Finance Applications of ML - Intermediate Handout

Machine Learning for Smarter Innovation

1 Finance Applications of ML - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.ensemble import RandomForestClassifier, GradientBoostingRegressor
from sklearn.metrics import classification_report, roc_auc_score
import warnings
warnings.filterwarnings('ignore')
```

1.2 1. Portfolio Optimization

1.2.1 Mean-Variance Optimization

```
import cvxpy as cp

def optimize_portfolio(returns, target_return=None, risk_free_rate=0.02):
    """
    Markowitz mean-variance optimization.

    Parameters:
    - returns: DataFrame of asset returns
    - target_return: Target portfolio return (None = max Sharpe)
    - risk_free_rate: Annual risk-free rate
    """
    n_assets = returns.shape[1]
    mu = returns.mean().values * 252  # Annualized
    Sigma = returns.cov().values * 252

    # Decision variable
    w = cp.Variable(n_assets)
```

```
    # Constraints
    constraints = [
        cp.sum(w) == 1,       # Fully invested
        w >= 0                # No short selling
    ]

    if target_return:
        constraints.append(mu @ w >= target_return)
        # Minimize risk for target return
        objective = cp.Minimize(cp.quad_form(w, Sigma))
    else:
        # Maximize Sharpe ratio (quadratic approximation)
        risk = cp.quad_form(w, Sigma)
        ret = mu @ w
        # Use risk-adjusted return as proxy
        objective = cp.Maximize(ret - 0.5 * risk)

    prob = cp.Problem(objective, constraints)
    prob.solve()

    weights = w.value
    port_return = mu @ weights
    port_risk = np.sqrt(weights @ Sigma @ weights)
    sharpe = (port_return - risk_free_rate) / port_risk

    return {
        'weights': dict(zip(returns.columns, weights)),
        'return': port_return,
        'risk': port_risk,
        'sharpe': sharpe
    }

# Usage
# returns = pd.DataFrame(...)  # Daily returns
# result = optimize_portfolio(returns)
```

1.2.2 Efficient Frontier

```
def plot_efficient_frontier(returns, n_points=50):
    """Plot the efficient frontier."""
    mu = returns.mean().values * 252
    Sigma = returns.cov().values * 252

    # Generate target returns
    min_ret = mu.min()
    max_ret = mu.max()
    target_returns = np.linspace(min_ret, max_ret, n_points)

    risks = []
    rets = []

    for target in target_returns:
        try:
            result = optimize_portfolio(returns, target_return=target)
            risks.append(result['risk'])
            rets.append(result['return'])
        except:
            continue

    plt.figure(figsize=(10, 6))
```

```
    plt.plot(risks, rets, 'b-', linewidth=2, label='Efficient Frontier')
    plt.scatter([np.sqrt(np.diag(Sigma))], [mu], c='red', s=100, label='
    Individual Assets')
    plt.xlabel('Risk (Std Dev)')
    plt.ylabel('Expected Return')
    plt.title('Efficient Frontier')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()
```

1.3 2. Risk Metrics

1.3.1 Value at Risk (VaR)

```
def calculate_var(returns, confidence=0.95, method='historical'):
    """
    Calculate Value at Risk.

    Methods:
    - 'historical': Empirical quantile
    - 'parametric': Assumes normal distribution
    - 'cornish_fisher': Adjusts for skewness/kurtosis
    """
    if method == 'historical':
        var = -np.percentile(returns, (1 - confidence) * 100)

    elif method == 'parametric':
        mu = returns.mean()
        sigma = returns.std()
        var = -(mu + sigma * stats.norm.ppf(1 - confidence))

    elif method == 'cornish_fisher':
        mu = returns.mean()
        sigma = returns.std()
        skew = stats.skew(returns)
        kurt = stats.kurtosis(returns)

        z = stats.norm.ppf(1 - confidence)
        z_cf = (z + (z**2 - 1) * skew / 6 +
                (z**3 - 3*z) * kurt / 24 -
                (2*z**3 - 5*z) * skew**2 / 36)
        var = -(mu + sigma * z_cf)

    return var

def calculate_cvar(returns, confidence=0.95):
    """Calculate Conditional VaR (Expected Shortfall)."""
    var = calculate_var(returns, confidence, method='historical')
    cvar = -returns[returns <= -var].mean()
    return cvar

# Usage
# var_95 = calculate_var(daily_returns, 0.95) * portfolio_value
# print(f"95% VaR: ${var_95:,.0f}")
```

1.3.2 Maximum Drawdown

```
def calculate_drawdown(returns):
    """Calculate maximum drawdown and drawdown series."""
    cum_returns = (1 + returns).cumprod()
    rolling_max = cum_returns.expanding().max()
    drawdown = (cum_returns - rolling_max) / rolling_max

    max_dd = drawdown.min()
    max_dd_end = drawdown.idxmin()

    # Find start of max drawdown
    peak_idx = cum_returns[:max_dd_end].idxmax()

    return {
        'max_drawdown': max_dd,
        'peak_date': peak_idx,
        'trough_date': max_dd_end,
        'drawdown_series': drawdown
    }
```

1.4 3. Credit Scoring Model

1.4.1 Feature Engineering for Credit

```
def engineer_credit_features(df):
    """Create features for credit scoring."""
    features = df.copy()

    # Debt ratios
    features['debt_to_income'] = df['total_debt'] / (df['annual_income'] + 1)
    features['credit_utilization'] = df['credit_used'] / (df['credit_limit'] +
     1)

    # Payment behavior
    features['avg_days_late'] = df['total_days_late'] / (df['num_payments'] +
    1)
    features['late_payment_ratio'] = df['late_payments'] / (df['total_payments
    '] + 1)

    # Account age
    features['avg_account_age'] = df['total_account_age'] / (df['num_accounts'
    ] + 1)

    # Inquiry intensity
    features['recent_inquiry_ratio'] = df['inquiries_6mo'] / (df['
    inquiries_total'] + 1)

    return features
```

1.4.2 Credit Scoring with Time-Based Split

```
def train_credit_model(X, y, date_column):
    """Train credit model with proper temporal validation."""
    # Sort by date
    X = X.sort_values(date_column)
    y = y.loc[X.index]
```

```
    # Time-based split
    tscv = TimeSeriesSplit(n_splits=5)

    scores = []
    feature_importance = []

    for train_idx, test_idx in tscv.split(X):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        y_train, y_test = y.iloc[train_idx], y.iloc[test_idx]

        # Remove date column for modeling
        X_train_model = X_train.drop(columns=[date_column])
        X_test_model = X_test.drop(columns=[date_column])

        # Scale features
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train_model)
        X_test_scaled = scaler.transform(X_test_model)

        # Train model
        model = GradientBoostingClassifier(
            n_estimators=100,
            max_depth=5,
            learning_rate=0.1,
            random_state=42
        )
        model.fit(X_train_scaled, y_train)

        # Evaluate
        y_pred_proba = model.predict_proba(X_test_scaled)[:, 1]
        auc = roc_auc_score(y_test, y_pred_proba)
        scores.append(auc)

        feature_importance.append(model.feature_importances_)

    print(f"Average AUC: {np.mean(scores):.4f} (+/- {np.std(scores):.4f})")

    # Average feature importance
    avg_importance = np.mean(feature_importance, axis=0)
    importance_df = pd.DataFrame({
        'feature': X.drop(columns=[date_column]).columns,
        'importance': avg_importance
    }).sort_values('importance', ascending=False)

    return model, scaler, importance_df
```

1.5 4. Trading Strategy Backtesting

1.5.1 Backtest Framework

```
class Backtester:
    def __init__(self, initial_capital=100000, transaction_cost=0.001):
        self.initial_capital = initial_capital
        self.transaction_cost = transaction_cost

    def run(self, prices, signals):
        """
```

```
        Run backtest.

        Parameters:
        - prices: Series of asset prices
        - signals: Series of signals (1=long, 0=flat, -1=short)
        """
        # Calculate returns
        returns = prices.pct_change()

        # Position changes
        position_changes = signals.diff().abs().fillna(0)

        # Strategy returns (signal * return - transaction costs)
        strategy_returns = signals.shift(1) * returns
        strategy_returns -= position_changes * self.transaction_cost

        # Portfolio value
        portfolio = self.initial_capital * (1 + strategy_returns).cumprod()

        # Metrics
        total_return = (portfolio.iloc[-1] / self.initial_capital) - 1
        ann_return = (1 + total_return) ** (252 / len(portfolio)) - 1
        ann_vol = strategy_returns.std() * np.sqrt(252)
        sharpe = ann_return / ann_vol if ann_vol > 0 else 0

        dd = calculate_drawdown(strategy_returns)

        return {
            'portfolio': portfolio,
            'total_return': total_return,
            'annual_return': ann_return,
            'annual_volatility': ann_vol,
            'sharpe_ratio': sharpe,
            'max_drawdown': dd['max_drawdown'],
            'num_trades': position_changes.sum()
        }

# Example: Moving Average Crossover
def ma_crossover_signals(prices, short_window=20, long_window=50):
    """Generate signals from moving average crossover."""
    short_ma = prices.rolling(short_window).mean()
    long_ma = prices.rolling(long_window).mean()

    signals = pd.Series(0, index=prices.index)
    signals[short_ma > long_ma] = 1
    signals[short_ma < long_ma] = -1

    return signals.fillna(0)
```

1.5.2 Walk-Forward Optimization

```
def walk_forward_optimization(prices, param_grid, window=252, step=63):
    """
    Walk-forward optimization to avoid overfitting.

    Parameters:
    - prices: Price series
    - param_grid: Dict of parameters to test
    - window: Training window size
    - step: Step size between windows
    """
```

```
    results = []

    for start in range(0, len(prices) - window - step, step):
        train_end = start + window
        test_end = min(train_end + step, len(prices))

        train_prices = prices.iloc[start:train_end]
        test_prices = prices.iloc[train_end:test_end]

        # Find best parameters on training set
        best_sharpe = -np.inf
        best_params = None

        for short in param_grid['short_window']:
            for long in param_grid['long_window']:
                if short >= long:
                    continue

                signals = ma_crossover_signals(train_prices, short, long)
                bt = Backtester()
                metrics = bt.run(train_prices, signals)

                if metrics['sharpe_ratio'] > best_sharpe:
                    best_sharpe = metrics['sharpe_ratio']
                    best_params = {'short': short, 'long': long}

        # Apply best parameters to test set
        test_signals = ma_crossover_signals(test_prices,
                                            best_params['short'],
                                            best_params['long'])
        bt = Backtester()
        test_metrics = bt.run(test_prices, test_signals)

        results.append({
            'period_start': test_prices.index[0],
            'period_end': test_prices.index[-1],
            'train_sharpe': best_sharpe,
            'test_sharpe': test_metrics['sharpe_ratio'],
            'best_params': best_params
        })

    return pd.DataFrame(results)
```

1.6 5. Fraud Detection

1.6.1 Anomaly Detection for Transactions

```
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler

def detect_fraud_anomalies(transactions, contamination=0.01):
    """
    Detect fraudulent transactions using Isolation Forest.

    Parameters:
    - transactions: DataFrame with transaction features
    - contamination: Expected proportion of anomalies
    """
```

```
    # Feature engineering
    features = transactions.copy()
    features['hour'] = pd.to_datetime(features['timestamp']).dt.hour
    features['day_of_week'] = pd.to_datetime(features['timestamp']).dt.
    dayofweek

    # Amount statistics per user
    user_stats = features.groupby('user_id')['amount'].agg(['mean', 'std'])
    features = features.merge(user_stats, on='user_id', suffixes=('', '_user')
    )
    features['amount_zscore'] = (features['amount'] - features['mean']) / (
    features['std'] + 1)

    # Select features for model
    model_features = ['amount', 'hour', 'day_of_week', 'amount_zscore']
    X = features[model_features].fillna(0)

    # Scale
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Isolation Forest
    iso_forest = IsolationForest(contamination=contamination, random_state=42)
    predictions = iso_forest.fit_predict(X_scaled)

    # -1 = anomaly, 1 = normal
    features['is_anomaly'] = predictions == -1
    features['anomaly_score'] = -iso_forest.score_samples(X_scaled)

    return features
```

1.7 6. Model Validation for Finance

1.7.1 Regulatory-Compliant Validation

```
def regulatory_validation_report(model, X_test, y_test, feature_names):
    """Generate validation report for regulatory compliance."""
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]

    report = {
        'model_performance': {
            'auc_roc': roc_auc_score(y_test, y_proba),
            'accuracy': (y_pred == y_test).mean(),
            'classification_report': classification_report(y_test, y_pred,
    output_dict=True)
        },
        'feature_importance': dict(zip(feature_names, model.
    feature_importances_)),
        'stability_tests': {},
        'fairness_metrics': {}
    }

    # Population Stability Index (PSI)
    # Compare score distribution between development and validation

    # Fairness metrics by protected class would go here
```

```
    return report
```

1.8 Common Financial Metrics

Metric	Formula	Interpretation
Sharpe Ratio	$(R_p - R_f)/\sigma_p$	Risk-adjusted return
Sortino Ratio	$(R_p - R_f)/\sigma_d$	Downside risk-adjusted
Information Ratio	$(R_p - R_b)/TE$	Active return per tracking error
Calmar Ratio	$R_p/MaxDD$	Return per drawdown

1.9 Practice Projects

1. **Portfolio Optimizer**: Build efficient frontier for ETF portfolio
2. **Credit Scorecard**: Develop interpretable credit model
3. **Pairs Trading**: Mean-reversion strategy with cointegration
4. **Fraud Detector**: Real-time anomaly detection system

1.10 Next Steps

- Implement more sophisticated models

- Add real market data (Alpha Vantage, Yahoo Finance)
- Study regulatory requirements in depth
- Read advanced handout for production deployment

In finance, a small edge compounded over time can be valuable. But beware of overfitting - if it looks too good to be true, it probably is.