|  |
| --- |
| Assignment 3  Big Data Engineering |
| Students:  Adam Downey (11718780)  Mark Brackenrig (12964298)  Hemendra Patil (13835246)  Werner Schott (10930194)  ) |



Table of Contents

[Overview 3](#_Toc74500023)

[Project Overview 3](#_Toc74500024)

[Part 1 – Produce Data 5](#_Toc74500025)

[Part 2 – KSQL 6](#_Toc74500026)

[Part 3 – Consume/Transform Data with Spark 9](#_Toc74500027)

[Spark Data frames 9](#_Toc74500028)

[Window Stream 12](#_Toc74500029)

[Spark Query 14](#_Toc74500030)

[Parquet Sink 14](#_Toc74500031)

[Data Visualisation 15](#_Toc74500032)

[Part 4 – Machine Learning Model 16](#_Toc74500033)

[Business Understanding 16](#_Toc74500034)

[Data Understanding 16](#_Toc74500035)

[Data Preparation 16](#_Toc74500036)

[Modelling 18](#_Toc74500037)

[Evaluation 18](#_Toc74500038)

[Recommendations 18](#_Toc74500039)

[Challenges/Learnings 20](#_Toc74500040)

[Next Steps & Recommendations 22](#_Toc74500041)

[Works Cited 23](#_Toc74500042)

[Appendix 24](#_Toc74500043)

[Appendix A – generate\_stocktrades\_stream function 24](#_Toc74500044)

[Appendix B – generate\_side\_stream function 25](#_Toc74500045)

[Appendix C – transform\_data function 26](#_Toc74500046)

[Appendix D – generate\_spark\_stream function 26](#_Toc74500047)

[Appendix E – Visualise function 27](#_Toc74500048)

# Overview

This project has been focused on producing a streaming application that is able to perform real time analytics and deploy a machine learning model for real time predictions. For this project we have focused on developing this streaming application using a generated data set that replicates stock trades through a continuous stream. The machine learning model has then been applied to predict whether a specific observation will be a ‘BUY’ or ‘SELL’ trade based-off the price, quantity and account. The intention is that the architecture, design and model should provide the start of a framework that could then be potentially applied to real streaming stock data in the future, whether for the application of the machine learning model, or other potential future models.

### Repository Structure

The repository follows the data science ‘cookie cutter’ repository repo structure. The following folders and files are important:

* **data/parquet\_output –** Folder containing parquet sink
* **models/ -** Folder containing pipeline and machine learning models
* **reports/ -** Folder containing word report and html version of final notebook
* **notebooks/ -** Folder containing relevant notebooks
* **notebooks/Assignment\_3\_final.ipynb –** final notebook for assignment
* **reports/Report.docx –** Final report

## Project Overview

The development of the application can be broken down into four key components:

* Connecting to a data source
* Developing KSQL queries
* Streaming using Spark
* Machine Learning using Spark

### Connecting to Data Source

Our project has been built off a generated dataset (rather than real-world) that simulates a continuous stream of stock trades. The KSQL Datagen application has been used to generate mock data through the ‘stock trades’ profile to send data into the ‘topic’ in Kafka.

### KSQL Queries

Creation of two streams and two aggregation tables using the KSQL DB stream processing capabilities in the Confluent Control Centre. This includes the generation of a raw data stream from the Kafka topic, two streams (one each) for the ‘BUY’ and ‘SELL’ side trades and aggregation tables for both the ‘BUY’ and ‘SELL’ side streams that summarises every 60 seconds of trades with tumbling windows.

### Spark Streaming

creation of a series of streaming data frames, window streams and queries based on the Kafka streams that had been created in confluent. This was used to make the data available in spark to support further modelling, visualisation and provide the foundation of the machine learning model. This included the creation of an output data ‘parquet sink’ to provide the static data set for the machine learning model as well as the development of a visualisation that shows changes in the underlying data being streams.

### Spark Machine Learning Model

Development of a machine learning model within the Spark environment that could be used to predict whether a trade will be on the ‘BUY’ or ‘SELL’ side based on the quantity, account and price used. This was then deployed to the live streaming data to make real time predictions. This model then provides the complete framework for producing future prediction models in a live environment that could use real time streaming data.

# Part 1 – Produce Data

Our project has been built around generated stock trade data that uses the KSQL Datagen application. This data source has been set up in the Confluent Control Centre as a connector. The steps to be followed to setup this data connector are as follows:

1. Open a browser and go to <http://localhost:9021/>
2. Select the available cluster
3. On the menu bar, select Connect
4. Click on the connect-default cluster in the Connect Clusters list.
5. Click on Add connector
6. Select DatagenConnector

To generate the same format data stream the following settings should be used:

* **Name:** connector\_stock\_trades
* **Connector.Class:** io.confluent.kafka.connect.datagen.DatagenConnector
* **Key.Converter:** org.appache.kafka.connect.storage.StringConverter
* **Max.Interval:** 100
* **Quickstart:** Stock\_Trades

This data generator will then leverage the Stock Trades profile to simulate the trading transactions.

The generated data stream will include the following key generated fields:

* **Side:** ‘BUY’ or ‘SELL’ representing whether the captured transactions relates to an order to buy or sell a specific stock
* **Quantity:** the volume of the specified stock to be included in the trade as an integer between 1 and 5,000
* **Symbol:** a set of characters representing the ‘stock code’ with each simulated stock having its own code value (e.g. ‘ZVV’) as a string value based on defined stock symbol options
* **Price:** the buy or sell price for the specific trading transaction as an integer between 5 and 1,000
* **Account:** the specific account value that is assigned to the trade as a string based on a list of defined potential account values
* **UserID:** an identifier field representing the initiator of the specific trade transaction as a string based on a defined range of users (User\_1 through to User\_9)

It is important to note that as this is being generated through the KSQL data generator all values within these fields are simulated outputs based on the define random generation patterns within the ‘Stock\_Trade’ profile.

This datagen output has been assigned to the Kafka Topic ‘stocktrades’ which will be used in the later KSQL queries.

# Part 2 – KSQL

As part of the data and streaming framework exploration a series of KSQL queries have been established within the ksqlDB capabilities within the Confluent control centre. These queries also provide some of the foundation for the Spark consumption/transformation and machine learning build in the later part of this project. The structure of the streams and tables that have been established are as per the flows shown below:

A picture containing diagram

Description automatically generated

Figure 1: KSQL Stream Flow

Creation of these streams and tables has relied on the use of KSQL code in the ksqlDB editor accessible through the Confluent Control Centre. To replicate these streams:

1. Open a browser and go to <http://localhost:9021/>
2. Select the available cluster
3. On the menu bar, select ksqlDB
4. Enter the code provided for each query in the ‘Editor’ tab and select ‘Run query’

### Steam 0.1 - STOCKTRADES

The first stream that has been created has been to access the values streamed through the ‘stocktrades’ topic that is being created with the underlying data generator. This stream contains no filtering and is focused on bringing through the raw data in the ‘AVRO’ format. The following KSQL code has been used to create the stream:

CREATE STREAM STOCKTRADES

   (SIDE STRING, QUANTITY INTEGER, SYMBOL STRING, PRICE INTEGER, ACCOUNT

    STRING, USERID STRING)

       WITH (KAFKA\_TOPIC='stocktrades', VALUE\_FORMAT='AVRO');

Stock trades query in AVRO format

### Steam 0.2 - STOCKTRADES\_JSON

The second stream that has been created is used to convert the original raw stream into the JSON format. While not important for the ksql components of this project, this conversion is necessary to align the data format with what is required for the SPARK components. Again this stream contains no filtering and relies on querying the complete raw stream (‘STOCKTRADES’) with an updated ‘VALUE\_FORMAT’ as ‘JSON’. For consistency of the data used, this stream has also been used as the basis for the creation of the filtered streams and tables. The complete list of fields and their formats for the generated data - excludes metadata - can be found in the description of the generator above. The following KSQL code can be used to create this stream:

CREATE STREAM STOCKTRADES\_JSON WITH (KAFKA\_TOPIC='STOCKTRADES\_JSON', VALUE\_FORMAT='JSON') AS SELECT

\* FROM STOCKTRADES STOCKTRADES

EMIT CHANGES;

Stock trades JSON query

### Stream 1 - SELL\_TRADES

The first filtered stream has been created to show only ‘SELL’ side trades, that is transactions that have been initiated to sell a specific stock at a specific value. This stream includes all the original fields of ‘STOCKTRADES\_JSON’, except the SIDE field and has been filtered for where the SIDE field in the original table is equal to ‘SELL’. The following KSQL code can be used to create this stream:

CREATE STREAM SELL\_TRADES WITH (KAFKA\_TOPIC='SELL\_TRADES', VALUE\_FORMAT='JSON') AS SELECT

STOCKTRADES\_JSON.QUANTITY QUANTITY,

STOCKTRADES\_JSON.SYMBOL SYMBOL,

STOCKTRADES\_JSON.PRICE PRICE,

STOCKTRADES\_JSON.ACCOUNT ACCOUNT,

STOCKTRADES\_JSON.USERID USERID

FROM STOCKTRADES\_JSON STOCKTRADES\_JSON

WHERE (STOCKTRADES\_JSON.SIDE = 'SELL')

EMIT CHANGES;

Sell Trades query

### Stream 2 - BUY\_TRADES

The second filtered stream has been created as the inverse of ‘Stream 1’ showing only the ‘BUY’ side trades, that is where a transaction has been initiated to purchase a quantity of stock at a specific price. This stream includes all the original fields of ‘STOCKTRADES\_JSON’, except the SIDE field and has been filtered for where the SIDE field in the original table is equal to ‘BUY’. The following KSQL code can be used to create this stream:

CREATE STREAM BUY\_TRADES WITH (KAFKA\_TOPIC='BUY\_TRADES', VALUE\_FORMAT='JSON') AS SELECT

STOCKTRADES\_JSON.QUANTITY QUANTITY,

STOCKTRADES\_JSON.SYMBOL SYMBOL,

STOCKTRADES\_JSON.PRICE PRICE,

STOCKTRADES\_JSON.ACCOUNT ACCOUNT,

STOCKTRADES\_JSON.USERID USERID

FROM STOCKTRADES\_JSON STOCKTRADES\_JSON

WHERE (STOCKTRADES\_JSON.SIDE = 'BUY')

EMIT CHANGES;

### Table 1 - AGG\_SELL\_ORDERS

Building off the filtered streams, the tables that have been created look to calculate aggregated statistics for the buy or sell-side stream over a defined tumbling window. Table 1 includes the aggregation of sell orders through a 60-second tumbling window for each different stock symbol. The fields include:

* **Symbol:** the characters identifying the specific stock which is also the key grouping field that has been used for this aggregation
* **Quantity\_Agg:** the sum of the quantity of trades (in this case ‘SELL’ transactions) for the defined window and stock symbol
* **Price\_Avg:** the average trade price across all the transactions (note in this case this is not weighted by the quantity but the average across trade lots)
* **Value\_Traded:** the total value of the stock in transactions across the window calculated by multiplying the value and quantity for each individual transactions and summing within the group

This table has been built with reference to the ‘SELL’ order filtered stream. The following KSQL code can be used to create this stream:

CREATE TABLE AGG\_SELL\_ORDERS WITH (KAFKA\_TOPIC='AGG\_SELL\_ORDERS', VALUE\_FORMAT='JSON') AS SELECT

SELL\_TRADES.SYMBOL SYMBOL,

SUM(SELL\_TRADES.QUANTITY) QUANTITY\_AGG,

AVG(SELL\_TRADES.PRICE) PRICE\_AVG,

SUM((SELL\_TRADES.QUANTITY \* SELL\_TRADES.PRICE)) VALUE\_TRADED

FROM SELL\_TRADES SELL\_TRADES

WINDOW TUMBLING ( SIZE 60 SECONDS )

GROUP BY SELL\_TRADES.SYMBOL

EMIT CHANGES;

Aggregated sell orders query

### Table 2 - AGG\_BUY\_ORDERS

Inverse of the ‘AGG\_SELL\_ORDERS’ table calculating the same aggregated statistics but built off the ‘BUY’ order filtered stream. This contains the exact same fields as Table 1 with aggregation calculations based on the same logic. The following KSQL code can be used to create this stream:

CREATE TABLE AGG\_BUY\_ORDERS WITH (KAFKA\_TOPIC='AGG\_BUY\_ORDERS', VALUE\_FORMAT='JSON') AS SELECT

BUY\_TRADES.SYMBOL SYMBOL,

SUM(BUY\_TRADES.QUANTITY) QUANTITY\_AGG,

AVG(BUY\_TRADES.PRICE) PRICE\_AVG,

SUM((BUY\_TRADES.QUANTITY \* BUY\_TRADES.PRICE)) VALUE\_TRADED

FROM BUY\_TRADES BUY\_TRADES

WINDOW TUMBLING ( SIZE 60 SECONDS )

GROUP BY BUY\_TRADES.SYMBOL

EMIT CHANGES;

Aggregated Buy Orders query

# Part 3 – Consume/Transform Data with Spark

The next component of the project has focused on utilising the Spark streaming capabilities to build a number of features on the streaming data set. The Spark streaming components can be broken down into:

* **Data Frames** - a series of Spark streaming dataframe that have been created to support the different components to be explored
* **Window Stream** - a window-based stream that utilises the watermark and window streaming capabilities of Spark
* **Query** - a Spark Query that has been used to produce the output data used for the model
* **Parquet Sink** - a function/code that can be used to output the Spark Query into a Parquet file as the basis for the Machine Learning model

**Data Visualisation** - a refreshable visualisation that can be used to visualise changes in the underlying streaming data

## Spark Data frames

### Query 1

To begin the Spark Streaming component of the project we have built two functions for ingesting data from the Kafka instances and the streams that have been created in KSQL. The first stream is focused on ingesting the raw data stream that has been created and utilises a function that has been created as ‘generate\_stocktrades\_stream’. The full function can be found in [Appendix A](#_Appendix_A_–).



This function takes one argument ‘keep\_stream’ identifying:

* **False -** only a queryable view of the stream should be returned
* **True -** both the queryable view and the streaming data frame should be returned by the function

The first part of the function is focused on ‘subscribing’ to the raw JSON stream of data in Kafka and creating a streaming data frame from this source. This has been set to begin streaming from the ‘latest’ for the ‘startingOffsets’ to limit the demand on memory in Spark which had been causing issues for some members of the team.

Text

Description automatically generated with medium confidence

Figure 2: Reading kafka stream

The next part of the function looks to convert the raw data frame from the ‘STOCKTADES\_JSON’ steam source into string values for both the ‘key’ and ‘value’ components of the stream.

Text

Description automatically generated

Figure 3: Converting JSON fields to String

The function then defines the schema that will be used to produce a JSON formatted stream that will be the next step in the stream transformation. This schema aligns to the original data format outlined as part of the production of the generated data for the raw stream.

Text

Description automatically generated

Figure 4: defining JSON value structure

The final step as part of the function stream is to then convert the data into the JSON format and then flatten this into the final format that will be used for the stream. As part of the flattening process, additional columns are brought into the final view (from the stream event metadata) including:

* **Event\_Key -** the event key for the specific observation in the streaming data
* **Event\_Topic -** the topic as the source of the underlying data
* **Event\_Timestamp -** the timestamp associated with the capture of the specific event (generated stream observation) in the topic

Text

Description automatically generated

Figure 5: writing to a JSON dataframe

The final return component of the function is then driven by the input for the ‘keep\_stream’ option. Under both it will return a queryable view named ‘stocktrades\_view’, however, if the ‘keep\_stream’ option is set to ‘True’ it will also return the streaming data frame itself.

Text

Description automatically generated

Figure 6: Returning Stocktrades stream

Putting this function into practice is as simple as assigning a variable as the generation function. We are then able to visualise the outputs from the streaming data frame by using the view that is created as part of the stream. Where the stream is no longer required (i.e. not to be used by other functions in our code) we will also stop the stream in each instance as part of conserving memory used by Spark.

Table

Description automatically generated

Figure 7: generate\_stocktrades\_stream in action

### Query 2

The second data frame stream function we have established is focused on accessing the filtered ‘BUY’ or ‘SELL’ streams that were set up in the KSQL streams above. This function follows the same structure as the raw data streaming data frame however includes an additional input to allow the function to generate both the ‘BUY’ or ‘SELL’ side streams.

Key differences include the ability to input a ‘SIDE’ as a parameter of the function which will be either ‘BUY’ or ‘SELL’. This will enable the function to determine which stream it should be accessing from Kafka as the source to create the streaming data frame.

Text

Description automatically generated

Figure 8: function accesses 'side' streams instead of main stream

This function then uses the same code as the original function until it gets to the return function at which point it will also dynamically create the view name based on the input. As this function is also not used for the window stream or parquet sink, it also does not include the option to return the streaming data frame, just the queryable view. However, this can easily be updated if the need arises in the future.

Text

Description automatically generated

Figure 9: returing stream

## Window Stream

The next component of the project looks to create a ‘windowed’ stream that includes aggregation calculations across a defined window and slide. The stream we have developed includes the following parameters:

* **Window Duration:** aggregations and calculation are across a 60 second time period
* **Slide Duration:** windows shift the calculation of the aggregation every 10 seconds

**Chart

Description automatically generated**

Figure 10: Window/slide duration diagram

The window stream is started by generating a stream of the raw stock trades and establishing the parameters as per above.

Text

Description automatically generated

Figure 11: Restart of Stream

We then created the windowed stream using these parameters and a ‘1-minute’ watermark. The windowed stream contains the following fields:

* **Window:** the start and end timestamp for each window
* **Symbol:** the specific stock symbol for each aggregation which has been used as the grouping dimension
* **No\_Trades:** the number of trade transactions for the grouping that has occurred during the window
* **Tot\_Quantity:** the total quantity of stocks that have been traded across the window for the specific symbol (buy and sell)

**Avg\_Price:** the unweighted average price the stock has been traded across the window (i.e. average price across the number of trades, not weighted by the number of stocks in each trade)

Text

Description automatically generated

Figure 12: Aggregated Dataframe

Similar to the other streams that have been created, we then create a queryable view to view the contents of the stream. As these streams are no longer required, post the query both the raw\_stream created and the windowed\_aggregation stream are then also terminated.

Table

Description automatically generated

Figure 13: Window Aggregation Stream

## Spark Query

Outside of the queries that have been developed for other components of the project, we have also developed a one off query to demonstrate the functionality. This query leverages the SQL query capabilities of Spark and looks to identify all BUY or SELL side trades in the streaming data set where the price is greater than 800. The query was set up based on the general raw data stream and can be replicated with the code below.

Graphical user interface, text, application, email

Description automatically generated

Figure 14: Query Example

## Parquet Sink

To support the development of the machine learning model, we required an extract of the streaming data that could be used to train the model. To provide this extract we have developed the capability of a Parquet Sink that is able to take the streaming data and write this to a file, in this case the Parquet format. Like the previous queries/functions, this has been based on the ‘generate\_stocktrades\_stream function that was developed.

Graphical user interface, text, application

Description automatically generated

Figure 15: Creating the Parquet Sink

## Data Visualisation

The final Spark component that has been developed (prior to the Machine Learning Model) is a visualization to show the changes in the data being captured through the stream. This visualisation has been set up as a function that is able to set the number of iterations and the time between each iteration to be processed (see [Appendix E](#_Appendix_E_–)). The visualisation is focused on demonstrating the total sum of the quantity of stock within the trades based on the specific ‘UserID’ they are associated with. This visualisation makes use of the matplotlib and seaborn packages for Python.

The visualisation itself contains two axes, vertical being the UserId and horizontal the total quantities of trades that have been made. The function contains default values for the two parameters (iterations and sleep timing) and an example using these two default parameters is shown below. In the live environment this visualisation will continue to update based on these parameters.

A picture containing graphical user interface

Description automatically generated

Figure 16: Function dynamically generating visualisations’

# Part 4 – Machine Learning Model

## Business Understanding

Since our stream was based on random stock trades, we felt that it would be interesting to see if we could predict whether the given trade was a ‘buy’ or a ‘sell’ trade based on the volume, price, user, and stock code. We felt this would be an interesting area, as our previous assignment covered a regression model, and this would allow us to explore classification models in spark.

Since our data was randomly generated, we did not have high expectations of the model. Nevertheless, we wanted to see whether the data generator produced correlated random data, or whether the data was truly random. For this model, we will use the AUROC metric (AUC) for the evaluation of the model.

## Data Understanding

The data was produced using a random data generator to create a spark streaming data frame. This data aimed to emulate streams of stock trades. We considered five independent variables for modelling:

* Price (int)- the price of the stock at the time of trade
* Quantity (int)- the amount of stock traded
* Symbol (string) - the stock code
* Account (string) - the account the trade was made from / to
* Userid (string) - the user making the trade.

Table

Description automatically generated

Figure 17: Summary stats from parquet file

 The target variable, ‘side’, consisted of two levels - ‘BUY’ or ‘SELL’.

## Data Preparation

Data preparation consisted of 3 main steps:

* Encoding the independent variables into an applicable data type.
* Scaling independent variables using the StandardScaler method
* Encoding the dependent variable to an appropriate data type.

### Encode Independent Variables

The symbol, account and userid fields were all string variables that could be considered categorical variables. To encode these variables, they first needed to be string indexed (so that they became integers) and then one hot encoded (OHE).

Text

Description automatically generated

Figure 18: Encoding pipeline for Machine Learning

Once these variables were encoded as  OHE variables, they were then assembled using the VectorAssembler method, which was then passed into a spark pipeline.

Graphical user interface, text, application, email

Description automatically generated

Figure 19: Full pipeline for Machine learning

The result created a new data frame with a ‘features’ column which represented the independent variables.

A picture containing text, device, screenshot

Description automatically generated

Figure 20: data frame after encoding pipeline

### Scale Independent Variables

As stated above, the independent variables were all at different scales. This can cause logistic regression to produce unexpected results. To avoid this, we used the StandardScaler method, which employs standard normalization to create a new ‘ScaledFeatures’ column.

### Encode Dependent Variable

The final step was to encode the target variable as an integer instead of a string to allow the model to be built. To do this, we used the StringIndexer method - converting ‘BUY’ into 1 and ‘SELL’ into 0.

Table

Description automatically generated

Figure 21: Dataframe with scaled features and encoded target

## Modelling

### Split into training and test sets

The first step of the modelling process is to split the data into training and test sets. Since there did not appear to be a specific order effect on the data, we decided to use a random 70%/30% training-test split on the data.

Graphical user interface, text

Description automatically generated

Figure 22: Splitting data into training and test set

### Model selection

We elected to only use logistic regression for modelling. After inspecting the data, we identified that the independent variables were uncorrelated with each other, making logistic regression a good candidate for modelling. We also wanted an explainable model, if, in fact, the data was not completely random.

We decided to maintain the default value for the regularization parameter ( λ=0.3) but set the elasticNetParam (α) to 0. We decided to set α to 0 to ensure that not all parameters shrunk to 0 as this would output the same value for every record.

## Evaluation

### Results

As expected, the model did not perform well. The AUC of the final model was 0.502 - barely above random chance. In addition, the accuracy of the model was quite weak, at just over 50%.

Text

Description automatically generated with low confidence

Figure 23: confusion matrix for test set

## Recommendations

In practice, a data scientist would never use a random data generator to produce a machine learning model. This model should not be deployed into a production environment. However, due to the task requirements, the model will be deployed to a Kafka stream anyway.

An improvement on this model would be to use a different data source for modelling. For example, using Alpaca markets[[1]](#footnote-1) would provide real-world streaming data for this task. Even though real data does not guarantee a viable model, results from this model would at least be interesting.

### Deployment

To deploy the model we created a new Kafka stream that added predictions to the original stream. This was done through the creation of two functions:

* ***Transform\_data -*** function which prepared a stream for modelling
* ***generate\_spark\_stream -*** function that inputs the prediction on the stream and publishes the stream to Kafka.

The transform data function (see [Appendix C](#_Appendix_C_–)) loads all the transformation models prior to modelling and applies that to the input dataframe. This dataframe is then provided to the generate\_spark\_stream function (see [Appendix D](#_Appendix_D_–)), along with a list of relevant columns to publish to the kafka topic.

Only sending a subset of columns was chosen because the features and scaledFeatures columns are not very interpretable by a human. Once the stream has been created, the topic is accessible through the confluent control center as shown below. This method provides flexibility by allowing multiple users (such as different departments) to access this data easily.

Graphical user interface, table, website

Description automatically generated

Figure 24: Prediction stream in confluent

# Challenges/Learnings

### Spark Memory Issues

#### Challenge

As with previous working with Spark in the virtual environment there continued to be issues around the memory that has been allocated to the environment and the processing required. This was particularly the case when looking to run queries on the Spark streaming data frames where they contained a significant amount of data.

#### Resolution

To temporarily resolve this challenge, the creation of the streaming data frame was altered to change the offset in the settings to the latest data only. This helped to limit the volume of data and processing required to facilitate the query and print-out/display of the data. However, this was not required for all members of the group as the issue continued to be related to the specific memory constraints of an individual's virtual environment.

#### Future Approach

In the future this issue can be resolved through two key steps:

* Defining a consistent environment for all members of the group so any issues and resolution can be applied uniformly across all members
* Working outside the virtual environment or with an environment that is able to access more memory so that the limitations are not reached on what has been a relatively low intensity processing requirement

### Docker set up on cloud environment

#### Challenge

Again, due to the differences in hardware and operating systems used across the members of the group, some challenges were identified in running the Docker environment. While the Docker setup was able to work on most machines, it was found to not operate on a Mac with an M1 chip that was being used by one member of the team. In addition, some challenges were found in being able to run the Docker setup on the cloud environment to work around this issue[[2]](#footnote-2).

#### Resolution

The team member with the M1 chip was able to set up the Docker container to run on a Linux EC2 instance. While the container was able to run (unlike on the local M1 machine), the user was still unable to run select queries without adjusting the docker-compose file for a different setup to the other team members.

#### Future Approach

While the approach provided the ability for the specific member to run the Docker instance, it did result in environment inconsistencies that could cause some issues in the replicability of code developed across the group members. In future, it may be necessary to move all members of the group to the cloud environment so there is a consistent approach, even if this may be initially more difficult than the local instance.

Graphical user interface, text, application

Description automatically generated

Figure 25: Example of query failure

### Developing Time Series Models with Spark Streaming

#### Challenge

An opportunity identified for the model was to develop a ‘time-series’ based prediction model that is able to predict the future trading price. This would be a particularly useful approach for applying the framework that had been developed to real world data. However, exploration of this modelling found that it was extremely difficult to develop a time-series based approach, and functionality such as the ‘lag’ function in Spark that could be used to develop the right inputs for the model were not possible in the case of Spark streaming data.

#### Resolution

Unfortunately within the scope of this current project it was not possible to overcome this challenge within the time available. As a work-around an alternate model was developed to provide a different type of prediction that is not reliant on the functions that are not supported by Spark streaming data.

#### Future Approach

With more time available to support an exploration of applying time series modelling it may be possible to find an appropriate work around. Some beginnings of this approach had been identified through research however more time was required to understand if these functions (e.g. mapGroupsWithState) would work as well as what would be required to configure these correctly with the data set. Further understanding would also be required as to how to manage the handling of data in memory (to provide access for the lag equivalent) and ensure this does not raise the same issues with Spark memory limitations.

# Next Steps & Recommendations

### Real World Data Application

This project provides the working and framework for adding machine learning predictions to a data stream. However, the choice of data set (generated) limits the immediate ability to provide meaningful results through what has been developed. What it does provide though is the basis and approach for applying this to a real-world data set. The fields that have been used as well as the quantity and velocity of the data created will be similar to what could be seen in a real-world trading example. This means it would be possible to apply this approach to a real-world trading data set, whether this is trades from equity markets or commodity marks such as the Alpaca markets.

### Modelling Approach

Through this project we were also limited in the types of models we were able to develop and their usefulness in a real-world situation. While we were able to produce a model that could be used to predict whether a trade would be a ‘BUY’ or ‘SELL’ transaction, in a real-world environment this would be information expected to already exist when the trade is initiated. To expand on what has been developed we will need to find opportunities to apply the ‘blueprint’ for the trading streaming data and develop a more insightful model for application. This may require us to find information that is not contained within the transaction itself or could otherwise look to more time-based measures (predicting a known value in the future) which would be more valuable for an application such as automated trading.

# Works Cited

Confluent Inc. (2021, April 8). *KSQL Quickstart Example not working with v5.3.1 in UI*. Retrieved from GitHub: https://github.com/confluentinc/ksql/issues/3374

# Appendix

## Appendix A – generate\_stocktrades\_stream function

Text

Description automatically generated

## Appendix B – generate\_side\_stream function

Text

Description automatically generated with medium confidence

## Appendix C – transform\_data function

Text

Description automatically generated

## Appendix D – generate\_spark\_stream function

Graphical user interface, text, application, email

Description automatically generated

## Appendix E – Visualise function

A picture containing graphical user interface

Description automatically generated

1. (Alpaca Markets, 2021) [↑](#footnote-ref-1)
2. (Confluent Inc., 2021) [↑](#footnote-ref-2)