Scalable Computing – Clustering of Big Data

Philip Andreadis (s4942906)
MSc Computing Science
University of Groningen
`f.andreadis@student.rug.nl`

Lorenzo Rota (s4895711)
MSc Computing Science
University of Groningen
`l.d.f.rota@student.rug.nl`

Marios Souroulla (s4765125)
MSc Computing Science
University of Groningen
`m.souroulla@student.rug.nl`

March 31, 2022

Keywords Scalable computing · Clustering · Big data · K-means · Apache Spark

1 Introduction

In this project we aim to realize our own scalable pipeline for retrieving real-world data, storing them in a distributed manner and applying to them a modified version of the K-means clustering algorithm after some basic preprocessing. To achieve this we have put together a set of available frameworks and built a robust infrastructure that deals with both historical and streaming data. The data source used is the Weather Api, which provides real time readings for multiple climatic factors generated from sensors placed in numerous cities around the world. Section 2 provides a detailed explanation of each framework used and how they are all integrated together into the final pipeline. In Section 3 we discuss the logic and implementation of the modified K-means algorithm and describe how the two separate cases of historical and streaming data are handled. Lastly, in Section 4 we provide the results of the clustering and comment on the scalability of our work.

2 Architecture

In this section we provide an overview of each individual component used in our pipeline along with an explanation on how they all work together. Figure 1 provides a visualization of the pipeline in the form of a flowchart.

Data The dataset used for this project consists of real-word weather data collected from the Weather Api at `https://www.weatherapi.com/`. The Api provides real-time weather data plus historical weather data for the past week along with many other services, such as weather forecast and geological or astronomy data. However, we narrow our interest to the historical and the live stream of data as the scope of the project necessitates. We chose to collect climatic features of each state capital of the US. Since there are multiple sensors generating readings for each city we aggregate them and derive a mean value for each feature and store them in the form of time-series with a 10 seconds time interval between each data point. It is important to mention that even if the data are stored as time-series, we do not treat them as such during the clustering phase. This means that during clustering the timestamp of each data point is irrelevant and is rather needed only occasionally to infer a time correlation between different readings from different cities. Hence, we create 50 different channels emitting the following weather attributes for the corresponding cities.

- City
- Timestamp
- Latitude
- Longitude
- Elevation
- Solar radiation

- Ultraviolet radiation (UV)
- Wind direction
- Wind chill
- Wind speed
- Humidity
- Temperature
- Atmospheric pressure
- Precipitation rate

PubNub Data Stream Network In order to retrieve the readings from the Weather API we have utilized the PubNub framework, which allows to turn any data source with an API into a ’data firehose’. This essentially means that we are able to stream weather readings of up to the past 7 days (max) in real time and push them into our pipeline.

Apache Kafka Apache Kafka is one of the most popular event streaming frameworks. We take advantage of this framework in order to fetch stream batches of weather readings, after they have been published from the PubNub network, and process them in real time. Kafka allows all this to take place in a distributed and fault-tolerant manner, hence it is a good fit for a scalable project. We use topics, one for the historic data, the *common* topic, and one for the streaming data, the *streaming topic*. Optimally we can have a total of 50 workers for each topic, and we also set the flush size for historic for the historic data to 10 000, and for streaming is 50, because we are expecting 50 data points. For the historic data this can be adjusted according to the total amount of data, in our case we are expecting a maximum of 1.64 Million (approximately) data points (for 7 days).

Hadoop Distributed File System (HDFS) HDFS is a fault-tolerant, distributed file system designed for applications that need to handle large amounts of data. Additionally, HDFS is suitable for batch processing proving itself useful for the data streaming part of our project. Regarding historical data, we are able to push them directly from PubNub and store them in HDFS in the form of Avro files. For the streaming files, we have a new one every 10 secs approximately, each file containing 50 data points (one for each city).

Apache Spark We use Spark as the final component of our pipeline and this is where the data preprocessing and the core logic of the modified K-means algorithm take place. Apache Spark is one of the most broadly used large-scale data analytics engine. Its main architectural component is the resilient distributed dataset (RDD), which works as an abstraction for parallel data processing in a fault tolerant manner. We use Spark in order to retrieve the weather data as an RDD, preprocess them and implement our own version of K-means on them.

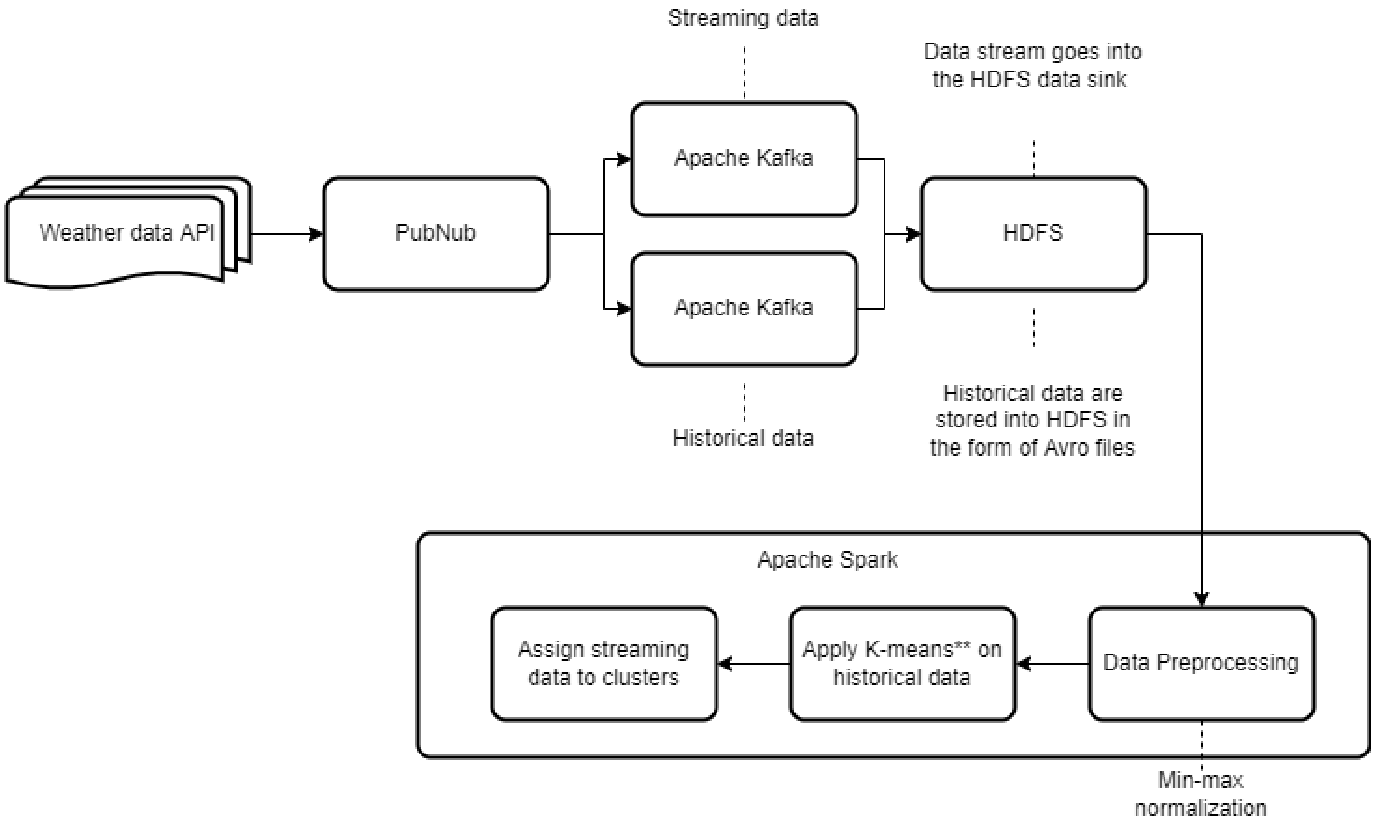
Streaming data
Data stream goes into the HDFS data sink
Weather data API
PubNub
Apache Kafka
Apache Kafka
HDFS
Historical data
Historical data are stored into HDFS in the form of Avro files
Apache Spark
Assign streaming data to clusters
Apply K-means** on historical data
Data Preprocessing
Min-max normalization

Fig. 1. Flowchart of the architecture

3 Core Algorithm

Preprocessing The first thing we do as soon as the weather data are loaded into Spark in the form of a Dataframe is to drop all null values and apply min-max normalization. This is necessary because the features are on different scales and the results of K-means on the original values would be greatly skewed. We also drop the columns ["timestamp", "longitude", "latitude", "city"] as they are not relevant for the clustering (they do not contain weather-related data).

K-means K-means is one of the most widely used methods of partitioning a dataset of unlabeled points into clusters. We apply this algorithm to the collected weather data in order to infer the *k* different weather conditions one might face, that have similar climatic characteristics. We implement our own version of K-means**, which is a slight alternation of the original algorithm, using Pyspark and the MapReduce programming paradigm in an effort to make the clustering of large amounts of data efficient. The only difference between K-means** and the original is the fact that each iteration is only applied to a sampled subset of the data. Specifically, the distances of each data point *x* to the centers of the clusters will only be computed with some probability β. The logical steps of K-means** are as follows:

1. Initialize $\beta < 1$ and $\alpha > 1$.
2. Choose randomly k data points from the dataset as the initial cluster centers.
3. Apply a Map operator on each data point *x* with the probability β. The Map operator computes the squared distance between *x* and each cluster center and assigns each point to the cluster with the minimum distance.
4. Apply the Reduce operator, which computes the mean of each cluster.
5. Define the new center of each cluster as the mean computed in step 4.
6. Set $\beta = \max(\beta \cdot \alpha, 1)$.
7. Repeat steps 3 to 6 until the distance between the old and the updated centers is less than a predefined convergence distance.

Historical and Streaming Data As already mentioned, our pipeline is intended to deal with bot historical data and batches of streaming data. Initially, given a set of historical data (e.g. weather readings of the past 7 days for each city) we run K-means** and generate the *k* clusters. On a second step, we retrieve a batch of new data points (i.e. streaming), compute the distances of each point to the clusters generated by the historical data and assign them to the closest cluster. Hence, this process resembles more a classification of the streaming data rather than form of clustering. The motivation behind this is the fact that the size of a batch of streaming data is significantly smaller than the size of the historical data used for creating the clusters. This means that in the case that we did the clustering again considering this time the new batch as well, the change of the cluster centers would be at most insignificant.

4 Results

In order to test the scalability of our K-means implementation we run both the original version of the algorithm and K-means** on different percentages of the historical data and compared the runtimes. Figure 2 shows the average runtime over 10 runs of both K-means and K-means**. We can notice that for 10%, 30%, and 90% the K-means** is faster, and for 50%, and 70% the original K-means is faster. This can be explained because the sub-sampling required for the K-means** adds an overhead, which is also increasing with the amount of data.

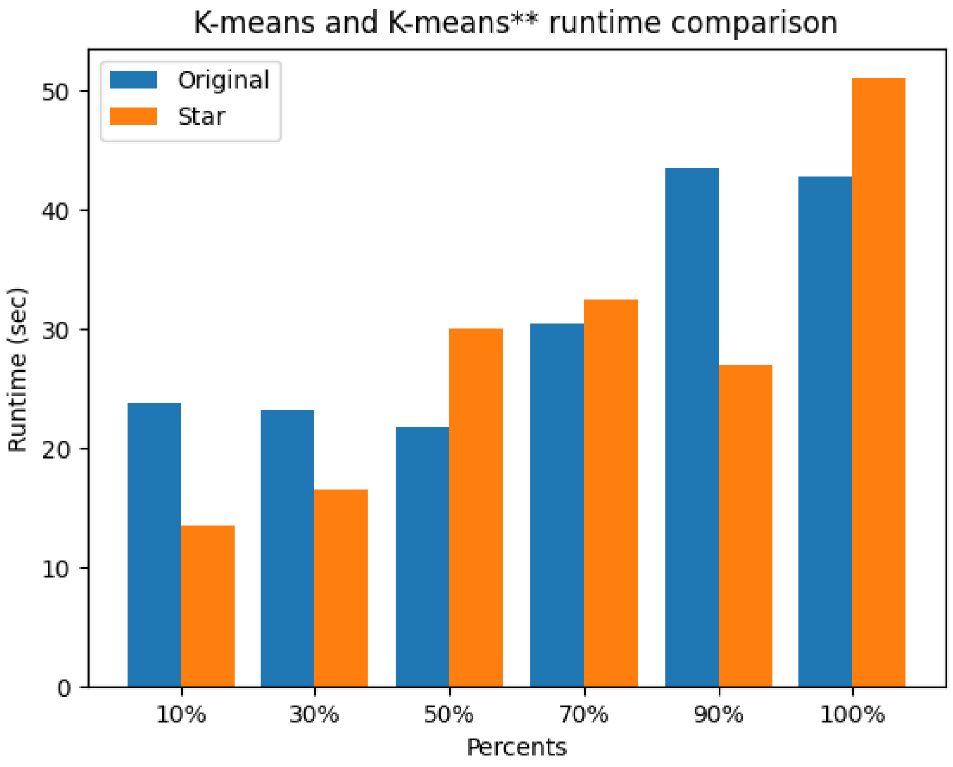

Fig. 2. Runtime comparison between K-means and K-means**