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, cross_val_score
from sklearn.ensemble import (
    RandomForestClassifier, GradientBoostingClassifier,
    GradientBoostingRegressor, IsolationForest
)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    classification_report, roc_auc_score, roc_curve,
    precision_recall_curve, confusion_matrix
)
import warnings
warnings.filterwarnings('ignore')

# Visualization settings
plt.rcParams.update({
    'font.size': 10,
    'axes.labelsize': 10,
    'figure.figsize': (10, 6)
})
```

This handout covers practical implementation of machine learning applications in quantitative finance. Finance presents unique challenges including non-stationary data, regime changes, survivorship bias, and strict regulatory requirements. The techniques apply across portfolio management, credit risk, algorithmic trading, and fraud detection.

1.2 1. Portfolio Optimization

1.2.1 Concept Overview

Modern Portfolio Theory, developed by Harry Markowitz, provides a mathematical framework for constructing portfolios that maximize expected return for a given level of risk. The key insight is that diversification can reduce portfolio risk without sacrificing expected returns. Portfolio optimization involves finding optimal asset weights that lie on the efficient frontier - the set of portfolios offering the highest return for each risk level.

The optimization problem balances expected returns (estimated from historical data) against portfolio variance (a function of the covariance matrix). Practical constraints include full investment (weights sum to 1), no short selling (weights non-negative), position limits, and sector constraints.

1.2.2 Implementation

```
import cvxpy as cp

class PortfolioOptimizer:
    """Mean-variance portfolio optimization with constraints."""

    def __init__(self, returns, risk_free_rate=0.02):
        """
        Initialize optimizer with return data.

        Parameters:
        - returns: DataFrame of daily asset returns
        - risk_free_rate: Annual risk-free rate for Sharpe calculation
        """
        self.returns = returns
        self.risk_free_rate = risk_free_rate
        self.n_assets = returns.shape[1]

        # Annualized statistics
        self.mu = returns.mean().values * 252
        self.Sigma = returns.cov().values * 252
        self.asset_names = returns.columns.tolist()

    def optimize(self, target_return=None, max_weight=0.4, min_weight=0.0):
        """
        Find optimal portfolio weights.

        Parameters:
        - target_return: Target annual return (None = maximize Sharpe)
        - max_weight: Maximum allocation to single asset
        - min_weight: Minimum allocation to single asset

        Returns:
        - Dictionary with weights and portfolio metrics
        """
        # Decision variable
        w = cp.Variable(self.n_assets)

        # Constraints
        constraints = [
            cp.sum(w) == 1,           # Fully invested
            w >= min_weight,          # Minimum weight
            w <= max_weight           # Position limit
        ]

        if target_return is not None:
            # Minimum variance for target return
```

```
            constraints.append(self.mu @ w >= target_return)
            objective = cp.Minimize(cp.quad_form(w, self.Sigma))
        else:
            # Maximize risk-adjusted return (Sharpe proxy)
            risk = cp.quad_form(w, self.Sigma)
            ret = self.mu @ w
            objective = cp.Maximize(ret - 0.5 * risk)

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

        if prob.status != 'optimal':
            raise ValueError(f"Optimization failed: {prob.status}")

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

        return {
            'weights': dict(zip(self.asset_names, weights)),
            'return': port_return,
            'risk': port_risk,
            'sharpe': sharpe,
            'status': prob.status
        }

    def efficient_frontier(self, n_points=50):
        """
        Generate efficient frontier points.

        Returns DataFrame with risk, return, sharpe for each point.
        """
        min_ret = self.mu.min()
        max_ret = self.mu.max()
        target_returns = np.linspace(min_ret * 0.8, max_ret * 1.2, n_points)

        results = []
        for target in target_returns:
            try:
                result = self.optimize(target_return=target)
                results.append({
                    'target_return': target,
                    'actual_return': result['return'],
                    'risk': result['risk'],
                    'sharpe': result['sharpe']
                })
            except:
                continue

        return pd.DataFrame(results)

    def plot_frontier(self, ax=None):
        """Plot efficient frontier with individual assets."""
        frontier = self.efficient_frontier()

        if ax is None:
            fig, ax = plt.subplots(figsize=(10, 6))

        # Efficient frontier
        ax.plot(frontier['risk'], frontier['actual_return'],
                'b-', linewidth=2, label='Efficient Frontier')
```

```
        # Individual assets
        asset_risks = np.sqrt(np.diag(self.Sigma))
        ax.scatter(asset_risks, self.mu, c='red', s=100,
                   label='Individual Assets', zorder=5)

        for i, name in enumerate(self.asset_names):
            ax.annotate(name, (asset_risks[i], self.mu[i]),
                        xytext=(5, 5), textcoords='offset points')

        # Maximum Sharpe portfolio
        max_sharpe = self.optimize()
        ax.scatter(max_sharpe['risk'], max_sharpe['return'],
                   c='green', s=200, marker='*', label='Max Sharpe')

        ax.set_xlabel('Risk (Annualized Std Dev)')
        ax.set_ylabel('Expected Return (Annualized)')
        ax.set_title('Efficient Frontier')
        ax.legend()
        ax.grid(True, alpha=0.3)

        return ax


# Usage example with simulated data
np.random.seed(42)
n_days = 252 * 3
assets = ['AAPL', 'GOOGL', 'MSFT', 'AMZN', 'JPM']
returns = pd.DataFrame(
    np.random.randn(n_days, len(assets)) * 0.02 + 0.0005,
    columns=assets
)

optimizer = PortfolioOptimizer(returns)
result = optimizer.optimize()
print(f"Optimal Sharpe: {result['sharpe']:.3f}")
print("Weights:", {k: f"{v:.2%}" for k, v in result['weights'].items()})
```

1.3 2. Risk Metrics and Management

1.3.1 Concept Overview

Financial risk management requires quantifying potential losses under adverse market conditions. Value at Risk (VaR) answers the question: "What is the maximum loss at a given confidence level?" For example, a 95% daily VaR of $1 million means there is a 5% chance of losing more than $1 million in a single day.

VaR has limitations - it does not describe how bad losses could be beyond the threshold. Conditional VaR (Expected Shortfall) addresses this by measuring the average loss when losses exceed the VaR threshold. Maximum drawdown measures the largest peak-to-trough decline, which is particularly relevant for assessing strategy performance over time.

1.3.2 Implementation

```
class RiskMetrics:
    """Calculate comprehensive risk metrics for portfolios."""

    def __init__(self, returns):
```

```
        """
        Initialize with return series.

        Parameters:
        - returns: Series or array of returns
        """
        self.returns = np.array(returns)

    def var(self, confidence=0.95, method='historical'):
        """
        Calculate Value at Risk.

        Parameters:
        - confidence: Confidence level (e.g., 0.95 for 95%)
        - method: 'historical', 'parametric', or 'cornish_fisher'

        Returns:
        - VaR as positive number (loss)
        """
        alpha = 1 - confidence

        if method == 'historical':
            # Empirical quantile from observed returns
            var = -np.percentile(self.returns, alpha * 100)

        elif method == 'parametric':
            # Assume normal distribution
            mu = self.returns.mean()
            sigma = self.returns.std()
            var = -(mu + sigma * stats.norm.ppf(alpha))

        elif method == 'cornish_fisher':
            # Adjust for skewness and kurtosis
            mu = self.returns.mean()
            sigma = self.returns.std()
            skew = stats.skew(self.returns)
            kurt = stats.kurtosis(self.returns)

            z = stats.norm.ppf(alpha)
            # Cornish-Fisher expansion
            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)

        else:
            raise ValueError(f"Unknown method: {method}")

        return var

    def cvar(self, confidence=0.95):
        """
        Calculate Conditional VaR (Expected Shortfall).

        Returns:
        - CVaR as positive number (expected loss beyond VaR)
        """
        var = self.var(confidence, method='historical')
        tail_returns = self.returns[self.returns <= -var]

        if len(tail_returns) == 0:
            return var
```

```
        return -tail_returns.mean()

    def max_drawdown(self):
        """
        Calculate maximum drawdown statistics.

        Returns:
        - Dictionary with max drawdown, peak date, trough date
        """
        cum_returns = (1 + self.returns).cumprod()
        running_max = np.maximum.accumulate(cum_returns)
        drawdown = (cum_returns - running_max) / running_max

        max_dd = drawdown.min()
        max_dd_idx = drawdown.argmin()

        # Find peak before trough
        peak_idx = cum_returns[:max_dd_idx + 1].argmax()

        return {
            'max_drawdown': max_dd,
            'peak_idx': peak_idx,
            'trough_idx': max_dd_idx,
            'drawdown_series': drawdown
        }

    def risk_metrics_summary(self, confidence=0.95, risk_free=0.02):
        """Generate comprehensive risk summary."""
        ann_return = (1 + self.returns.mean()) ** 252 - 1
        ann_vol = self.returns.std() * np.sqrt(252)

        # Downside deviation (only negative returns)
        negative_returns = self.returns[self.returns < 0]
        downside_dev = negative_returns.std() * np.sqrt(252) if len(
    negative_returns) > 0 else 0

        dd = self.max_drawdown()

        return {
            'annual_return': ann_return,
            'annual_volatility': ann_vol,
            'sharpe_ratio': (ann_return - risk_free) / ann_vol if ann_vol > 0
    else 0,
            'sortino_ratio': (ann_return - risk_free) / downside_dev if
    downside_dev > 0 else 0,
            'calmar_ratio': ann_return / abs(dd['max_drawdown']) if dd['
    max_drawdown'] != 0 else 0,
            f'var_{int(confidence*100)}': self.var(confidence),
            f'cvar_{int(confidence*100)}': self.cvar(confidence),
            'max_drawdown': dd['max_drawdown'],
            'skewness': stats.skew(self.returns),
            'kurtosis': stats.kurtosis(self.returns)
        }


# Usage
np.random.seed(42)
daily_returns = np.random.randn(500) * 0.015 - 0.001  # Slightly negative
    drift

metrics = RiskMetrics(daily_returns)
summary = metrics.risk_metrics_summary()
```

```
print("Risk Metrics Summary:")
for key, value in summary.items():
    print(f"  {key}: {value:.4f}")
```

1.4 3. Credit Scoring Model

1.4.1 Concept Overview

Credit scoring uses machine learning to predict the probability of default for loan applicants. Unlike general ML applications, credit models face regulatory scrutiny requiring interpretability, stability over time, and fairness across demographic groups. Models must generalize to future applicants whose characteristics may differ from historical data.

Feature engineering transforms raw applicant data into predictive signals. Common features include debt-to-income ratios, credit utilization, payment history patterns, and account age metrics. Time-based validation is essential because we can only train on past defaults to predict future ones - random cross-validation would create unrealistic information leakage.

1.4.2 Implementation

```
class CreditScorer:
    """Credit scoring model with proper temporal validation."""

    def __init__(self, n_estimators=100, learning_rate=0.1):
        """
        Initialize credit scoring model.

        Parameters:
        - n_estimators: Number of boosting iterations
        - learning_rate: Learning rate for gradient boosting
        """
        self.model = GradientBoostingClassifier(
            n_estimators=n_estimators,
            max_depth=5,
            learning_rate=learning_rate,
            random_state=42
        )
        self.scaler = StandardScaler()
        self.feature_names = None

    def engineer_features(self, df):
        """
        Create credit scoring features.

        Parameters:
        - df: DataFrame with raw applicant data

        Returns:
        - DataFrame with engineered features
        """
        features = pd.DataFrame(index=df.index)

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

```
        features['monthly_debt_ratio'] = df['monthly_payment'] / (df['
    monthly_income'] + 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)
        features['missed_payments'] = df['payments_missed']

        # Account characteristics
        features['avg_account_age'] = df['total_account_age'] / (df['
    num_accounts'] + 1)
        features['oldest_account'] = df['oldest_account_months']
        features['new_accounts_6mo'] = df['accounts_opened_6mo']

        # Inquiry patterns
        features['inquiries_6mo'] = df['inquiries_6mo']
        features['inquiry_intensity'] = df['inquiries_6mo'] / (df['
    inquiries_total'] + 1)

        # Credit mix
        features['num_credit_cards'] = df['num_credit_cards']
        features['num_installment_loans'] = df['num_installment_loans']

        return features

    def train_with_temporal_split(self, X, y, date_column, n_splits=5):
        """
        Train model with time-series cross-validation.

        Parameters:
        - X: Feature DataFrame with date column
        - y: Target Series (1 = default, 0 = no default)
        - date_column: Name of date column for temporal ordering
        - n_splits: Number of time-series splits

        Returns:
        - Dictionary with training results and metrics
        """
        # Sort by date
        X = X.sort_values(date_column)
        y = y.loc[X.index]

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

        scores = []
        feature_importance_list = []

        # Remove date for modeling
        X_model = X.drop(columns=[date_column])
        self.feature_names = X_model.columns.tolist()

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

            # Scale features
            X_train_scaled = self.scaler.fit_transform(X_train)
            X_test_scaled = self.scaler.transform(X_test)
```

```
            # Train
            self.model.fit(X_train_scaled, y_train)

            # Evaluate
            y_proba = self.model.predict_proba(X_test_scaled)[:, 1]
            auc = roc_auc_score(y_test, y_proba)
            scores.append(auc)
            feature_importance_list.append(self.model.feature_importances_)

            print(f"Fold {fold + 1}: AUC = {auc:.4f}")

        # Final model on all data
        X_scaled = self.scaler.fit_transform(X_model)
        self.model.fit(X_scaled, y)

        # Average feature importance
        avg_importance = np.mean(feature_importance_list, axis=0)
        self.importance_df = pd.DataFrame({
            'feature': self.feature_names,
            'importance': avg_importance
        }).sort_values('importance', ascending=False)

        return {
            'mean_auc': np.mean(scores),
            'std_auc': np.std(scores),
            'fold_scores': scores,
            'feature_importance': self.importance_df
        }

    def predict_proba(self, X):
        """Predict default probability for new applicants."""
        if self.feature_names is None:
            raise ValueError("Model not trained. Call
train_with_temporal_split first.")

        X_scaled = self.scaler.transform(X[self.feature_names])
        return self.model.predict_proba(X_scaled)[:, 1]

    def score_to_rating(self, probabilities, n_grades=10):
        """
        Convert probabilities to credit grades.

        Parameters:
        - probabilities: Array of default probabilities
        - n_grades: Number of grade buckets

        Returns:
        - Array of grade labels (A1, A2, ..., D3)
        """
        # Higher probability = worse grade
        percentiles = np.percentile(probabilities, np.linspace(0, 100,
n_grades + 1))

        grades = []
        labels = ['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3', 'D']

        for p in probabilities:
            for i, threshold in enumerate(percentiles[1:]):
                if p <= threshold:
                    grades.append(labels[min(i, len(labels) - 1)])
                    break
            else:
```

```
                grades.append(labels[-1])

        return grades


# Usage with simulated data
np.random.seed(42)
n_samples = 5000

credit_data = pd.DataFrame({
    'application_date': pd.date_range('2020-01-01', periods=n_samples, freq='D
    '),
    'annual_income': np.random.lognormal(11, 0.5, n_samples),
    'total_debt': np.random.lognormal(10, 0.8, n_samples),
    'credit_used': np.random.uniform(0, 10000, n_samples),
    'credit_limit': np.random.uniform(5000, 50000, n_samples),
    'monthly_income': np.random.lognormal(9, 0.5, n_samples),
    'monthly_payment': np.random.uniform(100, 2000, n_samples),
    'total_days_late': np.random.poisson(5, n_samples),
    'num_payments': np.random.randint(12, 120, n_samples),
    'late_payments': np.random.poisson(2, n_samples),
    'total_payments': np.random.randint(24, 200, n_samples),
    'payments_missed': np.random.poisson(1, n_samples),
    'total_account_age': np.random.uniform(12, 240, n_samples),
    'num_accounts': np.random.randint(1, 15, n_samples),
    'oldest_account_months': np.random.uniform(24, 300, n_samples),
    'accounts_opened_6mo': np.random.poisson(1, n_samples),
    'inquiries_6mo': np.random.poisson(2, n_samples),
    'inquiries_total': np.random.poisson(10, n_samples),
    'num_credit_cards': np.random.randint(0, 10, n_samples),
    'num_installment_loans': np.random.randint(0, 5, n_samples)
})

# Simulate defaults (correlated with features)
default_prob = 0.1 + 0.2 * (credit_data['debt_to_income'] := credit_data['
    total_debt'] / credit_data['annual_income'])
default_prob = np.clip(default_prob, 0, 0.5)
defaults = np.random.binomial(1, default_prob)

scorer = CreditScorer()
features = scorer.engineer_features(credit_data)
features['application_date'] = credit_data['application_date']

results = scorer.train_with_temporal_split(features, pd.Series(defaults), '
    application_date')
print(f"\nMean AUC: {results['mean_auc']:.4f} (+/- {results['std_auc']:.4f})")
```

1.5 4. Trading Strategy Backtesting

1.5.1 Concept Overview

Backtesting evaluates trading strategies on historical data to estimate future performance. However, backtesting is fraught with biases: look-ahead bias (using future information), survivorship bias (only testing on assets that survived), and overfitting (optimizing parameters to fit noise). Walk-forward optimization addresses overfitting by repeatedly optimizing on past data and testing on unseen future data.

A proper backtest framework accounts for transaction costs, slippage (price movement between signal and execution), and realistic position sizing. Results should be interpreted skeptically - strategies that

look good in backtests often underperform live.

1.5.2 Implementation

```
class Backtester:
    """Backtest trading strategies with realistic assumptions."""

    def __init__(self, initial_capital=100000, transaction_cost=0.001,
    slippage=0.0005):
        """
        Initialize backtester.

        Parameters:
        - initial_capital: Starting portfolio value
        - transaction_cost: Cost per trade (e.g., 0.001 = 0.1%)
        - slippage: Price impact per trade
        """
        self.initial_capital = initial_capital
        self.transaction_cost = transaction_cost
        self.slippage = slippage

    def run(self, prices, signals):
        """
        Run backtest on price series with signals.

        Parameters:
        - prices: Series of asset prices
        - signals: Series of signals (1=long, 0=flat, -1=short)

        Returns:
        - Dictionary with performance metrics and portfolio series
        """
        returns = prices.pct_change().fillna(0)

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

        # Strategy returns = position * return - costs
        strategy_returns = signals.shift(1).fillna(0) * returns
        strategy_returns -= position_changes * (self.transaction_cost + self.
    slippage)

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

        # Calculate metrics
        metrics = RiskMetrics(strategy_returns.values)
        risk_summary = metrics.risk_metrics_summary()

        total_trades = position_changes.sum()

        # Benchmark (buy and hold)
        benchmark_returns = returns
        benchmark_portfolio = self.initial_capital * (1 + benchmark_returns).
    cumprod()

        return {
            'portfolio': portfolio,
            'benchmark': benchmark_portfolio,
            'strategy_returns': strategy_returns,
            'total_return': (portfolio.iloc[-1] / self.initial_capital) - 1,
```

```
            'benchmark_return': (benchmark_portfolio.iloc[-1] / self.
    initial_capital) - 1,
            'num_trades': total_trades,
            **risk_summary
        }

    def plot_results(self, results, title='Strategy Performance'):
        """Plot backtest results."""
        fig, axes = plt.subplots(2, 2, figsize=(14, 10))

        # Portfolio value
        ax = axes[0, 0]
        results['portfolio'].plot(ax=ax, label='Strategy')
        results['benchmark'].plot(ax=ax, label='Benchmark', alpha=0.7)
        ax.set_title('Portfolio Value')
        ax.legend()
        ax.grid(True, alpha=0.3)

        # Drawdown
        ax = axes[0, 1]
        dd = RiskMetrics(results['strategy_returns'].values).max_drawdown()
        ax.fill_between(range(len(dd['drawdown_series'])),
                        dd['drawdown_series'] * 100, alpha=0.7)
        ax.set_title('Drawdown (%)')
        ax.grid(True, alpha=0.3)

        # Rolling Sharpe
        ax = axes[1, 0]
        rolling_ret = results['strategy_returns'].rolling(63).mean() * 252
        rolling_vol = results['strategy_returns'].rolling(63).std() * np.sqrt
    (252)
        rolling_sharpe = rolling_ret / rolling_vol
        rolling_sharpe.plot(ax=ax)
        ax.axhline(y=0, color='r', linestyle='--', alpha=0.5)
        ax.set_title('Rolling 3-Month Sharpe Ratio')
        ax.grid(True, alpha=0.3)

        # Return distribution
        ax = axes[1, 1]
        results['strategy_returns'].hist(bins=50, ax=ax, alpha=0.7)
        ax.axvline(x=0, color='r', linestyle='--')
        ax.set_title('Return Distribution')
        ax.grid(True, alpha=0.3)

        plt.suptitle(title, fontsize=14)
        plt.tight_layout()
        return fig


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)


def walk_forward_optimization(prices, param_grid, train_window=252,
    test_window=63):
```

```
    """
    Walk-forward optimization to reduce overfitting.

    Parameters:
    - prices: Price series
    - param_grid: Dictionary of parameters to test
    - train_window: Training period length (days)
    - test_window: Test period length (days)

    Returns:
    - DataFrame with optimization results by period
    """
    results = []
    total_len = len(prices)

    for start in range(0, total_len - train_window - test_window, test_window)
    :
        train_end = start + train_window
        test_end = min(train_end + test_window, total_len)

        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}

        # Test best parameters on out-of-sample data
        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] if hasattr(test_prices.index,
    '__getitem__') else start,
            'train_sharpe': best_sharpe,
            'test_sharpe': test_metrics['sharpe_ratio'],
            'test_return': test_metrics['total_return'],
            'best_short': best_params['short'],
            'best_long': best_params['long']
        })

    return pd.DataFrame(results)


# Usage
np.random.seed(42)
dates = pd.date_range('2020-01-01', periods=756, freq='D')
```

```
prices = pd.Series(100 * np.exp(np.cumsum(np.random.randn(756) * 0.02)), index
    =dates)

signals = ma_crossover_signals(prices, 20, 50)
bt = Backtester()
results = bt.run(prices, signals)

print(f"Strategy Return: {results['total_return']:.2%}")
print(f"Benchmark Return: {results['benchmark_return']:.2%}")
print(f"Sharpe Ratio: {results['sharpe_ratio']:.3f}")
print(f"Max Drawdown: {results['max_drawdown']:.2%}")
```

1.6 5. Fraud Detection

1.6.1 Concept Overview

Fraud detection uses anomaly detection to identify suspicious transactions that deviate from normal patterns. The challenge is extreme class imbalance - fraud typically represents less than 1% of transactions. Traditional classifiers tend to predict "not fraud" for everything, achieving high accuracy but missing actual fraud.

Isolation Forest is effective for fraud detection because it explicitly models anomalies rather than normal behavior. It works by randomly isolating observations - anomalies require fewer splits to isolate, leading to shorter path lengths. Feature engineering that captures user-level behavioral patterns (typical transaction amounts, times, locations) significantly improves detection.

1.6.2 Implementation

```
class FraudDetector:
    """Fraud detection using anomaly scoring."""

    def __init__(self, contamination=0.01):
        """
        Initialize fraud detector.

        Parameters:
        - contamination: Expected proportion of fraud (0-0.5)
        """
        self.contamination = contamination
        self.model = IsolationForest(
            contamination=contamination,
            n_estimators=100,
            random_state=42
        )
        self.scaler = StandardScaler()

    def engineer_features(self, transactions):
        """
        Create fraud detection features.

        Parameters:
        - transactions: DataFrame with transaction data

        Returns:
        - DataFrame with engineered features
        """
        df = transactions.copy()
```

```
        # Time features
        df['timestamp'] = pd.to_datetime(df['timestamp'])
        df['hour'] = df['timestamp'].dt.hour
        df['day_of_week'] = df['timestamp'].dt.dayofweek
        df['is_weekend'] = df['day_of_week'].isin([5, 6]).astype(int)
        df['is_night'] = df['hour'].isin(range(0, 6)).astype(int)

        # User-level statistics
        user_stats = df.groupby('user_id')['amount'].agg(['mean', 'std', '
    count'])
        user_stats.columns = ['user_mean', 'user_std', 'user_count']
        df = df.merge(user_stats, left_on='user_id', right_index=True)

        # Transaction deviation from user norm
        df['amount_zscore'] = (df['amount'] - df['user_mean']) / (df['user_std
    '] + 1)
        df['amount_ratio'] = df['amount'] / (df['user_mean'] + 1)

        # Velocity features (if timestamps are ordered)
        df = df.sort_values(['user_id', 'timestamp'])
        df['time_since_last'] = df.groupby('user_id')['timestamp'].diff().dt.
    total_seconds() / 3600
        df['time_since_last'] = df['time_since_last'].fillna(24)  # Default to
     24 hours

        return df

    def fit_predict(self, transactions):
        """
        Fit model and predict fraud scores.

        Parameters:
        - transactions: DataFrame with transaction data

        Returns:
        - DataFrame with fraud predictions and scores
        """
        features = self.engineer_features(transactions)

        # Select model features
        model_cols = ['amount', 'hour', 'day_of_week', 'is_weekend', 'is_night
    ',
                      'amount_zscore', 'amount_ratio', 'time_since_last']
        X = features[model_cols].fillna(0)

        # Scale and fit
        X_scaled = self.scaler.fit_transform(X)
        predictions = self.model.fit_predict(X_scaled)

        # Add results to features
        features['is_fraud_pred'] = predictions == -1
        features['fraud_score'] = -self.model.score_samples(X_scaled)

        return features

    def evaluate(self, predictions, true_labels):
        """
        Evaluate fraud detection performance.

        Parameters:
        - predictions: DataFrame from fit_predict
        - true_labels: Series with actual fraud labels
```

```
        Returns:
        - Dictionary with evaluation metrics
        """
        y_pred = predictions['is_fraud_pred'].astype(int)
        y_score = predictions['fraud_score']

        # Metrics
        cm = confusion_matrix(true_labels, y_pred)
        tn, fp, fn, tp = cm.ravel()

        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0

        return {
            'precision': precision,
            'recall': recall,
            'f1': 2 * precision * recall / (precision + recall) if (precision
    + recall) > 0 else 0,
            'auc_roc': roc_auc_score(true_labels, y_score),
            'true_positives': tp,
            'false_positives': fp,
            'false_negatives': fn,
            'true_negatives': tn
        }


# Usage with simulated data
np.random.seed(42)
n_transactions = 10000

transactions = pd.DataFrame({
    'transaction_id': range(n_transactions),
    'user_id': np.random.randint(0, 500, n_transactions),
    'amount': np.abs(np.random.lognormal(4, 1, n_transactions)),
    'timestamp': pd.date_range('2024-01-01', periods=n_transactions, freq='5
    min')
})

# Inject fraud (unusual amounts, odd hours)
fraud_idx = np.random.choice(n_transactions, size=int(n_transactions * 0.01),
    replace=False)
transactions.loc[fraud_idx, 'amount'] *= 10
true_fraud = np.zeros(n_transactions)
true_fraud[fraud_idx] = 1

detector = FraudDetector(contamination=0.02)
results = detector.fit_predict(transactions)
metrics = detector.evaluate(results, true_fraud)

print("Fraud Detection Results:")
for key, value in metrics.items():
    print(f"  {key}: {value:.4f}" if isinstance(value, float) else f"  {key}:
    {value}")
```

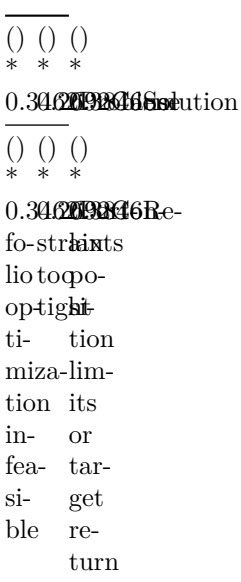
1.7 Common Financial Parameters

Application	Parameter	Typical Range	Notes
Portfolio Opt	Risk-free rate	0.01-0.05	Current Treasury rate
Portfolio Opt	Max position	0.2-0.4	Single asset limit
VaR	Confidence	0.95-0.99	Higher for regulation
VaR	Lookback	252-504 days	1-2 years typical
Credit Score	n_estimators	100-500	More for stability
Credit Score	max_depth	3-7	Shallow for interpretability
Backtest	Transaction cost	0.0005-0.002	Depends on asset class
Backtest	Slippage	0.0002-0.001	Higher for illiquid
Fraud	Contamination	0.001-0.05	Based on fraud rate
Fraud	n_estimators	100-200	More trees = stability

1.8 Practice Projects

1. **Multi-Asset Portfolio**: Build an optimizer for a portfolio of ETFs (SPY, TLT, GLD, etc.) with sector constraints and rebalancing rules. Compare monthly vs quarterly rebalancing and analyze turnover costs.
2. **Credit Scorecard Development**: Create an interpretable credit scoring model using logistic regression with weight-of-evidence encoding. Implement Population Stability Index (PSI) monitoring to detect model drift.
3. **Momentum Strategy Backtest**: Implement a cross-sectional momentum strategy that goes long top-decile performers and short bottom-decile. Use walk-forward optimization to select lookback periods and holding periods.
4. **Real-Time Fraud Detection**: Build a fraud detection pipeline that processes transactions in streaming fashion. Implement user-level feature computation with exponential moving averages for recency weighting.

1.9 Troubleshooting



() *	() *	() *		
0.3	4	2	9	Evolution
() *	() *	() *		
0.3	4	2	9	Re-
a- tive Sharpe in it back- test	eg- un- pro- ble- look- ahead bias	view sig- nals, check for		
() *	() *	() *		
0.3	4	2	9	Re-
mod AUC de- clin- ing	del train Gif re- cent data, mon- i- tor PSI			
() *	() *	() *		
0.3	4	2	9	Use
un- der- es- ti- mates risk	tail Cornish- Fisher or his- tor- i- cal VaR			
() *	() *	() *		
0.3	4	2	9	lower
de- teci- torna- high false pos- i- tives	tam- mon- tam- i- tion na- tion, tune thresh- old			

() *	() *	() *
0.3	4	2
() *	() *	() *
0.3	4	2
for-	fit-	sim-
sho	wa	sler
deg	tra-	strat-
da-	pas	egy,
tion		fewer
		pa-
		ram-
		e-
		ters
() *	() *	() *
0.3	4	2
tes	in	re-
un-	cos	ts
re-		is-
al-		tic
is-		trans-
tic		ac-
re-		tion
turns		costs
		and
		slip-
		page
() *	() *	() *
0.3	4	2
se-	spl	ture
ries		li- fu-
CV		rect- ture
da	tia	on ever
leak-		leaks
age		into
		train-
		ing

1.10 Next Steps

- Read the advanced handout for factor models and regime detection
- Implement live paper trading with realistic execution simulation
- Study regulatory requirements (Basel III, SR 11-7, CECL)
- Explore alternative data sources (satellite, web scraping, NLP on filings)
- Build production monitoring dashboards with drift detection

---

Financial ML requires unusual caution - markets are adversarial, non-stationary, and unforgiving of overfitting. A healthy skepticism of backtest results and rigorous out-of-sample validation separate successful practitioners from those who learned expensive lessons.