Topological Data Analysis of the Stock Market

Charles Njoroge[1]

[1]University of Chicago
[2]Physical Sciences Division

Monday 13th May, 2024

Abstract

In the rapidly evolving field of quantitative finance, I am integrating advanced mathematical techniques to try and offer novel insights into market structures and dynamics. My independent project leverages Topological Data Analysis (TDA) to enhance stock analysis and trading strategies. By using TDA, I extract persistent homology features from high-dimensional market data, capturing the underlying geometric and topological structures that conventional methods may overlook. These topological features are then utilized as input for a linear regression model to predict future stock price outcomes based on various market variables.

My approach involves constructing Vietoris-Rips complexes and calculating persistent barcodes, which may serve as robust indicators of market trends and anomalies. I train the linear regression model on these topological features, demonstrating its capability to uncover non-linear patterns and relationships inherent in the financial data. This integration of TDA and linear regression not only provides a deeper understanding of market behavior but also aims to enhance predictive accuracy and trading performance. The results indicate that incorporating topological features can offer insights that, with deeper tuning and market book data, can significantly improve the precision of stock price predictions, and may offer a powerful tool for seeking to gain a competitive edge in the market.

This method can be generalized for other types of data by applying Topological Data Analysis to uncover intrinsic geometric and topological structures within diverse datasets, such as in medical imaging, social networks, or other economic data. By extracting persistent homology features and integrating them with machine learning models, I can enhance the predictive power and analytical depth across various fields, providing robust insights and improved decision-making.

Contents

1 Introduction: Topological Data Analysis

Topological Data Analysis (TDA) is a field in computational topology that provides powerful tools for analyzing the shape of data. Unlike traditional statistical methods, which often focus on global properties, TDA captures the underlying topological and geometric structures, enabling the discovery of complex patterns and relationships within high-dimensional datasets.

At the core of TDA is the concept of *persistent homology*, which studies the multi-scale topological features of a data set. This technique involves constructing a family of simplicial complexes from the data and tracking the evolution of topological features such as connected components, loops, and voids across different scales. The result is a collection of *persistent barcodes* or *persistence diagrams*, which visually represent the birth and death of these features, providing insight into the intrinsic structure of the data in higher dimensional space.

TDA has been successfully applied in various domains, including biology, neuroscience, sensor networks, and finance, proving its versatility and effectiveness in uncovering hidden patterns that are often missed by other analytical methods.

1.1 Point Clouds

A point cloud is a set of points in a geometric space. In three dimensions, a point cloud is given by $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N\} \subset \mathbb{R}^3$, where each point $\mathbf{p}_i$ is represented as a vector in $\mathbb{R}^3$. This concept generalizes to an n-dimensional point cloud, defined as $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N\} \subset \mathbb{R}^n$ where each point $\mathbf{p}_i$.

1.2 Vietoris-Rips Complex

The Vietoris-Rips complex is a type of simplicial complex that is constructed from a point cloud. Given a set of points $\mathcal{P} = \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N\}$ and a distance parameter $\epsilon > 0$, the Vietoris-Rips complex $\mathcal{VR}_\epsilon(\mathcal{P})$ is defined as follows:

$$\mathcal{VR}_\epsilon(\mathcal{P}) = \{\sigma \subseteq \mathcal{P} \mid \text{diam}(\sigma) \leq \epsilon\},$$

where $\text{diam}(\sigma)$ is the diameter of the subset σ, which is the maximum pairwise distance between points in σ.

In simpler terms, the Vietoris-Rips complex connects points that are within a certain distance ϵ of each other, forming edges, triangles, and higher-dimensional simplices as more points are added within this distance. This construction helps in capturing the shape of the data at different scales.

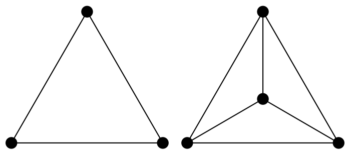

Figure 1: On the left is a simple example of a Vietoris-Rips complex with ϵ such that three points are connected to form a triangle. On the right, a tetrahedron formed in a Vietoris-Rips complex with four points and appropriate ϵ.

1.3 Simplex Tree

A simplex tree is a data structure used to efficiently store and manipulate simplicial complexes. A simplex is a generalization of the concept of a triangle or tetrahedron to arbitrary dimensions. For example, a k-simplex is defined as a set of $k+1$ vertices $\{v_0, v_1, \ldots, v_k\}$.

The simplex tree is designed to handle the combinatorial complexity of simplicial complexes by maintaining a tree structure where each node corresponds to a simplex, and its children correspond to the simplices that can be formed by adding one more vertex. This allows for efficient operations such as insertion, deletion, and traversal of simplices.

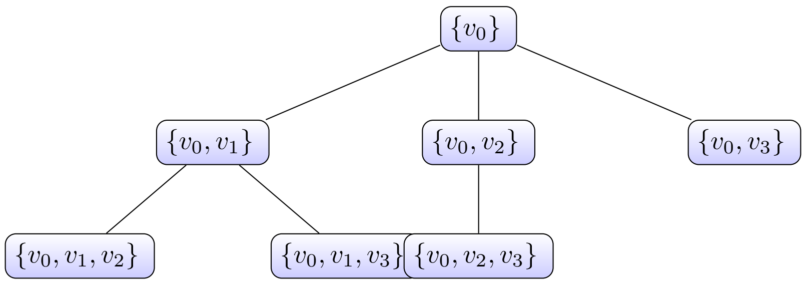

Figure 2: A simple example of a simplex tree. Each node represents a simplex, with edges indicating the addition of a new vertex.

1.4 KD Tree: Finding ϵ-Neighbors

A k-d tree (k-dimensional tree) is a binary tree used for organizing points in a k-dimensional space. It is particularly useful for range searches and nearest neighbor searches, making it efficient for finding points within a certain distance ϵ.

The algorithm to build a k-d tree involves the following steps: 1. Select the axis based on depth so that the axis cycles through all dimensions. 2. Sort the list of points and choose the median as the root. 3. Recursively construct the left and right subtrees using the median to split the points. It's important to note that in my implementatoin of a kd-tree, the tree is an N-Aray tree allowing for finder grained leveling of the data.

The construction of a k-d tree takes $O(N \log N)$ time, where N is the number of points. This is because each level of the tree requires sorting the points, and there are $O(\log N)$ levels.

Once the k-d tree is built, searching for all points within an ϵ distance can be done efficiently. The search algorithm involves: 1. Starting at the root, check if the current point is within ϵ distance. 2. Recursively search the subtree depending on the split dimension and distance. 3. Prune branches that cannot contain points within the ϵ distance, reducing unnecessary computations.

The search process has an average-case complexity of $O(\log N)$ per query, significantly reducing the computational burden compared to a naive $O(N^2)$ pairwise comparison.

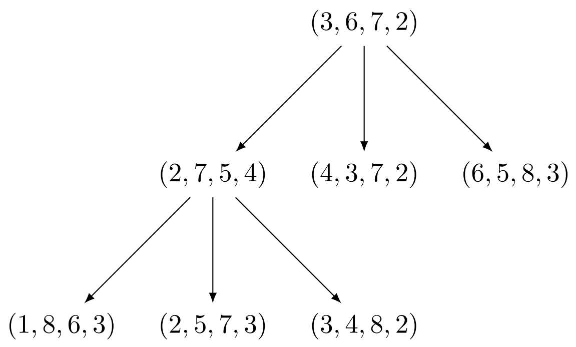

Figure 3: A k-d tree example in 4-dimensional space. Each node represents a point with 4 coordinates, with edges indicating the division based on the median of the selected axis. The varying arrow lengths represent different dimensions and distances. Note: Example is not exact.

1.5 Takens' Embedding

Takens' embedding theorem is a result in the study of dynamical systems and time series analysis. The theorem provides a method to reconstruct the state space of a dynamical system from a time series of observations. This reconstruction is achieved by constructing a phase space using time-delayed versions of the observed data.

Given a time series of observations $\{x(t)\}_{t=0}^{N}$, Takens' embedding theorem states that under certain conditions, it is possible to reconstruct the state space of the underlying dynamical system using delay coordinates. The reconstructed state space is defined as:

$$\mathbf{y}(t) = [x(t), x(t+\tau), x(t+2\tau), \ldots, x(t+(m-1)\tau)],$$

where τ is the time delay and m is the embedding dimension.

Some key points crucial to understanding Taken's Ebedding are:

- **Time Delay (τ)**: The time delay τ is an important parameter that determines the separation between consecutive elements in the delay vector. Choosing an appropriate τ is essential for a meaningful reconstruction of the phase space.
- **Embedding Dimension (m)**: The embedding dimension m determines the number of delayed observations used to form the delay vector. According to Takens' theorem, for most systems, an embedding dimension $m \geq 2d + 1$ is sufficient, where d is the dimension of the original state space.
- **Reconstruction of State Space**: The reconstructed state space using delay vectors is topologically equivalent to the original state space. This means that the qualitative features of the dynamical system, such as attractors and invariant measures, are preserved in the reconstructed space.

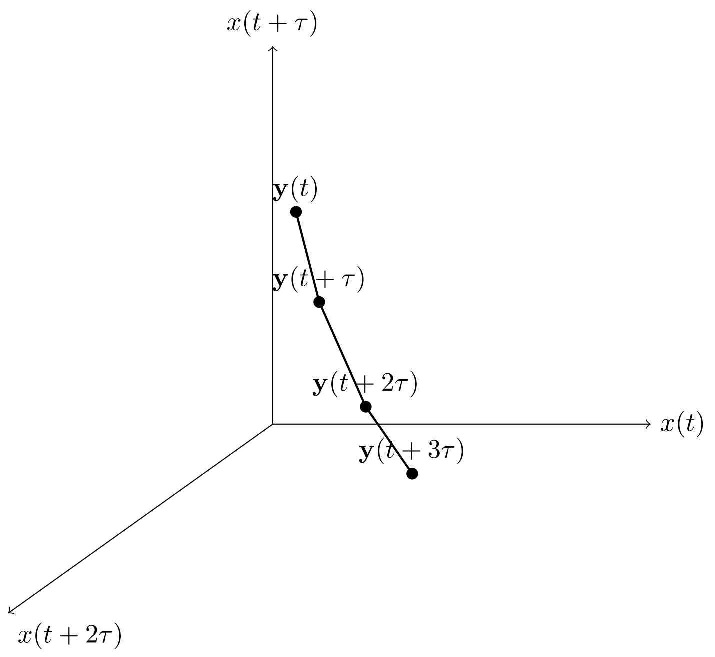

Figure 4: Illustration of Takens' embedding with time delay τ. The reconstructed state space is formed using delay coordinates from the time series into a 3-dimensional space.

In summary, the embedding theorem provides a framework for reconstructing the dynamics of a system from time series data. By selecting appropriate time delays and embedding dimensions, it is possible to reveal the underlying structure of complex dynamical systems, facilitating deeper analysis and understanding.

1.6 Linear Regression Model

Linear regression is a fundamental machine learning algorithm used for modeling the relationship between a dependent variable and one or more independent variables. The goal of linear regression is to find the best-fitting linear relationship between the variables, which can be used for prediction.

1.6.1 Model Description

Given a dataset with n observations, where each observation i consists of an independent variable x_i and a dependent variable y_i, the linear regression model can be represented as:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

Here:

- y_i is the dependent variable (response variable) for the i-th observation.
- x_i is the independent variable (predictor variable) for the i-th observation.
- β_0 is the intercept term, representing the value of y when x is zero.
- β_1 is the slope term, representing the change in y for a one-unit change in x.
- ϵ_i is the error term, accounting for the deviation of the observed values from the true linear relationship.

1.6.2 Objective Function

The objective of linear regression is to minimize the sum of squared errors (SSE) between the predicted values and the actual values. This is achieved by finding the optimal values of β_0 and β_1 that minimize the following cost function:

$$J(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2$$

1.6.3 Parameter Estimation

The optimal parameters β_0 and β_1 can be estimated using the least squares method. The estimates are given by:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

where $\bar{x}$ and $\bar{y}$ are the means of the independent and dependent variables, respectively.

1.6.4 Prediction

Once the parameters are estimated, the linear regression model can be used to make predictions. For a new input value x_{new}, the predicted output y_{pred} is given by:

$$y_{\text{pred}} = \hat{\beta}_0 + \hat{\beta}_1 x_{\text{new}}$$

1.6.5 Evaluation Metrics

The performance of the linear regression model can be evaluated using various metrics, including:

- **Mean Squared Error (MSE)**: Measures the average of the squares of the errors.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- **R-squared (R^2)**: Indicates the proportion of the variance in the dependent variable that is predictable from the independent variable.

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$$

2 Nexus

Nexus analyzes time series data by reconstructing the state space using Takens' embedding, building a simplicial complex, and computing persistent homology to provide insights into the underlying dynamics of the data. Afterwards, a linear regression model is used to take the computed persistent barcodes.

2.1 Algorithm

The `algorithm` is designed to analyze stock market data using Takens' embedding and persistence homology. This algorithm performs several key tasks, including data transformation, Takens' embedding generation, nearest neighbor search, simplex construction, and persistence pair computation. The results are then serialized and pushed onto a queue for further processing.

The algorithm retrieves closing price data and weighted volume averages for the last few minutes from the `DataStorage`. These values are then transformed for further analysis. A `TakensEmbeddingGenerator` object is used to generate Takens' embeddings for the transformed data. The embedding generator calculates the deltas and generates vectors for these embeddings, which are then stored in a KD-tree for efficient nearest neighbor search. Using the KD-tree, the algorithm performs a nearest neighbor search for each embedding point within a specified distance, ϵ, scaled by an increment factor. This search is executed asynchronously to improve performance. The results are collected into a set of unique neighbor indices. Then, a `SimplexTree` object is constructed using the nearest neighbors, and simplices are added to the tree. The simplices are then gathered and filtered to remove duplicates. Persistence pairs are computed from the filtered simplices, representing the birth and death times of topological features. The serialized `StockBarcode` message is then pushed onto the lock-free queue for further processing. If serialization fails, an error is logged.

2.2 Pseudocode

Algorithm 1 analyzeData

```
Require: storage, barcode_queue, file
1: data ← storage.getFieldValuesForLastMinutes("closing_price")
2: wva ← storage.getFieldValuesForLastMinutes("weighted_volume_average")
3: embedding_gen ← TakensEmbeddingGenerator($\tau = 1$, $m = 3$)
4: epsilon ← 0.0001
5: for each (symbol, prices) in data do
6:     if size(prices) > 10 then
7:         trans_data ← transform(prices)
8:         trans_wva ← transform(wva[symbol])
9:         category ← classify(trans_data, trans_wva)
10:        deltas ← embedding_gen.calculate_deltas(trans_data)
11:        vectors ← embedding_gen.generate(deltas)
12:        embeddings ← create_points(symbol, vectors)
13:        tree ← KDTree(embeddings)
14:        neighbors ← find_neighbors(tree, embeddings, epsilon)
15:        simplex_tree ← SimplexTree()
16:        for each (dist, neigh) in neighbors do
17:            simplex_tree.add_simplex(Simplex(neigh, dist))
18:        end for
19:        persistence_pairs ← compute_persistence_pairs(simplex_tree)
20:        stock_barcode ← create_stock_barcode(symbol, category, persistence_pairs)
21:        write_to_file(file, stock_barcode)
22:        barcode_queue.push(stock_barcode.serialize())
23:    end if
24: end for
```

2.3 Implementation

2.3.1 Protocol Buffers (Protobuf)

Protocol Buffers, commonly known as Protobuf, are a tool developed by Google for serializing structured data. It is useful for both data storage and communication protocols, such as gRPC (gRPC Remote

Procedure Call), providing a language-neutral, platform-neutral, extensible mechanism for serializing structured data.

In Nexus, I define several Protobuf messages to handle stock event data and feature engine communications.

The 'StockEvent' message represents individual stock events, containing fields such as the event name, stock symbol, tick volume, accumulated volume, opening price, weighted volume average price, opening tick price, closing tick price, high price, low price, volume-weighted average, average trade size, starting timestamp, ending timestamp, and a boolean indicating whether the stock is over-the-counter (OTC). These fields provide a comprehensive representation of stock trading events and their associated metadata. The 'StockEvents' message is a container for multiple 'StockEvent' messages, allowing me to group a series of stock events together. The 'StockMinuteAggregate' message captures aggregated stock data over a specified time window. It includes fields for the stock ticker symbol, trading volume, opening price, closing price, highest price, lowest price, the Unix timestamp for the start of the aggregation window, and the number of transactions within the window. This message helps in summarizing trading activities over minute-long intervals.

In the feature engine package, the 'FeatureEngineRequest' message is used to request data from the market, with fields indicating the name of the request, and boolean flags to control thread operations (kill or restart threads). The 'FeatureEngineResponse' message contains a simple string message as a response. The 'Barcode' message defines a topological feature with fields for dimension, birth, and death times, which are used in topological data analysis. The 'StockBarcode' message associates a stock ticker symbol with a series of 'Barcode' messages and a category field indicating the type of market condition (e.g., bullish, bearish, or neutral).

Finally, the 'FeatureEngineService' defines an RPC (Remote Procedure Call) service with a method 'CallMarket' that takes a 'FeatureEngineRequest' and returns a stream of 'FeatureEngineResponse' messages, enabling real-time market data processing and analysis.

2.3.2 Lock-Free Wait-Free Queue

A lock-free wait-free queue is a concurrent data structure that allows multiple threads to perform operations on the queue without the need for mutual exclusion mechanisms such as locks. This ensures that the queue operations are not only thread-safe but also avoid common issues associated with locks, such as deadlocks and contention.

The following describes the key components and operations of the lock-free wait-free queue implementation:

Node Structure

Each node in the queue contains:

- **data**: A unique pointer to the data of type `T`, ensuring that each node exclusively owns its data.
- **next**: An atomic pointer to the next node in the queue, which allows safe updates in a concurrent environment.

Queue Initialization

The queue is initialized with a dummy node, which acts as both the head and the tail of the queue initially. This dummy node simplifies the implementation by ensuring that the queue is never empty.

Push Operation

The `push` method inserts a new element at the end of the queue:

1. A new node is created to hold the data.
2. The tail of the queue is updated using an atomic compare-and-swap operation to ensure thread safety.
3. Once the tail is successfully updated, the previous tail's `next` pointer is updated to point to the new node.

Pop Operation The `pop` method removes an element from the front of the queue:

1. The head of the queue is read, and the `next` node is obtained.
2. If the `next` node is `nullptr`, the queue is empty.
3. Otherwise, the data from the `next` node is moved to a unique pointer for return.
4. The head is then updated to point to the `next` node, and the old head is deleted.

Peek Operation The `peek` method allows inspecting the front element of the queue without removing it:

1. The `next` node of the head is read.
2. If the `next` node is `nullptr`, the queue is empty.
3. Otherwise, the data of the `next` node is accessed and returned.

2.4 Kafka Consumer and Producer

Kafka is a distributed streaming platform commonly used for building real-time data pipelines and streaming applications. It allows for high-throughput, low-latency data processing and is composed of producers, brokers, and consumers. A Kafka consumer is responsible for subscribing to one or more topics and processing the stream of records produced to them. In the implementation, the consumer connects to a specified Kafka broker and subscribes to a topic. It continuously polls the Kafka broker for new messages, processes the received data, and handles various states of the message consumption, such as detecting the end of a partition, handling timeouts, and logging errors. The consumer operates in a thread-safe manner and can gracefully shut down when required.

A Kafka producer is responsible for sending records to one or more Kafka topics. The producer connects to a specified Kafka broker and can send messages to a designated topic. It handles the configuration of the producer, the creation of topics if they do not already exist, and the sending of messages. The producer ensures that messages are delivered reliably and efficiently, managing errors and retries if necessary. The producer also includes a polling mechanism to handle message acknowledgments and errors, and it provides a flush method to ensure that all outstanding messages are sent before shutting down.

Both the consumer and producer are implemented using the librdkafka library, a client library for Apache Kafka which provides a high-level, easy-to-use API for interacting with Kafka, ensuring that the operations are efficient and reliable.

2.5 HTTP Client Implementation

The HTTP client is designed to interact with remote servers over HTTP and WebSocket protocols. Utilizing the Boost.Asio library for asynchronous network programming and Boost.Beast for HTTP and WebSocket communication, this client is capable of secure SSL/TLS connections, making it suitable for accessing APIs and streaming data.

The `HttpClient` class template is initialized with an I/O context, SSL context, host, protocol, and API key. Upon construction, the client resolves the host and establishes a connection using TCP. It then performs an SSL handshake to secure the communication channel. Any errors during these steps are logged for debugging purposes. The client can upgrade the existing HTTP connection to a WebSocket connection. This is done by initializing a WebSocket stream over the SSL stream and performing a WebSocket handshake with the specified endpoint. This allows for bidirectional communication with the server, enabling real-time data streaming. The client parses incoming JSON strings into Protobuf messages, ensuring type safety and efficient serialization. The parsed events are then converted back to JSON strings and pushed onto a lock-free, wait-free queue for further processing. This approach leverages the strengths of both Protobuf for data serialization and the lock-free queue for efficient multi-threaded access.

To interact with authenticated APIs, the client sends an authentication message upon connection. It can also subscribe to specific data streams by sending subscription messages. These operations ensure that the client receives the necessary data from the server. The client enters a continuous listening loop, reading data from the WebSocket connection. Each received message is parsed and processed in real-time, ensuring that the client can handle live data streams efficiently. The client also maintains an output file for logging received messages. This ensures that data is not only processed in memory but also persisted for future reference. The logger is used extensively to record the operational status and any errors encountered, providing valuable insights during debugging and monitoring.

2.6 Logger

The Logger class is a singleton logging utility designed to provide thread-safe logging capabilities with various log levels. It ensures that log messages are formatted consistently and output to the console in a synchronized manner. Below are the key features and functionalities of the Logger class:

Log Levels
The Logger class supports four log levels:

- **INFO**: Informational messages that highlight the progress of the application.
- **WARNING**: Warning messages indicating a potential issue.
- **DEBUG**: Debug messages for detailed diagnostic information.
- **ERROR**: Error messages indicating a failure in the application.

Singleton Pattern
The Logger class is implemented as a singleton to ensure that only one instance of the logger exists throughout the application. This instance is accessed using the `getInstance` method.

Thread-Safe Logging
To ensure thread safety, the Logger class uses a `std::mutex` to protect log operations. This prevents concurrent threads from interleaving their log messages, which could lead to corrupted log output.

Log Message Formatting
Log messages are formatted to include the timestamp and the log level. The `formatLog` method generates a formatted string containing the current date and time, the log level, and the log message. The timestamp is generated using the `std::chrono` library.

Logging Methods
The Logger class provides several overloaded `log` methods to handle different types of log messages:

- `void log(LogLevel level, const std::string& message)`: Logs a simple string message at the specified log level.
- `template<typename... Args> void log(LogLevel level, std::format_string<Args...> formatStr, Args&&... args)`: Logs a formatted message using `std::format` with the specified arguments.
- `template<typename T> void log(LogLevel level, const std::vector<T>& values)`: Logs each element of a vector individually at the specified log level.

Setting Log Level
The static method `setLogLevel` allows the log level to be set dynamically. This controls the verbosity of the logging output, ensuring that only messages at or above the current log level are displayed.

Example Usage To use the Logger, one must first obtain the instance and then call the appropriate `log` method. For example:

```
Logger& logger = Logger::getInstance();
logger.log(Logger::LogLevel::INFO, "This is an informational message.");
logger.log(Logger::LogLevel::ERROR, "This is an error message: {}", errorCode);
```

The Logger class thus provides a robust, flexible, and thread-safe logging solution suitable for Nexus.

2.7 Advanced C++ Features in

Nexus leverages several advanced C++ features to ensure robustness, performance, and maintainability.

Advanced Use of CMake and Conan for Dependency Management The build system is designed using CMake, with advanced features to manage dependencies and generate protobuf and gRPC sources. CMake is used to set up Nexus's configuration, including setting the C++ standard to C++20, finding required packages, generating protobuf and gRPC files, and defining the library targets. Conan is used for dependency management, ensuring all required libraries are available and correctly configured.

Memory Ordering in Concurrent Data Structures Nexus includes a lock-free, wait-free queue using atomic operations with specific memory orderings to ensure thread safety and performance:

Memory ordering ensures the correct visibility of changes across threads, with `memory_order_relaxed`, `memory_order_acquire`, and `memory_order_release` providing the necessary synchronization.

Producer-Consumer Threads The project uses producer-consumer threading to share information between threads efficiently. The producer generates data and places it into a queue, while the consumer retrieves and processes this data. This pattern ensures smooth data flow and prevents bottlenecks, facilitating concurrent data processing.

Using `void_t` for Type Traits Nexus also uses advanced type traits in certain instances to check if a type `T` has a specific member, such as `descriptor()` for protobuf messages:

```
template<typename T, typename = void>
struct is_protobuf : std::false_type {};

template<typename T>
struct is_protobuf<T, std::void_t<decltype(T::descriptor())>> : std::true_type {};
```

This technique uses `std::void_t` to create a substitution failure, effectively enabling compile-time checks and ensuring type safety and correctness when dealing with protobuf messages.

By leveraging advanced C++ features such as sophisticated CMake configuration, memory ordering for concurrent data structures, producer-consumer threading, and type traits with `void_t`, this project achieves high performance, scalability, and maintainability. These techniques are crucial for developing robust, high-performance systems in modern C++.

3 Results

3.1 Feature Extraction and Trading Signals

The provided persistent barcodes and persistence diagrams for the stock symbols MPW, CCL, and IWM illustrate the topological features extracted from their respective price data. Persistent barcodes display the birth and death times of topological features, with different colors representing different dimensions. Persistence diagrams plot these features, showing their significance based on their distance from the

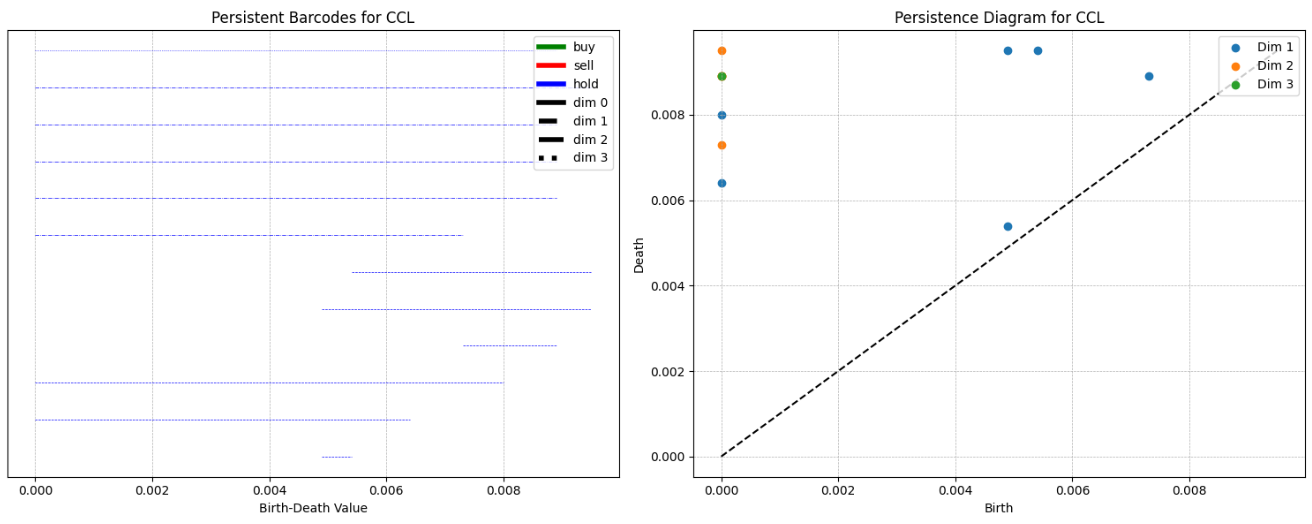

Figure 5: Persistent Barcode for CCL

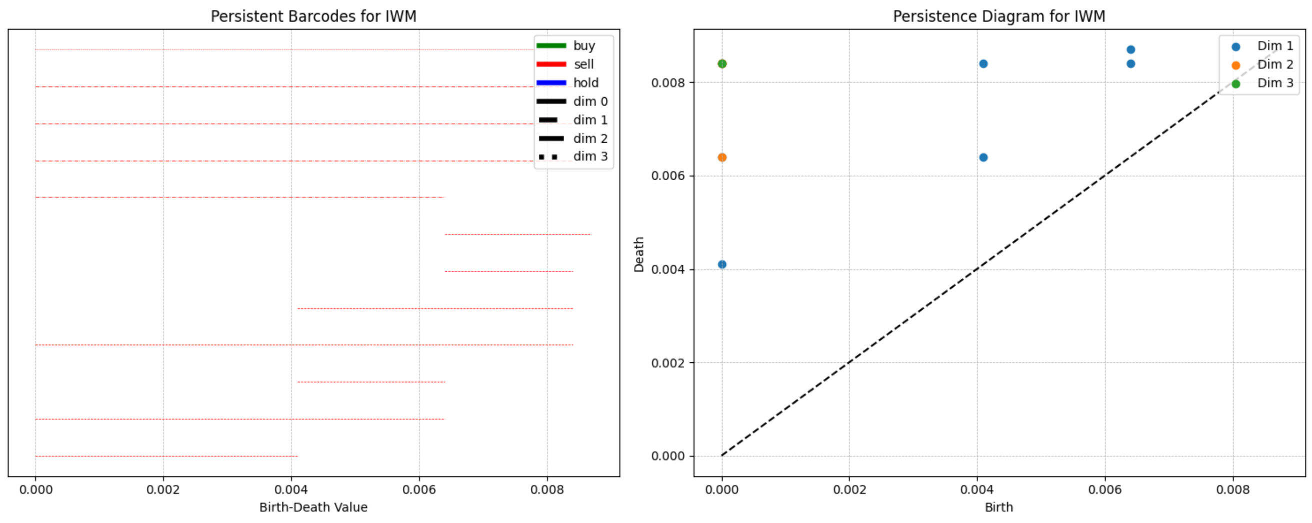

Figure 6: Persistent Barcode for IWM

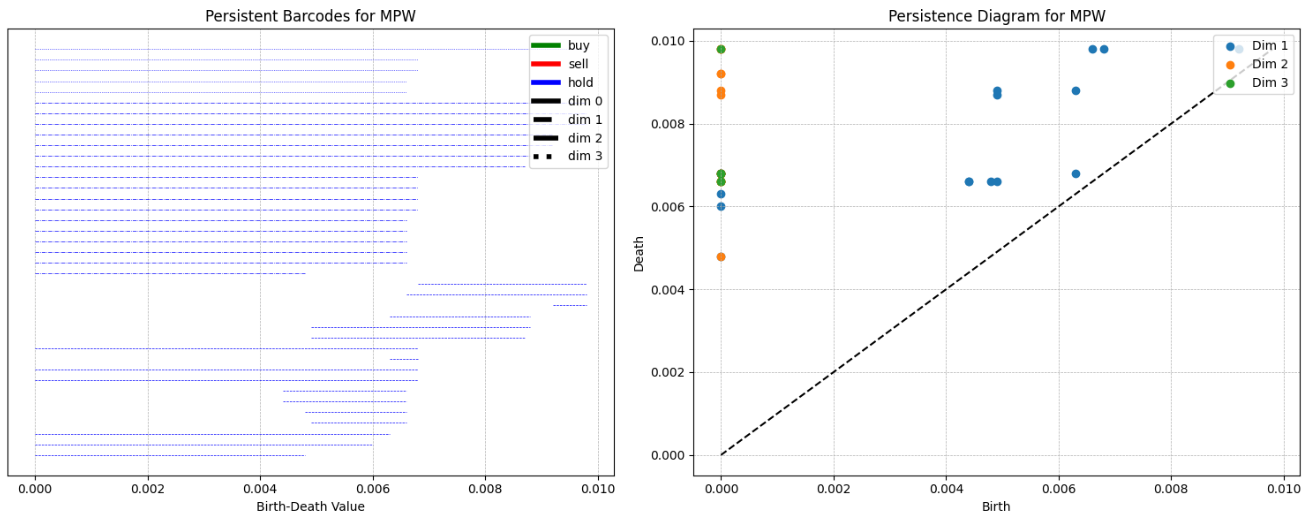

Figure 7: Persistent Barcode for IWM

diagonal. MPW and CCL exhibit a range of significant features across multiple dimensions, suggesting dynamic market conditions with various buy (green) and sell (red) signals. In contrast, IWM presents fewer and less significant features, indicating a more stable market with predominantly hold (blue) signals.

3.2 Future Analysis: Market-Wide Analysis Using TDA

In the future with significant computational resources I would like to continue this by studying the entire shape of the market. Analyzing the entire market as a whole using Topological Data Analysis (TDA) involves aggregating price data from multiple stocks to capture the global topological features of the market. By computing a barcode snapshot of the market, I can identify significant patterns and structures that are not apparent when analyzing individual stocks in isolation. This approach involves constructing a high-dimensional dataset where each point represents a stock's price or volume features over a given time period. The barcode snapshot, which captures the birth and death times of topological features, provides a comprehensive view of the market's behavior. Mathematically, this can be represented as:

$$\text{Market Barcodes} = \bigcup_{i=1}^{n} \text{Barcode}(\mathbf{x}_i, \tau, m)$$

where $\mathbf{x}_i$ denotes the feature vector of the i-th stock, τ is the time delay, and m is the embedding dimension. This union of barcodes across all stocks yields a global topological signature of the market, facilitating the identification of overarching trends and anomalies.

3.3 Results of Linear Model Training

The linear regression model was trained to classify stock trading signals based on topological features extracted from market data. The training and validation metrics provide insights into the model's performance and generalization ability. The model achieved a training loss of 0.2813 and a training accuracy of 92.67%. The low training loss indicates that the model is effectively minimizing the error on the training dataset. The high training accuracy demonstrates that the model has learned to correctly classify a significant majority of the training samples. The validation loss of 0.2665 and validation accuracy of 92.15% suggest that the model performs consistently well on unseen data. The validation loss being slightly lower than the training loss indicates that the model is not overfitting and generalizes well to new data. The validation accuracy, which is close to the training accuracy, further supports the model's robustness and reliability. The close alignment between training and validation metrics shows that the model has effectively captured the underlying patterns in the data without over fitting. The minimal difference between training and validation losses also suggests that the model is not under fitting and has an appropriate capacity to learn from the data. With high accuracy in both training and validation phases, the model shows potential for generating actionable trading signals (hold, buy, sell) based on topological data analysis (TDA) and various market structures. The reliable performance across different datasets implies that the model can be trusted to make informed trading decisions in real-world scenarios. This can significantly enhance the trading strategy by providing accurate predictions based on complex market data. While the results are promising, further improvements can be explored by fine-tuning hyperparameters, incorporating additional features, and experimenting with more complex models. Ensemble methods or hybrid models combining linear regression with other techniques might also be considered to enhance prediction accuracy and robustness.

Overall, the trained linear model demonstrates strong performance, indicating its efficacy in classifying trading signals derived from topological features of the market.

3.4 Mathematical Tools for Market Analysis

In addition to Topological Data Analysis (TDA), various other mathematical tools can be used to analyze the financial markets to offer novel insights. One notable approach is using fluid dynamics to capture the stochastic nature of market movements. The market, much like the waves of the ocean, exhibits complex, unpredictable patterns that can be modeled using principles from fluid dynamics and stochastic processes.

Fluid Dynamics and Stochastic Processes The market's price movements can be likened to the random flow of fluids, where prices fluctuate continuously due to the interaction of numerous market participants. Fluid dynamics, particularly the Navier-Stokes equations, can be adapted to model these fluctuations. The Navier-Stokes equations, which describe the motion of fluid substances, can be written as:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

where $\mathbf{u}$ is the velocity field, t is time, ρ is the fluid density, p is the pressure, ν is the kinematic viscosity, and $\mathbf{f}$ represents external forces.

In the context of financial markets, $\mathbf{u}$ can be interpreted as the price velocity, and $\mathbf{f}$ as external market influences (e.g., economic news, geopolitical events). By solving these equations, I can simulate the fluid-like behavior of market prices, capturing their stochastic nature.

Just as the ocean's surface is influenced by various factors such as wind, tides, and underwater currents, the financial market is shaped by myriad forces, both macroeconomic and microeconomic. Market prices rise and fall like waves, with periods of calm followed by sudden turbulence.

References

[1] G. Singh, F. Mémoli, and G. Carlsson, *Topological Methods for the Analysis of High Dimensional Data Sets and 3D Object Recognition*, Institute for Computational and Mathematical Engineering, Stanford University, California, USA, and Department of Mathematics, Stanford University, California, USA, 2007.

[2] N. Ravishanker and R. Chen, *Topological Data Analysis (TDA) for Time Series*, Journal of Time Series Analysis, 2019.

[3] J. Barnes and P. Hut, *A hierarchical $O(N \log N)$ force-calculation algorithm*, Nature, vol. 324, pp. 446-449, 1986.

[4] F. Chazal and B. Michel, *An Introduction to Topological Data Analysis: Fundamental and Practical Aspects for Data Scientists*, February 26, 2021.