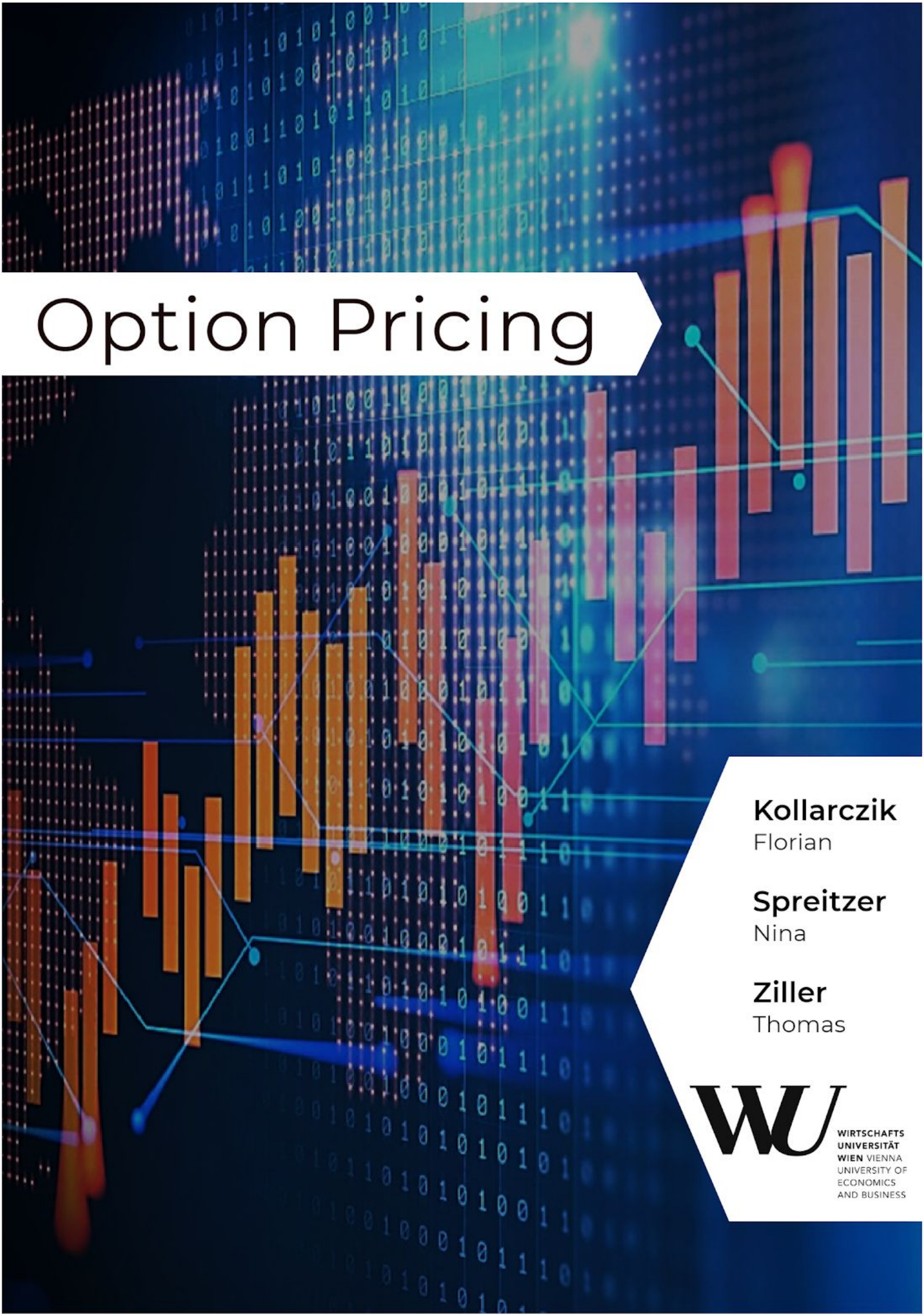


Option Pricing

Kollarczik
Florian

Spreitzer
Nina

Ziller
Thomas

WU
WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Vienna University of Economics and Business
Data Processing 2 Project - Team 3
Dr. Sabrina Kirrane
Winter Term

Introduction and Project Overview

Option Pricing is a project carried out by Kollarczik Florian, Spreitzer Nina and Ziller Thomas in regard to the Course "1508 Data Processing 2: Scalable Data Processing, Legal & Ethical Foundations of Data Science", led by Dr. Kirrane Sabrina. Its objective is to evaluate and compute financial options in real-time by streaming real-time stock data via Intrinio's API to later be processed and evaluated using the widely used Black-Scholes Model. The gathered raw stock data will include metrics such as closing-/ opening prices, high/low, volume, the exact date etc. For the pricing purposes the adjusted closing price – adjusted for both dividends and splits – will be taken for the calculations. The processing and analytics part will be done in Python 3 applying Apache Spark, Apache Kafka as well as necessary and compatible libraries.

This project is meant to make the students more comfortable within the Hadoop ecosystem, especially with Spark and Kafka. Since we got to choose our own topic and were able to approach the project solely how we wanted to, we managed to combine our tasks with a common interest of ours, the financial sector. Having said that, we are determined to continue improving our model and measuring results even after the final submission.

A big data-project was set as the target for every group and even though our project is not quite dealing with big data as of now, we designed it to be scalable and therefore able to manage every data input as long as adequate processing power is being used. Furthermore, our model not only works with one specific stock, but facilitates every single asset traded on the IEX (Investors Exchange). We decided to choose Apple (ticker: AAPL) for our purposes, as this is not only the highest valued company in the world, but also traded very extensively with a frequently changing price. In other words, this allows for effortless and visible changes in option prices, even though they might still be relatively small compared to the actual price difference, as options are non-linear, derivative financial instruments.

Definition Option according to Investopedia

"Options are financial instruments that are derivatives based on the value of underlying securities such as stocks. An options contract offers the buyer the opportunity to buy or sell—depending on the type of contract they hold—the underlying asset."[1]

[1] https://www.investopedia.com/terms/o/option.asp

Black-Scholes Model

In order to compute option prices, we employed the Black-Scholes Model. The model covers the dynamics of derivative financial instruments – primarily used for options – on financial markets. With the formula we can calculate the price of European call and put options, though we only covered call options for now. Rearranging the model to be able to calculate put options, would only require our model to change the initial formula slightly.

With the BS-Model we could also compute each individual Greek to see how various differences in the given parameters would change the final option price. In our case we only covered Delta ($\Delta = N(d_1)$) since only this Greek is essential to calculating the option price.

$$C = S_0 \cdot N(d_1) - Ke^{-r_f T} \cdot N(d_2)$$

Where:

$$d_1 = \frac{ln(\frac{S_0}{K}) + (r_f + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

Description

C = *Value of the call option*
S_0 = *Current value of the underlying asset*
K = *Strike price / price at which the option can be executed*
r_f = *Risk − free rate*
T = *Time until expiration date as a percentage of year*
σ = *Annualized Volatility of the underlying asset*
$N(d_1)$ = *The rate of change of the option price with respect to the price of the underlying asset*
$N(d_2)$ = *Probability of the option being "in the money"*

"In-the-money" (ITM) options are defined to have a lower strike price (K) than the current price of the underlying asset (S_0), while "out-of-the-money" (OTM) options have a higher strike price (K) than the current price of the underlying asset (S_0). Between these types are so called "at-the-money" (ATM) options, where the strike price (K) equals the current value of the underlying stock (S_0) for example.

Data Source: Intrinio

Legal

All data we made use of during our project was provided by Intrinio, which offers access to several financial data feeds. To use this service, we had to register for a trial account. Consequently, we received a key, which allows us to gather historical data. Using this service, requires the acceptance of the "Terms of Use" and "Privacy Policy". Both can be found easily on the website and provide many guidelines, regulations and policies of how to correctly use their services and which limitations come with it.

Unfortunately, we were facing some limitations with our trial account, as we were not able to access the real-time stock prices. Since our project was meant to contain a kafka-streaming part, we contacted the Intrinio-Team to get information on how we could gain permission to the real-time stock prices. After explaining what we are aiming to do with the data and that we are using it for a university project, we managed to extend our limitations and have access to the real-time stock prices through the API.

In the "Terms of Use" you can find a paragraph on "Personal Use; Limited License; Ownership", which provides an overview on how to use the account created. The following paragraph is of great importance for our project purpose.

"… without the prior written approval of Intrinio, you may not distribute, publicly perform or display, lease, sell, transmit, transfer, publish, edit, copy, create derivative works from, rent, sub-license, distribute, decompile, disassemble, reverse engineer or otherwise make unauthorized use of the Services"[2]

Due to the fact that we use the data to perform derivative work, such as prediction, calculations and further analysis, a written consent by the Intrinio-team was essential. As mentioned before, we received this approval by one of their employees.

Data

During our project we had to get access to different API-calls. Specifically, we used a stream for real-time stock-prices, as well as historical stock-prices. A single entry is represented in a nested JSON-format with different root level attributes. The majority of values are numeric, though exceptions include *ticker* (e.g.: AAPL) and *frequency* (e.g.: daily), that are represented as strings and a *date*-value.

[2] https://about.intrinio.com/term

Libraries and Frameworks

Standard libraries

- json
- requests
- pprint
- os
- sys
- datetime
- time

Producer

- KafkaProducer
- KafkaError
- _future_ (print_function)

Consumer

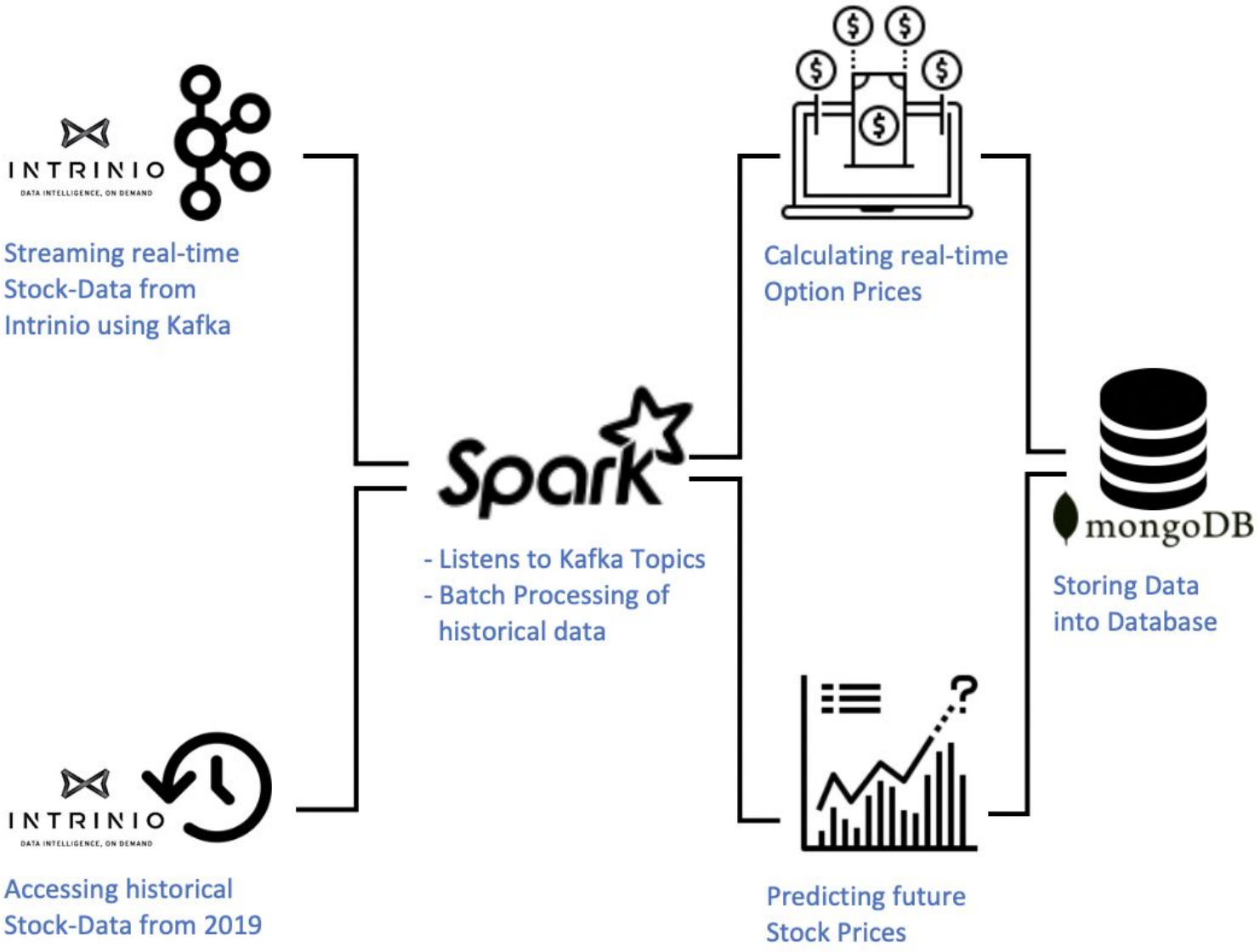
- findspark
- kafka (KafkaConsumer)
- pyspark.streaming (StreamingContext)
- pyspark.streaming.kafka (KafkaUtils)
- pyspark.sql (SparkSession, Row, SQLContext, functions)
- pyspark.sql.types (DateType)
- pyspark.sql.functions (abs, log, udf, stddev, sqrt)
- math
- numpy
- pymongo

Regression Model

Besides regular spark libraries:

- pyspark.ml.linalg (DenseVector)
- pyspark.ml.feature (StandardScaler)
- pyspark.ml.regression (LinearRegression)
- statistics
- mlflow.spark

Architecture



Code Description

Instruction of Code Usage

Before checking our notebooks, please read the README.txt file. By using our code, one has to follow a sequence to successfully run our code.

First of all, some libraries have to be installed: *pymongo* gets access to the mongodb database and *mlflow* in order to save our regression model. Kafka, Spark and MongoDB are already running on our server. Afterwards the code is ready to run. To start our real-time option-pricing calculation, the first thing to do is run the producer. Afterwards the consumer is able to retrieve the needed data. By starting the consumer, the data, including the calculated option prices, are automatically loaded into our mongodb database and can be called right away.

Description of Notebooks

Producer

In order to connect with the API, we created a function called *get_realtime_stock* with a parameter *(identifier)*, that allows you to choose what company you are looking at. As an example, we are using Apple, that can be accessed with the ticker AAPL (identifier). We have been using an URL request, which needs to include the provided API-Key. The data was pushed into a kafka producer, which produces messages every 5 seconds. The output shows the kafka-topic, which we called "data1", the partition 0 and the offset-number.

```
In [2]: def get_realtime_stock(identifier):
            url = "https://api-v2.intrinio.com/securities/" + str(identifier) +"/prices/realtime?api_key=OjkwODY5M2ZkY2M4ZGY1M2
            resp = requests.get(url).json()
            time.sleep(10)
            resp1 = requests.get(url).json()
            resp1["prev_price"] = resp["last_price"]
            #resp_json = json.loads(resp, encoding="utf-8")
            return resp1

In [ ]: from kafka import KafkaProducer
        from kafka.errors import KafkaError

        def on_send_success(record_metadata):
            print(record_metadata.topic)
            print(record_metadata.partition)
            print(record_metadata.offset)

        def on_send_error(excp):
            log.error('I am an errback', exc_info=excp)
            # handle exception

        producer = KafkaProducer(bootstrap_servers=['localhost:9092'], value_serializer=lambda m: json.dumps(m).encode('utf-8'
        INTERVAL = 5
        while True:
            data_points = get_realtime_stock('AAPL')
            data = {'updated_on': data_points['updated_on'], 'ticker': data_points['security']['ticker'],/
                    'last_price': data_points['last_price'],'prev_price': data_points['prev_price'] }
            message = data_points
            producer.send('data1', value=data).add_callback(on_send_success).add_errback(on_send_error)
            #time.sleep(INTERVAL)

        data1
        0
        8903
```

Consumer

The aim of our consumer is to consume the streaming data, process it, calculate the needed Option Prices and finally load the dataframe in the mongodb-database. First, we created a Spark-Session and configured the mongoDB-Spark-Connector. The function *get_sql_context_instance* provides the sql-singleton-context from the current context and the *write_mongo*-function finally writes every processed instance into the database.

In [4]:
```
# Import SparkSession
from pyspark.sql import SparkSession

# Build the SparkSession
spark = SparkSession.builder \
    .appName("master_consumer") \
    .config("spark.mongodb.input.uri", "mongodb://localhost/stocks_db.stock_coll") \
    .config("spark.mongodb.output.uri", "mongodb://localhost/stocks_db.stock_coll") \
    .getOrCreate()
    #.config("spark.executor.memory", "1gb") \
    #.master("local") \

sc = spark.sparkContext

sc.setLogLevel("WARN")
ssc = StreamingContext(sc, 10)
```

In [6]:
```
#    Spark SQL
from pyspark.sql import Row,SQLContext

def get_sql_context_instance(spark_context):
    if ('sqlContextSingletonInstance' not in globals()):
        globals()['sqlContextSingletonInstance'] = SQLContext(spark_context)
    return globals()['sqlContextSingletonInstance']
```

In [7]:
```
#    PyMongo
import pymongo

def write_mongo(df):
    try:
        # append the input df to the database
        df.write.format("com.mongodb.spark.sql.DefaultSource").mode("append").save()

    except:
        e = sys.exc_info()[0]
        print("Mongo Error: %s" % e)
```

We added a column to our dataframe, which provides the volatility. This value was calculated by multiplying the standard deviation with the square of 252 (trading days).

```
stockData = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri","mongodb://localhost/stocks_db.stockk
stockData  = stockData.drop("close", "high", "low", "open", "volumne", "frequency", "intraperiod")
stockData = stockData.withColumn("date", stockData["date"].cast(DateType()))
stockData.createOrReplaceTempView("AAPL")
stockData = spark.sql("SELECT * FROM AAPL WHERE date BETWEEN '2019-01-01' AND '2020-01-01'")
stockData = stockData.withColumn("volatility", abs(stockData["adj_close"]/stockData["prev_close"]-1))
stockData.show(truncate=False)
meanVol = stockData.select(stddev("volatility")*math.sqrt(252)).take(1)[0][0]
```

```
+-------------------------+---------+--------+--------+--------+-----------+----------+----------+-----------+------
---------------+
|_id                      |adj_close|adj_high|adj_low |adj_open|adj_volume |date      |prev_close|volume     |volati
lity           |
+-------------------------+---------+--------+--------+--------+-----------+----------+----------+-----------+------
---------------+
|[5e3d35671bc8b17f7ef416e4]|293.65   |293.68  |289.52  |289.93  |2.5247625E7|2019-12-31|291.52    |2.5247625E7|0.0073
065312843030306|
```

With the function *getOptionPrices* the existing RDDs of the Stream are first converted into Row RDDs. Then a PySpark SQL dataframe is created by using the *get_sql_context_instance*. After this processing, we come to the calculation part. We added further columns for d1 and d2, each providing out-of-the-money, in-the-money and at-the-money. With those it is possible to define the cdf-values and hence determine the final real-time option prices. Afterwards we used the sql-statement to select the coveted values, which should be included in the final dataframe. At the end the ultimate dataframe is loaded in mongoDB.

In [8]:
```
##    Defining external variables
#     Expiration time in months
T = 1/12
#     Square root of expiration time
T_s = (1/12)**0.5
#     Risk-free rate (constant  ->  source: https://www.statista.com/statistics/885892/average-risk-free-rate-austria/)
r = 0.013
#     Annualized historical volatility to the power of 2 (Assumption 252 Trading day)
#     set: 02-07-2020 till 01-01-2019
meanVolPow = meanVol**2
# Euler's number to the power of the negative risk-free rate times the expiration time
eu_exp = np.exp(-r * T)


def getOptionPrices(time, rdd):

    try:
        # Get SparkSQLContext from the current context
        sql_context = get_sql_context_instance(rdd.context)

        # Convert RDD to Row RDD
        row_rdd = rdd.map(lambda w: Row( updated_on=w['updated_on'], ticker=w['ticker'], last_price=w['last_price'], \
                                         prev_price=w["prev_price"]))
        row_rdd = rdd.map(lambda w: Row( updated_on=w['updated_on'], ticker=w['ticker'], last_price=w['last_price'], \
                                         prev_price=w["prev_price"]))
        #row_rdd = rdd.map(lambda w: Row(text=w['last_price']))
        # Create a DF of the Row RDD
        df = sql_context.createDataFrame(row_rdd)

        # Insert d1 and d2 for:
        #     out-of-the-money option (OTM)
        df = df.withColumn("d1_otm", (log(df["last_price"]/(df["last_price"]+20))+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d1_otm", (log(df["last_price"]/(df["last_price"]+20))+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d2_otm", (df["d1_otm"]-(meanVol * T_s)))
        #     at-the-money option (ATM)
        df = df.withColumn("d1_atm", (log(df["last_price"]/df["last_price"])+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d1_atm", (log(df["last_price"]/df["last_price"])+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d2_atm", (df["d1_atm"]-(meanVol * T_s)))
        #     in-the-money option (ITM)
        df = df.withColumn("d1_itm", (log(df["last_price"]/(df["last_price"]-20))+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d1_itm", (log(df["last_price"]/(df["last_price"]-20))+(r + 0.5 * meanVolPow)*T) / \
                           (meanVol * T_s))
        df = df.withColumn("d2_itm", (df["d1_itm"]-(meanVol * T_s)))

        # Creating "User Defined Function" in Spark, applying the cumulative distributed function for normal distributi
        norm_cdf = udf(lambda x: float(norm.cdf(x)))

        # Calculating probabilities for d1 and d2 (cumulative distribution function)
        #     out-of-the-money option (OTM)
        df = df.withColumn("d1_otm_cdf", norm_cdf(df['d1_otm']))
        df = df.withColumn("d2_otm_cdf", norm_cdf(df['d2_otm']))

        #     at-the-money option (ATM)
        df = df.withColumn("d1_atm_cdf", norm_cdf(df['d1_atm']))
        df = df.withColumn("d2_atm_cdf", norm_cdf(df['d2_atm']))

        #     in-the-money option (ITM)
        df = df.withColumn("d1_itm_cdf", norm_cdf(df['d1_itm']))
        df = df.withColumn("d2_itm_cdf", norm_cdf(df['d2_itm']))

        # Calculating option prices with Black-Scholes-Model
        df = df.withColumn("optionPrice_otm", df["last_price"] * df["d1_otm_cdf"] - (df["last_price"]+20) * eu_exp * \
                           df["d2_otm_cdf"])
        df = df.withColumn("optionPrice_atm", df["last_price"]*df["d1_atm_cdf"] - (df["last_price"]) * eu_exp * \
                           df["d2_atm_cdf"])
        df = df.withColumn("optionPrice_itm", df["last_price"]*df["d1_itm_cdf"] - (df["last_price"]-20) * eu_exp * \
                           df["d2_itm_cdf"])

        # Register table for SQL usage
        df.registerTempTable("stocks")

        ## Select whole data
        #stocks_df = sql_context.sql("select * from stocks")

        ## Select relevant data only
        stocks_df = sql_context.sql("select ticker, updated_on, last_price, prev_price, optionPrice_otm, \
                                     optionPrice_atm, optionPrice_itm from stocks")

        #  show df
        stocks_df.show(truncate=False)

        # write the resulting df to databse with pymongo and MongoDB Spark Connector
        write_mongo(stocks_df)

    # print errors in the output, in order to not stop the notebook
    except:
        e = sys.exc_info()[0]
        print("Process Error: %s" % e)
```

Afterwards we created a direct stream to kafka and used the function *getOptionPrices* for every incoming stock. The final output shows the created dataframe.

In [9]:
```
#    Defining Kafka Stream
kafkaStream1 = KafkaUtils.createStream(ssc, 'localhost:2181', groupId='spark-streaming-consumer', topics={'data1': 1})
```

In [ ]:
```
#    Input stream from producer with (lambda :  json.loads)
parsed_hist = kafkaStream1.map(lambda v: json.loads(v[1]))
#    Call previously defined function "getOptionPrices"
parsed_hist.foreachRDD(getOptionPrices)
#    Starting stream processing
ssc.start()
ssc.awaitTermination()
```

```
+------+-----------------------+----------+----------+------------------+------------------+------------------+
|ticker|updated_on             |last_price|prev_price|optionPrice_otm   |optionPrice_atm   |optionPrice_itm   |
+------+-----------------------+----------+----------+------------------+------------------+------------------+
|AAPL  |2020-02-07T20:16:44.463Z|318.89    |318.975   |1.5005144896275198|7.551152711650815 |21.469603546277938|
```

MongoDB

To access the data, which was loaded in the mongoDB we set up another spark session and configured the mongoDB-Spark-Connector again. By creating a dataframe of the constisting database, it is possible to show it for further analysis.

```
In [4]: # Import SparkSession
        from pyspark.sql import SparkSession

        spark = SparkSession.builder \
            .appName("master_consumer") \
            .config("spark.mongodb.input.uri", "mongodb://localhost/stocks_db.stock_coll") \
            .config("spark.mongodb.output.uri", "mongodb://localhost/stocks_db.stock_coll") \
            .getOrCreate()
```

```
In [22]: # create the master dataframe from the existing MongoDB database
         master_consumer_df = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri",
         "mongodb://localhost/stocks_db.stock_coll").load()
         master_consumer_df.show(1000, truncate=False)
```

```
+--------------------------+----------+------------------+------------------+------------------+----------+------+---
----------------------+
|_id                       |last_price|optionPrice_atm   |optionPrice_itm   |optionPrice_otm   |prev_price|ticker|upd
ated_on              |
+--------------------------+----------+------------------+------------------+------------------+----------+------+---
----------------------+
|[5e3dc4931bc8b1d3421dd0b9]|319.12    |7.556598994455783 |21.47261698396443 |1.5036856746231564|319.04    |AAPL  |202
0-02-07T20:11:38.502Z|
```

Linear Regression Model

Additionally, to the described option pricing algorithm above, we were interested in the accuracy of a linear regression model predicting the closing price of a trading day, assuming that besides the previous closing price and the opening price, the highest and the lowest price of this day has already been determined. First we downloaded the data from the provided Itrinio-URL and took a look into it.

```
In [2]: #  For usability: Bulk-Downloads are only available once a minute
        key1 = "OjMxMTlhMWRjMDQ5NDUxZTRhNjg0NDM5M2UwYzc3Njdk"
        key2 = "OjIwMWQ4MWU5ZGE5ZDcxMjRlY2NjOWE0N2ZlODI4Yzlj"


        #  API request function
        def get_hist(identifier, pages, key, stdate, enddate):
            url = "https://api-v2.intrinio.com/securities/"  + str(identifier) + "/prices?start_date=" + str(stdate) + "-01&end_date=
            resp = requests.get(url).json()
            #resp_json = json.loads(resp, encoding="utf-8")
            return resp
```

```
In [5]: aapl1980 = get_hist("AAPL", 10000, key2,"1980-01-01", "2020-01-31")
```

```
In [6]: aapl1980["stock_prices"]
```

```
Out[6]: [{'date': '2020-01-31',
          'intraperiod': False,
          'frequency': 'daily',
          'open': 320.93,
          'high': 322.68,
          'low': 308.29,
          'close': 309.51,
          'volume': 49897096.0,
          'adj_open': 320.170133759724,
          'adj_high': 321.915990283202,
          'adj_low': 307.560061498724,
          'adj_close': 308.777172903662,
          'adj_volume': 49897096.0},
         {'date': '2020-01-30',
          'intraperiod': False,
          'frequency': 'daily',
          'open': 320.5435,
          'high': 324.09,
          'low': 318.75,
```

In order to be able to use the data without calling the API again, we decided to export the received dictionary and load it into a SparkDataframe separately. To enable using Spark again and creating a SparkDataframe with the saved JSON-file, we had to import all necessary Spark frameworks, build a spark session and create a SparkContext.

In [7]:
```
#  Exporting history data
with open('data1.txt', 'w') as outfile:
    json.dump(aapl1980["stock_prices"], outfile)
```

In [8]:
```
import os, sys
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages org.mongodb.spark:mongo-spark-connector_2.11:2.4.0 --jars /usr/local/jar_file
```

In [9]:
```
# Import findspark
import findspark

# Initialize and provide path
findspark.init("/usr/share/spark/spark-2.3.2-bin-hadoop2.7/")

# Or use this alternative
findspark.init()
```

In [10]:
```
#    Spark
from pyspark import SparkContext
#    Spark Streaming
from pyspark.streaming import StreamingContext
#    Kafka
from pyspark.streaming.kafka import KafkaUtils
#    json parsing
import json
#    KafkaConsumer
from kafka import KafkaConsumer

from pyspark.streaming.kafka import KafkaUtils

import pyspark.sql.functions as functions
```

In [11]:
```
# Import SparkSession
from pyspark.sql import SparkSession

# Build the SparkSession
spark = SparkSession.builder \
   .master("local") \
   .appName("LinearRegressionModel") \
   .getOrCreate()
   #.config("spark.executor.memory", "1gb") \

# Create SparkContext
sc = spark.sparkContext
path = "./data1.txt"
# Load JSON to SparkDataframe
data = spark.read.json(path)
```

After dropping columns not providing any additional information, we counted the amount of categorical and numerical attributes. Due to the fact that no categorical data was relevant for our approach, we did not need to write a function for one-hot-encoding categorical values. The DenseVector function provided by PySpark creates a feature vector consisting of all attributes, except the one we want to predict (adj_close). In the next step, we scale the data using the PySpark StandardScaler, which normalizes each feature after fitting the dataframe to the scaler.

In [13]:
```
## determine categorical and numerical columns
categorical_cols = [item[0] for item in data.dtypes if item[1].startswith('string')]
print(categorical_cols)

numerical_cols = [item[0] for item in data.dtypes if item[1].startswith('int') | item[1].startswith('double')][:-1]
print(numerical_cols)

print("The data consists of " + str(len(categorical_cols)) + ' categorical features')
print("The data consists of " + str(len(numerical_cols)) + ' numerical features')
```

```
[]
['adj_close', 'adj_high', 'adj_low', 'adj_open']
The data consists of 0 categorical features
The data consists of 4 numerical features
```

In [16]:
```
# Import `DenseVector`
from pyspark.ml.linalg import DenseVector

# Define the `input_data`
input_data = data.rdd.map(lambda x: (x[0], DenseVector(x[1:])))

# Creating new dataframe with label (adj_close) and feature-vector
df = spark.createDataFrame(input_data, ["label", "features"])
```

Out[16]:
```
[Row(label=308.777172903662, features=DenseVector([321.916, 307.5601, 320.1701, 49897096.0])),
 Row(label=323.103172719166, features=DenseVector([323.3227, 317.9953, 319.7845, 31685808.0]))]
```

In [72]:
```
# Import `StandardScaler`
from pyspark.ml.feature import StandardScaler

# Initialize the `standardScaler`
standardScaler = StandardScaler(inputCol="features", outputCol="features_scaled")

# Fit the DataFrame to the scaler'
scaler = standardScaler.fit(df)

# Transform the data in `df` with the scaler
scaled_df = scaler.transform(df)
```

In order to validate the success of our model, we need to split the data in a training (80%) and a testing (20%) set. Using the randomSplit method, one can be (almost) sure, that there is no bias in the allocation of the data. The parameter *seed* provides the possibility to reuse the same sample allocation for further analysis.

In [53]:
```
# Split the data into train and test sets
train_data, test_data = scaled_df.randomSplit([.8,.2],seed=1)

# Import LinearRegression from pyspark.ml.regression
from pyspark.ml.regression import LinearRegression

# train the Linear Regression Model
lr = LinearRegression(labelCol="label", maxIter=10000, regParam=0.01, elasticNetParam=0.4)

# Fit the data to the model
lrModel = lr.fit(train_data)
```

In [22]:
```
# Generate predictions
predicted = linearModel.transform(test_data)

# Extract the predictions and the "known" correct labels
predictions = predicted.select("prediction").rdd.map(lambda x: x[0])
labels = predicted.select("label").rdd.map(lambda x: x[0])

# Zip `predictions` and `labels` into a list
predictionAndLabel = predictions.zip(labels).collect()

# Print out first 5 instances of `predictionAndLabel`
predictionAndLabel[:5]
```

Out[22]:
```
[(0.160175384699975, 0.156057571405883),
 (0.16530403088744425, 0.161306780625899),
 (0.1671493192972744, 0.163151097378878),
 (0.18483138287926867, 0.180884912311364),
 (0.19040972556873137, 0.18627599205084)]
```

Even though we tinkered with the parameter values of the linear regression model, we finally decided to stick with the original values taken from Dr. Kirrane's notebook. After fitting the model, the training date and predicting closing prices for the test data, we merged the actual and the predicted values into a list.

```
In [18]: #  Insert individual accuracy
for i in range(0, len(predictionAndLabel)):
    predictionAndLabel[i] = predictionAndLabel[i] + (1-abs((predictionAndLabel[i][1]-predictionAndLabel[i][0])/predicti

#  Calculate average accuracy
import statistics as stat
sumAcc = []
x = 0
for i in range(0, len(predictionAndLabel)):
    x = x + predictionAndLabel[i][2]
avgAcc = x / len(predictionAndLabel)
print("Assuming the lowest and highest price of a trading day is already reached, the model predicts \
the closing price of this day with an accuracy of " + str(round(avgAcc, 4)) + "%")

Assuming the lowest and highest price of a trading day is already reached, the model predicts the closing price of th
is day with an accuracy of 0.9893%
```

Using this list, we were able to calculate the accuracy of each prediction and finally come up with an average predicting accuracy of 98.93%. This appears to be quite realistic due to the fact that most of the time, the closing price is relatively near to the opening price and 1.07% on a 300$ stock is still a wide range which determines whether a trader wins or loses. Additionally to this, we assumed that the intraday high and low had already been reached and is well documented. Nevertheless, we are pleased to see our model performing accurately.

```
In [21]: # Taking a look into the model parameters
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))

# Summarize the model over the training set
trainingSummary = lrModel.summary
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
print("r2: %f" % trainingSummary.r2)
print("numIterations: %d" % trainingSummary.totalIterations)
trainingSummary.residuals.show()
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))

Coefficients: [0.4156679726541322,0.4880211808552577,0.09702150558981235,0.0]
Intercept: 0.0027245318742287997
RMSE: 0.435191
r2: 0.999935
numIterations: 945
+--------------------+
|           residuals|
+--------------------+
|-0.00370826187854...|
|-0.00360578941128...|
|-0.00360710009176...|
|-0.00378603847992...|
|-0.00379107955870...|
|-0.00372086457549...|
|-0.00361587156884...|
|-0.00361587156884...|
|-0.00379480995700...|
|-0.00372469579536...|
|-0.00356073175751...|

In [51]: #pip install mlflow

In [57]: # Saving the model --> reusability
import mlflow.spark
mlflow.spark.save_model(lrModel, "./model")
```

In the end, we printed some statistical background of the model. Most interestingly the correlation coefficients, representing the absolute change of the predicted value for each unit of the predicting variables (open | high | close | volume). In order to enrich the reusability of the trained model, we saved it using the save_model()-method provided by MLflow. For this, we first needed to install MLflow.

Encountered Problems

While working on this project, one of the major problems we have faced was our almost non-existing primer knowledge about the Apache Hadoop Ecosystem. Neither of us had any previous contact handling Big-Data-Frameworks and dealing with Streams. It took us a respectively long time to understand the practical application of the architecture behind our project and how we use Spark and Kafka. Especially running a stream with the data, we needed was at first more complicated than we thought it would be. Nevertheless, those problems encouraged us to dive deeper in specific topics, which then provided an advantage by solving many upcoming difficulties.

Another problem was our Jupyter-Notebook. Multiple times a day a loading-issue occurred and our notebook was not accessible for a specific period of time. Therefore, there was no possibility to continue our work appropriately within this timespan. Also, the kernel was not able to connect properly from time to time.

Additionally, one of the most nerve-wracking moments was five days before the submission deadline. We got a “NoBrokerAvailable”-Error for a whole day, that did not allow us to start any stream at all and consequently our ability to get further work done was accordingly limited this day. We sent the data science support team an email after trying to fix the problem by ourselves for a whole day. Fortunately, they were so kind and fixed our error within one night.

Limitations

During our project we have stumbled upon some limitations. First of all, the access to real-time stock data depends on trading-days. Stock prices are only altering between 15:30 and 22:00 CET on weekdays and the market is fully closed on weekends. Therefore, our real-time analysis was restricted to these time periods. Furthermore, we faced limitations at the beginning of the project by finding a real-time API for financial data. Especially real-time option-prices are not provided on a free basis and often only available for institutional status customers. Most financial-data websites only provide a change of option prices once a day, which was not useful for our approach. Even with the stock data we were just able to connect with a real-time stream because the Intrinio-team offered us this feature for seven days. Without this permission, there is no possibility to access that data for free as a student.

Future work

Having done the “messy” and core part for real-time pricing of financial options, we have created a great base for definite future work. By doing just that, we met and most likely even slightly exceeded our expectations, but we could definitely further elevate our project by implementing some additional features such as visualization-tools and extended information regarding the *Greeks* themselves for example. With the Black-Scholes Model and its extending formulas calculating each individual Greek would make option pricing more accurate, since only if you truly understand the Greeks and their respective meaning, you will succeed in option trading in the long run. To be successful in trading options, thus being the ultimate goal with this project, calculating option prices solely based on the standard BS-Model will not be sufficient as these formulas are widely known. Only slightly adapted models succeed in the long term, but as we still have to figure out which exact modifications we have to adapt our model with, we will stick with the basic model.

Additionally, we could implement a visually more pleasing interface to enrich reusability and enhance the user-experience. For this to happen, we could add various appealing widgets within our project, such as an easy to handle menu. Within that menu certain parameters such as the regarding underlying asset, different time periods or strike prices can be selected to further increase efficiency, rather than manually changing these metrics within the code itself.

If money is of no essence, another way of elevating our current project is to add real-time option prices to compare and contrast them to our computed findings. With this data embedded in our model, we would be able to specify our assumptions to rebuild the reality as precisely as possible. Moreover, the frequency of API-calls for our real-time stock data could also be improved to have even more specific and accurate findings with our model. This would enable us to always stay up to date to certain relevant market movements and to make it less complicated to adapt our initial formula and furthermore react quicker and more effectively.

Biases

At first glance it would seem like there would not be any biases since we are only working with numbers and formulas without any human input. This is only partially true since the price of the underlying asset is the result of thousands of people in the market acting in their own self-interest. Their behavior is full of various biases, ranging from relying on news, written by non-fully objective journalists to the market being biased itself. General perception is that markets rise steadily over time without any big hits such as market crashes due to natural disasters, epidemics etc. Even if those incidents occur, people strongly believe that the economy and therefore also the stock market rebounds to at least the same level as before the crash, if not, even outperform the previously achieved level.

The market could also be described as being biased for some of its participants having a lack of knowledge when it comes to general investment strategies or reacting to for example unprecedented market movements. This often leads to over- and undervalued stocks and other financial instruments and therefore to an imbalance of the market. Another and probably even one of the biggest biases within the financial industry is that market players, whether big or small, tend to chase trends. Especially when it comes to past trends, lots of participants favor assets with a good and very profitable track record, even though that does not necessarily imply that this history will continue in the future.[3]

Intrinio's service itself will most likely not have any obvious biases, as they are only screening the current stock feeds from stock exchanges – in our case the IEX – and do not handle the data.

Ethical Issues and Concerns

As we only worked with publicly available financial data and our own model, we do not interfere with anyone's privacy. While working on the project we paid close attention to abide by both the IEEE and ACM Code of Ethics, as well as the GDPR. We also acknowledged all data source providers accordingly and maintained a conscientious and self-reflecting transparency with our sources.

[3] https://www.investopedia.com/articles/investing/050813/4-behavioral-biases-and-how-avoid-them.asp

Credits

In addition to our own knowledge, a handful of resources helped us a lot, handling the major difficulties of this project. Particularly the provided Spark- and Kafka-Notebooks of Dr. Kirrane enabled us to get an easy start regarding the usage of the Spark environment and creating data streams through Kafka. For the second part of our project, we were able to use whole code chunks of the linear regression example as they fitted perfectly and led to an astonishing prediction performance. As only changing variable names would obviously look like a cheap try to sell this as our own, we fully acknowledge the previous work and decided to go with the same designation as full credit goes to Dr. Kirrane anyway.

Spreitzer Nina and Ziller Thomas were responsible for the creation and the handling of the environment in which our project performed on, while Kollarczik Florian was primarily focused on financial analysis. During our project, this work allocation proved to be the best working for everyone involved, as we could work on different fronts simultaneously and always had someone responsible for different steps. The Black-Scholes Model was taught in the Finance-specialization, hence Florian has already worked with these formulas and as a result to that, additional data sources are not required.

References

Cover picture, [online]
https://unsplash.com/license [07.02.2020]

Intrinio license, [online]
https://about.intrinio.com/term [07.02.2020]

Icons for architecture, [online]
https://iconscout.com/legal/licensing [07.02.2020]

Option definition, [online]
https://www.investopedia.com/terms/o/option.asp [07.02.2020]

The Spark documentation, [online]
https://spark.apache.org/docs/latest/api/python/pyspark.html [07.02.2020]

Biases in the financial sector, [online]
https://www.investopedia.com/articles/investing/050813/4-behavioral-biases-and-how-avoid-them.asp [07.02.2020]