Finance Applications of ML - Advanced Handout

Machine Learning for Smarter Innovation

1 Finance Applications of ML - Advanced Handout

Target Audience: Quantitative analysts and ML engineers **Duration**: 90 minutes reading **Level**: Advanced (mathematical foundations, production systems)

1.1 Portfolio Theory Foundations

1.1.1 Mean-Variance Optimization

Markowitz Problem:

$$\min_w \frac{1}{2} w^T \Sigma w$$

subject to: $\mu^T w \geq r_{target}$, $\mathbf{1}^T w = 1$, $w \geq 0$

Lagrangian:

$$\mathcal{L} = \frac{1}{2} w^T \Sigma w - \lambda_1(\mu^T w - r_{target}) - \lambda_2(\mathbf{1}^T w - 1)$$

Analytical Solution (unconstrained):

$$w^* = \frac{\Sigma^{-1}\mu}{\mathbf{1}^T \Sigma^{-1} \mu}$$

1.1.2 Black-Litterman Model

Combines market equilibrium with investor views:

$$\mu_{BL} = [(\tau\Sigma)^{-1} + P^T\Omega^{-1}P]^{-1}[(\tau\Sigma)^{-1}\Pi + P^T\Omega^{-1}Q]$$

Where: - Π = equilibrium excess returns - P = view pick matrix - Q = view vector - Ω = view uncertainty - τ = scalar (typically 0.05)

```
def black_litterman(returns, market_caps, views_P, views_Q, tau=0.05):
    """
    Black-Litterman model implementation.

    Parameters:
    - returns: Historical returns DataFrame
    - market_caps: Market capitalizations
    - views_P: View matrix (K x N)
    - views_Q: View vector (K x 1)
    - tau: Uncertainty scalar
    """
    Sigma = returns.cov().values * 252
```

```
    n = len(market_caps)

    # Market equilibrium weights
    w_mkt = market_caps / market_caps.sum()

    # Risk aversion coefficient
    delta = 2.5  # typical value

    # Equilibrium returns
    Pi = delta * Sigma @ w_mkt

    # View uncertainty (proportional to variance of view portfolios)
    Omega = np.diag(np.diag(views_P @ (tau * Sigma) @ views_P.T))

    # Black-Litterman expected returns
    inv_tau_sigma = np.linalg.inv(tau * Sigma)
    inv_omega = np.linalg.inv(Omega)

    M = np.linalg.inv(inv_tau_sigma + views_P.T @ inv_omega @ views_P)
    mu_bl = M @ (inv_tau_sigma @ Pi + views_P.T @ inv_omega @ views_Q)

    # Posterior covariance
    Sigma_bl = Sigma + M

    return mu_bl, Sigma_bl
```

1.2 Risk Measures

1.2.1 Value at Risk Mathematical Framework

Definition (VaR at confidence level α):

$$\text{VaR}_\alpha = -\inf\{x : P(L \leq x) \geq \alpha\}$$

Parametric VaR (assuming normality):

$$\text{VaR}_\alpha = -(\mu + \sigma\Phi^{-1}(1-\alpha))$$

Monte Carlo VaR:

```
def monte_carlo_var(returns, weights, n_simulations=10000, horizon=10,
    confidence=0.95):
    """
    Monte Carlo VaR with correlated asset returns.
    """
    mu = returns.mean().values
    Sigma = returns.cov().values

    # Cholesky decomposition for correlated simulations
    L = np.linalg.cholesky(Sigma)

    # Simulate returns
    z = np.random.standard_normal((n_simulations, len(mu)))
    simulated_returns = mu + z @ L.T

    # Scale to horizon
    simulated_returns = simulated_returns * np.sqrt(horizon)
```

```
    # Portfolio returns
    portfolio_returns = simulated_returns @ weights

    # VaR
    var = -np.percentile(portfolio_returns, (1 - confidence) * 100)

    return var, portfolio_returns
```

1.2.2 Expected Shortfall (CVaR)

$$\text{ES}_\alpha = -\mathbb{E}[L|L \leq -\text{VaR}_\alpha] = -\frac{1}{1-\alpha}\int_0^{1-\alpha} \text{VaR}_u \, du$$

Properties: - Coherent risk measure (satisfies subadditivity) - Tail risk sensitive - Convex optimization friendly

1.2.3 Risk Parity

Allocate risk equally across assets:

$$RC_i = w_i \frac{(\Sigma w)_i}{\sqrt{w^T \Sigma w}}$$

Objective: $RC_i = RC_j \; \forall i, j$

```
def risk_parity_weights(Sigma, budget=None):
    """
    Calculate risk parity weights.

    Parameters:
    - Sigma: Covariance matrix
    - budget: Risk budget (default: equal)
    """
    from scipy.optimize import minimize

    n = Sigma.shape[0]
    if budget is None:
        budget = np.ones(n) / n

    def objective(w):
        port_vol = np.sqrt(w @ Sigma @ w)
        marginal_contrib = Sigma @ w
        risk_contrib = w * marginal_contrib / port_vol
        target_contrib = budget * port_vol
        return np.sum((risk_contrib - target_contrib)**2)

    constraints = [{'type': 'eq', 'fun': lambda w: np.sum(w) - 1}]
    bounds = [(0.01, 1) for _ in range(n)]
    w0 = np.ones(n) / n

    result = minimize(objective, w0, method='SLSQP',
                      bounds=bounds, constraints=constraints)

    return result.x
```

1.3 Options Pricing and Greeks

1.3.1 Black-Scholes Formula

Call Option:

$$C = S_0 N(d_1) - K e^{-rT} N(d_2)$$

Where:

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

1.3.2 Greeks

Greek	Definition	Formula
Delta	$\partial C/\partial S$	$N(d_1)$
Gamma	$\partial^2 C/\partial S^2$	$\frac{N'(d_1)}{S\sigma\sqrt{T}}$
Vega	$\partial C/\partial \sigma$	$S N'(d_1)\sqrt{T}$
Theta	$\partial C/\partial T$	$\frac{S N'(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}N(d_2)$
Rho	$\partial C/\partial r$	$KTe^{-rT}N(d_2)$

1.3.3 Implied Volatility Surface

Learn the volatility smile using ML:

```
def train_vol_surface_model(options_data):
    """
    Train model to predict implied volatility.

    Features: moneyness, time to expiry, forward price
    Target: implied volatility
    """
    from sklearn.neural_network import MLPRegressor

    # Feature engineering
    X = options_data[['moneyness', 'time_to_expiry', 'forward_price']].copy()
    X['log_moneyness'] = np.log(X['moneyness'])
    X['sqrt_tte'] = np.sqrt(X['time_to_expiry'])

    y = options_data['implied_vol']

    # Train neural network
    model = MLPRegressor(hidden_layer_sizes=(64, 32, 16),
```

```
                          activation='relu',
                          solver='adam',
                          max_iter=1000)
    model.fit(X, y)

    return model
```

1.4 Time Series Models for Finance

1.4.1 GARCH(p,q) Model

Conditional variance:

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2$$

Stationarity condition: $\sum \alpha_i + \sum \beta_j < 1$

```
from arch import arch_model

def fit_garch(returns, p=1, q=1):
    """Fit GARCH model and forecast volatility."""
    model = arch_model(returns, vol='Garch', p=p, q=q, rescale=True)
    result = model.fit(disp='off')

    # Forecast
    forecast = result.forecast(horizon=5)
    vol_forecast = np.sqrt(forecast.variance.values[-1, :])

    return result, vol_forecast
```

1.4.2 Factor Models

Fama-French 3-Factor:

$$R_i - R_f = \alpha_i + \beta_i^{MKT}(R_M - R_f) + \beta_i^{SMB} \cdot SMB + \beta_i^{HML} \cdot HML + \epsilon_i$$

Principal Component Factors:

```
from sklearn.decomposition import PCA

def extract_statistical_factors(returns, n_factors=5):
    """Extract statistical factors from returns."""
    pca = PCA(n_components=n_factors)
    factors = pca.fit_transform(returns)

    # Factor loadings
    loadings = pca.components_.T

    # Variance explained
    var_explained = pca.explained_variance_ratio_

    return factors, loadings, var_explained
```

1.5 Credit Risk Modeling

1.5.1 Probability of Default (PD)

Merton Model (structural):

$$PD = N\left(-\frac{\ln(V_0/D) + (\mu - \sigma^2/2)T}{\sigma\sqrt{T}}\right)$$

Logistic Regression (reduced form):

$$\log\left(\frac{PD}{1-PD}\right) = \beta_0 + \sum_i \beta_i X_i$$

1.5.2 Loss Given Default (LGD)

$$\text{LGD} = 1 - \text{Recovery Rate}$$

Modeling approaches: - Beta regression (bounded 0-1) - Two-stage model (default vs recovery)

1.5.3 Expected Loss

$$EL = PD \times LGD \times EAD$$

Where EAD = Exposure at Default

1.6 Algorithmic Trading Considerations

1.6.1 Market Microstructure

Bid-Ask Spread Cost:

$$\text{Cost} = \frac{\text{Ask} - \text{Bid}}{2 \times \text{Mid}}$$

Market Impact (square-root model):

$$\text{Impact} = \sigma\sqrt{\frac{V}{ADV}}$$

Where V = trade volume, ADV = average daily volume

1.6.2 Execution Algorithms

TWAP (Time-Weighted Average Price): - Divide order into equal slices over time - Minimize timing risk

VWAP (Volume-Weighted Average Price): - Trade proportionally to expected volume profile - Benchmark for execution quality

Implementation Shortfall:

$$IS = \text{Side} \times (P_{avg} - P_{decision}) \times Q$$

1.7 Production ML Systems

1.7.1 Real-Time Feature Engineering

```
class FeaturePipeline:
    """Production-ready feature engineering."""

    def __init__(self, lookback_windows=[5, 20, 60]):
        self.lookback_windows = lookback_windows
        self.scalers = {}

    def compute_features(self, prices, volumes):
        features = {}

        for window in self.lookback_windows:
            # Returns
            returns = prices.pct_change(window)
            features[f'return_{window}d'] = returns

            # Volatility
            features[f'vol_{window}d'] = prices.pct_change().rolling(window).
    std()

            # Volume ratio
            features[f'vol_ratio_{window}d'] = volumes / volumes.rolling(
    window).mean()

            # Momentum
            features[f'momentum_{window}d'] = prices / prices.shift(window) -
    1

        return pd.DataFrame(features)

    def fit_transform(self, X):
        X_scaled = X.copy()
        for col in X.columns:
            self.scalers[col] = StandardScaler()
            X_scaled[col] = self.scalers[col].fit_transform(X[[col]])
        return X_scaled

    def transform(self, X):
        X_scaled = X.copy()
        for col in X.columns:
            X_scaled[col] = self.scalers[col].transform(X[[col]])
        return X_scaled
```

1.7.2 Model Monitoring

```
class ModelMonitor:
    """Monitor model performance in production."""

    def __init__(self, baseline_metrics):
        self.baseline = baseline_metrics
        self.history = []

    def evaluate(self, y_true, y_pred, timestamp):
        metrics = {
            'timestamp': timestamp,
            'accuracy': (y_true == y_pred).mean(),
```

```
            'auc': roc_auc_score(y_true, y_pred) if len(np.unique(y_true)) > 1
    else np.nan
        }
        self.history.append(metrics)

        # Check for drift
        alerts = []
        if metrics['accuracy'] < self.baseline['accuracy'] * 0.9:
            alerts.append(f"Accuracy degraded: {metrics['accuracy']:.3f}")

        return metrics, alerts

    def calculate_psi(self, expected, actual, bins=10):
        """Population Stability Index."""
        expected_perc = np.histogram(expected, bins=bins)[0] / len(expected)
        actual_perc = np.histogram(actual, bins=bins)[0] / len(actual)

        # Avoid division by zero
        expected_perc = np.clip(expected_perc, 1e-6, 1)
        actual_perc = np.clip(actual_perc, 1e-6, 1)

        psi = np.sum((actual_perc - expected_perc) * np.log(actual_perc /
    expected_perc))
        return psi
```

1.8 Regulatory Compliance

1.8.1 SR 11-7 Model Risk Management

Requirements: 1. Model development with sound theory 2. Independent validation 3. Ongoing monitoring 4. Documentation and audit trail 5. Limitations clearly stated

1.8.2 Model Documentation Template

```
## Model Documentation

### 1. Model Purpose
- Business use case
- Decision supported

### 2. Methodology
- Algorithm description
- Assumptions
- Limitations

### 3. Data
- Training data description
- Feature definitions
- Data quality checks

### 4. Performance
- Validation metrics
- Benchmark comparison
- Sensitivity analysis

### 5. Implementation
- Technical architecture
```

```
- Monitoring procedures
- Escalation thresholds

### 6. Governance
- Model owner
- Review schedule
- Change log
```

1.9 References

1. Markowitz, H. (1952). "Portfolio Selection"
2. Black, F., & Litterman, R. (1992). "Global Portfolio Optimization"
3. Merton, R. (1974). "On the Pricing of Corporate Debt"
4. Bollerslev, T. (1986). "Generalized Autoregressive Conditional Heteroskedasticity"
5. Federal Reserve SR 11-7: "Guidance on Model Risk Management"

In quantitative finance, rigor matters. Models that work in backtests often fail in production. Always validate assumptions, monitor performance, and maintain healthy skepticism.