mean per Stock):")
print(mean_predicted_returns.head(30))

# =====================================
# STEP 2: SELECT TOP-K STOCKS
# =====================================
k = 30  # Number of top stocks
top_k_tickers = mean_predicted_returns.head(k).index.tolist()
print(f"\nTop {k} stocks selected for the portfolio:\n{top_k_tickers}")

# =====================================
# STEP 3: HISTORICAL RETURNS AND COVARIANCE
# =====================================
historical_returns = pivoted_close[top_k_tickers].pct_change().dropna()
cov_matrix = historical_returns.cov()
expected_returns = mean_predicted_returns[top_k_tickers].values

# =====================================
# STEP 4: PORTFOLIO OPTIMIZATION (with min weight constraint)
# =====================================
def negative_sharpe_ratio(weights, expected_returns, cov_matrix,␣
 ↪risk_free_rate=0.0):
    portfolio_return = np.dot(weights, expected_returns)
    portfolio_volatility = np.sqrt(np.dot(weights.T, np.dot(cov_matrix,␣
 ↪weights)))
    return -(portfolio_return - risk_free_rate) / portfolio_volatility

# Set minimum weight constraint
min_w = 0.02  # each asset at least 2%
if k * min_w > 1:
```

```
    raise ValueError("Minimum weight per asset too large; not feasible.")

# Constraints:
# 1) Weights sum to 1
eq_constraints = {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}

# 2) Each weight >= min_w
ineq_constraints = []
for i in range(k):
    ineq_constraints.append({'type': 'ineq', 'fun': lambda x, i=i: x[i] -
 ↪min_w})

constraints = [eq_constraints] + ineq_constraints

bounds = tuple((0, 1) for _ in range(k))

# Initial guess: allocate min_w to each, distribute remainder equally
remaining = 1 - k * min_w
if remaining < 0:
    raise ValueError("No feasible solution: sum of min weights exceeds 1.")
initial_weights = np.full(k, min_w) + (remaining / k)

result = minimize(
    negative_sharpe_ratio,
    initial_weights,
    args=(expected_returns, cov_matrix),
    method='SLSQP',
    bounds=bounds,
    constraints=constraints
)

optimal_weights = result.x

# ====================================
# STEP 5: PORTFOLIO SUMMARY
# ====================================
portfolio_return = np.dot(optimal_weights, expected_returns)
portfolio_volatility = np.sqrt(np.dot(optimal_weights.T, np.dot(cov_matrix,
 ↪optimal_weights)))
sharpe_ratio = portfolio_return / portfolio_volatility

print("\nOptimal Portfolio Weights:")
for ticker, weight in zip(top_k_tickers, optimal_weights):
    print(f"{ticker}: {weight:.4f}")

print("\nPortfolio Performance:")
print(f"Expected Portfolio Return: {portfolio_return:.4%}")
```

```
print(f"Portfolio Volatility: {portfolio_volatility:.4%}")
print(f"Sharpe Ratio: {sharpe_ratio:.4f}")
```

[ ]:
```
# Pie Chart
plt.figure(figsize=(12, 8))
plt.pie(optimal_weights,
        labels=[f'{ticker}\n{weight:.1%}' for ticker, weight in␣
 ↪zip(top_k_tickers, optimal_weights)],
        autopct='%1.1f%%',
        pctdistance=0.85)
plt.title('Portfolio Allocation (Pie Chart)')
plt.show()

# Print Summary Statistics
print("\nPortfolio Statistics:")
print(f"Number of stocks: {len(top_k_tickers)}")
print(f"Largest allocation: {max(optimal_weights):.1%} ({top_k_tickers[np.
 ↪argmax(optimal_weights)]})")
print(f"Smallest allocation: {min(optimal_weights):.1%} ({top_k_tickers[np.
 ↪argmin(optimal_weights)]})")
print(f"Average allocation: {np.mean(optimal_weights):.1%}")
```

0.4 Result Visualization

[ ]:
```
# Define the cutoff date for analysis (end of 2024)
end_date = '2023-12-31'

# Slice historical_returns up to end_date if needed
historical_returns_2024 = historical_returns.loc[:end_date]

# Get S&P 500 data only up to 2024
sp500 = yf.download('^GSPC',
                    start=historical_returns_2024.index[0],
                    end=end_date)['Adj Close']
sp500.index = sp500.index.tz_localize(None)  # Remove timezone info

# Calculate daily portfolio returns for the truncated period
portfolio_daily_returns = pd.Series(
    np.dot(historical_returns_2024, optimal_weights),
    index=historical_returns_2024.index.tz_localize(None)
)

# Calculate cumulative returns for the truncated period
portfolio_cumulative = (1 + portfolio_daily_returns).cumprod()

sp500_returns = sp500.pct_change().dropna()
sp500_cumulative = (1 + sp500_returns).cumprod()
```

```python
# Performance comparison plot (up to end of 2024)
plt.figure(figsize=(12, 6))
plt.plot(portfolio_cumulative, label='Optimized Portfolio')
plt.plot(sp500_cumulative, label='S&P 500')
plt.title('Portfolio Performance vs S&P 500 (Up to 2024)')
plt.xlabel('Date')
plt.ylabel('Cumulative Return')
plt.legend()
plt.grid(True)
plt.show()

def calculate_metrics(returns):
    """Calculate performance metrics."""
    if isinstance(returns, np.ndarray):
        returns = pd.Series(returns)

    annual_return = np.mean(returns) * 252
    annual_volatility = np.std(returns) * np.sqrt(252)
    sharpe_ratio = annual_return / annual_volatility

    cum_returns = (1 + returns).cumprod()
    rolling_max = cum_returns.expanding().max()
    drawdowns = cum_returns/rolling_max - 1
    max_drawdown = drawdowns.min()

    return {
        'Annual Return': annual_return,
        'Max Drawdown': max_drawdown
    }

# Calculate metrics for truncated period
portfolio_metrics = calculate_metrics(portfolio_daily_returns)
sp500_metrics = calculate_metrics(sp500_returns)

# Display metrics comparison
metrics_df = pd.DataFrame({
    'Optimized Portfolio': portfolio_metrics,
    'S&P 500': sp500_metrics
})
print("\nPerformance Metrics (Up to 2024):")
print(metrics_df.round(4))
```

[ ]:
```python
# Save the trained model
joblib.dump(final_model, 'final_model.pkl')
print("Model saved successfully as 'final_model.pkl'")
```

0.5 Trying different ML techniques

1. Classification Task (Up/Down)
2. XGBoost

[ ]:
```
# 1) Convert the regression target to a classification target (up/down).
# 2) Introduce a lag in sentiment features.

# Introduce a lag in sentiment features to simulate that the market reacts␣
↪after some delay:
lag_days = 3
for col in ['net_sentiment', 'weighted_sentiment', 'net_sentiment_D',␣
↪'net_sentiment_R', 'total_trades', 'avg_trade_size']:
    merged[col] = merged.groupby('Ticker')[col].shift(lag_days)

# Drop rows with NaN after shifting
merged = merged.dropna(subset=['Future_Return'] +␣
↪['net_sentiment','weighted_sentiment','net_sentiment_D','net_sentiment_R','total_trades','a

# Convert to classification: 1 if Future_Return > 0, else 0
merged['Future_Up'] = (merged['Future_Return'] > 0).astype(int)

feature_cols = [
    'net_sentiment',
    'weighted_sentiment',
    'net_sentiment_D',
    'net_sentiment_R',
    'total_trades',
    'avg_trade_size',
    'mom_5d',
    'mom_20d',
    'vol_20d',
    'ma_20d'
]

X = merged[feature_cols]
y = merged['Future_Up']

# Time series split
tscv = TimeSeriesSplit(n_splits=3)

param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, None],
    'min_samples_leaf': [1, 5, 10]
}

clf = RandomForestClassifier(random_state=42)
```

```
grid_search = GridSearchCV(
    clf,
    param_grid,
    scoring='accuracy',  # or 'roc_auc'
    cv=tscv,
    n_jobs=-1,
    verbose=1
)
grid_search.fit(X, y)

print("Best Params:", grid_search.best_params_)
print("Best CV Accuracy:", grid_search.best_score_)

# Final evaluation on an out-of-sample period
cutoff_date = pd.to_datetime("2022-12-31")
merged['Traded'] = pd.to_datetime(merged['Traded']).dt.normalize()

train_mask = merged['Traded'] < cutoff_date
test_mask = merged['Traded'] >= cutoff_date

X_train, y_train = X[train_mask], y[train_mask]
X_test, y_test = X[test_mask], y[test_mask]

final_clf = RandomForestClassifier(**grid_search.best_params_, random_state=42)
final_clf.fit(X_train, y_train)
y_pred = final_clf.predict(X_test)
y_prob = final_clf.predict_proba(X_test)[:,1]

print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("Test AUC:", roc_auc_score(y_test, y_prob))
```

[ ]:

```
# ====================================
# STEP 1: PREPARE DATA
# ====================================

# We'll use a shorter horizon (5 days) for future returns:
future_horizon = 5
future_returns = pivoted_close.shift(-future_horizon) / pivoted_close - 1.0

# Melt future_returns to long format
future_returns_long = future_returns.stack().reset_index()
future_returns_long.columns = ['Traded', 'Ticker', 'Future_Return']

# Ensure ticker and date formatting matches sentiment
if not all(col in daily_sentiment.columns for col in ['Traded','Ticker']):
    sentiment_df = daily_sentiment.reset_index()
else:
```

```python
    sentiment_df = daily_sentiment.copy()

sentiment_df['Ticker'] = sentiment_df['Ticker'].str.upper().str.strip()
future_returns_long['Ticker'] = future_returns_long['Ticker'].str.upper().str.
 ↪strip()

# Filter sentiment to ensure overlap (adjust date as needed)
sentiment_df = sentiment_df[sentiment_df['Traded'] >= '2016-01-01']

# Normalize/truncate times
sentiment_df['Traded'] = sentiment_df['Traded'].dt.normalize()
future_returns_long['Traded'] = future_returns_long['Traded'].dt.normalize()

# Merge
merged = pd.merge(sentiment_df, future_returns_long, on=['Traded', 'Ticker'],␣
 ↪how='inner').dropna(subset=['Future_Return'])

if merged.empty:
    print("Merged dataframe is empty. Check ticker/date overlap before␣
 ↪proceeding.")
else:
    # ====================================
    # STEP 2: ADD ADDITIONAL PRICE FEATURES
    # ====================================
    price_features = pivoted_close.copy()

    # Daily returns
    daily_ret = price_features.pct_change()

    # 5-day momentum
    mom_5d = daily_ret.rolling(window=5).mean()

    # 20-day momentum
    mom_20d = daily_ret.rolling(window=20).mean()

    # 20-day volatility
    vol_20d = daily_ret.rolling(window=20).std()

    # 20-day moving average price
    ma_20d = price_features.rolling(window=20).mean()

    def stack_feature(feat_df, col_name):
        df_long = feat_df.stack().reset_index()
        df_long.columns = ['Traded', 'Ticker', col_name]
        df_long['Traded'] = df_long['Traded'].dt.normalize()
        return df_long
```

```python
mom_5d_long = stack_feature(mom_5d, 'mom_5d')
mom_20d_long = stack_feature(mom_20d, 'mom_20d')
vol_20d_long = stack_feature(vol_20d, 'vol_20d')
ma_20d_long = stack_feature(ma_20d, 'ma_20d')

# Merge all additional features into merged
merged = pd.merge(merged, mom_5d_long, on=['Traded','Ticker'], how='left')
merged = pd.merge(merged, mom_20d_long, on=['Traded','Ticker'], how='left')
merged = pd.merge(merged, vol_20d_long, on=['Traded','Ticker'], how='left')
merged = pd.merge(merged, ma_20d_long, on=['Traded','Ticker'], how='left')

# Drop rows with no price features
merged = merged.dropna(subset=['mom_5d','mom_20d','vol_20d','ma_20d'])

# ====================================
# STEP 3: SELECT FEATURES AND TARGET
# ====================================
feature_cols = [
    'net_sentiment',
    'weighted_sentiment',
    'net_sentiment_D',
    'net_sentiment_R',
    'total_trades',
    'avg_trade_size',
    'mom_5d',
    'mom_20d',
    'vol_20d',
    'ma_20d'
]
target_col = 'Future_Return'

X = merged[feature_cols]
y = merged[target_col]

# ====================================
# STEP 4: TIME SERIES SPLIT
# ====================================
tscv = TimeSeriesSplit(n_splits=3)

# ====================================
# STEP 5: HYPERPARAMETER TUNING WITH XGBOOST
# ====================================
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [3, 5, 10],
    'learning_rate': [0.01, 0.05, 0.1]
}
```

```
    model = XGBRegressor(random_state=42, tree_method='hist', n_jobs=-1)  # Use␣
↪hist for faster training if large dataset

    grid_search = GridSearchCV(
        model,
        param_grid,
        scoring='r2',
        cv=tscv,
        n_jobs=-1,
        verbose=1
    )
    grid_search.fit(X, y)

    print("Best Params:", grid_search.best_params_)
    print("Best CV R^2 Score:", grid_search.best_score_)

    # Train final model on entire dataset before cutoff, then test on a hold-out
    cutoff_date = pd.to_datetime("2022-12-31")
    # Ensure Traded is naive datetime
    merged['Traded'] = merged['Traded'].dt.tz_localize(None)

    train_mask = merged['Traded'] < cutoff_date
    test_mask = merged['Traded'] >= cutoff_date

    X_train, y_train = X[train_mask], y[train_mask]
    X_test, y_test = X[test_mask], y[test_mask]

    final_model = XGBRegressor(**grid_search.best_params_, random_state=42,␣
↪tree_method='hist', n_jobs=-1)
    final_model.fit(X_train, y_train)
    y_pred = final_model.predict(X_test)

    print("Test R^2 Score with improved setup:", r2_score(y_test, y_pred))

    # Feature importances
    importances = final_model.feature_importances_
    print("Feature Importances:", dict(zip(feature_cols, importances)))
```