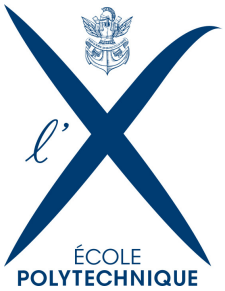

Real-Time Cryptocurrency Symbol Prediction Using Kafka

-Data Stream Processing-

Authors : Zakaria **AKIL**, Maha **KHADRAOUI**

Ecole Polytechnique | IP Paris

August 6, 2025

Contents

1 Introduction

Trading is the activity of buying and selling financial assets with the goal of generating profits from price variations and fluctuations in these assets. The concept of time and speed is crucial in the world of finance, and even more so in trading, as they directly impact profitability and risk. This is why **algorithmic trading** has emerged, to automate the buying and selling processes based on predefined rules, such as setting thresholds or conditions grounded in mathematical reasoning. This automation not only saves time, but also enhances efficiency.

Algorithmic trading proves to be particularly useful in the cryptocurrency market, which is characterized by its uncontrolled and often extreme volatility. It is in this dynamic and unpredictable environment that our project finds its purpose.

In our work, our goal is to develop real-time machine learning methods to predict market trends for specific **cryptocurrencies**. The goal is to accelerate and optimize the trading process, making it faster and more effective. To achieve this, we utilize data from **Binance**, one of the largest platforms in the world where users can buy, sell, and trade cryptocurrencies. This allows us to access real-time data streams, which serve as the foundation for powering our models.

We have chosen to study the evolution of five famous cryptocurrencies: BTCUSDT, ETHUSDT, ADAUSDT, XRPUSDT and GALAUSDT. For this purpose, we employ a Kafka architecture specifically designed for this project, along with various streaming models, to generate real-time predictions of market closing prices. The results are then displayed on a dashboard, where we compare our models' predictions with the actual values and visualize their performance using multiple metrics.

Furthermore, we performe a comparison between **batch** and **streaming** models to evaluate the strengths and weaknesses of each approach, ultimately aiming to identify the most effective methods for cryptocurrency trading.

2 Kafka Architecture

2.1 The Model We Implemented

Kafka is a distributed event store and stream-processing platform. It enables applications to produce (i.e., send) and consume (i.e., read) data in real-time with high performance. One of the platform's key strengths is its ability to handle large volumes of data with low latency. In our project, we use Kafka to manage the flow of stock market data.

When addressing this challenge, several architectural options were available. However, we opted for a design that struck the best balance between practicality, generality, efficiency, and simplicity. This approach not only allowed us to handle all the data efficiently but also ensured the results could be interpreted easily.

We chose to implement an architecture with two topics, one producer (DataProduce), one component acting as both a consumer for the first topic and a producer for the second topic (ConsumeTrainProduce), and a final consumer. This setup provided us with a well-structured and manageable data flow, ensuring that each stage was clearly defined and aligned with our objectives. This logical structure allowed for seamless integration and efficient management of our data streams, offering the flexibility needed to adapt and analyze results effectively.

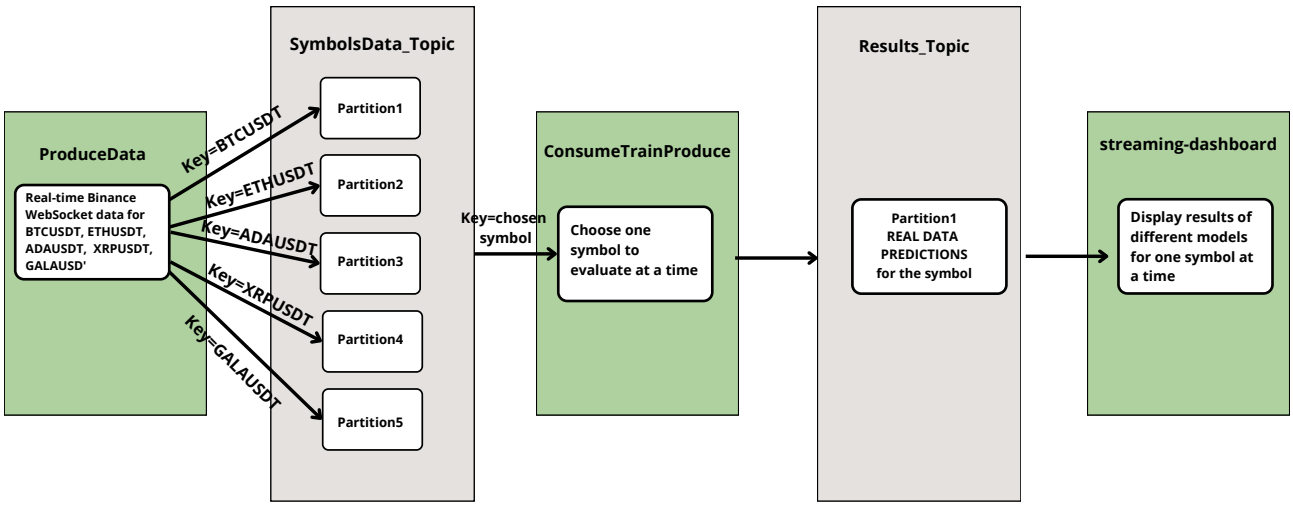

Figure .1: Kafka Architecture Used

We chose to use real-time WebSocket data to retrieve all relevant information for five specific cryptocurrency symbols. The **ProduceData** component fetches this data every five seconds and sends it directly to a single topic in a structured format, ensuring the data is easy to use in subsequent stages. We opted to use a single topic containing five partitions, leveraging the concept of a key. In fact, the **SymbolicData_Topic** contains five partitions, and we leverage the concept of a key to route data for each symbol to its corresponding partition. By using a number of partitions equal to the number of symbols we are processing, each symbol ends up in its own dedicated partition, which improves data organization, scalability, and parallel processing.

Next, the **ConsumeTrainProduce** component consumes the data for a specific symbol based on user requirements. It processes the data, trains the selected data stream models specified by the user, and then produces these results into the second topic, **Results_Topic**. Additionally, it sends the processed version of the data used for training, allowing real-time comparison.

The second topic contains a single partition dedicated to the real and predicted data for the selected symbol (with the ability to choose a different symbol for study each time). Finally, **StreamingDashboard** acts as a consumer in the form of an application, enabling us to visualize predictions and evaluate results.

2.2 Alternative Architectures That Could Have Been Considered

We could have designed two alternative architectures to address this problem. Below, we describe these approaches and their respective trade-offs:

- **Architecture Using a Second Topic with Five Partitions**

  In this approach, the second topic would also include five partitions, allowing the results of predictions from different models for all five symbols to be processed and available simultaneously. While this could increase parallelism and provide insights across all symbols at once, it could significantly slow down the system due to higher computational overhead. Additionally, managing concurrent processing for multiple partitions might increase system complexity and introduce synchronization challenges. For simplicity and to maintain efficiency, we opted to focus on one symbol at a time.

- **Architecture Using Two Topics Per Symbol**

  Another possible architecture would involve creating two separate topics for each symbol: one for sending training data and another for sending the results for that specific symbol. This design would result in multiple producers and consumers interacting with numerous topics simultaneously. While this approach could provide better data organization per symbol, it has significant drawbacks. It would complicate the system architecture, increase resource usage, and require more complex configurations and maintenance and it would have made monitoring and debugging more challenging.

3 Batch Learning

3.1 Introduction

In this project, we have a dedicated a batch learning component where we apply traditional machine learning models to historical data. The data was collected using Binance's API client by retrieving current data and rolling back to gather 100 instances, spanning from '2025-01-22' to '2022-04-29'. The primary objective of this project remains consistent: to forecast stock prices for specific cryptocurrency symbols.

In this section, we experiment with various models, comparing their performance. These include statistical models like ARIMA/ARIMAX and Prophet, and basic machine learning models such as SVR, XGBoost, and Random Forest. By doing so, we aim to evaluate their effectiveness and understand the potential of different techniques in predicting cryptocurrency price trends.

3.2 Data Exploration

In order to better understand the cryptocurrency data that we collected and we'll be using, we performed an exploratory data analysis in order to identify patterns and potential challenges in the dataset. The historical data used includes open, high, low, close, and volume values for specific cryptocurrency symbols.

Figure .2 presents a sample of the historical cryptocurrency data, showcasing the structure and key features of the dataset. Additionally, Figure .3 illustrates a specific example of the daily closing price trend BTCUSDT over time, where we can see significant fluctuations and trends that will guide our forecasting models.

3.3 Stationarity and Time Series Analysis

We will study the stationarity of time series which is an essential property in time series forecasting because most models, such as ARIMA, assume that the data are stationary. By definition, a stationary time series has a constant mean, variance and autocorrelation over time. In order to produce reliable forecasts, stationarity is necessary and therefore non-stationary data that exhibit trends or seasonality will need to be transformed.

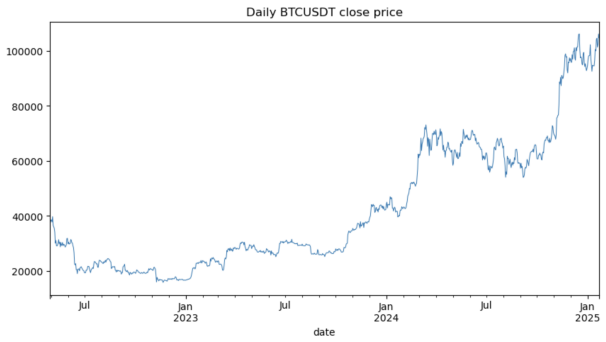

Figure .2: Historical Cryptocurrency Data Sample

date	open	high	low	close	volume
2022-04-29	39742.06	39925.25	38175.00	38596.11	51453.65715
2022-04-30	38596.11	38795.38	37578.20	37630.80	35321.18989
2022-05-01	37630.80	38675.00	37386.38	38468.35	38812.24104
2022-05-02	38468.35	39167.34	38052.00	38525.16	53200.92628
2022-05-03	38525.16	38651.51	37517.80	37728.95	40316.45358

Figure .3: Daily BTCUSDT Closing Price Trend

Dickey-Fuller Test for Stationarity

In order to see if a time series is stationary or not, we use the Augmented Dickey-Fuller (ADF) which is a statistical test whith these hypotheses:

- **Null Hypothesis (H_0)**: The data is non-stationary.
- **Alternative Hypothesis (H_1)**: The data is stationary.

The test provides a p-value, where:

- A high p-value ($p > 0.05$) means that the data is non-stationary.
- A low p-value ($p \leq 0.05$) means that the data is stationary.

With our dataset, we see that the ADF test applied to the original cryptocurrency closing price data returned a high p-value, confirming non-stationarity. So in order to make it stationnary we differentiate the data once, and that's when the test indicated stationarity which means that it is now suitable for modeling.

Visualizing Stationarity

To visually analyze stationarity, we plotted the rolling mean and rolling standard deviation for the original data (Figure .4) and after differencing (Figure .5). Significant deviations between the rolling mean and the original series in Figure .4 confirm non-stationarity. In contrast, Figure .5 shows a stable rolling mean and standard deviation which means that we now have the stationarity.

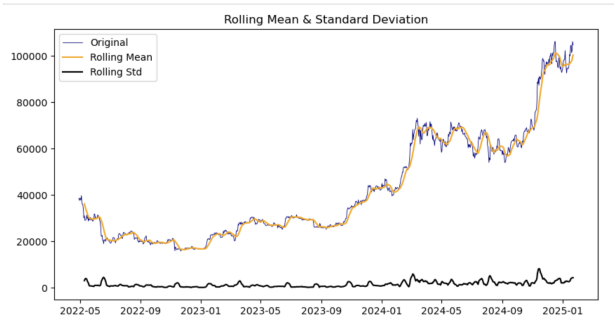

Figure .4: Rolling Mean and Standard Deviation (Before Differencing)

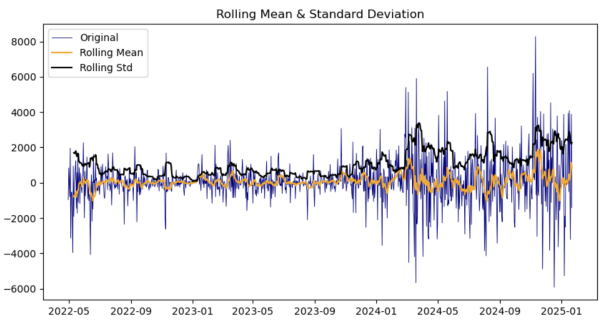

Figure .5: Rolling Mean and Standard Deviation (After Differencing)

Autocorrelation and partial autocorrelation

We analyze tools that will allow us to identify how past values of the series influence future values, thus allowing us to understand the structure of the data.

- **Autocorrelation (ACF):** measures the degree of correlation between the time series and its lagged versions (i.e. the same series lagged by one or more time steps), this will also allow us to identify patterns that persist over time.
- **Partial autocorrelation (PACF):** takes into account all indirect influences. To better explain, we can take the example of the PACF at lag 2, it measures the correlation between the current value and the value of two previous time steps, removing the influence of the intermediate value at $t-1$.

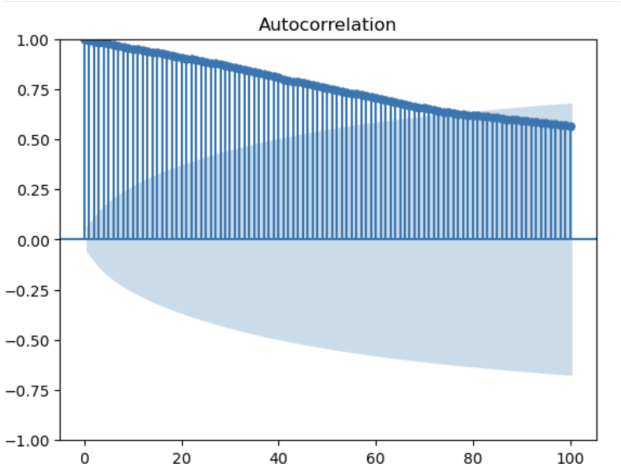

Figure .6: ACF Plot (Before Differencing)

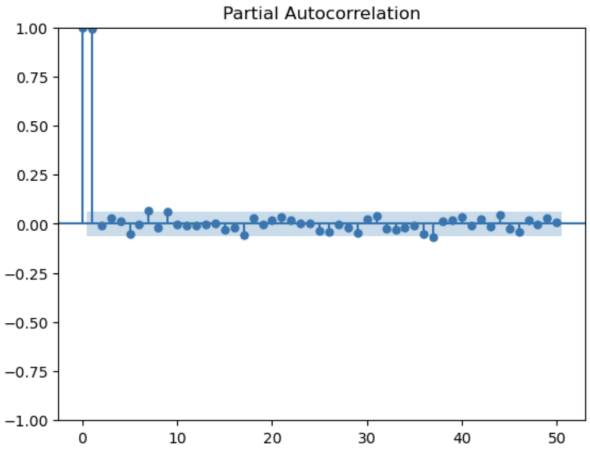

Figure .7: PACF Plot (Before Differencing)

Before differentiation, non-stationarity is proven by the figures above. In fact, we observe a slow decrease of the bars with a large number of significant lags exceeding the tolerance zone which reflects the correlation with past values. Similarly, the PACF shows two bars exceeding the blue zone which suggests a strong direct correlation between the current value of the series and the two immediately preceding values. After this peak the values become insignificant and do not exceed the confidence zone and therefore they do not have a direct effect.

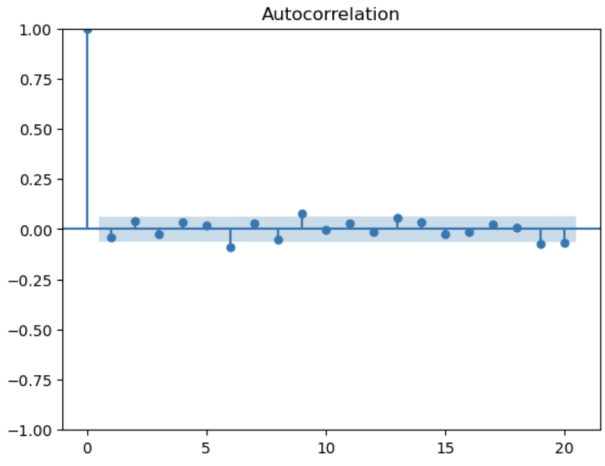

Figure .8: ACF Plot (After Differencing)

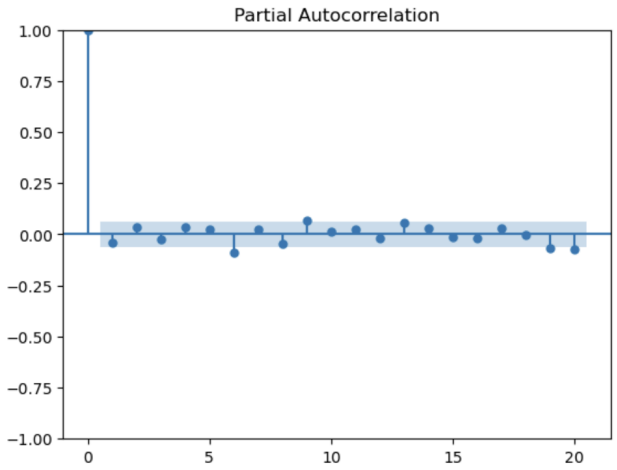

Figure .9: PACF Plot (After Differencing)

After differencing, the ACF shows a rapid drop after lag 1, indicating that the series is now stationary. Besides, the PACF shows a single significant peak at lag 1, suggesting that an autoregressive term (p=1) is appropriate to model the series.

3.4 Statistical Models

The stationarity analysis proved that an ARIMA model should be used with parameters $p = 1$, $d = 1$, and $q = 1$. We will use these parameters to try both the ARIMA and ARIMAX models for our data.

ARIMA Model

The ARIMA (AutoRegressive Integrated Moving Average) model is one of the most commonly used statistical models for time series forecasting. It captures relationships between lagged observations, adjusts for trends, and accounts for moving average components.

The Figure .10 shows that the ARIMA model predicts the overall trend with success because the predicted values closely align with the real values. But, for the test values we found a constant prediction that does not match the evolution of the real values. This is also shown in the residual plot (Figure .11), where we have big values at the end of thee plot, indicating that the model struggles to capture the complexity and volatility of cryptocurrency data during the test period.

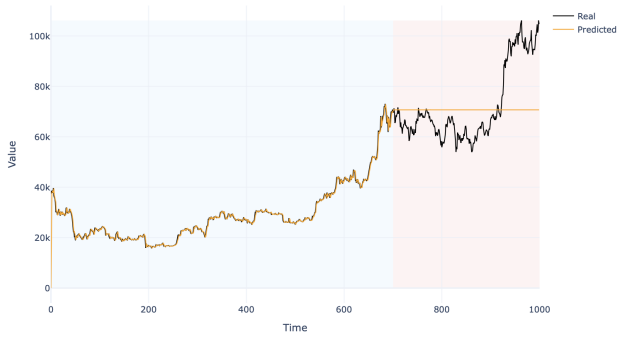

Figure .10: ARIMA Model Predictions

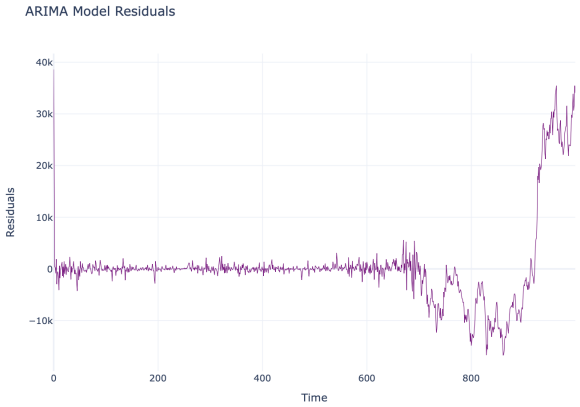

Figure .11: Residuals of ARIMA Model

ARIMAX Model

The ARIMAX (AutoRegressive Integrated Moving Average with Exogenous variables) model extends ARIMA by incorporating external, exogenous variables that may influence the target variable. This addition allows the model to leverage additional contextual information in order to improve its predictions. In our case, as the exogenous variable, we used the Xtest that contains the other value of price for that specific day(open, high, low,volume). This addition allows the model to leverage additional contextual information in order to improve its predictions.

The results of the ARIMAX model, shown in Figure .12, are quite surprin-sing as they almost perfectly align with the real values during both the training and testing periods and therefore outperform the ARIMA models. The residual plot (Figure .13) shows a distribution with very low variance indicating a good model fit. This demonstrates the importance of incorporating additional information in complex time series forecasting tasks.

Figure .12: ARIMAX Model Predictions

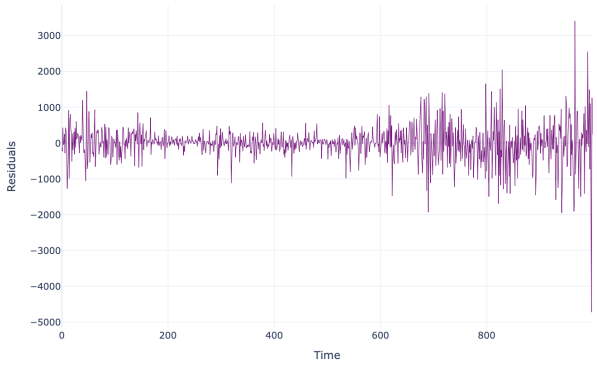

Figure .13: Residuals of ARIMAX Model

3.5 Prophet

Prophet is an open-source forecasting tool developed by Facebook that is designed to handle time series data with strong seasonality and trends. It models the data as a combination of three main components:

- **Trend**: Captures the overall upward or downward trajectory of the data over time.
- **Seasonality**: Accounts for repeating patterns in the data (e.g., daily, weekly, yearly).
- **Holiday effects**: Includes custom events or holidays that may impact the data.

figure .14 shows that during the training phase, the Prophet model manages to more or less capture the general trend of the cryptocurrency price, but it fails to adapt to the high volatility and strong fluctuations of the data. This is clearer for the test data where we see the predicted values that diverge considerably from the real values but that keep a general trend that is more or less correct which is here an increase in the curve;

This can be explained by the fact that Prophet favors long-term trends over short-term variations which is a limitation in very volatile environments like cryptocurrency markets where there is not really a clear notion of seasonality. We therefore conclude that the linear trend assumption of this model leads to oversimplified forecasts that do not reflect the complexity of the data.

Despite its limitations, Prophet remains a valuable tool for applications where trend and seasonality are the primary focus. However, for datasets dominated by high volatility, more advanced models, such as ARIMA, ARIMAX, or machine learning approaches, may yield better predictions.

3.6 Exploration of Classic Machine Learning Models

In addition to the statistical models, we also tested classic machine learning models to evaluate their performance on cryptocurrency price predictions. Specifically, we trained and tested three models: Random Forest, XGBoost, and Support Vector Regression (SVR). To enhance model performance, we added several statistical metrics derived from the basic features (open, high, low, and volume) to the dataset. Furthermore, a grid search was performed for each model to identify the optimal hyperparameters.

Model Overview

We recall a little the logic behind each model:

- **Random Forest**: An ensemble learning method that builds multiple decision trees and combines their outputs to improve accuracy and reduce overfitting.

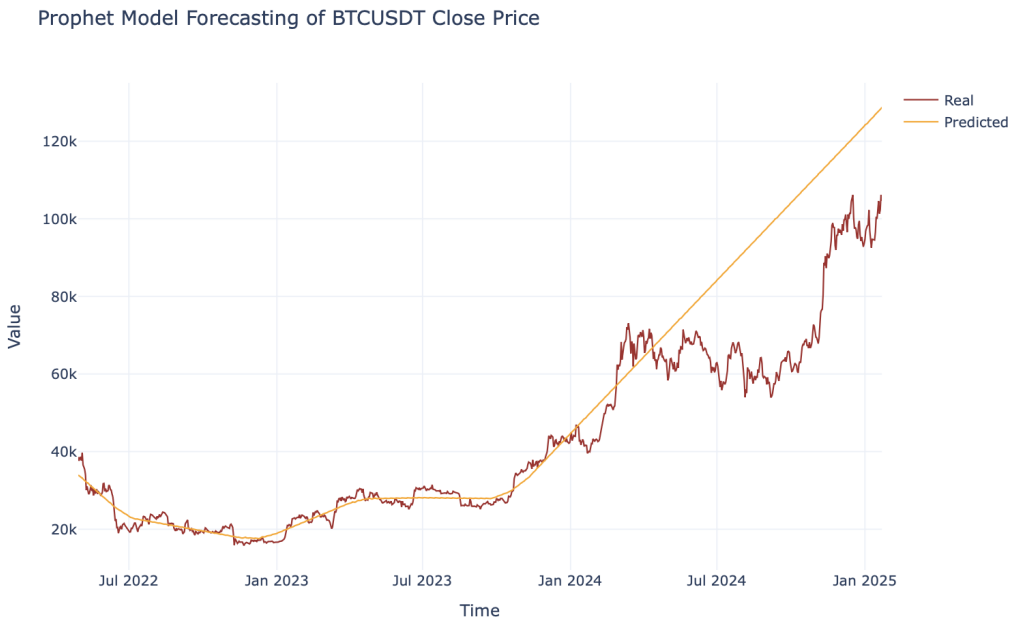

Figure .14: Prophet Model Forecasting of BTC/USDT Close Price

- **XGBoost**: An optimized gradient boosting algorithm that focuses on minimizing the loss function through sequential training and regularization to prevent overfitting.
- **SVR (Support Vector Regression)**: A regression model that finds the optimal hyperplane to minimize prediction error within a given margin, making it robust to outliers.

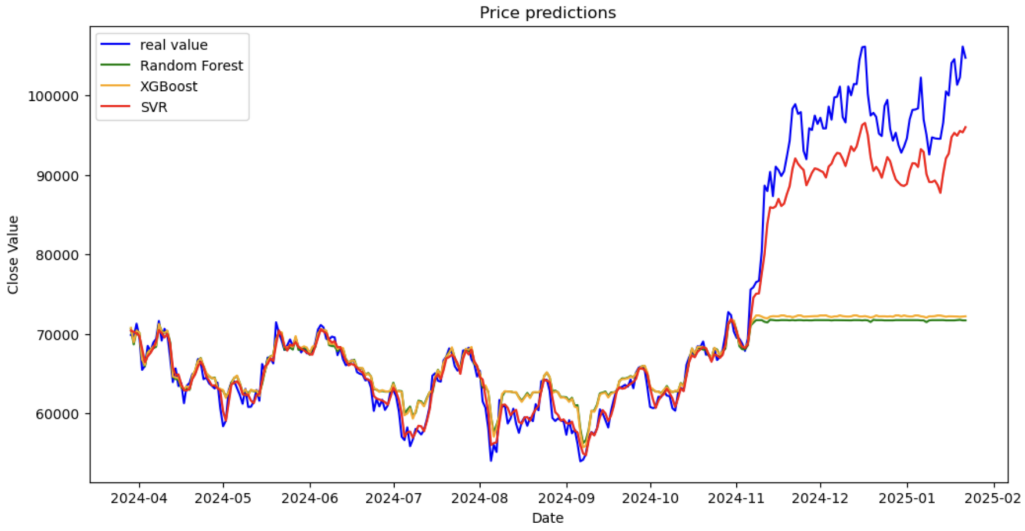

Figure .15: Price Predictions Using Random Forest, XGBoost, and SVR

Results and Analysis

The predictions from these models are visualized in Figure .15. Surprisingly, SVR outperformed the other models in terms of providing realistic predictions that closely follow the trends of the real values, especially during the test period. This suggests that SVR's ability to generalize well and handle small datasets effectively made it a suitable choice for our problem.

In contrast, Random Forest and XGBoost exhibited limitations in capturing the sudden spikes and volatility in the data. Random Forest produced predictions that were overly smoothed, failing to adapt to abrupt price changes, while XGBoost struggled with high volatility and showed less precise predictions during critical points.

These results highlight that, while ensemble methods like Random Forest and XGBoost are powerful, they may require additional tuning or feature engineering to handle extreme market fluctuations effectively. On the other hand, SVR demonstrated its strength in adapting to the inherent volatility of cryptocurrency markets, providing one of the best performances observed so far.

3.7 Batch Models Across Different Symbols

We would like to clarify that this batch learning section was developed to explain the different models used, showcasing the results obtained for a single specific symbol. However, when the same models are applied to historical data from other symbols, the results are generally similar but can occasionally exhibit surprising differences. The figures below, for instance, present the metrics obtained using the SARIMA and SARIMAX models for three different symbols BTC, ADA and ETH.

Symbol	Model	MAE	MSE	MAPE	R2
BTC	SARIMA	11593.6100	14890.30	0.1502	−0.0100
ADA	SARIMA	0.2618	0.28	0.5560	−0.1612
ETH	SARIMA	513.8800	642.67	0.1881	−0.9103

Figure .16: Metrics for SARIMA across three symbols.

Symbol	Model	MAE	MSE	MAPE	R2
BTC	SARIMAX	530.276058	752.081310	0.007348	0.997423
ADA	SARIMAX	0.109048	0.116646	0.234734	0.798542
ETH	SARIMAX	50414.401429	51513.345621	16.306808	−12273.034677

Figure .17: Metrics for SARIMAX across three symbols.

4 Stream/Online Learning

In the previous section, we explored the feasibility of predicting cryptocurrency prices using the classic batch learning techniques. Although batch learning offered valuable insights and some good prediction results on past trading data, it has limitations when dealing with dynamic and ever-changing market conditions. Training a model on the past to predict the whole future sometimes requires high complexity models and could not even adapt to the change of the concept. Real-world financial markets are characterized by:

- Non-stationarity: Market trends and volatility can shift rapidly, making historical data less relevant for future predictions.
- Concept drift: The relationships between features (e.g., trading volume, news sentiment) and the target variable (close price) can change over time.
- Data streams: New data arrive continuously, requiring the model to adapt and learn from the latest information.

To address these challenges, we shift to the class of online models, which are better adapted to this task. These models are capable of learning incrementally as data arrives in real time, while also making predictions based on their current knowledge of the data. Alternating between training and predictions allows the model to continuously adjust its parameters to reflect evolving market dynamics. Additionally, many algorithms can be implemented in parallel to detect concept drift in the data, which is not feasible with batch learning methods where data is grouped for the model to learn the resulting training distribution.

In this section, we will use streaming algorithms from River [3], a machine learning library for dynamic data streams and continual learning in Python. River provides a wide range of state-of-the-art learning methods, data generators/transformers, performance metrics, and evaluators for various stream learning problems.

We tested multiple stream/adaptive regressors to predict the prices of three different cryptocurrency symbols. Using Binance's Client API, we saved data locally to simulate a stream for testing. Testing online models on simulated streams before applying them directly to real-time streams enables efficient and fair comparisons between models, as they are evaluated on the same data under the same conditions. Furthermore, this approach allows us to draw conclusions about the adaptability and learning speed of each algorithm before deploying it in a live setting.

4.1 Feature engineering

In this project, we enriched the raw cryptocurrency price data (open, high, low, volume, n_trades) with several technical indicators commonly used in financial analysis. These indicators provide additional insights into price dynamics and help predictive models more effectively capture market trends and behaviors. Below is a description of the main indicators used:

Exponential Moving Average (EMA) The EMA is a moving average that gives more weight to recent prices, allowing it to capture current trends more quickly than the Simple Moving Average (SMA). It is defined as:

$$\text{EMA}_t = \alpha \cdot \text{Price}_t + (1 - \alpha) \cdot \text{EMA}_{t-1}, \tag{.1}$$

where:

- $\alpha = \frac{2}{n+1}$ is the smoothing factor,
- n is the period for the calculation.

Simple Moving Average (SMA) The SMA is an arithmetic average of prices over a given period n, calculated as:

$$\text{SMA}_t = \frac{1}{n} \sum_{i=t-n+1}^{t} \text{Price}_i. \tag{.2}$$

Stochastic Oscillator The Stochastic Oscillator measures the relative position of the closing price with respect to its range (high-low) over a given period n. It is defined as:

$$K_t = \frac{\text{Close}_t - \text{Low}_{t-n}}{\text{High}_{t-n} - \text{Low}_{t-n}} \cdot 100, \tag{.3}$$

where:

- Close_t is the current closing price,
- Low_{t-n} is the lowest price over the last n periods,
- High_{t-n} is the highest price over the last n periods.

Moving Average Convergence Divergence (MACD) The MACD is a momentum indicator that shows the relationship between two EMAs, and the EMA of the MACD, is often used to generate buy or sell signals.

$$\text{MACD}_t = \text{EMA}_{\text{fast},t} - \text{EMA}_{\text{slow},t}, \tag{.4}$$

where:

- EMA_{fast} is the fast EMA (e.g., 10 periods),
- EMA_{slow} is the slow EMA (e.g., 30 periods).

Relative Strength Index (RSI) The RSI is a famous momentum indicator that measures the speed and magnitude of price changes. It is defined as:

$$\text{RSI}_t = 100 - \frac{100}{1 + RS_t}, \tag{.5}$$

where:

$$RS_t = \frac{\text{Average Gain}_t}{\text{Average Loss}_t}. \tag{.6}$$

These indicators were calculated at each timestep in the streaming data. By combining these features with raw price data (*open*, *high*, *low*, *volume*, *n_trades*), we constructed a rich and representative feature set that served as input to the prediction models in the phase of model testing.

4.2 Streaming Models

Linear Regression

Linear regression model is considered a basic statistical model to predict the prices of financial assets. One of the strengths of this model is its interpretability. The 'debug one' method provides a breakdown of predictions, illustrating how each feature contributes to the final output. This allows a detailed understanding of the model's behavior even when embedded in a pipeline. By combining its simplicity, flexibility, and interpretability, it served as a baseline model for our analysis as its linear nature may underestimate the complexity of the problem.

Passive-Aggressive Regressor

The Passive-Aggressive Regressor is a linear online learning algorithm that is well-suited for data stream scenarios. This model aims to find a balance between adjusting to new data (aggressiveness) and maintaining stability with previously learned data (passiveness).

The update rule in Passive-Aggressive regression is designed to minimize a hinge loss function while staying as close as possible to the current model parameters. Its ability to dynamically adjust to concept drift made it a valuable tool for predicting financial trends in a streaming environment. While efficient and adaptable, the model may struggle with non-linear patterns in data, which are common in financial markets

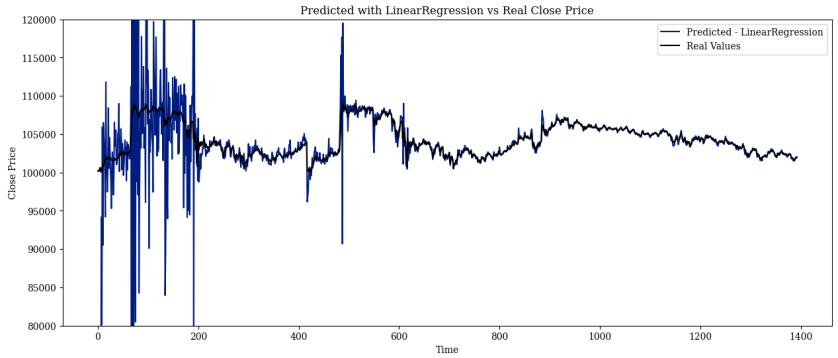

Figure .18: BTCUSDT : PAR predictions vs Real stream

Adaptive Random Forest Regressor

Random forest is also considered as a foundational interpretable ensemble algorithm as it delivers impurity-based results and associate an importance score to the features. River introduced an adaptive version of

RF that can be trained on streams. We will not present the detailed algorithm in this report, for more details see [1].

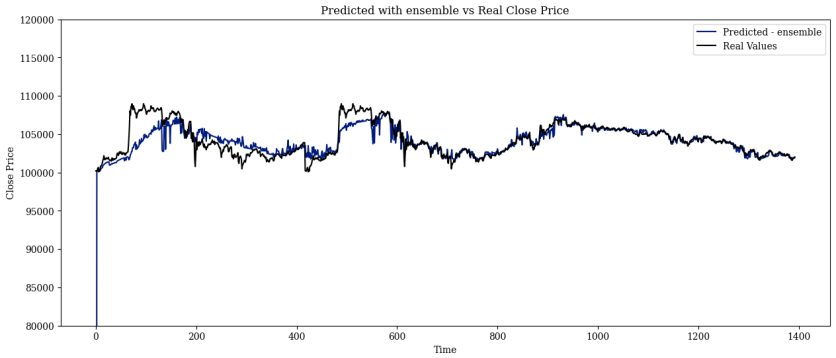

Figure .19: BTCUSDT : ARF predictions vs Real stream

K-Nearest Neighbor Regressor

The K-Nearest Neighbor (KNN) Regressor is a non-parametric model that predicts the target value of a query instance by aggregating the target values of its nearest neighbors in feature space. It is particularly useful for capturing local patterns in the data.

This model works by storing past samples in a first-in, first-out (FIFO) buffer and dynamically querying the closest neighbors when making predictions. The closeness of neighbors is determined by a distance metric, and the prediction is computed using an aggregation method such as the mean, median, or weighted mean of the neighbors' target values. The KNN Regressor was applied in this project to leverage its ability to capture local patterns in the evolving cryptocurrency data stream.

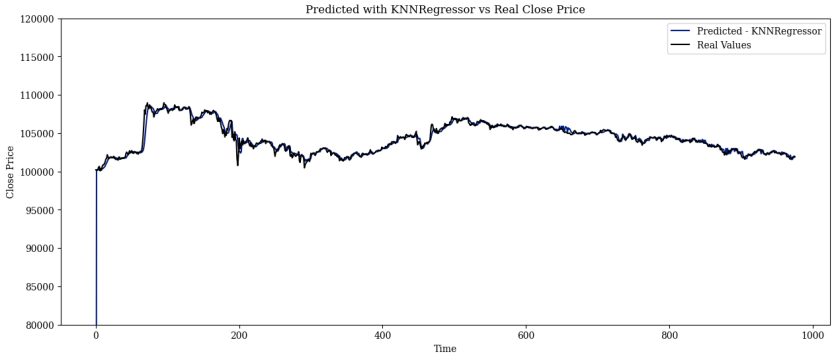

Figure .20: BTCUSDT : KNN predictions vs Real stream

4.3 Models Comparison & Analysis

Metric	LinearRegression	ARFRegressor	KNNRegressor	PAReg
R^2	-1.900216e+12	-4.50	-1.82	-2.212259e+03
MAE	3.505076e+08	876.8530	334.45	8.644792e+04
RMSE	2.653096e+09	4.515088e+03	3.233308e+03	9.054562e+04
MAPE	8.532932e+05	0.83868	0.025804	8.287282e+01
CPUTime	0.00045	0.01654	0.02208	0.00025

Table .1: Metrics Comparison for Different Regressors

Remarks :

- As illustrated in Table 1 above, the KKN Regressor demonstrates superior performance relative to other online models across all evaluated metrics, with the exception of CPU time, where simpler models exhibit greater efficiency. The adaptive RandomForest also performs commendably and can be juxtaposed with the KKN. However, its delayed adaptation and learning from the stream place it at a disadvantage compared to the KKN, which rapidly aligns with the signal and adjusts to the price swiftly.

- The values of the metrics in the table may be misleading, as they might suggest that the models are performing poorly on the problem. However, it is important to note that these metrics are incremental, meaning they average over all samples processed so far. Each time a new sample arrives, the instance-wise metric is calculated and added to the cumulative average. As a result, the losses appear large because the metrics still account for the initial predictions, which were far from the actual signal, as the models were just beginning to learn. This explanation also applies to the R^2 score. This behavior can be better understood by observing the figures comparing the real price to the predicted one, or by using alternative methods for computing metrics, such as windowed metrics (averaging over a fixed window of recent samples) or instance-wise metrics (evaluating performance on individual samples).

4.4 Concept Drift Analysis

After identifying the best online regressor based on various regression metrics, we further analyzed the learning process and examined common trends across all regressors throughout the stream.

In real-time forecasting, we observed multiple instances of performance drops across all models. These consistent drops suggest they are tied to the trading stream itself rather than specific model limitations. This phenomenon, known as *concept drift* in data stream processing, reflects changes in the underlying data distribution over time. While these shifts cannot always be explained by technical analysis, they can be detected and addressed.

To develop a robust online regressor, it is crucial to incorporate a concept drift detector. Such a detector triggers alerts when a data concept change is detected, allowing the model to adapt—for instance, by clearing old data in its window and learning the new trend or distribution.

Concept drift detectors rely on key hyperparameters, including window size and sensitivity (*delta*), which must be tuned based on the nature and frequency of drifts as well as the specific application:

- **Delta (sensitivity):** With a fixed error window, a delta value closer to 1 increases the likelihood of detecting concept drifts, as it lowers the confidence threshold required to signal a change.

- **Window size:** With a fixed detector and confidence level, larger error windows reduce drift detections because averaging smoothens out volatility, making errors more uniform.

For our study, we could have used concept drift detectors such as **SEED**, **ADWIN**, or **STEPD**, which are implemented in libraries like CapyMOA [2] or River [3]. However, due to time constraints, we did not integrate them into our real-time forecasting application.

5 Comparaison Batch and Stream Learning

Here , we try to contrast batch learning and stream learning approaches in order to highlight the main differences:

Table .2: Comparison of Batch and Streaming Learning

Aspect	Batch Learning	Streaming Learning
Adaptability	✗ Limited adaptability to new data without retraining.	✓ Quickly adapts to changes and concept drift in the data stream.
Data Processing	✓ Processes the entire dataset at once, offering comprehensive historical analysis.	✓ Processes data incrementally as it arrives, enabling real-time predictions.
Outlier Sensitivity	✓ Less sensitive to noise or outliers due to training on full datasets.	✗ Prone to being influenced by anomalies, which can affect accuracy in volatile markets.
Performance	✓ Excels in identifying long-term trends and patterns.	✓ Optimized for short-term predictions and immediate insights.
Resource Demand	✓ Requires less computational power during inference.	✗ Demands continuous computational resources for real-time processing.
Use Case	✓ Best for static and retrospective analyses.	✓ Ideal for dynamic and time-sensitive applications.

6 Conclusion

This project allowed us to delve into stock market forecasting, a highly complex and ever-relevant task, particularly with the rise of cryptocurrencies, which have introduced an unprecedented level of volatility compared to traditional supervised markets. This makes the field a continuously evolving area of research that requires further exploration. Our approach contributes to this subject by deploying batch and streaming models to analyze performance and attempt to forecast the evolution of different symbols. Using Kafka and River for real-time processing, along with batch modeling using ARIMA/X, Prophet, and some traditional models like SVR, we achieved interesting results that highlight the complexity of this domain.

In summary, the decision between batch and streaming models hinges on the specific requirements of the forecasting task. Batch models provide a balanced approach with reliable performance and efficient training, while streaming models excel in adaptability and real-time forecasting accuracy, making them particularly valuable in dynamic environments where rapid adjustments to shifting trends are crucial.

7 References

—

1. Heitor Murilo Gomes, Jean Paul Barddal, Luis Eduardo Boiko, Albert Bifet (2018). *Adaptive random forests for data stream regression*, PaperLink

2. CapyMOA, API Webpage, GitHub Repository

3. Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, Albert Bifet (2021). *River: machine learning for streaming data in Python*, https://arxiv.org/abs/2012.04740

4. Binance API Python package. Binance API Guide link