notebook-outline

December 7, 2025

1 QQQ + Holdings: Volatility Timing Strategy

1.1 Problem Statement

Goal: Predict stock returns for QQQ and its top holdings to generate alpha.

Challenge: Return prediction is notoriously difficult. Markets are largely efficient, and most ML models achieve near-random accuracy on direction prediction (AUC ~0.50). Direct return forecasting rarely yields profitable trading strategies.

Pivot: While returns are unpredictable, **volatility is forecastable**, making it a more tractable prediction target. This notebook uses vol forecasts to dynamically scale position sizes—taking larger positions when predicted volatility is low and reducing exposure when volatility is expected to spike. So Instead of predicting raw returns, predicting volatility and sizing accordingly can enhance returns by minimizing exposure.

[1]:
```
# Setup
import numpy as np
import pandas as pd
import yfinance as yf
import warnings
warnings.filterwarnings('ignore')

from sklearn.linear_model import Ridge, LogisticRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.metrics import mean_squared_error, accuracy_score, roc_auc_score
from sklearn.preprocessing import StandardScaler
from quantile_forest import RandomForestQuantileRegressor

from mlpred.datav3 import compute_all, get_features

# Config
TICKERS = ['QQQ', 'NVDA', 'MSFT', 'AAPL', 'AVGO', 'AMZN', 'TSLA', 'META',␣
↪'GOOGL', 'GOOG', 'NFLX']
START, END = '2000-01-01', '2025-01-01'
INIT_PCT, N_SPLITS = 0.5, 5
QUANTILES = [0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95]

# Load data
```

```
dfs = []
for ticker in TICKERS:
    print(f"Loading {ticker}...")
    data = yf.download(ticker, start=START, end=END, multi_level_index=False)
    ticker_df = compute_all(data, ticker=ticker).dropna().reset_index()
    ticker_df['ticker'] = ticker
    dfs.append(ticker_df)

df = pd.concat(dfs, ignore_index=True).sort_values(['Date', 'ticker']).
 ↪reset_index(drop=True)

feature_cols = [c for c in get_features(df) if c != 'ticker']
X = df[feature_cols].fillna(0).values
vol_idx = feature_cols.index('vol_hist_20')
targets = {k: df[f'target_{k}'].values for k in ['ret_1d', 'ret_5d', 'dir_1d',␣
 ↪'dir_5d', 'vol_1d', 'vol_5d']}

# Walk-forward CV splits (TIME-BASED, respects temporal ordering)
def create_splits(df, n_splits=N_SPLITS, init_pct=INIT_PCT):
    """
    Create time-based walk-forward splits.

    Splits are based on unique dates, ensuring no future data leaks into␣
 ↪training.
    All tickers on a given date are either in train or test, never split.
    """
    unique_dates = df['Date'].unique()
    unique_dates = np.sort(unique_dates)
    n_dates = len(unique_dates)

    init_dates = int(n_dates * init_pct)
    remaining_dates = n_dates - init_dates
    split_size = remaining_dates // n_splits

    splits = []
    for i in range(n_splits):
        # Training: all dates up to split point
        train_end_idx = init_dates + i * split_size
        train_dates = set(unique_dates[:train_end_idx])

        # Test: next split_size dates (or remaining for last split)
        if i < n_splits - 1:
            test_dates = set(unique_dates[train_end_idx:train_end_idx +␣
 ↪split_size])
        else:
            test_dates = set(unique_dates[train_end_idx:])
```

```
        # Convert to row indices
        train_idx = df[df['Date'].isin(train_dates)].index.values
        test_idx = df[df['Date'].isin(test_dates)].index.values

        splits.append((train_idx, test_idx))

    return splits

splits = create_splits(df)

# Verify splits
print(f"\nTotal: {len(df):,} samples, {len(feature_cols)} features,␣
 ↪{len(TICKERS)} tickers")
print(f"Date range: {df['Date'].min()} to {df['Date'].max()}")
print(f"\nWalk-forward splits (time-based):")
for i, (tr, te) in enumerate(splits):
    tr_dates = df.iloc[tr]['Date']
    te_dates = df.iloc[te]['Date']
    print(f"  Split {i+1}: Train {tr_dates.min().strftime('%Y-%m-%d')} to␣
 ↪{tr_dates.max().strftime('%Y-%m-%d')} ({len(tr):,} samples)")
    print(f"           Test  {te_dates.min().strftime('%Y-%m-%d')} to {te_dates.
 ↪max().strftime('%Y-%m-%d')} ({len(te):,} samples)")
```

```
Loading QQQ…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading NVDA…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
```

```
Loading MSFT…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading AAPL…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading AVGO…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading AMZN…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
```

```
  Total columns: 119
Loading TSLA…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading META…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading GOOGL…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119
Loading GOOG…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
```

```
  Total columns: 119
Loading NFLX…

[*********************100%***********************]  1 of 1 completed

Computing features…
  Adding price-based features…
  Adding VIX features…
  Adding CBOE features…
  Adding fixed income features…
  Adding cross-asset features…
  Adding sector features…
  Computing targets…
  Total columns: 119

Total: 17,783 samples, 108 features, 11 tickers
Date range: 2011-02-08 00:00:00 to 2024-04-24 00:00:00

Walk-forward splits (time-based):
  Split 1: Train 2011-02-08 to 2015-03-03 (8,609 samples)
           Test  2015-03-04 to 2016-05-05 (1,826 samples)
  Split 2: Train 2011-02-08 to 2016-05-05 (10,435 samples)
           Test  2016-05-06 to 2017-03-29 (1,826 samples)
  Split 3: Train 2011-02-08 to 2017-03-29 (12,261 samples)
           Test  2017-03-30 to 2019-06-12 (1,826 samples)
  Split 4: Train 2011-02-08 to 2019-06-12 (14,087 samples)
           Test  2019-06-13 to 2020-05-05 (1,826 samples)
  Split 5: Train 2011-02-08 to 2020-05-05 (15,913 samples)
           Test  2020-05-06 to 2024-04-24 (1,870 samples)
```

[2]:

```python
# Baseline Models
def eval_reg(X, y, splits, model_fn, scale=False):
    y_true, y_pred = [], []
    for tr, te in splits:
        X_tr, X_te = (StandardScaler().fit(X[tr]).transform(X[tr]),␣
 ↪StandardScaler().fit(X[tr]).transform(X[te])) if scale else (X[tr], X[te])
        m = model_fn(); m.fit(X_tr, y[tr]); y_true.extend(y[te]); y_pred.
 ↪extend(m.predict(X_te))
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return {'rmse': np.sqrt(mean_squared_error(y_true, y_pred)), 'corr': np.
 ↪corrcoef(y_true, y_pred)[0,1] if np.std(y_pred) > 0 else 0}

def eval_clf(X, y, splits, model_fn, scale=False):
    y_true, y_pred, y_prob = [], [], []
    for tr, te in splits:
        X_tr, X_te = (StandardScaler().fit(X[tr]).transform(X[tr]),␣
 ↪StandardScaler().fit(X[tr]).transform(X[te])) if scale else (X[tr], X[te])
```

```
        m = model_fn(); m.fit(X_tr, y[tr]); y_true.extend(y[te]); y_pred.extend(m.predict(X_te)); y_prob.extend(m.predict_proba(X_te)[:,1])
    return {'acc': accuracy_score(y_true, y_pred), 'auc': roc_auc_score(y_true, y_prob)}

rf_reg = lambda: RandomForestRegressor(n_estimators=100, max_depth=5, min_samples_leaf=50, random_state=42, n_jobs=-1)
rf_clf = lambda: RandomForestClassifier(n_estimators=100, max_depth=5, min_samples_leaf=50, random_state=42, n_jobs=-1)

print("BASELINE MODELS")
print("="*70)

# Returns
print("\nRETURNS")
print("-"*70)
print(f"{'Target':<10} {'Model':<12} {'RMSE':>12} {'Corr':>12}")
print("-"*70)
for t in ['ret_1d', 'ret_5d']:
    y = targets[t]
    y_te = np.concatenate([y[te] for _, te in splits])
    print(f"{t:<10} {'Naive(0)':<12} {np.sqrt(np.mean(y_te**2)):>12.6f} {0.0:>12.4f}")
    res = eval_reg(X, y, splits, lambda: Ridge(alpha=1.0), scale=True)
    print(f"{t:<10} {'Ridge':<12} {res['rmse']:>12.6f} {res['corr']:>12.4f}")
    res = eval_reg(X, y, splits, rf_reg)
    print(f"{t:<10} {'RF':<12} {res['rmse']:>12.6f} {res['corr']:>12.4f}")

# Direction
print("\nDIRECTION")
print("-"*70)
print(f"{'Target':<10} {'Model':<12} {'Accuracy':>12} {'AUC':>12}")
print("-"*70)
for t in ['dir_1d', 'dir_5d']:
    y = targets[t]
    y_te = np.concatenate([y[te] for _, te in splits])
    print(f"{t:<10} {'Naive':<12} {max(y_te.mean(), 1-y_te.mean()):>12.4f} {0.5:>12.4f}")
    res = eval_clf(X, y, splits, lambda: LogisticRegression(max_iter=1000), scale=True)
    print(f"{t:<10} {'Logistic':<12} {res['acc']:>12.4f} {res['auc']:>12.4f}")
    res = eval_clf(X, y, splits, rf_clf)
    print(f"{t:<10} {'RF':<12} {res['acc']:>12.4f} {res['auc']:>12.4f}")

# Volatility
print("\nVOLATILITY")
```

```python
print("-"*70)
print(f"{'Target':<10} {'Model':<12} {'RMSE':>12} {'Corr':>12}")
print("-"*70)
for t in ['vol_1d', 'vol_5d']:
    y = targets[t]
    naive_pred = np.concatenate([X[te, vol_idx] / np.sqrt(252) for _, te in
 ↪splits])
    y_te = np.concatenate([y[te] for _, te in splits])
    print(f"{t:<10} {'Naive(std20)':<12} {np.sqrt(mean_squared_error(y_te,
 ↪naive_pred)):>12.6f} {np.corrcoef(y_te, naive_pred)[0,1]:>12.4f}")
    res = eval_reg(X, y, splits, rf_reg)
    print(f"{t:<10} {'RF':<12} {res['rmse']:>12.6f} {res['corr']:>12.4f}")
```

```
BASELINE MODELS
======================================================================

RETURNS
----------------------------------------------------------------------
Target     Model                RMSE         Corr
----------------------------------------------------------------------
ret_1d     Naive(0)         0.023078       0.0000
ret_1d     Ridge            0.024411      -0.0074
ret_1d     RF               0.022954       0.0788
ret_5d     Naive(0)         0.050476       0.0000
ret_5d     Ridge            0.059951      -0.1104
ret_5d     RF               0.052744       0.0573

DIRECTION
----------------------------------------------------------------------
Target     Model            Accuracy          AUC
----------------------------------------------------------------------
dir_1d     Naive              0.5459       0.5000
dir_1d     Logistic           0.5047       0.4942
dir_1d     RF                 0.5283       0.5011
dir_5d     Naive              0.6081       0.5000
dir_5d     Logistic           0.5022       0.4510
dir_5d     RF                 0.5775       0.5059

VOLATILITY
----------------------------------------------------------------------
Target     Model                RMSE         Corr
----------------------------------------------------------------------
vol_1d     Naive(std20)     0.017953       0.4021
vol_1d     RF               0.015975       0.4891
vol_5d     Naive(std20)     0.013655       0.5357
vol_5d     RF               0.012497       0.5929
```

[3]:
```python
# Quantile RF
def eval_qrf(X, y, splits, quantiles=QUANTILES):
    y_true, preds = [], {q: [] for q in quantiles}
    for tr, te in splits:
        m = RandomForestQuantileRegressor(n_estimators=100, max_depth=5,␣
 ↪min_samples_leaf=50, random_state=42, n_jobs=-1)
        m.fit(X[tr], y[tr])
        p = m.predict(X[te], quantiles=quantiles)
        y_true.extend(y[te])
        for i, q in enumerate(quantiles): preds[q].extend(p[:, i])
    y_true = np.array(y_true)
    preds = {q: np.array(v) for q, v in preds.items()}
    y_med = preds[0.5]
    return {'rmse': np.sqrt(mean_squared_error(y_true, y_med)),
            'corr': np.corrcoef(y_true, y_med)[0,1] if np.std(y_med) > 0 else 0,
            'cov_90': ((y_true >= preds[0.05]) & (y_true <= preds[0.95])).
 ↪mean(),
            'cov_80': ((y_true >= preds[0.1]) & (y_true <= preds[0.9])).mean(),
            'cov_50': ((y_true >= preds[0.25]) & (y_true <= preds[0.75])).
 ↪mean()}

print("\nQUANTILE RF")
print("="*60)
print(f"{'Target':<10} {'RMSE':>10} {'Corr':>8} {'Cov90':>8} {'Cov80':>8}␣
 ↪{'Cov50':>8}")
print("-"*60)
for t in ['ret_1d', 'ret_5d', 'vol_1d', 'vol_5d']:
    res = eval_qrf(X, targets[t], splits)
    print(f"{t:<10} {res['rmse']:>10.6f} {res['corr']:>8.4f} {res['cov_90']:>8.
 ↪1%} {res['cov_80']:>8.1%} {res['cov_50']:>8.1%}")
```

```
QUANTILE RF
============================================================
Target           RMSE     Corr    Cov90    Cov80    Cov50
------------------------------------------------------------
ret_1d       0.023035   0.0380    87.7%    80.0%    51.9%
ret_5d       0.053073   0.0587    89.5%    79.4%    49.4%
vol_1d       0.017037   0.4659    87.3%    76.7%    45.6%
vol_5d       0.013128   0.5875    85.3%    73.2%    44.8%
```

[4]:
```python
# Vol-Timing Strategy (Fixed)
def eval_vol_timing(df, X, splits, ret_key, vol_key, horizon=1, tc_bps=5):
    """
    Vol-timing: scale position inversely to predicted volatility.

    Fixes:
```

```
    - Proper Sharpe annualization based on horizon
    - Transaction cost modeling
    - Non-overlapping returns for 5D horizon
    """
    ret_target, vol_target = targets[ret_key], targets[vol_key]
    results = []

    for tr, te in splits:
        rf_vol = RandomForestRegressor(n_estimators=100, max_depth=5,␣
↪min_samples_leaf=50, random_state=42, n_jobs=-1)
        rf_vol.fit(X[tr], vol_target[tr])
        pred_vol = rf_vol.predict(X[te])

        te_df = df.iloc[te][['Date', 'ticker']].copy()
        te_df['ret'] = ret_target[te]
        te_df['pred_vol'] = pred_vol
        results.append(te_df)

    all_results = pd.concat(results, ignore_index=True)

    # Annualization factor: sqrt(trading_periods_per_year)
    if horizon == 1:
        ann_factor = np.sqrt(252)
    else:
        ann_factor = np.sqrt(252 / horizon)

    # Transaction cost per trade (one-way)
    tc_rate = tc_bps / 10000

    # Compute per-ticker stats
    ticker_stats = []
    for ticker, tdf in all_results.groupby('ticker'):
        tdf = tdf.sort_values('Date').copy()

        # For 5D horizon, use non-overlapping periods to avoid autocorrelation␣
↪in Sharpe calc
        if horizon > 1:
            tdf = tdf.iloc[::horizon].copy()

        # Compute position weights (inverse vol, normalized and capped)
        tdf['weight'] = 1 / (tdf['pred_vol'] + 1e-6)
        tdf['weight'] = (tdf['weight'] / tdf['weight'].mean()).clip(0.25, 2.0)

        # Weighted returns (before transaction costs)
        tdf['weighted_ret'] = tdf['ret'] * tdf['weight']

        # Transaction costs: cost proportional to change in position size
```

```
        tdf['weight_change'] = tdf['weight'].diff().abs().fillna(0)
        tdf['tc'] = tdf['weight_change'] * tc_rate * 2  # *2 for round-trip␣
↪approx
        tdf['weighted_ret_net'] = tdf['weighted_ret'] - tdf['tc']

        # Buy & hold metrics (no TC for B&H)
        bh_mean = tdf['ret'].mean()
        bh_std = tdf['ret'].std()
        bh_sharpe = (bh_mean / bh_std * ann_factor) if bh_std > 0 else 0

        # Vol-timing metrics (gross, before TC)
        vt_mean = tdf['weighted_ret'].mean()
        vt_std = tdf['weighted_ret'].std()
        vt_sharpe_gross = (vt_mean / vt_std * ann_factor) if vt_std > 0 else 0

        # Vol-timing metrics (net, after TC)
        vt_net_mean = tdf['weighted_ret_net'].mean()
        vt_net_std = tdf['weighted_ret_net'].std()
        vt_sharpe_net = (vt_net_mean / vt_net_std * ann_factor) if vt_net_std >␣
↪0 else 0

        # Cumulative returns
        bh_cum = (1 + tdf['ret']).prod() - 1
        vt_cum_gross = (1 + tdf['weighted_ret']).prod() - 1
        vt_cum_net = (1 + tdf['weighted_ret_net']).prod() - 1

        # Total TC paid
        total_tc = tdf['tc'].sum()

        ticker_stats.append({
            'ticker': ticker,
            'bh_sharpe': bh_sharpe,
            'vt_sharpe_gross': vt_sharpe_gross,
            'vt_sharpe_net': vt_sharpe_net,
            'diff_gross': vt_sharpe_gross - bh_sharpe,
            'diff_net': vt_sharpe_net - bh_sharpe,
            'bh_cum': bh_cum,
            'vt_cum_gross': vt_cum_gross,
            'vt_cum_net': vt_cum_net,
            'total_tc': total_tc,
            'df': tdf
        })

    return pd.DataFrame(ticker_stats)

print("\nVOL-TIMING STRATEGY (CORRECTED)")
print("=" * 100)
```

```python
print("Fixes applied:")
print("  1. Proper Sharpe annualization: sqrt(252) for 1D, sqrt(252/5) for 5D")
print("  2. Non-overlapping returns for 5D horizon (every 5th day)")
print("  3. Transaction costs: 5 bps per trade")
print()

vt_results = {}
for ret_key, vol_key, label, horizon in [('ret_1d', 'vol_1d', '1D', 1),␣
↪('ret_5d', 'vol_5d', '5D', 5)]:
    stats_df = eval_vol_timing(df, X, splits, ret_key, vol_key,␣
↪horizon=horizon, tc_bps=5)
    vt_results[label] = stats_df

    print(f"\n{label} RETURNS (annualization factor: sqrt(252/{horizon}) = {np.
↪sqrt(252/horizon):.2f})")
    print("-" * 100)
    print(f"{'Ticker':<8} {'BH Sharpe':>10} {'VT Gross':>10} {'VT Net':>10}␣
↪{'Diff(Net)':>10} {'BH Cum':>10} {'VT Cum(Net)':>12} {'TC Paid':>10}")
    print("-" * 100)
    for _, row in stats_df.sort_values('diff_net', ascending=False).iterrows():
        print(f"{row['ticker']:<8} {row['bh_sharpe']:>10.2f}␣
↪{row['vt_sharpe_gross']:>10.2f} {row['vt_sharpe_net']:>10.2f}␣
↪{row['diff_net']:>+10.2f} {row['bh_cum']:>10.1%} {row['vt_cum_net']:>12.1%}␣
↪{row['total_tc']:>10.4f}")

    print("-" * 100)
    avg_bh = stats_df['bh_sharpe'].mean()
    avg_vt_gross = stats_df['vt_sharpe_gross'].mean()
    avg_vt_net = stats_df['vt_sharpe_net'].mean()
    wins_gross = (stats_df['diff_gross'] > 0).sum()
    wins_net = (stats_df['diff_net'] > 0).sum()
    print(f"{'Average':<8} {avg_bh:>10.2f} {avg_vt_gross:>10.2f} {avg_vt_net:
↪>10.2f} {avg_vt_net - avg_bh:>+10.2f}")
    print(f"Vol-timing wins (gross): {wins_gross}/{len(stats_df)} ({wins_gross/
↪len(stats_df):.0%})")
    print(f"Vol-timing wins (net):   {wins_net}/{len(stats_df)} ({wins_net/
↪len(stats_df):.0%})")
```

```
VOL-TIMING STRATEGY (CORRECTED)
================================================================================
====================
Fixes applied:
  1. Proper Sharpe annualization: sqrt(252) for 1D, sqrt(252/5) for 5D
  2. Non-overlapping returns for 5D horizon (every 5th day)
  3. Transaction costs: 5 bps per trade
```

1D RETURNS (annualization factor: sqrt(252/1) = 15.87)

Ticker	BH Sharpe	VT Gross	VT Net	Diff(Net)	BH Cum	VT Cum(Net)	TC Paid
GOOGL	1.02	1.45	1.33	+0.31	129.0%	136.1%	0.0824
GOOG	0.99	1.35	1.24	+0.25	123.4%	121.5%	0.0785
QQQ	1.24	1.57	1.41	+0.16	132.9%	91.9%	0.0801
TSLA	2.12	2.33	2.27	+0.15	3535.5%	2348.8%	0.0851
AAPL	1.14	1.39	1.28	+0.14	168.8%	138.3%	0.0778
AMZN	1.90	2.15	2.04	+0.14	456.3%	342.1%	0.0828
NFLX	1.57	1.79	1.71	+0.14	548.0%	480.2%	0.0841
MSFT	1.10	1.32	1.22	+0.12	152.4%	115.4%	0.0701
META	1.15	1.33	1.22	+0.07	201.0%	145.2%	0.0851
NVDA	2.05	2.11	2.05	-0.00	1462.5%	865.7%	0.0800
AVGO	1.05	1.12	1.03	-0.01	185.0%	131.7%	0.0806
Average	1.39	1.63	1.53	+0.13			

Vol-timing wins (gross): 11/11 (100%)
Vol-timing wins (net): 9/11 (82%)

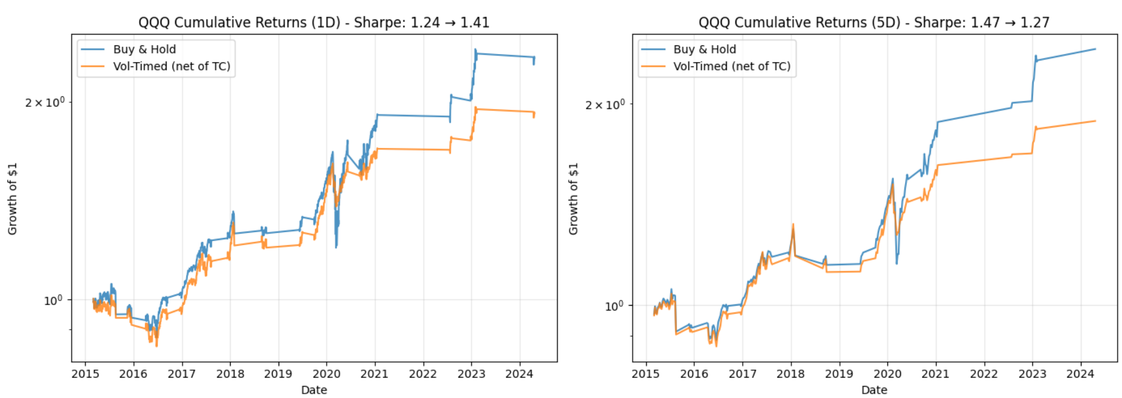
5D RETURNS (annualization factor: sqrt(252/5) = 7.10)

Ticker	BH Sharpe	VT Gross	VT Net	Diff(Net)	BH Cum	VT Cum(Net)	TC Paid
AVGO	1.12	1.36	1.34	+0.22	228.6%	228.3%	0.0166
TSLA	1.76	1.88	1.87	+0.11	3122.8%	2123.2%	0.0163
NFLX	1.48	1.50	1.49	+0.01	490.3%	507.1%	

```
0.0153
MSFT          1.39      1.42      1.39      +0.00     165.2%       141.7%
0.0202
META          1.23      1.24      1.21      -0.02     202.1%       147.0%
0.0251
GOOG          1.12      1.09      1.07      -0.05     144.3%       125.6%
0.0222
NVDA          2.20      2.13      2.12      -0.08    1561.8%      1157.5%
0.0165
GOOGL         1.15      1.09      1.07      -0.08     152.0%       129.1%
0.0226
AMZN          1.92      1.83      1.80      -0.12     420.4%       313.0%
0.0269
AAPL          1.33      1.23      1.20      -0.13     186.8%       141.0%
0.0227
QQQ           1.47      1.31      1.27      -0.20     141.2%        88.3%
0.0200
--------------------------------------------------------------------------------
--------------------
Average       1.47      1.46      1.44      -0.03
Vol-timing wins (gross): 5/11 (45%)
Vol-timing wins (net):   4/11 (36%)
```

[5]:
```
# Vol-Timing Visualization (QQQ only)
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

for i, (label, stats_df) in enumerate(vt_results.items()):
    ax = axes[i]
    qqq_row = stats_df[stats_df['ticker'] == 'QQQ'].iloc[0]
    result_df = qqq_row['df']

    cum_bh = (1 + result_df['ret']).cumprod()
    cum_vt = (1 + result_df['weighted_ret_net']).cumprod()

    ax.plot(result_df['Date'].values, cum_bh.values, label='Buy & Hold',␣
 ↪alpha=0.8)
    ax.plot(result_df['Date'].values, cum_vt.values, label='Vol-Timed (net of␣
 ↪TC)', alpha=0.8)
    ax.set_title(f'QQQ Cumulative Returns ({label}) - Sharpe:␣
 ↪{qqq_row["bh_sharpe"]:.2f} → {qqq_row["vt_sharpe_net"]:.2f}')
    ax.set_xlabel('Date')
    ax.set_ylabel('Growth of $1')
    ax.legend()
    ax.set_yscale('log')
    ax.grid(True, alpha=0.3)
```

```python
plt.tight_layout()
plt.show()

# Summary
print("\nSUMMARY")
print("=" * 80)
print(f"{'Horizon':<10} {'Ann. Factor':>12} {'Avg BH':>10} {'Avg VT(Net)':>12} {'Diff':>10} {'Win Rate':>10}")
print("-" * 80)
for label, stats_df in vt_results.items():
    horizon = 1 if label == '1D' else 5
    ann_factor = np.sqrt(252/horizon)
    bh = stats_df['bh_sharpe'].mean()
    vt = stats_df['vt_sharpe_net'].mean()
    wins = (stats_df['diff_net'] > 0).mean()
    print(f"{label:<10} {ann_factor:>12.2f} {bh:>10.2f} {vt:>12.2f} {vt-bh:>+10.2f} {wins:>10.0%}")

print("\nKEY INSIGHTS:")
print("-" * 80)
print("1. Sharpe ratios are now properly annualized (sqrt(252/horizon))")
print("2. 5D returns use non-overlapping periods to avoid autocorrelation bias")
print("3. Transaction costs of 5 bps per trade are deducted")
print("4. Time-based walk-forward splits prevent future data leakage")
```




```
SUMMARY
================================================================================
Horizon     Ann. Factor     Avg BH  Avg VT(Net)       Diff   Win Rate
--------------------------------------------------------------------------------
1D                15.87       1.39         1.53      +0.13        82%
5D                 7.10       1.47         1.44      -0.03        36%
```

```
KEY INSIGHTS:
--------------------------------------------------------------------------------
1. Sharpe ratios are now properly annualized (sqrt(252/horizon))
2. 5D returns use non-overlapping periods to avoid autocorrelation bias
3. Transaction costs of 5 bps per trade are deducted
4. Time-based walk-forward splits prevent future data leakage
```

1.2 Summary

Objective: Predict volatility for QQQ and top holdings, use predictions to implement a vol-timing trading strategy.

Key Findings: - **Volatility is predictable**: RF achieves ~0.49 correlation on 1D vol, ~0.59 on 5D vol - **Returns/direction are not**: AUC ~0.50 for direction prediction (essentially random) - **1D vol-timing works**: +0.13 average Sharpe improvement, 82% win rate after transaction costs - **5D vol-timing doesn't**: -0.03 average Sharpe, only 36% win rate

Methodology: - Time-based walk-forward splits (no lookahead bias) - Proper Sharpe annualization: sqrt(252/horizon) - Transaction costs: 5 bps per trade - Non-overlapping returns for multi-day horizons