market-prediction-elsie (1)

November 28, 2025

[1]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
 ↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
 ↪docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
 ↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
 ↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
 ↪outside of the current session
```

```
/kaggle/input/market-prediction/train.csv
/kaggle/input/market-prediction/test.csv
```

[2]:
```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import Lasso, Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, TimeSeriesSplit
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy import stats
import warnings
warnings.filterwarnings('ignore')
```

1 DATA LOADING

[3]:
```
# Load both datasets
train_df = pd.read_csv('/kaggle/input/market-prediction/train.csv')
test_df = pd.read_csv('/kaggle/input/market-prediction/test.csv')

print("Train data shape:", train_df.shape)
print("Test data shape:", test_df.shape)
print("\nTarget column:", 'lagged_market_forward_excess_returns')
```

```
Train data shape: (9021, 98)
Test data shape: (10, 99)

Target column: lagged_market_forward_excess_returns
```

2 Preprocess train and test data

[4]:
```
# Competition-specific data preparation
def prepare_competition_features(test_df, is_training=True):
    """Prepare features for the competition format"""
    # Features to exclude
    exclude_cols = ['date_id', 'is_scored']
    if is_training:
        exclude_cols.extend(['lagged_forward_returns', 'lagged_risk_free_rate',
                             'lagged_market_forward_excess_returns'])
    else:
        exclude_cols.extend(['lagged_forward_returns', 'lagged_risk_free_rate'])

    feature_columns = [col for col in test_df.columns if col not in␣
 ↪exclude_cols]
    X = test_df[feature_columns].copy()

    # Add volatility features
    X['volatility_D'] = X[[f'D{i}' for i in range(1, 10) if f'D{i}' in X.
 ↪columns]].std(axis=1)

    e_cols = [col for col in X.columns if col.startswith('E')]
    if e_cols:
        X['volatility_E'] = X[e_cols].std(axis=1)

    m_cols = [col for col in X.columns if col.startswith('M')]
    if m_cols:
        X['volatility_M'] = X[m_cols].std(axis=1)

    if is_training:
        y = test_df['lagged_market_forward_excess_returns'].copy()
        return X, y
```

```
    else:
        return X

# Prepare training data
X_train, y_train = prepare_competition_features(test_df, is_training=True)
X_test = prepare_competition_features(test_df, is_training=False)

print("Training features shape:", X_train.shape)
print("Test features shape:", X_test.shape)
```

```
Training features shape: (10, 97)
Test features shape: (10, 98)
```

[5]:
```
# Competition-optimized predictor for S&P500 allocations
class SP500AllocationPredictor:
    def __init__(self):
        self.models = {
            'rf': RandomForestRegressor(n_estimators=100, random_state=42,␣
 ↪max_depth=7),
            'gbm': GradientBoostingRegressor(n_estimators=100, random_state=42,␣
 ↪max_depth=4),
            'lasso': Lasso(alpha=0.001, random_state=42, max_iter=2000),
            'ridge': Ridge(alpha=0.1, random_state=42)
        }
        self.scaler = StandardScaler()
        self.volatility_threshold = None
        self.return_scaling_factor = None

    def calculate_volatility_threshold(self, returns, percentile=70):
        """Calculate volatility threshold for position sizing"""
        if len(returns) < 10:
            self.volatility_threshold = returns.std()
        else:
            volatility = returns.rolling(window=10, min_periods=1).std()
            self.volatility_threshold = np.percentile(volatility.dropna(),␣
 ↪percentile)
        return self.volatility_threshold

    def calculate_return_scaling(self, returns):
        """Calculate scaling factor to convert predictions to allocations"""
        # Use the inverse of return std for scaling predictions to reasonable␣
 ↪allocation ranges
        self.return_scaling_factor = 1.0 / (returns.std() + 1e-8)
        return self.return_scaling_factor

    def fit(self, X, y):
        """Fit models on training data"""
```

```
        # Handle NaN values
        X = X.fillna(X.mean())
        y = y.fillna(y.mean())

        # Scale features
        X_scaled = self.scaler.fit_transform(X)

        # Calculate thresholds and scaling factors
        self.calculate_volatility_threshold(pd.Series(y))
        self.calculate_return_scaling(pd.Series(y))

        # Train each model
        for name, model in self.models.items():
            model.fit(X_scaled, y)

        return self

    def predict_allocations(self, X, current_volatility=None):
        """Predict S&P500 allocations in range [0, 2]"""
        X = X.fillna(X.mean())
        X_scaled = self.scaler.transform(X)

        # Get predictions from all models
        predictions = {}
        for name, model in self.models.items():
            predictions[name] = model.predict(X_scaled)

        # Ensemble prediction (weighted average)
        weights = {'rf': 0.4, 'gbm': 0.4, 'lasso': 0.1, 'ridge': 0.1}
        ensemble_pred = sum(weights[name] * predictions[name] for name in␣
↪predictions)

        # Convert predictions to allocations (0-2 range)
        allocations = self._convert_to_allocations(ensemble_pred,␣
↪current_volatility)

        return allocations, ensemble_pred

    def _convert_to_allocations(self, predictions, current_volatility=None):
        """Convert raw predictions to allocations in [0, 2] range"""
        # Scale predictions to reasonable range
        if self.return_scaling_factor is not None:
            scaled_predictions = predictions * self.return_scaling_factor
        else:
            scaled_predictions = predictions * 10  # fallback scaling

        # Apply volatility adjustment if provided
```

```
        if current_volatility is not None and self.volatility_threshold is not␣
↪None:
            if current_volatility > self.volatility_threshold:
                # Reduce allocation during high volatility
                vol_penalty = self.volatility_threshold / (current_volatility +␣
↪1e-8)
                scaled_predictions = scaled_predictions * vol_penalty

        # Convert to allocations using sigmoid-like function scaled to [0, 2]
        # Using tanh transformation: maps to [-1, 1] then shift to [0, 2]
        allocations = np.tanh(scaled_predictions) + 1

        # Ensure allocations are in [0, 2] range
        allocations = np.clip(allocations, 0, 2)

        return allocations

# Initialize and train the model
predictor = SP500AllocationPredictor()
predictor.fit(X_train, y_train)

print("Model training completed!")
print(f"Volatility threshold: {predictor.volatility_threshold:.6f}")
print(f"Return scaling factor: {predictor.return_scaling_factor:.6f}")
```

```
Model training completed!
Volatility threshold: 0.006507
Return scaling factor: 182.386384
```

[6]:
```
# Generate competition submissions
def generate_submission(predictor, test_df, X_test, output_file='submission.
↪csv'):
    """Generate competition submission file with allocations"""
    # Calculate current volatility for test set
    current_volatility = X_test['volatility_D'].mean() if 'volatility_D' in␣
↪X_test.columns else None

    # Predict allocations for test set
    allocations, predictions = predictor.predict_allocations(X_test,␣
↪current_volatility)

    # Create submission dataframe
    submission = pd.DataFrame({
        'date_id': test_df['date_id'],
        'allocation': allocations
    })
```

```
    # Validate allocations
    print("Allocation statistics:")
    print(f"  Min: {allocations.min():.4f}")
    print(f"  Max: {allocations.max():.4f}")
    print(f"  Mean: {allocations.mean():.4f}")
    print(f"  Std: {allocations.std():.4f}")

    # Save to CSV
    submission.to_csv(output_file, index=False)
    print(f"\nSubmission saved to {output_file}")

    return submission, predictions

# Generate submission file
submission, test_predictions = generate_submission(predictor, test_df, X_train)
```

```
Allocation statistics:
  Min: 0.9681
  Max: 1.0326
  Mean: 1.0054
  Std: 0.0203

Submission saved to submission.csv
```

[7]:
```
# Validate submission format
print("Submission preview:")
print(submission.head(10))
print(f"\nSubmission shape: {submission.shape}")

# Check that all allocations are within valid range [0, 2]
valid_range = (submission['allocation'] >= 0) & (submission['allocation'] <= 2)
print(f"Allocations within valid range [0, 2]: {valid_range.all()}")
if not valid_range.all():
    print("WARNING: Some allocations outside valid range!")
    print(f"Min allocation: {submission['allocation'].min()}")
    print(f"Max allocation: {submission['allocation'].max()}")
```

```
Submission preview:
   date_id  allocation
0     8980    1.009755
1     8981    0.977268
2     8982    0.968132
3     8983    1.018201
4     8984    1.031367
5     8985    0.988180
6     8986    1.008126
7     8987    1.009056
8     8988    1.011684
```

```
9     8989    1.032634

Submission shape: (10, 2)
Allocations within valid range [0, 2]: True
```

3 Model evaluation

[8]:
```python
# Backtest on training data (for validation)
def backtest_strategy(predictor, X_train, y_train, train_df):
    """Backtest the strategy on training data to estimate performance"""
    # Use time series split for realistic backtest
    from sklearn.model_selection import TimeSeriesSplit

    tscv = TimeSeriesSplit(n_splits=5)
    sharpe_ratios = []
    returns = []

    for train_idx, val_idx in tscv.split(X_train):
        # Split data
        X_fold_train, X_fold_val = X_train.iloc[train_idx], X_train.
↪iloc[val_idx]
        y_fold_train, y_fold_val = y_train.iloc[train_idx], y_train.
↪iloc[val_idx]

        # Train model on fold
        fold_predictor = SP500AllocationPredictor()
        fold_predictor.fit(X_fold_train, y_fold_train)

        # Predict allocations on validation
        current_volatility = X_fold_val['volatility_D'].mean() if␣
↪'volatility_D' in X_fold_val.columns else None
        allocations, _ = fold_predictor.predict_allocations(X_fold_val,␣
↪current_volatility)

        # Calculate strategy returns (allocation * market excess returns)
        strategy_returns = allocations * y_fold_val.values

        # Calculate Sharpe ratio
        if len(strategy_returns) > 1 and np.std(strategy_returns) > 0:
            sharpe = np.mean(strategy_returns) / np.std(strategy_returns)
            sharpe_ratios.append(sharpe)
            returns.extend(strategy_returns)

    if sharpe_ratios:
        avg_sharpe = np.mean(sharpe_ratios)
        total_return = np.prod(1 + np.array(returns)) - 1 if returns else 0
        print(f"Backtest Sharpe Ratio: {avg_sharpe:.4f}")
```

```
        print(f"Total Return: {total_return:.4f}")
    else:
        print("Insufficient data for backtest")

    return sharpe_ratios

print("=== BACKTEST RESULTS ===")
sharpe_ratios = backtest_strategy(predictor, X_train, y_train, train_df)
```

```
=== BACKTEST RESULTS ===
Insufficient data for backtest
```

[9]:
```
# Feature importance analysis for competition strategy
def analyze_competition_features(predictor, feature_names):
    """Analyze which features drive the allocation decisions"""
    rf_model = predictor.models['rf']
    importance_scores = rf_model.feature_importances_

    # Create feature importance dataframe
    importance_df = pd.DataFrame({
        'feature': feature_names,
        'importance': importance_scores
    }).sort_values('importance', ascending=False)

    print("\n=== TOP 15 FEATURES DRIVING ALLOCATIONS ===")
    print(importance_df.head(15))

    return importance_df

feature_importance = analyze_competition_features(predictor, X_train.columns.
 ↪tolist())
```

```
=== TOP 15 FEATURES DRIVING ALLOCATIONS ===
         feature  importance
19           E19    0.109962
61            P2    0.100581
89            V5    0.058506
46           M17    0.054923
40           M11    0.042981
63            P4    0.042216
26            E7    0.041276
82           V10    0.032285
95  volatility_E    0.031848
48            M2    0.028249
65            P6    0.021607
55            M9    0.019837
68            P9    0.019117
```

```
59           P12    0.018038
37            I9    0.017663
```

[10]:
```python
# Strategy explanation and rationale
def explain_strategy(predictor, test_predictions, submission):
    """Explain the strategy and its rationale"""
    print("\n" + "="*60)
    print("COMPETITION STRATEGY EXPLANATION")
    print("="*60)

    print("\nSTRATEGY RATIONALE:")
    print("1. ENSEMBLE APPROACH: Combines Random Forest, Gradient Boosting,␣
 ↪Lasso, and Ridge")
    print("2. VOLATILITY MANAGEMENT: Reduces allocation during high volatility␣
 ↪periods")
    print("3. RISK-CONTROLLED: Uses sigmoid transformation to ensure␣
 ↪allocations in [0, 2]")
    print("4. ADAPTIVE: Learns from multiple feature types (D, E, M, P, S, V␣
 ↪groups)")

    print(f"\nALLOCATION DISTRIBUTION:")
    print(f"  Mean Allocation: {submission['allocation'].mean():.3f}")
    print(f"  Allocation Std: {submission['allocation'].std():.3f}")

    # Count allocations in different ranges
    low_alloc = (submission['allocation'] < 0.5).sum()
    med_alloc = ((submission['allocation'] >= 0.5) & (submission['allocation']␣
 ↪< 1.5)).sum()
    high_alloc = (submission['allocation'] >= 1.5).sum()

    print(f"\nALLOCATION BREAKDOWN:")
    print(f"  Conservative (<0.5): {low_alloc} days ({low_alloc/
 ↪len(submission)*100:.1f}%)")
    print(f"  Moderate (0.5-1.5): {med_alloc} days ({med_alloc/
 ↪len(submission)*100:.1f}%)")
    print(f"  Aggressive (>1.5): {high_alloc} days ({high_alloc/
 ↪len(submission)*100:.1f}%)")

    print("\nKEY FEATURES:")
    top_features = feature_importance.head(5)['feature'].tolist()
    for i, feature in enumerate(top_features, 1):
        print(f"  {i}. {feature}")

explain_strategy(predictor, test_predictions, submission)
```

```
============================================================
```

```
COMPETITION STRATEGY EXPLANATION
============================================================

STRATEGY RATIONALE:
1. ENSEMBLE APPROACH: Combines Random Forest, Gradient Boosting, Lasso, and
Ridge
2. VOLATILITY MANAGEMENT: Reduces allocation during high volatility periods
3. RISK-CONTROLLED: Uses sigmoid transformation to ensure allocations in [0, 2]
4. ADAPTIVE: Learns from multiple feature types (D, E, M, P, S, V groups)

ALLOCATION DISTRIBUTION:
  Mean Allocation: 1.005
  Allocation Std: 0.021

ALLOCATION BREAKDOWN:
  Conservative (<0.5): 0 days (0.0%)
  Moderate (0.5-1.5): 10 days (100.0%)
  Aggressive (>1.5): 0 days (0.0%)

KEY FEATURES:
  1. E19
  2. P2
  3. V5
  4. M17
  5. M11
```

[11]:
```python
# Final competition submission preparation
print("\n" + "="*60)
print("FINAL COMPETITION SUBMISSION")
print("="*60)

print("  Model trained on provided features")
print("  Allocations constrained to [0, 2] range")
print("  Volatility-aware position sizing")
print("  Ensemble approach for robust predictions")
print(f"  Submission file created with {len(submission)} predictions")

print("\nSUBMISSION VALIDATION:")
print(f"- All allocations in [0, 2]: {(submission['allocation'].between(0, 2)).
 ↪all()}")
print(f"- Unique date_ids: {submission['date_id'].nunique() ==␣
 ↪len(submission)}")
print(f"- No missing values: {not submission.isnull().any().any()}")

print("\nNEXT STEPS:")
print("1. Submit 'submission.csv' through the competition API")
print("2. Monitor leaderboard performance")
```

```python
print("3. Consider feature engineering or model tuning if needed")
```

```
============================================================
FINAL COMPETITION SUBMISSION
============================================================
  Model trained on provided features
  Allocations constrained to [0, 2] range
  Volatility-aware position sizing
  Ensemble approach for robust predictions
  Submission file created with 10 predictions

SUBMISSION VALIDATION:
- All allocations in [0, 2]: True
- Unique date_ids: True
- No missing values: True

NEXT STEPS:
1. Submit 'submission.csv' through the competition API
2. Monitor leaderboard performance
3. Consider feature engineering or model tuning if needed
```