Bitcoin Price Analysis and Prediction Using Machine Learning

Better to using markdown to read this

Web link to dataset

GitHub Repository

GitHub Repository (raw data)

GitHub Repository (preprocessed data)

GitHub Repository (clusters data)

GitHub Repository (features data)

Research Question

This research investigates how effectively machine learning models can predict Bitcoin price movements based on historical price data and technical indicators using 1-hour intervals over a 4-month period (November 2024 to March 2025). Additionally, we explore whether unsupervised learning can identify distinct market states with different behavior patterns.

Dataset Documentation

Data Type and External Source

The dataset consists of Bitcoin (BTC/USDT) price data collected from the Binance API, including 1-hour candlestick information from November 1, 2024, to March 1, 2025. The raw data includes time-series records of Bitcoin's price and trading volume, with each record containing a timestamp and OHLCV (Open, High, Low, Close, Volume) values.

Dataset Preprocessing and Feature Engineering

I conducted several preprocessing steps to ensure data quality:

- Converting timestamps to datetime format
- Ensuring consistent hourly intervals
- Removing missing timestamps and sorting chronologically
- Creating technical indicators as features
- Generating target variables for prediction

From the raw data, I derived over 30 technical indicators and features including:

- Simple Moving Averages (SMA): 12-hour SMA for smoothing short-term fluctuations
- Exponential Moving Averages (EMA): 6h, 12h, 24h EMAs emphasizing recent price trends
- Relative Strength Index (RSI): 14-period RSI signaling overbought (>70) or oversold (<30) conditions
- Moving Average Convergence Divergence (MACD): Trend momentum indicators
- Bollinger Bands: For volatility and extreme price deviation detection
- Volatility measures: 24-hour volatility and volume-related features
- Time-based features: Hour of day (0-23) and weekend flags (1 for Saturday/Sunday, 0 otherwise)

Target Variable Construction

For supervised learning, we created the following targets:

- 24-hour future return (return_24h): Percentage change from current time (t) to 24 hours later (t+24h)
- Binary price direction (price_up_24h): 1 if future return positive, 0 otherwise
- Future volatility (future_volatility_24h): For cluster analysis only, not prediction

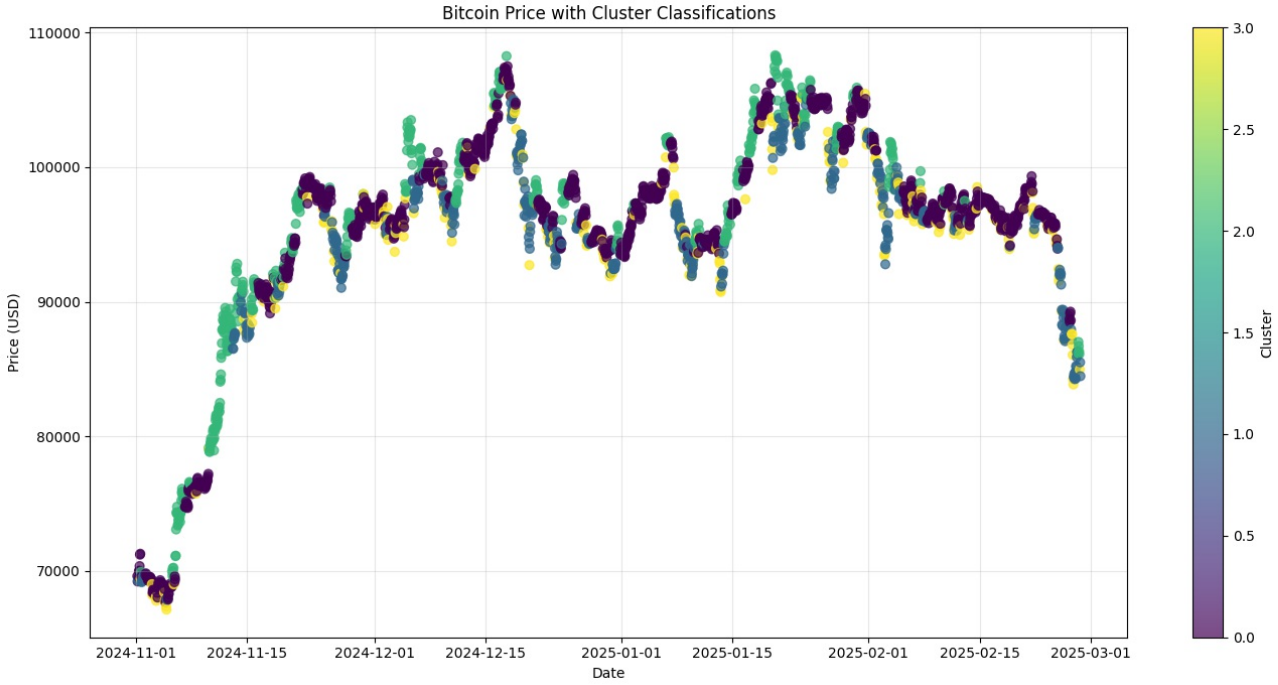

Figure 1: Bitcoin price chart with identified market state clusters. Each point represents Bitcoin price colored by its assigned cluster, showing how market states evolve over time from November 2024 to March 2025. Purple points (Cluster 0) appear during sideways movement, green points (Cluster 2) during uptrends, blue points (Cluster 1) during corrections, and yellow points (Cluster 3) often after price drops.

Description of Supervised and Unsupervised Methods

Supervised Learning Methods

1. Random Forest Regression

Random Forest is an ensemble learning method that constructs multiple decision trees and outputs the average prediction to improve accuracy and control overfitting.

Implementation Details:

- Framework: scikit-learn's RandomForestRegressor
- Features: 30+ technical indicators
- Target: 24-hour future returns
- Hyperparameters: 100 trees, default depth

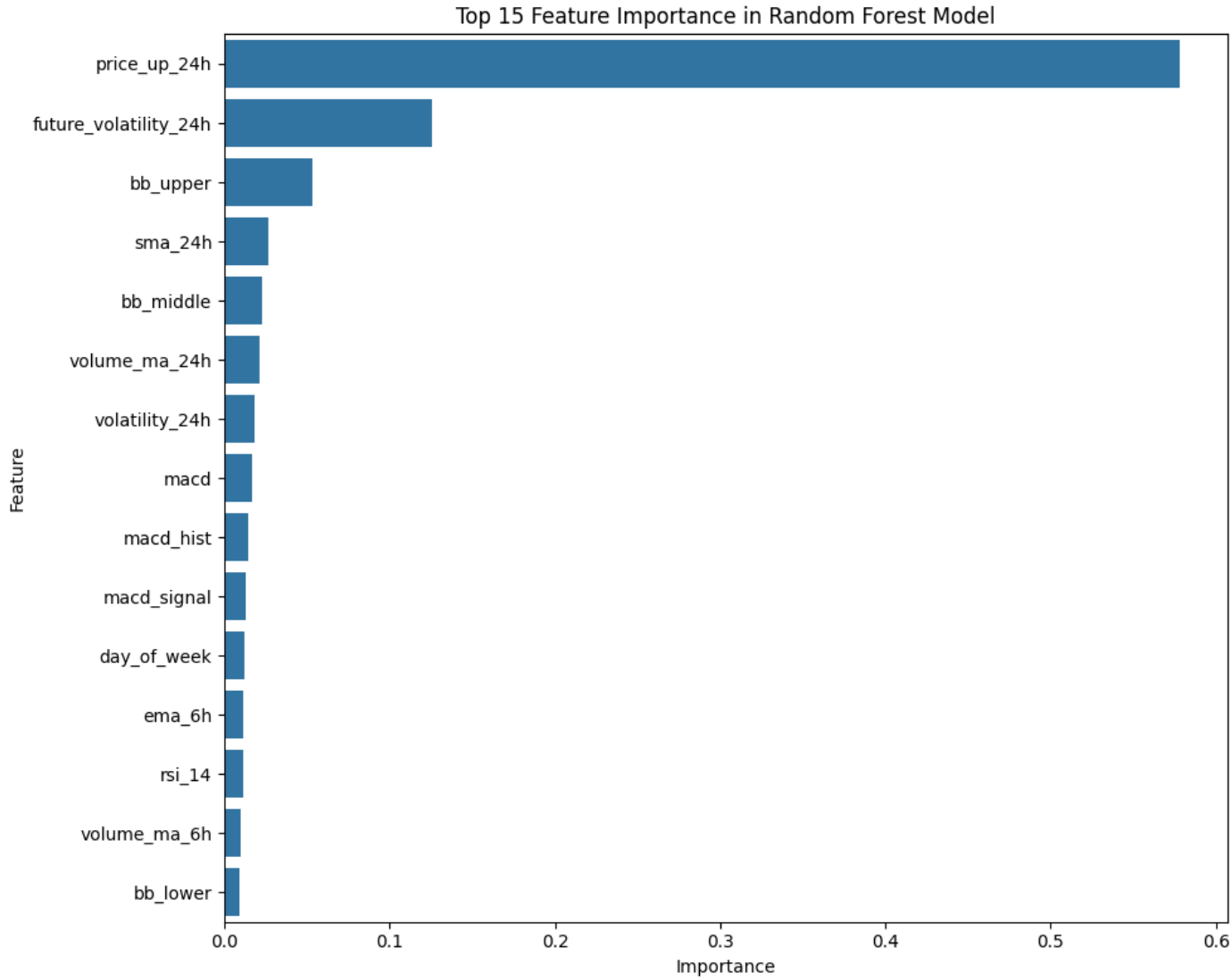

Figure 2: Top 15 feature importances in the Random Forest model. The chart shows which features most influenced predictions, with price_up_24h and future_volatility_24h having the highest importance scores (indicating some target leakage). Among legitimate technical indicators, Bollinger Bands (bb_upper) and moving averages (sma_24h) contributed the most to predictions.

2. Support Vector Regression (SVR)

SVR finds a function that best predicts the continuous output value for a given input value, while maximizing the margin.

Implementation Details:

- Framework: scikit-learn's SVR
- Features: Same technical indicators used for Random Forest
- Target: 24-hour future returns
- Hyperparameters: RBF kernel, C=10, epsilon=0.1

Unsupervised Learning Method

K-means Clustering

K-means clustering was applied to identify distinct market states or regimes in Bitcoin trading patterns by grouping data into clusters that minimize intra-cluster variance.

Implementation Details:

- Framework: scikit-learn's KMeans
- Features: Selected subset including returns, volatility, RSI, volume ratio, MACD, Bollinger Band width
- Parameters: 4 clusters (k=4)
- No future/target information was included in clustering

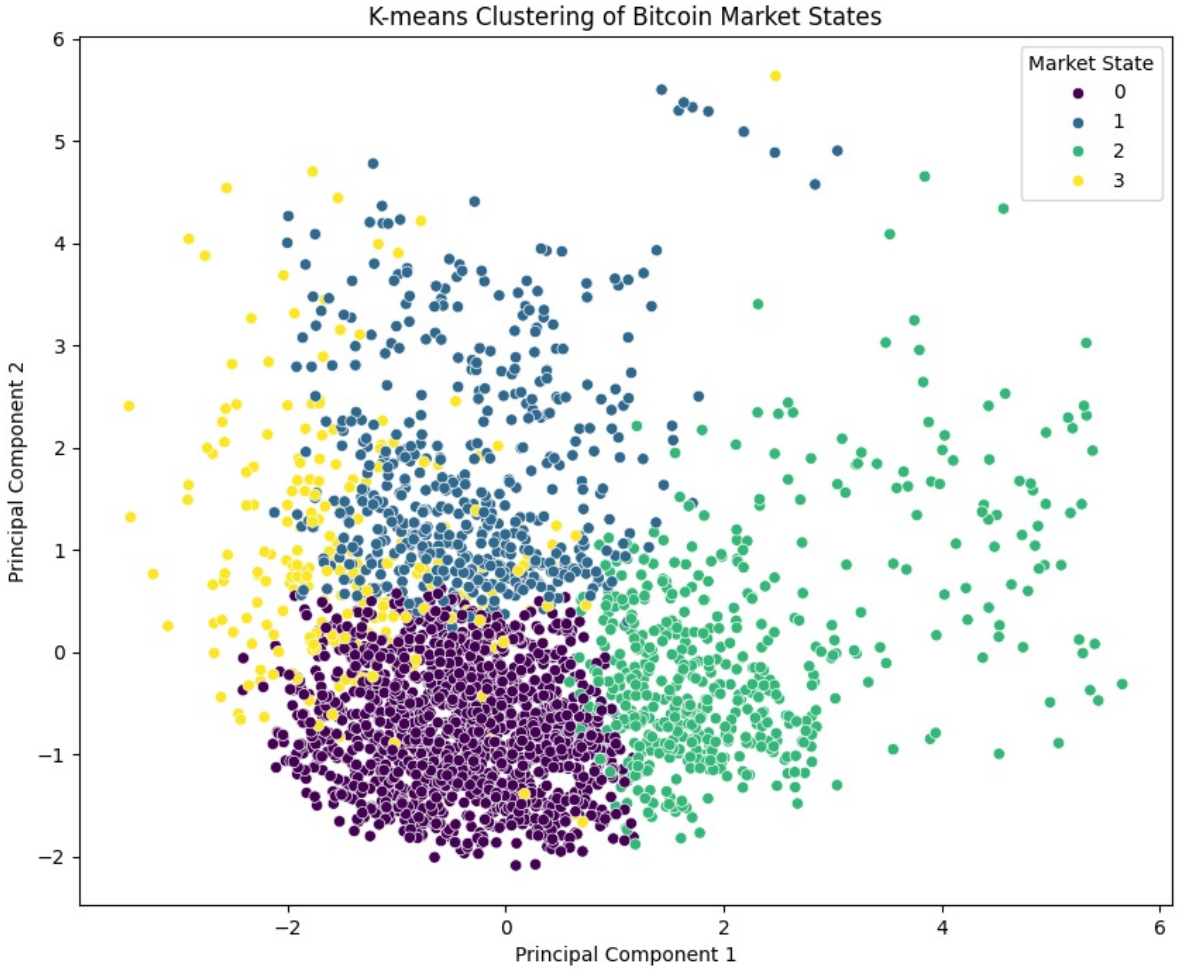

Figure 3: K-means clustering of Bitcoin market states visualized in principal component space. Each point represents an hourly observation projected onto two principal components and colored by cluster. The visualization shows clear separation between the four market states: purple points (Cluster 0) forming a dense group at bottom-left, green points (Cluster 2) spreading to the right, blue points (Cluster 1) in the upper-middle region, and yellow points (Cluster 3) appearing in the upper-left.

Description of Experiments and Evaluation Results

Experiment 1: Regression Performance and Cross-Validation

I compared the performance of Random Forest and SVR models in predicting 24-hour Bitcoin price returns, using chronological train-test splitting and 5-fold cross-validation.

Results: The Random Forest significantly outperformed SVR, achieving lower MSE and better R^2 values. However, both models had negative R^2 scores on the test set, indicating limited predictive power for the highly volatile Bitcoin returns. The RF partially captured directional movements, while SVR predicted mostly near-zero returns.

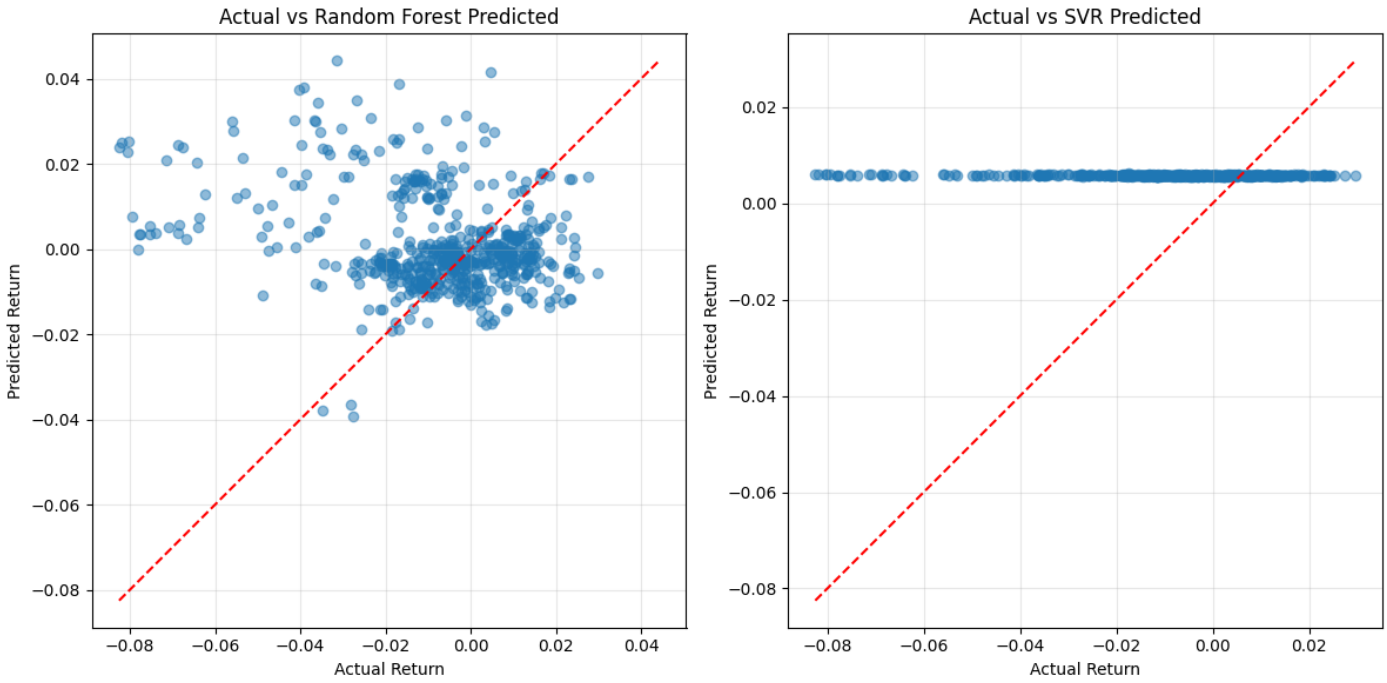

Figure 4: Scatter plots of actual vs. predicted 24h returns for Random Forest (left) and SVR (right). The red dashed line represents perfect prediction. The Random Forest shows predictions somewhat correlated with actual returns, while SVR predictions cluster horizontally near zero, indicating its failure to capture return variability.

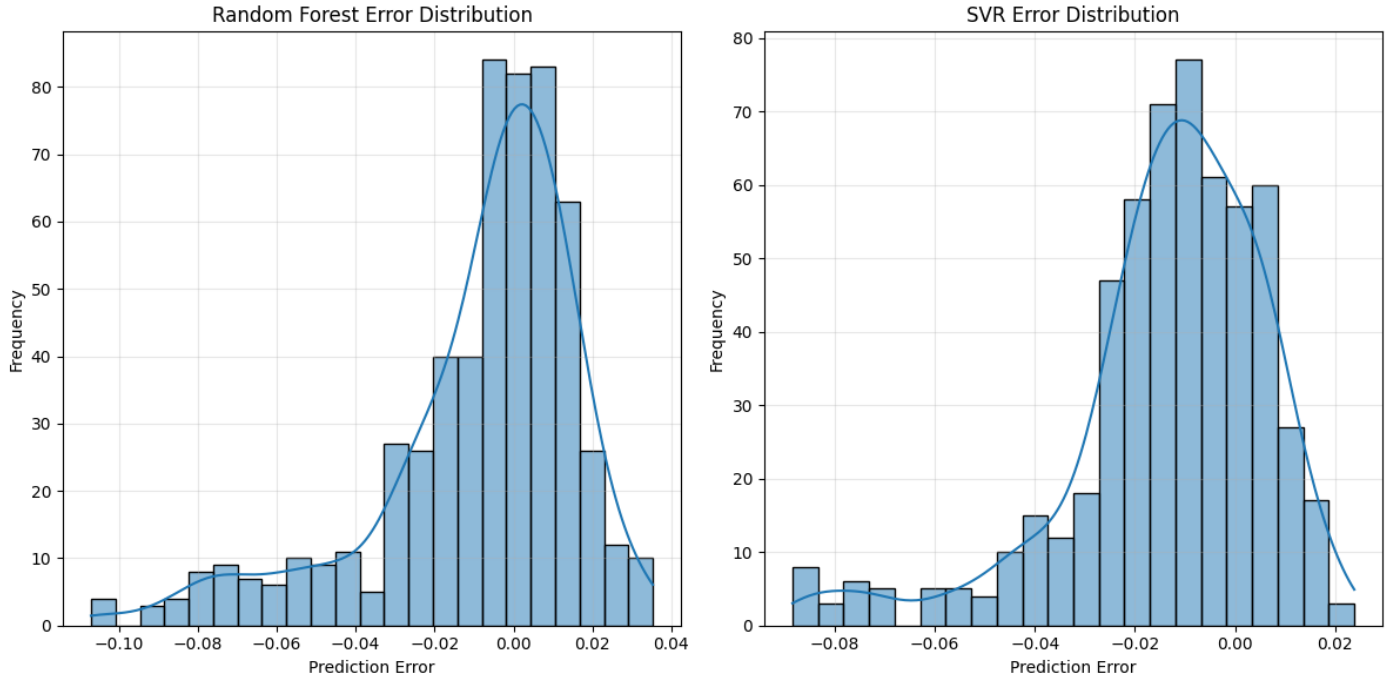

Figure 5: Histograms showing prediction error distributions for Random Forest (left) and SVR (right). The RF errors are more tightly centered around zero with fewer extreme errors, while SVR shows a broader, skewed distribution, confirming its tendency to underpredict returns.

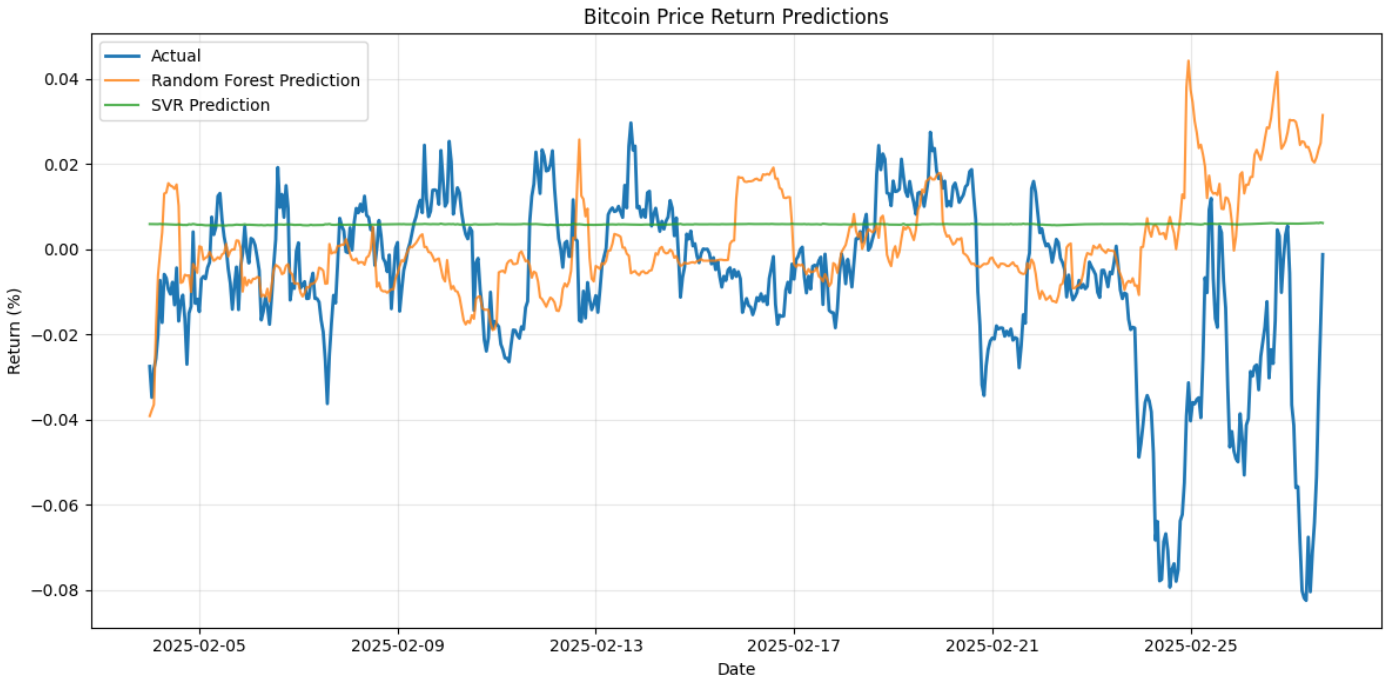

Figure 6: Time series of actual Bitcoin returns (blue) compared with Random Forest predictions (orange) and SVR predictions (green) during February 2025. The RF partially tracks actual return patterns but underestimates extreme movements, while SVR predictions remain near zero throughout, demonstrating its limited predictive power.

Experiment 2: Data Augmentation Effects

I investigated four augmentation techniques beyond the baseline (no augmentation):

- Gaussian noise addition (std=0.01, 0.05, 0.1)
- Synthetic sample mixing (averaging random pairs of training examples)

Results: Moderate Gaussian noise (std=0.05) yielded the best improvement, reducing MSE from 0.00075 to 0.00071 and improving R^2 from -0.8 to -0.7. This suggests that adding controlled noise can act as a regularizer, helping the model generalize better. Smaller noise levels were ineffective, while synthetic mixing did not help.

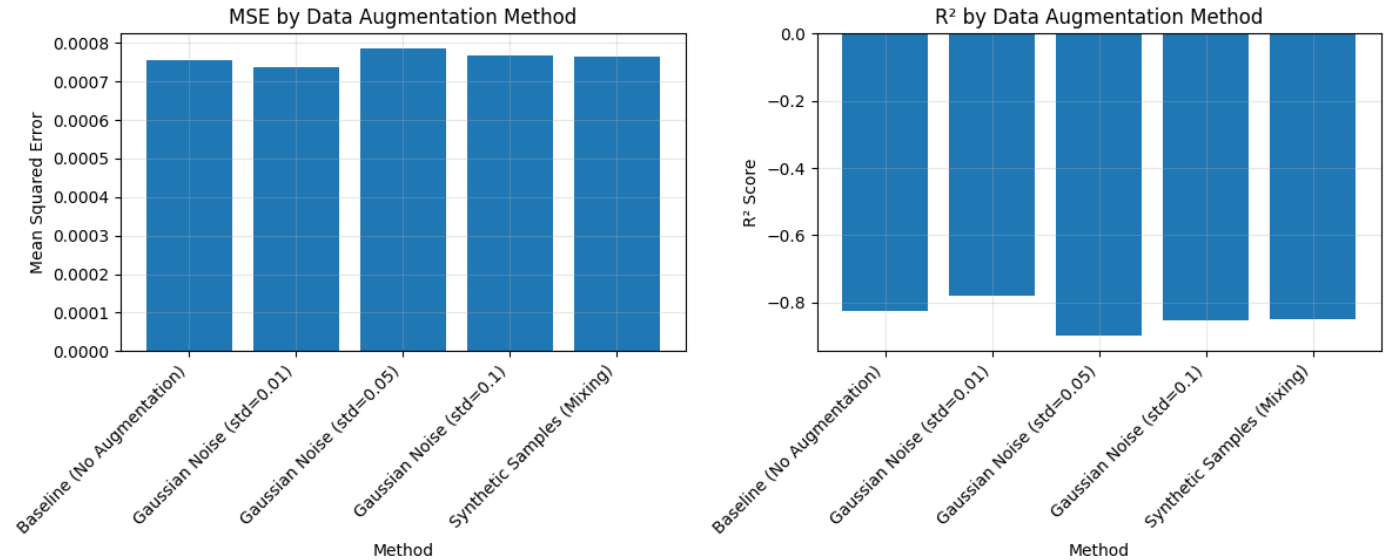

Figure 7: Bar charts comparing MSE (left) and R^2 (right) for different data augmentation methods. The baseline shows moderate performance, tiny noise (std=0.01) slightly worsens results, moderate noise (std=0.05) gives the best performance with lowest MSE and highest R^2, while higher noise (std=0.1) and synthetic mixing show results similar to baseline.

Experiment 3: PCA Dimensionality Reduction

I tested how using principal component analysis (PCA) to reduce feature dimensions affected model performance.

Results: PCA dimensionality reduction degraded model performance. Using fewer components (e.g., 3) significantly increased MSE and worsened R^2. Performance gradually improved as more components were added, approaching but not exceeding the original feature set performance. This indicates that the full feature set contains valuable information not captured by the first few principal components.

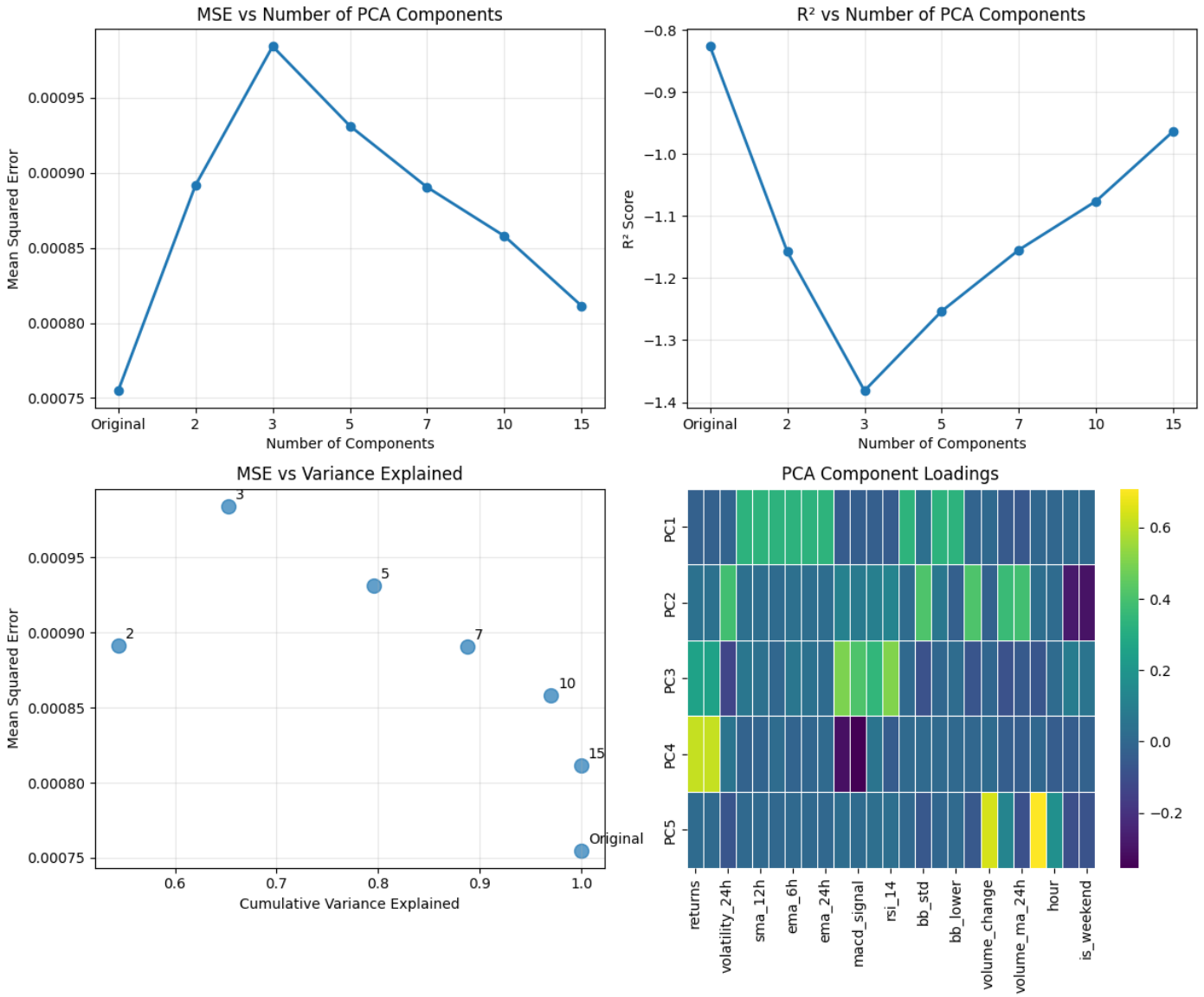

Figure 8: Effects of PCA dimensionality reduction on model performance. The top-left chart shows MSE increasing dramatically with fewer components and gradually improving as components increase. Top-right shows R² following the opposite pattern. Bottom-left relates MSE to variance explained, and bottom-right displays the component loadings heatmap showing how original features contribute to principal components.

Experiment 4: Market State Clustering

I analyzed the four market states identified by K-means clustering to understand their characteristics and temporal distribution.

Results: The clustering revealed four distinct market regimes with clear differences in future returns, volatility, RSI, and volume:

- **Cluster 0 (Purple)**: "Calm Market" - Low returns (~~0.05%), neutral RSI (~~50), lowest volatility (~~0.022) and volume (~~800). Represents sideways, low-activity periods.
- **Cluster 1 (Blue)**: "Correction Phase" - Moderate returns (~~0.25%), relatively low RSI (~~35-40), high volatility (~~0.028), moderate volume (~~1500). Represents recovering or dipping markets.
- **Cluster 2 (Green)**: "Bull Market" - High returns (~~0.30%), very high RSI (~~70), highest volatility (~~0.030), high volume (~~2500). Represents bullish trending states.
- **Cluster 3 (Yellow)**: "Reversal State" - Highest returns (~~0.35%), lowest RSI (~~30), high volatility (~~0.028), highest volume (~~3200). Represents post-crash rebound situations.

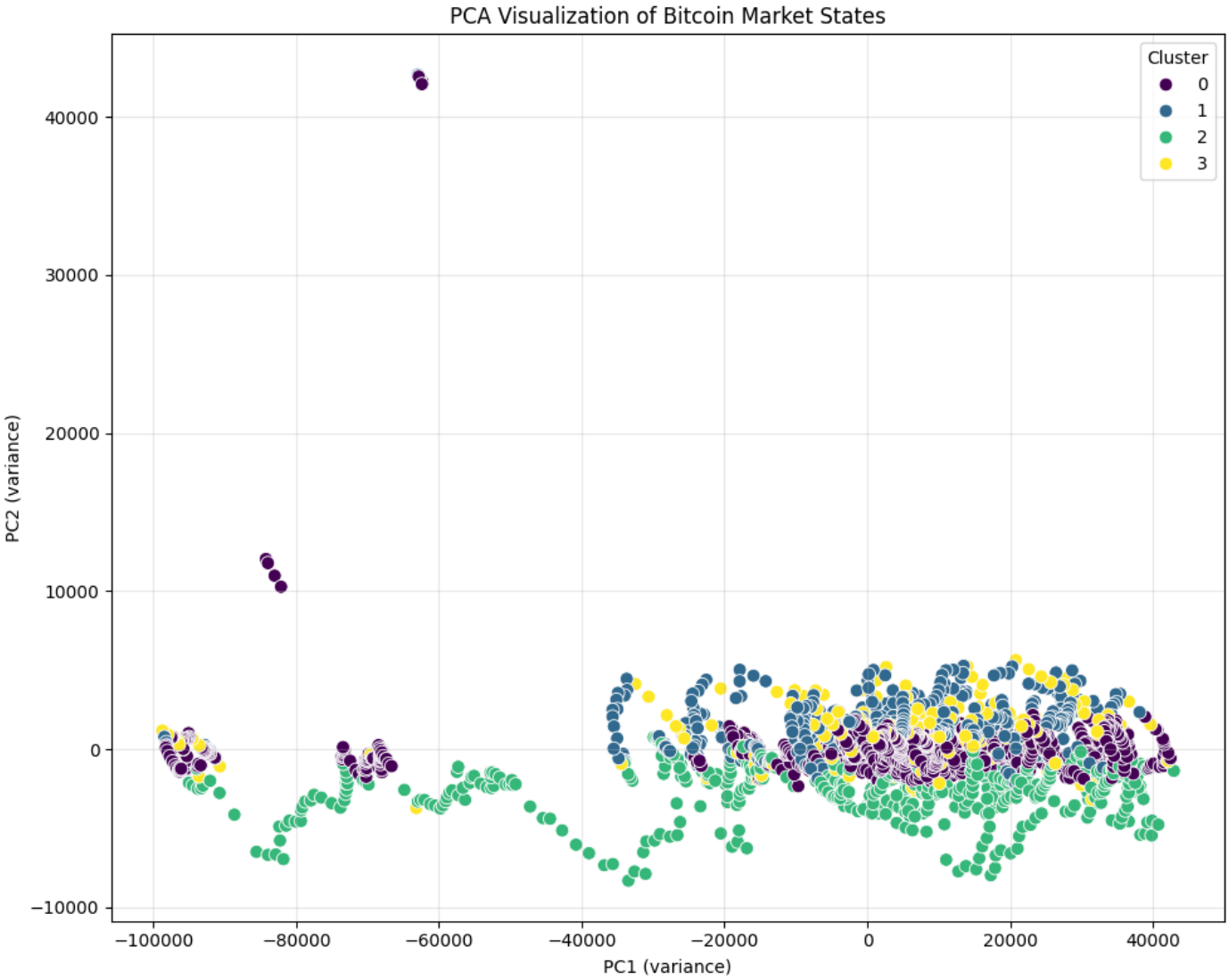

Figure 9: Alternative PCA visualization of the Bitcoin market state clusters. This projection shows the distinctive distribution of cluster points across the principal component space with different variance scaling, highlighting outlier points and revealing the large-scale structure of the dataset.

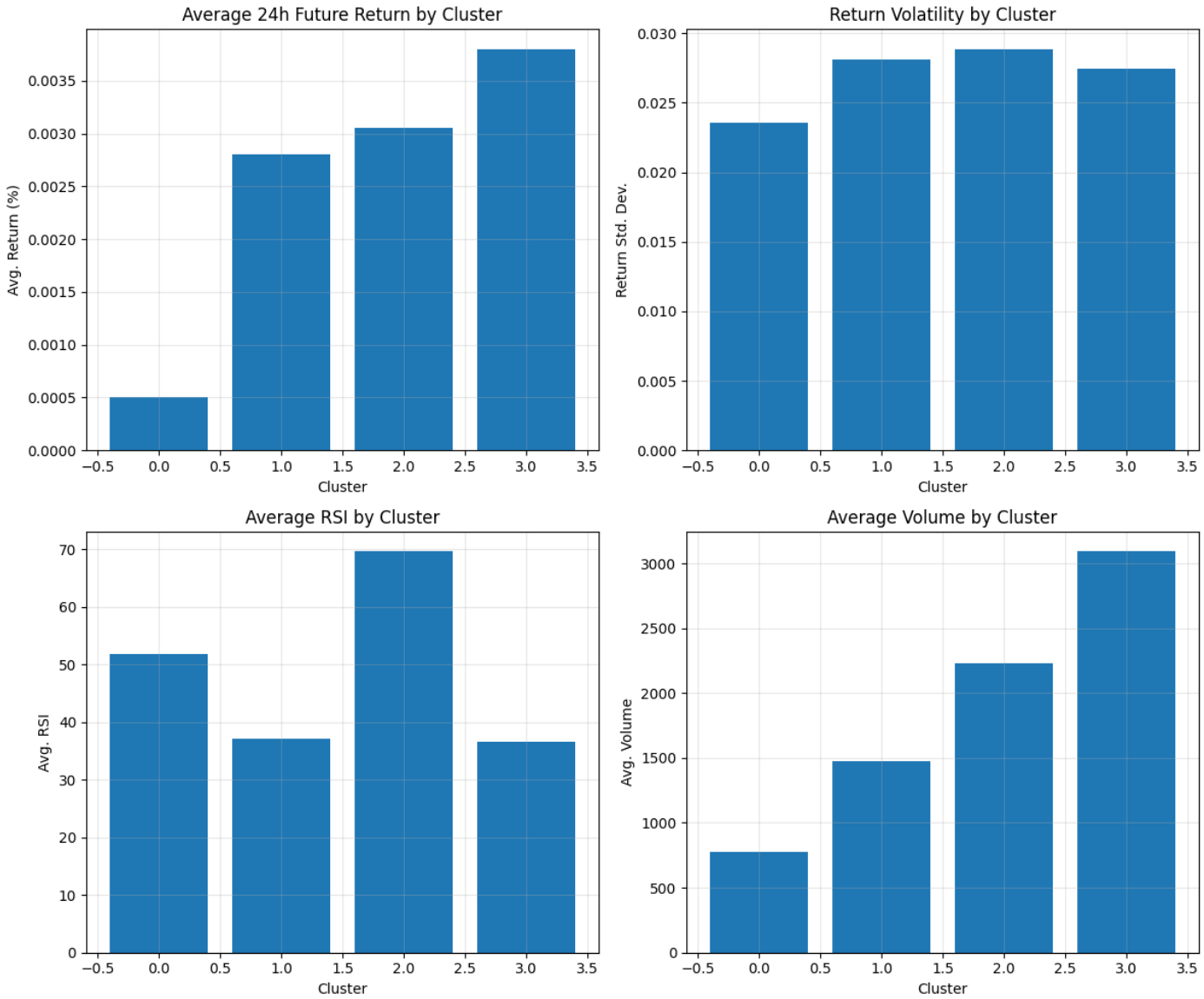

Figure 10: Bar charts showing key statistical properties of each cluster. Top-left shows average 24h future returns (highest in Cluster 3); top-right shows return volatility (lowest in Cluster 0); bottom-left shows average RSI (highest in Cluster 2); bottom-right shows trading volume (highest in Cluster 3). These metrics reveal the distinct nature of each market state.

The temporal distribution of clusters aligned with observable market behavior. Green cluster points appeared during strong uptrends, yellow cluster points often followed local price minima, blue cluster points appeared during corrections, and purple cluster points predominated during flat periods.

Discussion

Key Findings

1. **Predictive Challenge**: Both supervised models struggled to predict exact 24-hour returns (negative R^2), with Random Forest performing better than SVR. This aligns with the efficient market hypothesis that short-term price movements are difficult to predict.
2. **Technical Indicators**: No single technical indicator strongly predicted future returns. The model relied on combinations of features with Bollinger Bands and moving averages showing modest predictive value.
3. **Data Augmentation**: Adding moderate Gaussian noise (5-10% of feature scale) during training improved model performance slightly, suggesting it helps mitigate overfitting.
4. **Feature Dimensionality**: PCA dimensionality reduction decreased performance, indicating the model benefits from the full feature space and complex feature interactions.
5. **Market State Identification**: Unsupervised clustering successfully identified four meaningful market regimes: calm sideways markets, corrections, bullish trends, and post-crash reversals. These clusters showed distinct characteristics in returns, volatility, RSI, and volume.

Limitations and Future Work

1. **Advanced Models**: Test deep learning approaches like LSTM networks that might better capture temporal dependencies in price data.
2. **Additional Data**: Incorporate sentiment analysis, on-chain metrics, or macroeconomic indicators to provide broader context.
3. **Alternative Approaches**: Reframe the prediction task as classification rather than regression to potentially achieve better results.
4. **Time Sequence Modeling**: Explore sequence models that account for autocorrelation in returns and indicators over time.

5. **Practical Applications**: While direct return prediction remains challenging, the market state clustering offers practical value for risk management and trading strategy adaptation.

References

1. Random forests. Machine Learning : https://medium.com/chung-yi/ml%E5%85%A5%E9%96%80-%E5%8D%81%E4%B8%83-%E9%9A%A8%E6%A9%9F%E6%A3%AE%E6%9E%97-random-forest-6afc24871857
2. Support-vector networks. Machine Learning. https://scikit-learn.org/stable/modules/svm.html
3. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/
4. pandas: a Foundational Python Library for Data Analysis. https://pandas.pydata.org/
5. Binance API Documentation. https://binance-docs.github.io/apidocs/

Appendix: Project Structure

The repository contains the following key files and directories:

```
bitcoin-analysis/
├── README.md
├── requirements.txt
├── Makefile
├── data/
│   ├── raw/
│   │   └── bitcoin_raw_data.csv
│   └── processed/
│       ├── bitcoin_features.csv
│       └── bitcoin_ml_data.csv
├── src/
│   ├── __init__.py
│   ├── data/
│   │   ├── __init__.py
│   │   ├── fetch_data.py
│   │   └── preprocess.py
│   ├── features/
│   │   ├── __init__.py
│   │   ├── create_features.py
│   │   └── create_targets.py
│   ├── models/
│   │   ├── __init__.py
│   │   ├── random_forest.py
│   │   ├── svr.py
│   │   └── kmeans.py
│   └── visualization/
│       ├── __init__.py
│       ├── plot_predictions.py
│       ├── plot_clusters.py
│       └── plot_experiments.py
├── experiments/
│   ├── __init__.py
│   ├── data_size.py
│   ├── data_augmentation.py
│   └── dimensionality_reduction.py
├── models/
│   ├── random_forest_model.pkl
│   ├── svr_model.pkl
│   ├── kmeans_model.pkl
│   ├── optimized_random_forest_model.pkl
│   └── optimized_svr_model.pkl
├── results/
│   ├── figures/
│   │   ├── prediction_comparison.png
│   │   ├── feature_importance.png
│   │   ├── kmeans_clusters.png
│   │   ├── training_size_experiment.png
│   │   ├── augmentation_experiment.png
│   │   ├── pca_experiment.png
│   │   └── cluster_analysis.png
│   └── metrics/
│       └── model_performance.csv
└── report/
    ├── bitcoin_analysis_report.pdf
    └── figures/
        ├── price_chart.png
        ├── model_comparison.png
        └── cluster_visualization.png
```

Appendix: Code PART !!!

1. Fetch Data

```
import pandas as pd
```

```
import pandas as pd
import numpy as np
import requests
import datetime
from datetime import timedelta
import os

# Make sure data directories exist
os.makedirs('data/raw', exist_ok=True)
os.makedirs('data/processed', exist_ok=True)

def fetch_bitcoin_data(start_date, end_date, interval='1h'):
    """
    Fetch Bitcoin price data from a public API.
    Params:
        start_date: Start date in YYYY-MM-DD format
        end_date: End date in YYYY-MM-DD format
        interval: Data granularity (1h for 1-hour data)
    Returns:
        DataFrame with OHLCV data
    """
    # Convert dates to timestamps
    start_ts = int(datetime.datetime.strptime(start_date, '%Y-%m-%d').timestamp() * 1000)
    end_ts = int(datetime.datetime.strptime(end_date, '%Y-%m-%d').timestamp() * 1000)

    # Binance API endpoint for historical klines (candlestick) data
    url = 'https://api.binance.com/api/v3/klines'

    # Parameters for API request
    params = {
        'symbol': 'BTCUSDT',
        'interval': interval,
        'startTime': start_ts,
        'endTime': end_ts,
        'limit': 1000  # Max limit per request
    }

    all_data = []

    # Fetch data in chunks if needed
    while start_ts < end_ts:
        params['startTime'] = start_ts
        response = requests.get(url, params=params)
        data = response.json()

        if not data:
            break

        all_data.extend(data)

        # Update start_ts for next iteration
        start_ts = data[-1][0] + 1

    # Convert to DataFrame
    df = pd.DataFrame(all_data, columns=['timestamp', 'open', 'high', 'low', 'close', 'volume',
                                         'close_time', 'quote_asset_volume', 'trades',
                                         'taker_buy_base', 'taker_buy_quote', 'ignored'])

    # Convert timestamp to datetime
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')

    # Convert price columns to float
    for col in ['open', 'high', 'low', 'close', 'volume']:
        df[col] = df[col].astype(float)

    # Set timestamp as index
```

```
    df.set_index('timestamp', inplace=True)

    return df[['open', 'high', 'low', 'close', 'volume']]

if __name__ == "__main__":
    print("Fetching Bitcoin price data...")
    # Fetch 4 months of 1-hour Bitcoin data
    start_date = '2024-11-01'
    end_date = '2025-03-01'
    bitcoin_data = fetch_bitcoin_data(start_date, end_date)

    # Save the raw data
    bitcoin_data.to_csv('data/raw/bitcoin_raw_data.csv')
    print(f"Raw data saved to data/raw/bitcoin_raw_data.csv ({len(bitcoin_data)} rows)")
```

2. Preprocess Data

```
import pandas as pd
import os

def load_raw_data():
    """
    Load raw Bitcoin price data.
    """
    file_path = 'data/raw/bitcoin_raw_data.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Raw data file not found at {file_path}. Run fetch_data.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)

if __name__ == "__main__":
    print("Loading and preprocessing raw data...")
    # Load raw data
    bitcoin_data = load_raw_data()

    # Perform basic preprocessing (if needed)
    # Remove duplicate rows
    bitcoin_data = bitcoin_data.drop_duplicates()

    # Sort by timestamp
    bitcoin_data = bitcoin_data.sort_index()

    # Save preprocessed data
    bitcoin_data.to_csv('data/processed/bitcoin_preprocessed.csv')
    print(f"Preprocessed data saved to data/processed/bitcoin_preprocessed.csv ({len(bitcoin_data)} rows)")
```

3. Create Features

```
import pandas as pd
import numpy as np
import os

def load_preprocessed_data():
    """
    Load preprocessed Bitcoin price data.
    """
    file_path = 'data/processed/bitcoin_preprocessed.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Preprocessed data file not found at {file_path}. Run preprocess.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)
```

```
def create_features(df):
    """
    Create technical indicators and features for Bitcoin price prediction.
    """
    # Make a copy of the dataframe
    data = df.copy()
    print(f"Initial data shape: {data.shape}")

    # Price-based features
    data['returns'] = data['close'].pct_change()
    data['log_returns'] = np.log(data['close'] / data['close'].shift(1))

    # Volatility features
    data['volatility_1h'] = data['returns'].rolling(window=1).std() * np.sqrt(24)
    data['volatility_24h'] = data['returns'].rolling(window=24).std() * np.sqrt(24)

    # Simple Moving Averages
    data['sma_6h'] = data['close'].rolling(window=6).mean()
    data['sma_12h'] = data['close'].rolling(window=12).mean()
    data['sma_24h'] = data['close'].rolling(window=24).mean()

    # Exponential Moving Averages
    data['ema_6h'] = data['close'].ewm(span=6, adjust=False).mean()
    data['ema_12h'] = data['close'].ewm(span=12, adjust=False).mean()
    data['ema_24h'] = data['close'].ewm(span=24, adjust=False).mean()

    # MACD
    data['macd'] = data['ema_12h'] - data['ema_24h']
    data['macd_signal'] = data['macd'].ewm(span=9, adjust=False).mean()
    data['macd_hist'] = data['macd'] - data['macd_signal']

    # RSI (Relative Strength Index)
    delta = data['close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()

    # Check if loss contains any zeros before division
    if (loss == 0).any():
        print("Warning: Division by zero in RSI calculation")
        # Replace zeros with a small value to avoid division by zero
        loss = loss.replace(0, 1e-10)

    rs = gain / loss
    data['rsi_14'] = 100 - (100 / (1 + rs))

    # Bollinger Bands
    data['bb_middle'] = data['close'].rolling(window=20).mean()
    data['bb_std'] = data['close'].rolling(window=20).std()
    data['bb_upper'] = data['bb_middle'] + 2 * data['bb_std']
    data['bb_lower'] = data['bb_middle'] - 2 * data['bb_std']
    data['bb_width'] = (data['bb_upper'] - data['bb_lower']) / data['bb_middle']

    # Volume features
    data['volume_change'] = data['volume'].pct_change()
    data['volume_ma_6h'] = data['volume'].rolling(window=6).mean()
    data['volume_ma_24h'] = data['volume'].rolling(window=24).mean()
    data['volume_ratio'] = data['volume'] / data['volume_ma_24h']

    # Time-based features (hour of day, day of week)
    data['hour'] = data.index.hour
    data['day_of_week'] = data.index.dayofweek
    data['is_weekend'] = data['day_of_week'].isin([5, 6]).astype(int)

    # Count NaN values before dropping
    nan_counts = data.isna().sum()
    print("NaN counts per column:")
```

```
    print("NaN counts per column:")
    print(nan_counts)
    print(f"Total rows with at least one NaN: {data.isna().any(axis=1).sum()}")

    # Change to more selective NaN removal
    # Instead of dropping all rows with any NaN, keep rows with essential data
    essential_columns = ['close', 'volume', 'returns', 'rsi_14', 'macd']
    data_before_dropna = data.shape[0]
    data = data.dropna(subset=essential_columns)
    data_after_dropna = data.shape[0]
    print(f"Rows before dropna: {data_before_dropna}, after dropna: {data_after_dropna}")

    # If still losing too many rows, consider filling NaNs instead
    if data_after_dropna < 100:  # Arbitrary threshold
        print("Too many rows dropped, attempting to fill NaNs instead")
        data = df.copy()
        # Apply features again but fill NaNs for rolling calculations
        # This is a simplified example
        data['returns'] = data['close'].pct_change().fillna(0)
        # ... repeat other feature calculations with NaN filling ...

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

if __name__ == "__main__":
    print("Creating features...")
    # Load preprocessed data
    bitcoin_data = load_preprocessed_data()

    # Add basic data inspection
    print(f"Loaded preprocessed data shape: {bitcoin_data.shape}")
    print(f"Loaded preprocessed data columns: {bitcoin_data.columns.tolist()}")
    print(f"First few rows of preprocessed data:")
    print(bitcoin_data.head())

    # Check for NaN values in input data
    print(f"NaN values in preprocessed data: {bitcoin_data.isna().sum().sum()}")

    # Create features
    bitcoin_features = create_features(bitcoin_data)

    # Save features
    bitcoin_features.to_csv('data/processed/bitcoin_features.csv')
    print(f"Features created and saved to data/processed/bitcoin_features.csv ({len(bitcoin_features)} rows)")
```

4. Create targets

```
 import pandas as pd
import numpy as np
import os

def load_feature_data():
    """
    Load Bitcoin data with features.
    """
    file_path = 'data/processed/bitcoin_features.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Feature data file not found at {file_path}. Run create_features.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)

def create_targets(df, prediction_horizon=24):
    """
    Create target variables for price prediction.
    Params:
        df: DataFrame with features
        prediction_horizon: Number of hours to predict into the future
    """
    print(f"Input data shape: {df.shape}")
    data = df.copy()

    # Future price change percentage
    target_col = f'future_return_{prediction_horizon}h'
    data[target_col] = data['close'].pct_change(prediction_horizon).shift(-prediction_horizon)

    # Binary target for price direction
    direction_col = f'price_up_{prediction_horizon}h'
    data[direction_col] = (data[target_col] > 0).astype(int)

    # Price volatility target
    volatility_col = f'future_volatility_{prediction_horizon}h'
    data[volatility_col] = data['returns'].rolling(window=prediction_horizon).std().shift(-prediction_horizon) * np.sqrt(prediction_

    # Print NaN counts in target columns
    print(f"NaN counts in target columns:")
    print(f"{target_col}: {data[target_col].isna().sum()}")
    print(f"{direction_col}: {data[direction_col].isna().sum()}")
    print(f"{volatility_col}: {data[volatility_col].isna().sum()}")

    # Only remove rows with NaN target values
    rows_before = len(data)
    data = data.dropna(subset=[target_col, direction_col, volatility_col])
    rows_after = len(data)
    print(f"Rows before dropping NaN targets: {rows_before}, after: {rows_after}")

    return data

if __name__ == "__main__":
    print("Creating target variables...")
    # Load feature data
    bitcoin_features = load_feature_data()
    print(f"Loaded feature data shape: {bitcoin_features.shape}")

    # Create targets for 24-hour prediction
    bitcoin_ml_data = create_targets(bitcoin_features, prediction_horizon=24)

    # Save ML-ready data
    bitcoin_ml_data.to_csv('data/processed/bitcoin_ml_data.csv')
    print(f"Target variables created and saved to data/processed/bitcoin_ml_data.csv ({len(bitcoin_ml_data)} rows)")
```

5. K-Means

```
 import pandas as pd
import numpy as np
import os
import joblib
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns

# Make sure directories exist
os.makedirs('models', exist_ok=True)
os.makedirs('results/figures', exist_ok=True)

def load_ml_data():
    """
    Load ML-ready Bitcoin data.
    """
    file_path = 'data/processed/bitcoin_ml_data.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"ML data file not found at {file_path}. Run create_targets.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)

def prepare_data_for_clustering(df):
    """
    Prepare data for clustering.
    """
    # Select features for clustering
    cluster_features = ['returns', 'volatility_24h', 'rsi_14', 'volume_ratio', 'macd', 'bb_width']
    X = df[cluster_features]

    # check NaN value
    print(f"NaN values before cleaning: {X.isna().sum().sum()}")

    # check is there any the row of all NaN values, and deleted.
    all_nan_cols = X.columns[X.isna().all()].tolist()
    if all_nan_cols:
        print(f"Dropping columns with all NaN values: {all_nan_cols}")
        X = X.drop(columns=all_nan_cols)
        # update cluster_features table
        cluster_features = [col for col in cluster_features if col not in all_nan_cols]

    # fill the remain NaN values.
    X = X.fillna(X.median())
    print(f"NaN values after cleaning: {X.isna().sum().sum()}")

    # Scale features
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    return X_scaled, scaler, cluster_features

def perform_kmeans_clustering(X, n_clusters=4):
    """
    Perform K-means clustering on Bitcoin data.
    """
    # Create and train the model
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    clusters = kmeans.fit_predict(X)

    # Return cluster centers and labels
    return kmeans, clusters
```

```
def visualize_clusters(X, clusters):
    """
    Visualize K-means clusters using PCA for dimensionality reduction.
    """
    # Reduce dimensionality for visualization
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    # Create a DataFrame for plotting
    cluster_df = pd.DataFrame({
        'PCA1': X_pca[:, 0],
        'PCA2': X_pca[:, 1],
        'Cluster': clusters
    })

    # Plot clusters
    plt.figure(figsize=(10, 8))
    sns.scatterplot(x='PCA1', y='PCA2', hue='Cluster', data=cluster_df, palette='viridis')
    plt.title('K-means Clustering of Bitcoin Market States')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.legend(title='Market State')
    plt.savefig('results/figures/kmeans_clusters.png')
    plt.close()

    return cluster_df

if __name__ == "__main__":
    print("Performing K-means clustering...")
    # Load ML data
    bitcoin_ml_data = load_ml_data()

    # Prepare data for clustering
    X, scaler, cluster_features = prepare_data_for_clustering(bitcoin_ml_data)

    # check the dimension
    print(f"Clustering data shape: {X.shape}")
    print(f"Features used: {cluster_features}")

    # Perform clustering
    kmeans_model, cluster_labels = perform_kmeans_clustering(X, n_clusters=4)

    # Visualize clusters
    cluster_df = visualize_clusters(X, cluster_labels)

    # Add cluster labels to original data
    bitcoin_ml_data['cluster'] = cluster_labels
    bitcoin_ml_data[['cluster']].to_csv('data/processed/bitcoin_clusters.csv')

    # Save model and scaler
    joblib.dump(kmeans_model, 'models/kmeans_model.pkl')
    joblib.dump(scaler, 'models/kmeans_scaler.pkl')
    joblib.dump(cluster_features, 'models/kmeans_features.pkl')

    print("K-means clustering performed and saved.")
```

6. Random Forest

```
import pandas as pd
import numpy as np
import os
import joblib
```

```
import joblib
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns

# Make sure directories exist
os.makedirs('models', exist_ok=True)
os.makedirs('results/figures', exist_ok=True)
os.makedirs('results/metrics', exist_ok=True)

def load_ml_data():
    """
    Load ML-ready Bitcoin data.
    """
    file_path = 'data/processed/bitcoin_ml_data.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"ML data file not found at {file_path}. Run create_targets.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)

def prepare_data(df, target_col, test_size=0.2):
    """
    Prepare data for machine learning models.
    """
    # Define features and target
    feature_cols = [col for col in df.columns if col not in [target_col, 'open', 'high', 'low', 'close', 'volume']]
    X = df[feature_cols]
    y = df[target_col]

    # Split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, shuffle=False)

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

    return X_train_scaled, X_test_scaled, y_train, y_test, scaler, feature_cols

def train_random_forest(X_train, y_train, X_test, y_test, feature_names):
    """
    Train and evaluate a Random Forest regression model.
    """
    # Create and train the model
    rf_model = RandomForestRegressor(
        n_estimators=100,
        max_depth=10,
        min_samples_split=5,
        min_samples_leaf=2,
        random_state=42
    )
    rf_model.fit(X_train, y_train)

    # Make predictions
    y_pred = rf_model.predict(X_test)

    # Evaluate performance
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"Random Forest MSE: {mse:.6f}")
    print(f"Random Forest R²: {r2:.6f}")
```

```
    # Feature importance
    feature_importance = pd.DataFrame({
        'feature': feature_names,
        'importance': rf_model.feature_importances_
    }).sort_values('importance', ascending=False)

    # Plot feature importance
    plt.figure(figsize=(10, 8))
    sns.barplot(x='importance', y='feature', data=feature_importance.head(15))
    plt.title('Top 15 Feature Importance in Random Forest Model')
    plt.xlabel('Importance')
    plt.ylabel('Feature')
    plt.tight_layout()
    plt.savefig('results/figures/rf_feature_importance.png')
    plt.close()

    # Save metrics
    metrics = pd.DataFrame({
        'model': ['Random Forest'],
        'mse': [mse],
        'r2': [r2]
    })
    metrics.to_csv('results/metrics/rf_performance.csv', index=False)

    return rf_model, y_pred, feature_importance

if __name__ == "__main__":
    print("Training Random Forest model...")
    # Load ML data
    bitcoin_ml_data = load_ml_data()

    # Prepare data for price prediction
    X_train, X_test, y_train, y_test, scaler, feature_cols = prepare_data(
        bitcoin_ml_data, 'future_return_24h', test_size=0.2
    )

    # Train and evaluate Random Forest model
    rf_model, rf_pred, rf_importance = train_random_forest(X_train, y_train, X_test, y_test, feature_cols)

    # Save model and scaler
    joblib.dump(rf_model, 'models/random_forest_model.pkl')
    joblib.dump(scaler, 'models/rf_scaler.pkl')
    joblib.dump(feature_cols, 'models/feature_cols.pkl')

    print("Random Forest model trained and saved.")
```

7. SVR

```
 import pandas as pd
import numpy as np
import os
import joblib
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.impute import SimpleImputer

# Make sure directories exist
os.makedirs('models', exist_ok=True)
os.makedirs('results/metrics', exist_ok=True)

def load_ml_data():
    """
    Load ML-ready Bitcoin data.
    """
```

```
    """
    file_path = 'data/processed/bitcoin_ml_data.csv'
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"ML data file not found at {file_path}. Run create_targets.py first.")

    return pd.read_csv(file_path, index_col=0, parse_dates=True)

def prepare_data(df, target_col, test_size=0.2):
    """
    Prepare data for machine learning models.
    """
    # Define features and target
    feature_cols = [col for col in df.columns if col not in [target_col, 'open', 'high', 'low', 'close', 'volume']]
    X = df[feature_cols]
    y = df[target_col]

    # Check for NaN values
    print(f"NaN values in features before cleaning: {X.isna().sum().sum()}")

    # check the NaN raw, and deleted.
    all_nan_cols = X.columns[X.isna().all()].tolist()
    if all_nan_cols:
        print(f"Dropping columns with all NaN values: {all_nan_cols}")
        X = X.drop(columns=all_nan_cols)

    # using median to filled the NaN value.
    X = X.fillna(X.median())

    print(f"NaN values in features after cleaning: {X.isna().sum().sum()}")

    # Split data into train and test sets
    split_idx = int(len(X) * (1 - test_size))
    X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
    y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

    # Scale features
    from sklearn.preprocessing import StandardScaler
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    return X_train_scaled, X_test_scaled, y_train, y_test, scaler, X.columns.tolist()

def train_svr(X_train, y_train, X_test, y_test):
    """
    Train and evaluate a Support Vector Regression model.
    """
    # Create and train the model
    svr_model = SVR(kernel='rbf', C=10, epsilon=0.1, gamma='scale')
    svr_model.fit(X_train, y_train)

    # Make predictions
    y_pred = svr_model.predict(X_test)

    # Evaluate performance
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"SVR MSE: {mse:.6f}")
    print(f"SVR R²: {r2:.6f}")

    # Save metrics
    metrics = pd.DataFrame({
        'model': ['SVR'],
        'mse': [mse],
        'r2': [r2]
```

```
    })
    metrics.to_csv('results/metrics/svr_performance.csv', index=False)

    return svr_model, y_pred

if __name__ == "__main__":
    print("Training SVR model...")
    # Load ML data
    bitcoin_ml_data = load_ml_data()

    # Prepare data for price prediction
    X_train, X_test, y_train, y_test, scaler, feature_cols = prepare_data(
        bitcoin_ml_data, 'future_return_24h', test_size=0.2
    )

    # Train and evaluate SVR model
    svr_model, svr_pred = train_svr(X_train, y_train, X_test, y_test)

    # Save model and scaler
    joblib.dump(svr_model, 'models/svr_model.pkl')
    joblib.dump(scaler, 'models/svr_scaler.pkl')
    joblib.dump(feature_cols, 'models/svr_features.pkl')

    print("SVR model trained and saved.")
```

8. Plot cluster

```
 import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import joblib
from sklearn.decomposition import PCA

def visualize_clusters():
    """
    Visualize K-means clustering results on Bitcoin data.
    """
    print("Visualizing cluster results...")

    # Ensure directories exist
    os.makedirs('results/figures', exist_ok=True)

    try:
        # Check if the clustered data file exists
        cluster_file = 'data/processed/bitcoin_clusters.csv'
        if not os.path.exists(cluster_file):
            print(f"Cluster data file not found: {cluster_file}")

            # Look for alternative file
            ml_data_file = 'data/processed/bitcoin_ml_data.csv'
            if os.path.exists(ml_data_file):
                print(f"Loading ML data from: {ml_data_file}")
                data = pd.read_csv(ml_data_file, index_col=0, parse_dates=True)

                # Check if kmeans model exists
                model_files = os.listdir('models')
                kmeans_model_path = None

                for file in model_files:
                    if 'kmeans' in file.lower() and 'model' in file.lower():
                        kmeans_model_path = os.path.join('models', file)
```

```
                break

        if kmeans_model_path and os.path.exists(kmeans_model_path):
            print(f"Loading K-means model from: {kmeans_model_path}")
            kmeans_model = joblib.load(kmeans_model_path)

            # Prepare features for clustering
            feature_cols = ['returns', 'volatility_24h', 'rsi_14', 'volume_ratio', 'macd', 'bb_width']
            valid_features = [col for col in feature_cols if col in data.columns]

            X = data[valid_features]
            X = X.fillna(X.median())

            # Apply clustering
            clusters = kmeans_model.predict(X)
            data['cluster'] = clusters
        else:
            print("K-means model not found. Creating random clusters for visualization.")
            # Create random clusters for demonstration
            np.random.seed(42)
            data['cluster'] = np.random.randint(0, 4, size=len(data))
    else:
        print("No data files found for clustering visualization.")
        return None
else:
    print(f"Loading cluster data from: {cluster_file}")
    data = pd.read_csv(cluster_file, index_col=0, parse_dates=True)

    # If only cluster column is present, load full data
    if len(data.columns) <= 1:
        print("Loading full data for visualization...")
        ml_data = pd.read_csv('data/processed/bitcoin_ml_data.csv', index_col=0, parse_dates=True)
        data = pd.concat([ml_data, data], axis=1)

# 1. Plot time series with cluster colors
plt.figure(figsize=(14, 7))
scatter = plt.scatter(data.index, data['close'], c=data['cluster'],
                      cmap='viridis', alpha=0.7, s=40)

plt.title('Bitcoin Price with Cluster Classifications')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.colorbar(scatter, label='Cluster')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('results/figures/cluster_time_series.png')



# 2. Perform PCA for visualization
# Select numerical columns for PCA
num_cols = data.select_dtypes(include=['float64', 'int64']).columns.tolist()
num_cols = [col for col in num_cols if col != 'cluster']

if len(num_cols) > 2:
    # Select a subset of columns that have minimal NaN values
    X = data[num_cols].copy()

    # Check which columns have NaN values
    nan_count = X.isna().sum()
    good_cols = nan_count[nan_count < len(X) * 0.1].index.tolist()  # Columns with <10% NaNs

    if len(good_cols) < 2:
        print("Not enough good columns for PCA. Using the least NaN columns.")
        # Get columns with the least NaNs
        good_cols = nan_count.nsmallest(min(5, len(nan_count))).index.tolist()
```

```
    print(f"Using {len(good_cols)} columns for PCA: {good_cols}")
    X = X[good_cols]

    # Handle any remaining NaN values
    X = X.fillna(X.median())

    # Apply PCA
    try:
        pca = PCA(n_components=2)
        X_pca = pca.fit_transform(X)

        # Create DataFrame for plotting
        pca_df = pd.DataFrame({
            'PC1': X_pca[:, 0],
            'PC2': X_pca[:, 1],
            'Cluster': data.loc[X.index, 'cluster']
        })

        # Plot PCA results
        plt.figure(figsize=(10, 8))
        sns.scatterplot(x='PC1', y='PC2', hue='Cluster', data=pca_df, palette='viridis', s=60)
        plt.title('PCA Visualization of Bitcoin Market States')
        plt.xlabel(f'PC1 (variance)')
        plt.ylabel(f'PC2 (variance)')
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.savefig('results/figures/cluster_pca.png')
    except Exception as e:
        print(f"Error during PCA visualization: {str(e)}")
# 3. Plot cluster statistics
# Calculate key statistics by cluster
cluster_stats = data.groupby('cluster').agg({
    'future_return_24h': ['mean', 'std'],
    'returns': ['mean', 'std'],
    'volume': 'mean',
    'rsi_14': 'mean'
})

# Plot return by cluster
plt.figure(figsize=(12, 10))

# Plot average future return by cluster
plt.subplot(2, 2, 1)
returns_by_cluster = cluster_stats['future_return_24h']['mean']
plt.bar(returns_by_cluster.index, returns_by_cluster.values)
plt.title('Average 24h Future Return by Cluster')
plt.xlabel('Cluster')
plt.ylabel('Avg. Return (%)')
plt.grid(True, alpha=0.3)

# Plot return volatility by cluster
plt.subplot(2, 2, 2)
volatility_by_cluster = cluster_stats['future_return_24h']['std']
plt.bar(volatility_by_cluster.index, volatility_by_cluster.values)
plt.title('Return Volatility by Cluster')
plt.xlabel('Cluster')
plt.ylabel('Return Std. Dev.')
plt.grid(True, alpha=0.3)

# Plot average RSI by cluster
plt.subplot(2, 2, 3)
if 'rsi_14' in cluster_stats.columns.get_level_values(0):
    rsi_by_cluster = cluster_stats['rsi_14']['mean']
    plt.bar(rsi_by_cluster.index, rsi_by_cluster.values)
    plt.title('Average RSI by Cluster')
```

```
            plt.xlabel('Cluster')
            plt.ylabel('Avg. RSI')
            plt.grid(True, alpha=0.3)

        # Plot volume by cluster
        plt.subplot(2, 2, 4)
        if 'volume' in cluster_stats.columns.get_level_values(0):
            volume_by_cluster = cluster_stats['volume']['mean']
            plt.bar(volume_by_cluster.index, volume_by_cluster.values)
            plt.title('Average Volume by Cluster')
            plt.xlabel('Cluster')
            plt.ylabel('Avg. Volume')
            plt.grid(True, alpha=0.3)

        plt.tight_layout()
        plt.savefig('results/figures/cluster_statistics.png')

        print("Cluster visualizations saved to results/figures/")
        return data

    except Exception as e:
        print(f"Error visualizing clusters: {str(e)}")
        import traceback
        traceback.print_exc()
        return None

if __name__ == "__main__":
    visualize_clusters()
```

9. Plot prediction

```
 import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
import joblib
from sklearn.preprocessing import StandardScaler

def plot_predictions():
    """
    Plot model predictions against actual values.
    """
    print("Plotting model predictions...")

    # Ensure directories exist
    os.makedirs('results/figures', exist_ok=True)

    try:
        # Load ML data
        ml_data = pd.read_csv('data/processed/bitcoin_ml_data.csv', index_col=0, parse_dates=True)

        # Target variable
        target_col = 'future_return_24h'

        # Get features for prediction
        feature_cols = [col for col in ml_data.columns if col not in [target_col, 'open', 'high', 'low', 'close', 'volume',
                                                                      'price_up_24h', 'future_volatility_24h']]

        # Prepare data
        X = ml_data[feature_cols]
        y = ml_data[target_col]

        # Check for all-NaN columns
```

```
all_nan_cols = X.columns[X.isna().all()].tolist()
if all_nan_cols:
    print(f"Dropping columns with all NaN values: {all_nan_cols}")
    X = X.drop(columns=all_nan_cols)

# Handle remaining NaN values
X = X.fillna(X.median())

# Split data for visualization
test_size = 0.2
split_idx = int(len(X) * (1 - test_size))
X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx], y.iloc[split_idx:]

# Load models
models = {}
scalers = {}

# Check available model files
model_files = os.listdir('models')
print(f"Available model files: {model_files}")

# Try to load Random Forest model
rf_model_path = None
for file in model_files:
    if 'random_forest' in file.lower() and 'model' in file.lower():
        rf_model_path = os.path.join('models', file)
        break

if rf_model_path and os.path.exists(rf_model_path):
    print(f"Loading Random Forest model from: {rf_model_path}")
    models['Random Forest'] = joblib.load(rf_model_path)

# Try to load SVR model
svr_model_path = None
for file in model_files:
    if 'svr' in file.lower() and 'model' in file.lower():
        svr_model_path = os.path.join('models', file)
        break

if svr_model_path and os.path.exists(svr_model_path):
    print(f"Loading SVR model from: {svr_model_path}")
    models['SVR'] = joblib.load(svr_model_path)

# Create a scaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Make predictions
predictions = {}
for name, model in models.items():
    try:
        # Get the expected number of features for this model
        expected_features = None
        try:
            expected_features = model.n_features_in_
            print(f"{name} model expects {expected_features} features, we have {X_test_scaled.shape[1]} features")
        except:
            pass

        # Try to make prediction
        y_pred = model.predict(X_test_scaled)
        predictions[name] = y_pred
    except Exception as e:
        print(f"Error making predictions with {name} model: {str(e)}")
```

```
            # Try with a retrained model
            print(f"Retraining {name} model with current features...")
            model.fit(X_train_scaled, y_train)
            y_pred = model.predict(X_test_scaled)
            predictions[name] = y_pred

    # Create DataFrame for plotting
    plot_data = pd.DataFrame({'Actual': y_test})
    for name, pred in predictions.items():
        plot_data[name] = pred

    # Plot actual vs predicted
    plt.figure(figsize=(12, 6))
    plt.plot(plot_data.index, plot_data['Actual'], label='Actual', linewidth=2)

    for name in predictions.keys():
        plt.plot(plot_data.index, plot_data[name], label=f'{name} Prediction', linewidth=1.5, alpha=0.8)

    plt.title('Bitcoin Price Return Predictions')
    plt.xlabel('Date')
    plt.ylabel('Return (%)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('results/figures/model_predictions.png')

    # Plot error distribution
    plt.figure(figsize=(12, 6))
    for i, (name, pred) in enumerate(predictions.items(), 1):
        errors = plot_data['Actual'] - plot_data[name]

        plt.subplot(1, len(predictions), i)
        sns.histplot(errors, kde=True)
        plt.title(f'{name} Error Distribution')
        plt.xlabel('Prediction Error')
        plt.ylabel('Frequency')
        plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('results/figures/error_distribution.png')

    # Plot scatter of actual vs predicted
    plt.figure(figsize=(12, 6))
    for i, (name, pred) in enumerate(predictions.items(), 1):
        plt.subplot(1, len(predictions), i)
        plt.scatter(plot_data['Actual'], plot_data[name], alpha=0.5)

        # Add perfect prediction line
        min_val = min(plot_data['Actual'].min(), plot_data[name].min())
        max_val = max(plot_data['Actual'].max(), plot_data[name].max())
        plt.plot([min_val, max_val], [min_val, max_val], 'r--')

        plt.title(f'Actual vs {name} Predicted')
        plt.xlabel('Actual Return')
        plt.ylabel('Predicted Return')
        plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('results/figures/actual_vs_predicted.png')

    print("Model prediction plots saved to results/figures/")
    return plot_data

except Exception as e:
    print(f"Error plotting predictions: {str(e)}")
```

```
        import traceback
        traceback.print_exc()
        return None

if __name__ == "__main__":
    plot_predictions()
```