```
!pip install pandas numpy scikit-learn xgboost plotly seaborn statsmodels joblib
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (2.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (1.26.4)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.0)
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (2.1.3)
Requirement already satisfied: plotly in /usr/local/lib/python3.11/dist-packages (5.24.1)
Requirement already satisfied: seaborn in /usr/local/lib/python3.11/dist-packages (0.13.2)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.11/dist-packages (0.14.4)
Requirement already satisfied: joblib in /usr/local/lib/python3.11/dist-packages (1.4.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas) (2.8.
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas) (2024.2)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-packages (from xgboost) (2.21.5)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.11/dist-packages (from plotly) (9.0.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from plotly) (24.2)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in /usr/local/lib/python3.11/dist-packages (from seaborn) (3.
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels) (1.0.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,>=
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,>=3.4-
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,>=3.4->s
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib!=3.6.1,>
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pan
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import TimeSeriesSplit
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
import xgboost as xgb
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from datetime import datetime
import joblib
import warnings
warnings.filterwarnings('ignore')
```

```
def load_and_clean_data(file_path):
    """Load and clean Bloomberg format data"""
    # Read the CSV file
    print("Loading data...")
    df = pd.read_csv(file_path)

    # Extract ticker row (row 6) for column names
    ticker_row = df.iloc[5]  # Changed from 6 to 5 since Python is 0-based
    actual_columns = []

    # Process column names
    for i, val in enumerate(ticker_row):
        if pd.notna(val) and str(val).strip() != '':
            actual_columns.append(str(val).strip())
        else:
            actual_columns.append(f'Asset_{i}')

    # Get data starting from row 7
    data_df = df.iloc[7:].copy()
    data_df.columns = actual_columns

    # Create numeric columns
```

```
    numeric_df = pd.DataFrame()
    for col in data_df.columns:
        try:
            # Replace special values with NaN
            series = data_df[col].replace(['#N/A N/A', 'N/A', '', 'NaN'], np.nan)
            # Convert to numeric
            numeric_series = pd.to_numeric(series, errors='coerce')
            # Only keep columns with at least 50% non-null values
            if numeric_series.notna().sum() > len(numeric_series) * 0.5:
                numeric_df[col] = numeric_series
        except Exception as e:
            print(f"Skipping column {col}: {str(e)}")
            continue

    # Set index
    try:
        if 'Date' in data_df.columns:
            dates = pd.to_datetime(data_df['Date'], format='%m/%d/%Y', errors='coerce')
            numeric_df.index = dates
        else:
            numeric_df.index = pd.date_range(start='2000-01-01', periods=len(numeric_df), freq='B')
    except Exception as e:
        print(f"Error setting dates: {str(e)}")
        numeric_df.index = pd.date_range(start='2000-01-01', periods=len(numeric_df), freq='B')

    # Forward fill and backfill missing values
    numeric_df = numeric_df.fillna(method='ffill').fillna(method='bfill')

    print(f"Processed data shape: {numeric_df.shape}")
    print("Available columns:", numeric_df.columns.tolist())

    return numeric_df


def create_features(df, windows=[5, 10, 20, 50]):
    """Create comprehensive feature set"""
    features = pd.DataFrame(index=df.index)

    for col in df.columns:
        print(f"Creating features for {col}")
        # Basic price features
        features[f'{col}_raw'] = df[col]

        # Returns
        features[f'{col}_return_1d'] = df[col].pct_change()
        for window in windows:
            features[f'{col}_return_{window}d'] = df[col].pct_change(window)

        # Moving averages and related
        for window in windows:
            # Simple moving average
            ma = df[col].rolling(window=window).mean()
            features[f'{col}_ma_{window}'] = ma
            features[f'{col}_ma_ratio_{window}'] = df[col] / ma

            # Exponential moving average
            ema = df[col].ewm(span=window).mean()
            features[f'{col}_ema_{window}'] = ema
            features[f'{col}_ema_ratio_{window}'] = df[col] / ema

            # Volatility
            features[f'{col}_volatility_{window}'] = features[f'{col}_return_1d'].rolling(window).std()

            # Moving average crossovers
            if window > windows[0]:
                features[f'{col}_ma_cross_{windows[0]}_{window}'] = (
                    features[f'{col}_ma_{windows[0]}'] > features[f'{col}_ma_{window}']
                ).astype(int)

        # Technical indicators
        # RSI
        delta = df[col].diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
```

```
        loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
        rs = gain / loss
        features[f'{col}_rsi'] = 100 - (100 / (1 + rs))

        # MACD
        exp12 = df[col].ewm(span=12, adjust=False).mean()
        exp26 = df[col].ewm(span=26, adjust=False).mean()
        macd = exp12 - exp26
        signal = macd.ewm(span=9, adjust=False).mean()
        features[f'{col}_macd'] = macd - signal

        # Bollinger Bands
        for window in [20]:
            rolling_mean = df[col].rolling(window=window).mean()
            rolling_std = df[col].rolling(window=window).std()
            features[f'{col}_bb_upper_{window}'] = rolling_mean + (rolling_std * 2)
            features[f'{col}_bb_lower_{window}'] = rolling_mean - (rolling_std * 2)
            features[f'{col}_bb_position_{window}'] = (
                (df[col] - features[f'{col}_bb_lower_{window}']) /
                (features[f'{col}_bb_upper_{window}'] - features[f'{col}_bb_lower_{window}'])
            )

    # Remove infinite values
    features = features.replace([np.inf, -np.inf], np.nan)
    features = features.fillna(method='ffill').fillna(method='bfill')

    # Drop highly correlated features
    corr_matrix = features.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
    features = features.drop(columns=to_drop)

    print(f"Final feature shape: {features.shape}")
    return features


def create_crash_labels(df, threshold=-0.02, window=5):
    """Create forward-looking crash labels"""
    # Identify market columns (looking for common market index names)
    market_cols = [col for col in df.columns
                  if any(market in col.upper()
                        for market in ['SP', 'S&P', 'NASDAQ', 'DAX', 'FTSE', 'INDEX'])]

    # If no market columns found, use the first 3 columns
    if not market_cols:
        market_cols = df.columns[:3]

    print("Using market columns:", market_cols)

    # Calculate returns
    market_returns = df[market_cols].pct_change()

    # Forward-looking crash labels
    crashes = pd.DataFrame(index=df.index)
    for i in range(window):
        crashes[f'day_{i}'] = (market_returns.shift(-i) < threshold).any(axis=1)

    labels = crashes.any(axis=1).astype(int)
    print(f"Created labels with {labels.sum()} crash events")

    return labels



def analyze_first_rows(file_path, n_rows=10):
    """Analyze the first few rows of the CSV to understand its structure"""
    df = pd.read_csv(file_path, nrows=n_rows)
    print("\nFirst few rows of raw data:")
    print(df.head(n_rows))

    print("\nColumn names:")
    print(df.columns.tolist())

    # Look at specific rows
```

```
        # Look at specific rows
        print("\nRow 5 (ticker row):")
        print(df.iloc[5])

        return df

    # Let's analyze the data first
    print("Analyzing CSV structure...")
    raw_df = analyze_first_rows('FinancialMarketData.csv')

    # Then load and process the data
    df = load_and_clean_data('FinancialMarketData.csv')

    if df.shape[1] == 0:
        raise ValueError("No valid numeric columns found in the data. Please check the data format.")
```

```
Unnamed: 7         NaN
Unnamed: 8         NaN
Unnamed: 9         NaN
Unnamed: 10        NaN
Unnamed: 11        NaN
Unnamed: 12        NaN
Unnamed: 13        NaN
Unnamed: 14        NaN
Unnamed: 15        NaN
Unnamed: 16        NaN
Unnamed: 17        NaN
Unnamed: 18        NaN
Unnamed: 19        NaN
Unnamed: 20        NaN
Unnamed: 21        NaN
Unnamed: 22        NaN
Unnamed: 23        NaN
Unnamed: 24        NaN
Unnamed: 25        NaN
Unnamed: 26        NaN
Unnamed: 27        NaN
Unnamed: 28        NaN
Unnamed: 29        NaN
Unnamed: 30        NaN
Unnamed: 31        NaN
Unnamed: 32        NaN
Unnamed: 33        NaN
Unnamed: 34        NaN
Unnamed: 35        NaN
Unnamed: 36        NaN
Unnamed: 37        NaN
Unnamed: 38        NaN
Unnamed: 39        NaN
Unnamed: 40        NaN
Unnamed: 41        NaN
Unnamed: 42        NaN
Unnamed: 43        NaN
Unnamed: 44        NaN
Unnamed: 45        NaN
Unnamed: 46        NaN
Unnamed: 47        NaN
Unnamed: 48        NaN
Unnamed: 49        NaN
Unnamed: 50        NaN
Unnamed: 51        NaN
Unnamed: 52        NaN
Unnamed: 53        NaN
Unnamed: 54        NaN
Unnamed: 55        NaN
Unnamed: 56        NaN
Unnamed: 57        NaN
Unnamed: 58        NaN
Unnamed: 59        NaN
Name: 5, dtype: object
Loading data...
Processed data shape: (1147, 57)
Available columns: ['284.250', 'Asset_4', 'Asset_5', 'Asset_6', 'Asset_7', 'Asset_8', 'Asset_9', 'Asset_10', 'Ass
```

```
    def analyze_time_series(df):
        """Perform time series analysis with proper error handling"""
        for col in df.columns[:3]:  # Analyze first 3 assets
```

```
        print(f"\nAnalyzing {col}")
        series = df[col].replace([np.inf, -np.inf], np.nan).dropna()

        if len(series) == 0:
            print(f"No valid data for {col}")
            continue

        # Stationarity test
        try:
            adf_result = adfuller(series)
            print('ADF Statistic:', adf_result[0])
            print('p-value:', adf_result[1])
        except Exception as e:
            print(f"Could not perform ADF test: {str(e)}")

        # Plot time series components
        plt.figure(figsize=(15, 10))

        # Original series
        plt.subplot(411)
        plt.plot(series)
        plt.title(f'{col} - Original Series')

        # Returns with handling for infinities
        returns = series.pct_change()
        returns = returns.replace([np.inf, -np.inf], np.nan).dropna()
        plt.subplot(412)
        plt.plot(returns)
        plt.title('Returns')

        # Volatility
        volatility = returns.rolling(20).std()
        plt.subplot(413)
        plt.plot(volatility)
        plt.title('20-day Rolling Volatility')

        # Returns distribution (with finite values only)
        plt.subplot(414)
        plt.hist(returns, bins='auto')
        plt.title('Returns Distribution')

        plt.tight_layout()
        plt.show()


from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
import graphviz
import pydotplus
from IPython.display import Image


# Add to imports
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
import graphviz
import pydotplus
from IPython.display import Image

def train_decision_tree(features, labels, max_depth=5):
    """
    Train and visualize decision tree
    """
    # Scale features
    scaler = RobustScaler()
    features_scaled = scaler.fit_transform(features)
    features_scaled = pd.DataFrame(features_scaled, columns=features.columns, index=features.index)

    # Create train/test split using time series split
    tscv = TimeSeriesSplit(n_splits=5)
    splits = list(tscv.split(features_scaled))
    train_index, test_index = splits[-1]  # Use last split

    X_train = features_scaled.iloc[train_index]
    X_test = features_scaled.iloc[test_index]
```

```
        y_train = labels.iloc[train_index]
        y_test = labels.iloc[test_index]

        # Train decision tree
        dt = DecisionTreeClassifier(
            max_depth=max_depth,
            min_samples_split=20,
            min_samples_leaf=10,
            class_weight='balanced',
            random_state=42
        )

        dt.fit(X_train, y_train)

        # Evaluate
        y_pred = dt.predict(X_test)
        print("\nDecision Tree Performance:")
        print(classification_report(y_test, y_pred))

        # Plot confusion matrix
        plt.figure(figsize=(8, 6))
        cm = confusion_matrix(y_test, y_pred)
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
        plt.title('Decision Tree Confusion Matrix')
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.show()

        # Plot ROC curve
        y_pred_proba = dt.predict_proba(X_test)[:, 1]
        fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
        roc_auc = auc(fpr, tpr)

        plt.figure(figsize=(8, 6))
        plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
        plt.plot([0, 1], [0, 1], 'k--')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('Decision Tree ROC Curve')
        plt.legend()
        plt.show()

        # Visualize decision tree
        plt.figure(figsize=(20,10))
        plot_tree(dt,
                 feature_names=features.columns,
                 class_names=['No Crash', 'Crash'],
                 filled=True,
                 rounded=True,
                 fontsize=10)
        plt.title('Decision Tree Visualization')
        plt.show()

        # Feature importance
        importance = pd.DataFrame({
            'feature': features.columns,
            'importance': dt.feature_importances_
        }).sort_values('importance', ascending=False)

        plt.figure(figsize=(12, 6))
        sns.barplot(data=importance.head(20), x='importance', y='feature')
        plt.title('Decision Tree Feature Importance')
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

        return dt, scaler

    def analyze_decision_paths(dt, features, feature_names):
        """
        Analyze decision paths and rules
        """
        n_nodes = dt.tree_.node_count
```

```
    children_left = dt.tree_.children_left
    children_right = dt.tree_.children_right
    feature = dt.tree_.feature
    threshold = dt.tree_.threshold

    # Get decision path for each node
    node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
    is_leaves = np.zeros(shape=n_nodes, dtype=bool)
    stack = [(0, 0)]  # (node_id, depth)

    while len(stack) > 0:
        node_id, depth = stack.pop()
        node_depth[node_id] = depth

        is_split_node = children_left[node_id] != children_right[node_id]
        if is_split_node:
            stack.append((children_left[node_id], depth + 1))
            stack.append((children_right[node_id], depth + 1))
        else:
            is_leaves[node_id] = True

    print("\nDecision Tree Rules:")
    for i in range(n_nodes):
        if is_leaves[i]:
            print(f"\nLeaf node {i} (depth {node_depth[i]}):")
            continue

        feature_name = feature_names[feature[i]]
        threshold_value = threshold[i]

        print(f"\nNode {i} (depth {node_depth[i]}):")
        print(f"If {feature_name} <= {threshold_value:.2f}:")
        print(f"    go to node {children_left[i]}")
        print(f"else:")
        print(f"    go to node {children_right[i]}")


def train_ensemble_models(features, labels):
    """Train ensemble models"""
    # Scale features
    scaler = RobustScaler()
    features_scaled = scaler.fit_transform(features)
    features_scaled = pd.DataFrame(features_scaled, columns=features.columns, index=features.index)

    # Time series split
    tscv = TimeSeriesSplit(n_splits=5)
    splits = list(tscv.split(features_scaled))
    train_index, test_index = splits[-1]

    X_train = features_scaled.iloc[train_index]
    X_test = features_scaled.iloc[test_index]
    y_train = labels.iloc[train_index]
    y_test = labels.iloc[test_index]

    # Initialize models
    models = {
        'Random Forest': RandomForestClassifier(
            n_estimators=200,
            max_depth=10,
            min_samples_split=5,
            class_weight='balanced',
            random_state=42
        ),
        'XGBoost': xgb.XGBClassifier(
            n_estimators=200,
            max_depth=5,
            learning_rate=0.1,
            scale_pos_weight=5,
            eval_metric='logloss'
        ),
        'Gradient Boosting': GradientBoostingClassifier(
            n_estimators=200,
            max_depth=5,
```

```
            learning_rate=0.1,
            random_state=42
        )
    }

    results = {}
    for name, model in models.items():
        print(f"\nTraining {name}...")
        model.fit(X_train, y_train)

        # Predictions
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)[:, 1]

        print(f"\n{name} Performance:")
        print(classification_report(y_test, y_pred))

        # Plot confusion matrix
        plt.figure(figsize=(8, 6))
        cm = confusion_matrix(y_test, y_pred)
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
        plt.title(f'{name} Confusion Matrix')
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.show()

        # Plot feature importance
        if hasattr(model, 'feature_importances_'):
            importances = pd.DataFrame({
                'feature': features.columns,
                'importance': model.feature_importances_
            }).sort_values('importance', ascending=False)

            plt.figure(figsize=(12, 6))
            sns.barplot(data=importances.head(20), x='importance', y='feature')
            plt.title(f'{name} Top 20 Features')
            plt.xticks(rotation=45)
            plt.tight_layout()
            plt.show()

        results[name] = {
            'model': model,
            'predictions': y_pred,
            'probabilities': y_pred_proba
        }

    return models, scaler, (X_test, y_test), results


# Add these functions to your existing code

def train_ensemble_models(features, labels):
    """Train ensemble models"""
    # Scale features
    scaler = RobustScaler()
    features_scaled = scaler.fit_transform(features)
    features_scaled = pd.DataFrame(features_scaled, columns=features.columns, index=features.index)

    # Time series split
    tscv = TimeSeriesSplit(n_splits=5)
    splits = list(tscv.split(features_scaled))
    train_index, test_index = splits[-1]

    X_train = features_scaled.iloc[train_index]
    X_test = features_scaled.iloc[test_index]
    y_train = labels.iloc[train_index]
    y_test = labels.iloc[test_index]

    # Initialize models
    models = {
        'Random Forest': RandomForestClassifier(
            n_estimators=200,
```

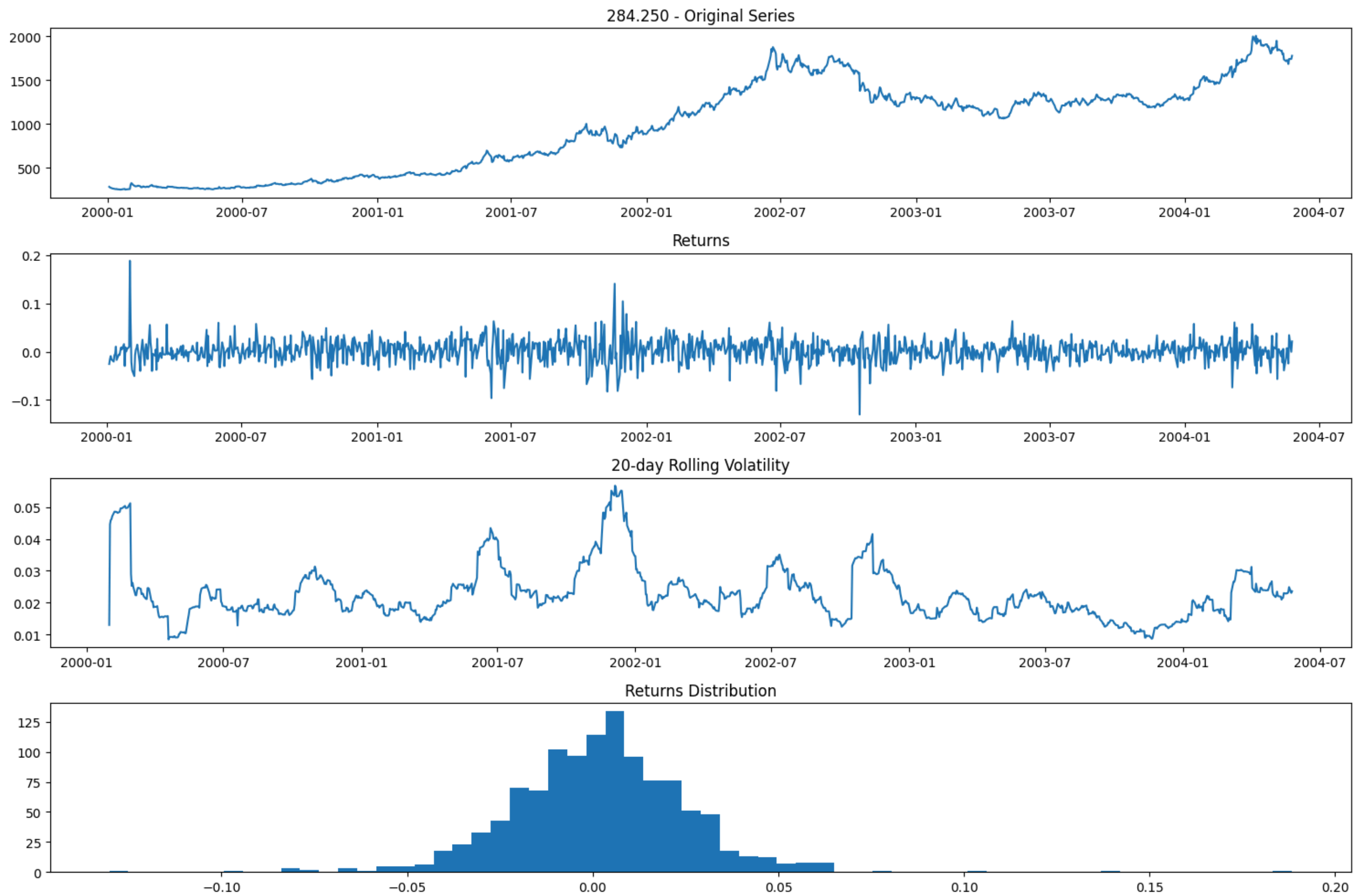
```
            max_depth=10,
            min_samples_split=5,
            class_weight='balanced',
            random_state=42
        ),
        'XGBoost': xgb.XGBClassifier(
            n_estimators=200,
            max_depth=5,
            learning_rate=0.1,
            scale_pos_weight=5,
            eval_metric='logloss'
        ),
        'Gradient Boosting': GradientBoostingClassifier(
            n_estimators=200,
            max_depth=5,
            learning_rate=0.1,
            random_state=42
        )
    }

    results = {}
    for name, model in models.items():
        print(f"\nTraining {name}...")
        model.fit(X_train, y_train)

        # Predictions
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)[:, 1]

        print(f"\n{name} Performance:")
        print(classification_report(y_test, y_pred))

        # Plot confusion matrix
        plt.figure(figsize=(8, 6))
        cm = confusion_matrix(y_test, y_pred)
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
        plt.title(f'{name} Confusion Matrix')
        plt.ylabel('True Label')
        plt.xlabel('Predicted Label')
        plt.show()

        # Plot feature importance
        if hasattr(model, 'feature_importances_'):
            importances = pd.DataFrame({
                'feature': features.columns,
                'importance': model.feature_importances_
            }).sort_values('importance', ascending=False)

            plt.figure(figsize=(12, 6))
            sns.barplot(data=importances.head(20), x='importance', y='feature')
            plt.title(f'{name} Top 20 Features')
            plt.xticks(rotation=45)
            plt.tight_layout()
            plt.show()

        results[name] = {
            'model': model,
            'predictions': y_pred,
            'probabilities': y_pred_proba
        }

    return models, scaler, (X_test, y_test), results

def evaluate_models(results, test_data):
    """Evaluate all models and ensemble"""
    X_test, y_test = test_data

    # Compare ROC curves
    plt.figure(figsize=(10, 6))
    for name, result in results.items():
        fpr, tpr, _ = roc_curve(y_test, result['probabilities'])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.2f})')
```

```
    # Add ensemble
    ensemble_proba = np.mean([result['probabilities'] for result in results.values()], axis=0)
    fpr, tpr, _ = roc_curve(y_test, ensemble_proba)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'Ensemble (AUC = {roc_auc:.2f})', linestyle='--')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Model Comparison - ROC Curves')
    plt.legend()
    plt.show()

    # Ensemble predictions
    ensemble_pred = (ensemble_proba > 0.5).astype(int)
    print("\nEnsemble Model Performance:")
    print(classification_report(y_test, ensemble_pred))

    return ensemble_pred, ensemble_proba

def visualize_predictions(features, predictions_df):
    """Visualize predictions over time with proper error handling"""
    fig = go.Figure()

    # Add actual values
    fig.add_trace(go.Scatter(
        x=predictions_df.index,
        y=predictions_df['Actual'],
        name='Actual',
        mode='markers',
        marker=dict(size=10)
    ))

    # Add probabilities
    fig.add_trace(go.Scatter(
        x=predictions_df.index,
        y=predictions_df['Ensemble_Probability'],
        name='Crash Probability',
        line=dict(width=2)
    ))

    # Add original price data
    for col in features.columns[:3]:  # Plot first 3 assets
        try:
            # Use the original column directly
            normalized_price = (features[col] - features[col].mean()) / features[col].std()
            fig.add_trace(go.Scatter(
                x=features.index,
                y=normalized_price,
                name=f'{col} (normalized)',
                opacity=0.3
            ))
        except Exception as e:
            print(f"Error plotting {col}: {str(e)}")
            continue

    fig.update_layout(
        title='Market Crash Predictions vs Actual',
        yaxis_title='Value',
        template='plotly_white',
        height=600
    )
    fig.show()



def create_features(df, windows=[5, 10, 20, 50]):
    """Create comprehensive feature set with proper column naming"""
    features = pd.DataFrame(index=df.index)

    for col in df.columns:
        print(f"Creating features for {col}")
```

```
        # Basic price features
        features[f'{col}'] = df[col]  # Store original values without _raw suffix

        # Returns
        features[f'{col}_return_1d'] = df[col].pct_change()
        for window in windows:
            features[f'{col}_return_{window}d'] = df[col].pct_change(window)

        # Moving averages and related
        for window in windows:
            # Simple moving average
            ma = df[col].rolling(window=window).mean()
            features[f'{col}_ma_{window}'] = ma
            features[f'{col}_ma_ratio_{window}'] = df[col] / ma

            # Exponential moving average
            ema = df[col].ewm(span=window).mean()
            features[f'{col}_ema_{window}'] = ema
            features[f'{col}_ema_ratio_{window}'] = df[col] / ema

            # Volatility
            returns = df[col].pct_change()
            features[f'{col}_volatility_{window}'] = returns.rolling(window).std()

            # Moving average crossovers
            if window > windows[0]:
                features[f'{col}_ma_cross_{windows[0]}_{window}'] = (
                    features[f'{col}_ma_{windows[0]}'] > features[f'{col}_ma_{window}']
                ).astype(int)

        # Technical indicators
        # RSI
        delta = df[col].diff()
        gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
        loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
        rs = gain / loss
        features[f'{col}_rsi'] = 100 - (100 / (1 + rs))

        # MACD
        exp12 = df[col].ewm(span=12, adjust=False).mean()
        exp26 = df[col].ewm(span=26, adjust=False).mean()
        macd = exp12 - exp26
        signal = macd.ewm(span=9, adjust=False).mean()
        features[f'{col}_macd'] = macd - signal

        # Bollinger Bands
        rolling_mean = df[col].rolling(window=20).mean()
        rolling_std = df[col].rolling(window=20).std()
        features[f'{col}_bb_upper'] = rolling_mean + (rolling_std * 2)
        features[f'{col}_bb_lower'] = rolling_mean - (rolling_std * 2)
        features[f'{col}_bb_position'] = (
            (df[col] - features[f'{col}_bb_lower']) /
            (features[f'{col}_bb_upper'] - features[f'{col}_bb_lower'])
        )

    # Remove infinite values
    features = features.replace([np.inf, -np.inf], np.nan)
    features = features.fillna(method='ffill').fillna(method='bfill')

    # Drop highly correlated features
    corr_matrix = features.corr().abs()
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
    to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
    features = features.drop(columns=to_drop)

    print(f"Final feature shape: {features.shape}")
    return features


def main():
    # Load and prepare data
    print("Loading data...")
    df = load_and_clean_data('FinancialMarketData.csv')
```

```
    # Analyze time series
    print("\nPerforming time series analysis...")
    analyze_time_series(df)

    # Create features
    print("\nGenerating features...")
    features = create_features(df)

    # Create labels
    print("\nGenerating labels...")
    labels = create_crash_labels(df)

    # Train decision tree
    print("\nTraining decision tree...")
    dt_model, dt_scaler = train_decision_tree(features, labels)

    # Analyze decision paths
    print("\nAnalyzing decision paths...")
    analyze_decision_paths(dt_model, features, features.columns)

    # Train ensemble models
    print("\nTraining ensemble models...")
    ensemble_models, ensemble_scaler, test_data, results = train_ensemble_models(features, labels)

    # Evaluate models
    print("\nEvaluating models...")
    ensemble_pred, ensemble_proba = evaluate_models(results, test_data)

    # Create predictions dataframe
    X_test, y_test = test_data
    predictions_df = pd.DataFrame({
        'Actual': y_test,
        'Ensemble_Probability': ensemble_proba,
        'Ensemble_Prediction': ensemble_pred
    }, index=X_test.index)

    for name, result in results.items():
        predictions_df[f'{name}_Probability'] = result['probabilities']
        predictions_df[f'{name}_Prediction'] = result['predictions']

    # Visualize predictions
    print("\nVisualizing predictions...")
    visualize_predictions(features, predictions_df)

    # Save models and artifacts
    print("\nSaving models and artifacts...")
    joblib.dump(dt_model, 'decision_tree_model.pkl')
    joblib.dump(dt_scaler, 'dt_scaler.pkl')

    for name, model in ensemble_models.items():
        joblib.dump(model, f'{name.lower().replace(" ", "_")}_model.pkl')
    joblib.dump(ensemble_scaler, 'ensemble_scaler.pkl')
    joblib.dump(features.columns, 'feature_names.pkl')

    predictions_df.to_csv('predictions.csv')
    print("\nProcessing completed successfully!")

    return predictions_df, features, results

if __name__ == "__main__":
    predictions_df, features, results = main()
```

```
Loading data...
Loading data...
Processed data shape: (1147, 57)
Available columns: ['284.250', 'Asset_4', 'Asset_5', 'Asset_6', 'Asset_7', 'Asset_8', 'Asset_9', 'Asset_10', 'Asset_

Performing time series analysis...

Analyzing 284.250
ADF Statistic: -0.6202571563530737
p-value: 0.8663387303135373
```




```
Analyzing Asset_4
ADF Statistic: -6.519933595426635
p-value: 1.048326104484659e-08
```