A machine learning infrastructure for Aline using Amazon Web Services

Victor Hwasser
Abstract

A machine learning infrastructure for Aline using Amazon Web Services

Victor Hwasser

With the rise of Internet of Things, cloud computing has become an increasingly important concept. Applications can be run on hardware with limited capacity due to the heavy computing being transferred to data centers over the internet. This has created new opportunities to what mobile applications are able to do and opened up new horizons for tailor-made and user-centered solutions for users through machine learning and the use of data integration.

In this report we developed an architecture and infrastructure for a mobile application named Aline. Aline is a learning platform that recommends learning material to its users according to the users learning objectives. The purpose of the infrastructure was to deploy a framework for supplying the recommendation system with data and providing these recommendations for the end user. The architecture was built around AWS and AWS Lambda was used as core of the infrastructure. AWS Lambda controlled the transfer of datasets from an external database to the AWS Data Lake, interacted with the machine learning environment and handled https requests from the Aline application to the AWS. This infrastructure consumed 3.5-5.5 seconds to make recommendations. However, we further optimized it to 1.2-2.6 seconds. Furthermore, the system was tested for scaling using 100,000 test users. We also developed a machine learning algorithm that successfully demonstrated the usability of our architecture. The architecture can be further improved by storing datasets in another format or integrating the Data Lake and the database to a common solution.
8 Appendix A: Early work
8.1 Technologies used
8.1.1 Apache Spark
8.2 Early designs
8.3 Early implementation of transferring data
8.3.1 Design and implementation
8.3.2 Results
9 Appendix B: Code Examples
1 Introduction

1.1 Aline - Learn Better

Aline is a learning platform that provide users (called learners) with learning materials and encourage them in taking an active role in their learning by showcasing their progress. It exists as a mobile application for Android and Iphone (more information can be reached at www.alinebetter.com).

When a learner sign up for Aline they specify what subjects and categories they are interested in learning about. Learners will then be fed with material based on these settings. Learning material can be of various formats, such as news paper articles, academic papers, podcast-episode and video-clips. Aline is currently in a closed beta version and the users are currently expected to upload learning material into the app themselves. Uploaded material can then be accessed by other users. But the aim is to use algorithms to upload new learning material by mining trusted internet sources, rather than relying on manual uploading from learners. Uploaded learning material - both automatically mined material and non-categorized manually uploaded material - needs to be clustered, filtered and categorized using previous uploaded material as reference. Another aim is for the application is to learn from user interaction and provide recommendations. The learners will be able to track their progress and how well they’re achieving their learning goals. Additionally learners can also join groups based on their interests and get material based on the groups purpose.

1.2 Goals and expectations

The goal of this project is to provide an efficient computational infrastructure for processing huge amounts of data for a machine learning application. It should follow the principles of reliability, scalability and maintainability of building distributed systems [5, p. 6]. Another principle to keep in mind is the cost principle, which means the actual price paid, since cloud services charge for both storage, transfer and execution time. For a complete and functioning infrastructure there has to be an integration of the Aline mobile application, its database and various machine learning models. Aline strives for machine learning behaviour in a number of aspects, these include:

- A web crawler application which mines trusted sources on the internet for new learning material with the help of machine learning algorithms.
• An application that processes and classifies incoming learning material, possibly denying inappropriate material containing illegal or offensive content.

• Adaption of the mobile platform by learning how the user interacts with it.

• An application that creates daily recommendations through machine learning algorithms.

Due to time limitations, this report will be limited to developing the infrastructure of creating recommendations. A big consideration in building the infrastructure is that the database will be written to, modified and read by millions of users in real-time. Therefore the data must both be easily accessible, quick to modify and fast to analyze.

The performance of the infrastructure will be measured using the Python time-library and Amazon CloudWatch which stores text logs from services used within Amazon Web Service-ecosystem. Since Aline is in closed beta version and lacks large amount of data, an application need to be implemented to generate pseudo user’s data that is adequate for simulating the processing and transportation of data similar to that of larger user base. To achieve this the pseudo user dataset needs to have the same structure and
similar content as the real user dataset and will be tested for performance, scalability and reliability.

The company has yet to develop a machine learning model for predicting recommendations, therefore a basic machine learning model needs to be implemented. Creating a model was not part of the project plan, but was added at the end of the project. The accuracy of the model is out of scope for this report and it will be replaced by a new model by the company in the near future. The purpose of the model is to make sure the infrastructure works well as a whole.

The safety and security of the infrastructure is not part of the project plan and is also out of the scope for this report.

2 Background

2.1 Technologies Used

In order to understand this report, some basic concept and tools need to be explained. Aline uses MongoDB as it’s main user database and thus its inner workings need to be explained for choices made in this report. All machine learning will take place inside of Amazon SageMaker, which uses S3 as its primary storage, thus both these services will need to be explained.

2.1.1 MongoDB

MongoDB is a software for handling NoSQL databases, which is a form of semi-structured databases consisting of JSON-like objects called documents. A MongoDB-client has functionality for importing, exporting and aggregating data from a MongoDB-server. It has support for replicated data sets and horizontal scaling, which provides reliability and high performance for distributed data systems [14]. MongoDB official site provides storage and virtual machine clusters for handling MongoDB-databases through a service called MongoDB Atlas, but users can also set up their own databases using MongoDB-server. MongoDB is built for many fast user requests and is thus optimized for fast response-time. in fact MongoDB didn’t support multi-document transactions at all until version 4.0 in year 2018, if more than one

---

\(^{i}\)“Not only SQL”.

\(^{ii}\)JavaScript Object Notation is a text-based format for storing and exchanging data.
document needed to be updated MongoDB would use a bulk system which serialized all operations needed and then execute them one-and-one [8].

The Python-library *PyMongo* will be used for connecting Python-applications with MongoDB.

### 2.1.2 Amazon S3

S3 is a Data Lake storage provided by Amazon. A Data Lake is a solution for storing huge amounts of raw data, contrary to that of a database which expects semi-structured or structured data. Many implementations of Data Lakes are based on the HDFS implemented in Apache Hadoop. Data lakes are expected to be operated on with the principles of *extract, transform, load*, previously found in data warehouses, which is a solution for storing huge amounts of structured data [4]. Amazon S3 is the standard storage solution for various cloud services at Amazon Web Services (AWS).

### 2.1.3 Amazon SageMaker

Amazon SageMaker is a cloud service for creating, deploying and processing machine learning models. It contains a large amount of tools and functionality to assist data scientists, machine learning engineers and developers. SageMaker is centered around the use of *notebooks*, but can also be used externally by the SageMaker API. A Jupyter Notebook (or simply called a notebook) is a web-based interactive platform where you can write code segments, notes and documentation. The strength with notebooks is the ease of experimentation since code segments, meta data and graphs can be executed and evaluated within the environment itself. SageMaker allows users to deploy notebooks in virtual machines instances [9] and execute them using SageMaker’s customized Apache Spark-cluster. But any Spark cluster, such as Amazon’s dynamic cluster solution EMR is also supported with the use of Amazons notebook-extension called Sparkmagic [11].

An integrated development environment (IDE) called SageMaker Studio is included in the service for working with notebooks. A data scientist could use a SageMaker Studio to pre-processes data, train and create machine learning model, analyze the results and store models to S3 for production. SageMaker also contain functionality for creating so called *Training Jobs* for training stored models, allowing for real-time prediction with the help of so

---

In this report these deployed notebooks will simply be called SageMaker Notebooks
called EndPoints or make predictions in batch using Batch Transform Jobs. The primary source of storing and retrieving data in SageMaker is with S3 in either JSON- or CSV-data format[iv][16].

2.2 Related work

Dadeck L. developed a tool for automatic deployment of machine learning applications. A console interface is built for providing an automated infrastructure for machine learning using Amazon AWS services. The application is deployed directly in the cloud using Amazons serverless solution Amazon Elastic Container Service. The automation tool allows the user to select and execute Amazon SageMaker notebooks for training machine learning models and store the result in a bucket within Amazon S3.

It also provides automation for deploying scalable applications and enables the execution of deployment techniques and viewing metrics [13].

Naranjo et al. developed a serverless gateway for a machine learning architecture. The serverless gateway is built using Amazon Web Services Lambda and SCAR. SCAR is a framework for executing Docker container images[v] on Lambda. The purpose of the work was to implement an architecture for reducing the complexity of using artificial intelligence via a web interface. The user can simply select a model, upload data into a S3 bucket and after the processing is done the predictions are stored in a folder within the bucket[10].

In a study by Alipour et al. they construct a cloud microservice architecture that scales in width with the help of predictions. According to the paper most cloud services uses reactive and proactive strategies to help customers with automated scaling. Proactive strategies include the possibility for customers to make the architecture scale in width due to a schedule and reactive strategies makes use of threshold values. The architecture is build upon Amazon Web Services and uses services such as Lambda and Amazon Machine Learning (the predecessor of SageMaker) to assist with the predictive machine learning models. The data comes from monitoring of the workload of microservices and the architecture has scheduled runs of multiple models that learns from the data and adapt the scaling accordingly[1].

---

iv CSV is a file format for storing structured data.

v Containers are light-weight environments containing an executable and all dependencies to run the executable.
3 Design

The design section contains an explanation of the infrastructure (or lack of) at the start of this report, a short description of the database collection that are to be used and a prototype of an architecture. The end of the section contains a brief description of the prototype for of a small application which creates additional test-data for the infrastructure and the design of a temporal machine learning model.

3.1 An explanation of the initial infrastructure

The Aline mobile application is built using JavaScript with the JavaScript runtime environment Node.js in it’s back-end. It uploads and fetches its user data using a semi-structured database handled by MongoDB Atlas. This is appropriate since both JavaScript and MongoDB works with JSON-structured objects and since MongoDB is optimized for many small user requests. Since data in MongoDB is stored in semi-structured form the data need to be transformed into structured form before it is used by machine learning algorithms.

The machine learning models were being written using SageMaker Notebooks. There are several challenges with this setup. Primarily that SageMaker functionality outside Notebooks manages its data from Amazon S3, which makes the current dependency on MongoDB problematic. Secondly that the data of the machine learning models are being pre-processed using Pandas DataFrames. Pandas is a Python-tool for data analysis and data manipulation and is a common tool among data scientists. A Pandas DataFrame is a matrix like data structure that contains rows and columns of data. Thus a more precise problem definition of this report would be how to build an infrastructure connecting MongoDB, Amazon SageMaker, Pandas and a mobile application.

3.2 Data at hand

There were currently about 20 collections of datasets, however most of them were still incomplete, that is containing very few examples or missing a lot of data fields. The most relevant datasets for machine learning were library and people. library contains information about learning material such as category, subjects, brief descriptions, release data and how many times people
3.3 Design proposal

An infrastructure could be designed connecting SageMaker with the mobile application using middleware. Middleware could be set up on virtual machines in the AWS-ecosystem or by deploying applications directly to the cloud using one of AWS services. The first design proposal was to use Amazon’s virtual machines called EC2 for the middleware, but an application within EC2 cannot be triggered using the AWS event system EventBridge. This would require implementing a new scheduler and event system. A decision was made by the supervisor to avoid ”reinventing the wheel” and instead rely on a serverless cloud solution from AWS. Since AWS Lambda was used as the core component in the architectures of some related works\[10\][1] and mentioned in various guides on how to implement AWS solutions for big data, AWS Lambda seemed like the obvious choice. AWS Lambda, is a serverless solution which enables scripts written in several programming languages (in-
cluding Python and JavaScript) to be deployed and executed directly in the cloud. Lambda script can be triggered by EventBridge by a time schedule, by changes in S3 or by some other AWS service event.

Since a lot of SageMaker functionality depends on the use of input data from S3, data need to be transferred from MongoDB Atlas to S3 by either JSON- or CSV-data format. After the data is stored, a SageMaker notebook could be executed with AWS Lambda and its data stored in a S3 bucket at completion, as used in the work of Leon Radeck. The middleware could then be scheduled to fetch the data from the Amazon S3 bucket and deliver it to a MongoDB collection. However, activating and de-activating a SageMaker notebook seems like an inflexible solution.

A better solution was found while exploring the SageMaker environment and documentation. SageMaker Interference is a part of the SageMaker platform which contains features for deploying machine learning models. A model can be produced using SageMaker and predictions can be made using the model, interference and supplied input data. With interference predictions can be made either in real-time using Endpoints or in scheduled batch jobs using Batch Transform Jobs. A guide on the AWS-website gave inspiration to an idea of using an application external from the AWS-ecosystem and make https requests to a Lambda application which in its turn can use SageMaker EndPoints for predictions. The guide can be found at https://aws.amazon.com/blogs/machine-learning/call-an-amazon-sagemaker-model-endpoint-using-amazon-api-gateway-and-aws-lambda/. This solution was used for the final design in this report and will be further explained in the next section.

3.4 Providing user recommendations

3.4.1 Suggestion of a design for fetching predictions

A premise was that the machine learning model would take some kind of user input and return recommendations, in the shape of a list containing the id of learning material. If all predictions are made in batch daily and then stored in a MongoDB-cluster they can be reached directly from the cluster. However, some users may not log in to the application each day and some users may even be inactive. Thus superfluous processing can be avoided if

VI A provided link on how to do this for the interested: https://medium.com/analytics-vidhya/a-guide-to-schedule-sagemaker-notebooks-a7a09eb641f6
Figure 3: The final suggested prototype. This architecture makes use of HTTPS requests from an external application to get inside AWS and trigger a Lambda application for fetching predictions.

the predictions are made in real-time using Endpoints. According to the AWS documentation the proper way to use Endpoints is to make a HTTPS request from an application to a REST API using AWS API Gateway. API Gateway can then forward the request to a Lambda Function, which is in turn can preprocess data supplied from the request and invoke the Endpoint. After a prediction has been made with SageMaker through the EndPoint it will be returned to the Lambda function which will post-process the prediction and store it in a MongoDB collection. Then the Lambda function can forward the prediction back the application through the REST API. Since the Aline mobile application is written in JavaScript, the application making HTTPS requests will be written in JavaScript using Node.js to make the integration simple for the people developing the mobile application.

API Gateway has the feature of using cache to avoid unnecessary predictions using a cache. If a prediction already exists in the cache, the prediction can be forwarded back directly to the user. Only in the case of a cache miss

\[vii\] A deeper explanation of a REST API is out of scope for this thesis, it can be seen as an abstraction of a "gateway" which allows communication between the AWS ecosystem and the outside world through the use of HTTPS requests.
a new prediction will be generated. But since a MongoDB collection will contain the most recent predictions, the application could instead connect directly to the cluster, check if a prediction was made recently and return it directly to the application, otherwise forwarding the request to the REST API.

3.4.2 A provisional model for recommendations

Due to some delays a machine learning model failed to deliver on time, thus a provisional machine learning model for recommendations had to be implemented in order to make sure the infrastructure worked. The model is based on the movie recommendation model at https://towardsdatascience.com/machine-learning-recommender-engine-with-aws-sagemaker-4892a9e4a858. Aline users can bookmark learning material they like in the the mobile application, and the principle of this model is to divide users into clusters based on those bookmarks. Thus recommendations can be made from bookmarked material of other users within the same cluster. A machine learning algorithm called K-Means is used for dividing users into clusters. A K-Means algorithm, is an unsupervised algorithm, meaning it doesn’t need a label (or target value) to guide the algorithm. The algorithm divides examples (in this case users) into $K \in \mathbb{N}$ clusters based on their features (columns of data). New users can be fitted into an existing clusters, and existing users can be refitted into other clusters if their user behavior change over time.

There are two option on how to fit users into cluster: Either fit all users into clusters in bulk, or fit users into clusters in real-time on demand, using a SageMaker EndPoint. If done in bulk all existing users can be fitted into clusters daily, and predicting recommendations could have the following algorithm:

1. Looking up which cluster the user belongs to.
2. Find all bookmarked learning material within the cluster.
3. Exclude all learning material already bookmarked by the user.

Fitting users into cluster in real-time on demand would mean new users getting fitted into clusters faster, but on the same time they would be fitted with other users with possible zero bookmarks, meaning there could be

---

viiiMachine learning algorithms are out of scope for this report, but feel free to explore K-Means and clustering algorithms if you feel compelled.
very few recommendations. It could also mean quite inaccurate predictions because the lack of bookmarked content. The batch solution of providing recommendations for very recent user may actually be more accurate. Another issue with fitting data on demand could be users getting moved into new clusters, thus leaving current cluster recommendations obsolete. This would result in having to refresh cluster recommendations at each call, possibly multiple times a second if the user base grows large enough. Thus fitting users into clusters in bulk seems like the most reasonable solution. User’s that hasn’t yet been fitted into a cluster could get recommendations based on the coming roadmap 2.0 which contain information of what the user has selected as preferred content at registration as well as the progress toward these goals. Until the roadmap 2.0 is ready a temporary system of random recommendations can be developed as a replacement.

For this solution two application has to be created. One application for pre-processing data and another application for refreshing the user clusters.

3.5 Design of test-data

As stated in the introduction, Aline is still in closed beta. Thus there are not enough of user profiles to test the reliability and scalability of the infrastructure for creating clusters and recommendations. Creating artificial users for testing the data system may not be optimal from a data scientists point of view, however it may be sufficient to test how the system handles heavy loads and how good the system scales. Performance analysis could be performed to test clustering and recommendations for a million users. The artificial user data must still be realistic enough to work with the pre-processing and machine learning algorithms currently deployed with the system. Thus requiring the same columns containing the same data types as the real user data.

4 Implementation

4.1 Creating test-data

For the artificial user profiles to be as correct and realistic as possible, the software first imports the database collection library, containing all learning material. It contains the id, title, a brief description and all meta data such
as subject and media type of each learning material. First of all subjects and media types were extracted from library, then users are generated with randomly picked sets of preference in subjects and media types. Then each user gets assigned two sets - liked and disliked - which are subsets of the learning material. The material in each set is based on which subjects and media types were picked as preferred. The newly generated database is then saved to disk as a CSV-file which can be imported into a MongoDB collection. The application can be run from a terminal and requires the user to specify the number of rows of test data to be created.

4.2 Transferring data

An assumption was that simply using PyMongo to import data directly to the SageMaker Notebook would be the best solution. However, a requirement posed from a company machine learning engineer was to provide files to S3. This way data transfers from MongoDB on each execution of a SageMaker Notebook can be avoided. Another reason for transferring data from MongoDB into S3 is due to that some functionality in SageMaker requires data to be stored in S3 in either JSON- or CSV-format. Thus data needs to be transferred from MongoDB to Amazon S3 daily. The process of transferring data was as following: Import the data using MongoDB and store the data inside a Pandas DataFrame. Then the AWS Data Wrangler-method to_csv was used to store the Pandas DataFrame as a CSV-file to a S3 bucket.

Importing a whole collection may lead high memory usage whenever the collections grows large enough, this will result in the application eventually running out of memory and crash. To avoid the application to crash when transferring huge data sets and thus breaking the principle of durability, the application was built splitting data into chunks of a specified number of rows. The chunks could then be processed for machine learning separately for increased scalability. The algorithm of the program may be explained as follows: Importing a chunk of for example 100 000 rows, exporting them to a CSV-file in a S3 bucket, clearing the memory and continuing with the next chunk until completion. But to let the application wait for the chunk to be written as a CSV-file and exported to a bucket before continuing importing more data is wasteful and Lambda bills the owner by the second. Thus to comply to the principles of scalability and cost, the application was implemented to spawn a new thread at the completion of importing a chunk. The spawned thread will then write a CSV-file, export it to a bucket and then
clear the memory of the data, while the main program continue importing rows of data.

For being able to use non-AWS-libraries so called *Layers* had to be created. The process of creating layers was to set up a local programming environment for Python inside a folder in the AWS Console, install the libraries locally, compress the folder to a zip-file and then uploading the zip-file to S3. The zip-file can then be used as a Layer to import the libraries wanted.

4.3 The recommendation system

4.3.1 Requesting recommendations

As seen in figure 4, requesting a user recommendation from the machine learning algorithm at AWS with the mobile application requires two applications. A *Node.js* application called *recommendation_request* that takes a user id and searches a MongoDB collection for recent predictions. If a recent prediction exists it will be returned immediately. If no predictions

---

Figure 4: A chain of events, from a recommendation request to a response.

---

\textsuperscript{15}The AWS Console is an text-based-interface to a virtual machine tied to an IAM-account.
exists or if the prediction is to old (being more than 24 hours old) user data is imported from MongoDB, then a https request is sent to the API Gateway with the user id and data, which executes the second application `fetch_batch_recommendations` inside AWS Lambda. This application will pre-process the user data and find out which cluster the user belongs to, fetch recommendations from within the cluster, excluding all content already read by the user. If the user doesn’t belong to a cluster or is an outlier (that is being alone in a cluster), then recommendations will be generated at random. This is a temporary solution and should be replaced with data based on the coming roadmap 2.0 mentioned in the design-section.

In the last step the Lambda application returns the information back to the Node.js application which will store the predictions in the MongoDB database and return them to the mobile application. If the Lambda application crashes or cannot be reached, the Node.js-application will regress to using previous recommendations. The connection to the MongoDB cluster is open through the whole process of the Node.js-application to reduce latency.

### 4.3.2 Generating cluster recommendations

![Architecture](image)

Figure 5: The chain of events from datasets at MongoDB to user clusters with recommendations stored at S3.
To create the user clusters, the data had to pass a series of events seen in figure 5, thus four applications was required:

- *Supply_datasets_for_ML* application which pre-process user data and stores it for training or fitting users into clusters created by the K-Means algorithm. MongoDB uses a lot of memory due to caching, so to avoid the program to crash from running out of memory the caller must choose a max allowed chunk size, the program will start exporting when a chunk has reached its max allowed chunk size. The program delegates the work of exporting a chunk to a separate thread while the main thread continues to import data.

- *Recommendations_preprocess* which counts how many chunks of *Learners* that has been transferred and then assigns each chunk to a subprocess called *recommendations_preprocess_worker* to increase scalability. Durability will also be increased since the program will be less likely to run out of memory or be killed by Lambda for taking too long.

- *Recommendations_preprocess_worker* pre-processes a chunk of data and store the data in a folder. The pre-processing of works by first fetching both one chunk of the *Learners*-dataset and the *Learning Material* dataset from S3. Then the learners set take the *Bookmarks*-columns, which contains the id’s of all bookmarked learning material and split the data to separate row. The learner sets then get joined together with the learning material by the id’s of the bookmarked learning material. Now the dataset is split into three sets, which contains 1-hot rows of user material and columns of some property of each of those materials. One set uses *Category*, another *Subject* and the last one *Media_Type* containing the medium of the material. Then the mean value of what subjects, categories and media type each user consumes is calculated. The last step is merging the three sets back together and the results can be seen in figure 6.

- *Recommendations_refresh* ties users to clusters, possibly reassigning users previously assigned to other cluster if their behavior has changed and feeds clusters with new content, thus renewing the user recommendations on a daily basis.

EventBridge is used for scheduling the execution of each application which can be seen in figure 5. A SageMaker Training Job was also created for re-
The Training Job executes a stored model, using the pre-processed input generated by recommendations_refresh. A thing to keep in mind is that each time a Training Job has been executed all former clusters will be absolute, thus each Training Job must be followed by a runtime of recommendations_refresh. An assumption is made that the clusters themselves probably won’t change that often, thus clusters rarely needs retraining and thus no Training Job is scheduled.

A SageMaker notebook was used for experimenting with the model, including pre-processing and deployment of the model as an EndPoint. The SageMaker Breast Cancer Predictions-sample notebook was used as a template when deploying the model.

5 Evaluation and Results

5.1 Moving data between MongoDB and Amazon S3

Testing was performed on test data, containing 100 000 pseudo-users. For these tests a memory size of 1570 mb was specified in Lambda, thus using two processing cores at about 0.88% of their full speed. A single-threaded solution of the application had an execution time of 21.9 seconds, using 413 mb of memory. A multi-threaded solution on the same data had an execution time of about 16.5 seconds, using 603 mb of memory. Even though a method is used to clear the memory of a list before a thread has terminated, the

*For the interested, the sample notebook is described in the following link: https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_applying_machine_learning/breast_cancer_prediction/Breast%20Cancer%20Prediction.html*
memory usage is actually higher than the version of the application not splitting the data into chunks. However, this may be due to the Python garbage collector not executing since the memory is not close to being full.

5.2 Response time of requesting recommendations

The whole chain of events seen in figure 4 took roughly between 3.5-5.5 seconds to perform. The JavaScript Date-object was used to measure time in milliseconds. To find and retrieve a recent prediction from MongoDB took about 0.55 seconds on average. An idea of lowering the time consumption of the whole chain of event was to make use of Amazon S3 when being inside the AWS-ecosystem to avoid having to connect to MongoDB multiple time and instead make all transfers within AWS. But just making a https request from the Node.js-application and make the lambda function return a number immediately without any addition processing turned out to have a time consumption between 1.5-2.3 seconds, thus being much slower than using MongoDB if a prediction has already been processed recently.

Three connections are done to MongoDB in this solution:

1. From the JavaScript application: Retrieving current user recommendation to find out how recent it is.

2. From the Lambda application: Retrieving user data from the user id to make a prediction on it.

3. From the Lambda application: Storing the the prediction to user recommendations.

Since the connection to MongoDB is already open in 1, a thought emerged: "why not retrieve 2 immediately before sending the data to Gateway API?". This showed to actually reduce time consumption of the whole chain down to 2.37-3.23 seconds. This will also reduce time spent inside Lambda, which is important since AWS charge Lambda by seconds executed. Since the Lambda application will actually return the data back to the JavaScript application, another thought emerged: "why not store the new prediction to MongoDB from the JavaScript application directly? Actually, why even let the JavaScript application wait for the data to be stored before returning at all when this can happen asynchronous?". The JavaScript application was
rewritten, almost completely resulting in a time consumption of the whole
chain of events to be reduced to between 1.2-2.6 seconds.

Experimentation was made both on fetching recommendations from clus-
ters stored on S3 and by making a fake requests from an EndPoint using
a BreastCancer model found on https://aws.amazon.com/blogs/machine-
learning/call-an-amazon-sagemaker-model-endpoint-using-amazon-api-gateway-
and-aws-lambda/. The purpose of making predictions on the BreastCancer
model was to test the performance of SageMaker EndPoints before the tem-
poral model was implemented. The time spent on sending and receiving
data back from an EndPoint didn’t differ from fetching recommendations
from the clusters. This results in the conclusion that EndPoints are fairly
fast and effective.

5.3 Generating recommendations

The model was trained and fitted using a SageMaker Notebook. Training the
model took roughly two minutes and deploying the model using an EndPoint
for making predictions took a between three and ten minutes.

The chunks generated by `recommendations_preprocess_worker` are about
0.42% the size of the actual `Learners`-chunk processed. Scalability was tested
with test data of 100 000 pseudo users. Testing was done both with all pseudo
users merged into one process and with four chunks of 25 000 pseudo users
each and running on separate processes. Data was extracted from AWS
CloudWatch and the results can be seen in figure 7. When running the
process with all data, the memory consumption was at most 3806 mb. The
high memory utilization is probably due to the step where the algorithm
is merging each user will all its learning material. During this step 100
rows of data became 4000 rows in earlier experimentation with a SageMaker
Notebook, those 4000 rows contains a lot of duplicated data, but since this
model is a temporary solution further optimizations of the pre-processing
algorithm will not considered.

Fitting users into clusters took less than one second on a dataset con-
sisting of about 100 pre-processed users. The number of users per clusters
may vary from 1 to about 40. Clusters consisting of only one user are con-
sidered outliers, which in this case means users which do not share interests
with other users. These users could instead be fed data generated using the
coming roadmap 2.0. When the K-Means algorithm was set to $K = 20$ (that
is fitting the users into 20 clusters), there was three outliers in total. When
the algorithm was set to $K = 15$, there was two outliers in total.

Since the machine learning model had to be implemented during the last weeks of the report, the maintainability and modularity of the system was put to the test. No changes whatsoever had to be made to the application that fetches data from AWS or the application that transfers data. Only minor changes had to be made to the Lambda application that fetches predictions and since all data pre-processing in the SageMaker Notebooks are made with Python and the Pandas-library, the pre-processing function in Lambda can simply be replaced with a new solution made with a SageMaker Notebook, since Lambda has full support for Python and Pandas.
6 Future work

Extensions to the infrastructure could include expanding the number of databases compatible and making various optimizations. For instance some SageMaker algorithms support incremental training \([6, \text{ p. 1}]\). With incremental training a machine learning model can learn from new data, using the previous trained model. This way only the most recent data would be imported at a specified time interval (for example each day) and SageMaker could perform incremental training on the existing algorithm and therefore both minimizing data transfer and training time.

A request for future expansion from the supervisor was to replace MongoDB all together for storing user data with something "faster for big data", preferable an Amazon-service. Some storage and database services were researched for this. For instance replacing MongoDB for Amazon S3 is not a good solution since data can not be appended to, meaning each user interaction would mean rewriting a new file, which would mean millions of files with millions of users, and as seen in Appendix A: Early implementation of transferring data reading multiple files from S3 is terribly slow. AWS also has its NoSQL database called DocumentDB, but it is build upon the MongoDB API, and thus it is doubtful to expect a large increase in performance. Amazon Athena is an Amazon SQL-database, but it is built upon S3 and seems to have the same performance issues as S3 when working with many small data entries.

Another solution for improving performance of data transfer would be replacing CSV-format for Apache Parquete-format. Parquete files can be reduced to under 10% of the size of a CSV-file containing the same data and increase query performance significantly \([18]\).

7 Conclusion

The initial idea in this report was to use Apache Spark and virtual machines as the center piece for the machine learning infrastructure for Aline. However dependencies on the database MongoDB, the machine learning environment SageMaker and Amazons cloud providing service AWS and requirements from the machine learning engineer at Aline resulted in the initial design being scratched and replaced with a new design based on AWS-services. And due to delays I had to implement the machine learning model myself.
The complete infrastructure solution consists of four parts. The first part of making requests from the mobile application, one part where AWS SageMaker provides predictions, the second part of transferring data from the database to the Data Lake and a third part of pre-processing data and refreshing the machine learning model. An application was also implemented to generate pseudo-data for making sure the system could withstand heavy loads of content. The system as a whole is made with big datasets in mind and the ability to scale in width, which was made possible by transferring data in chunks and pre-process the chunks separately. Thus the system complies to the principles of scalability and durability. Cost-effectiveness has also been taken into account during the whole process. The whole system is built to be modular and has proven to be easy to replace for other machine learning models, thus following the principle of maintainability.

The greatest difficulty of designing and implementing a machine learning infrastructure turned out to be width of knowledge required in multiple domains, spawning from database-providers, the ecosystem of a chosen cloud provider to knowledge of serverless deployment of code. AWS cloud service offerings are vast and according to Amazon there are over 200 services, therefore resulting in the effect of ”one cannot see the forest because of all trees”. The speed of which the field of big data is growing makes guides written only a few years ago absolute or less effective than currently existing solutions. While researching big data infrastructures and data engineering tools, new techniques och services would often be stumbled upon at random. For example SageMaker Interference were found while exploring the SageMaker page on AWS. AWS Lambda and EventBridge was found when researching how to automate the workflow of SageMaker notebooks. Thus a lot of time was spend on research and development and the infrastructure had to be re-designed several times, often during the middle of implementation. But in the end an effective and functioning solution could be implemented. This would be a general solution for a user that aims to create a data pipeline using Pandas DataFrames as the main tool for processing.

8 Appendix A: Early work

This appendix contains information about earlier designs, implementations and results leading to the final infrastructure.
8.1 Technologies used

8.1.1 Apache Spark

Apache Spark is a framework for processing data on cloud clusters [17]. A cloud cluster consists of multiple virtual machines working together to perform some kind of computation. Apache Spark is written in the programming language Scala, but is available in other common languages, such as Java and Python. Apache Spark uses an immutable data structure called resilient distributed dataset (RDD), which are easy to distribute among a cluster and is also fault-resistant. Spark make use of lazy evaluation, and remembers which transformations to perform due to a directed acyclic graph. Since RDDs are immutable, every step in a long sequence of transformations are remembered, this is called lineage. If an RDD is lost, Spark can make use of this lineage and re-process only what is deemed necessary. Apache Spark is a part of the Apache Hadoop-ecosystem and thus expects to use the Hadoop Distributed File System (HDFS) [15], but other file systems are also supported. Spark includes libraries for machine learning, SQL, streaming and graphs, making it a complete solution for processing big data.

8.2 Early designs

Due to previous experience of using Apache Spark, this was thought of as the obvious choice for an integral part of an infrastructure for processing big data.

Amazon offers their cloud solution Amazon EMR which offers a complete Apache Hadoop cluster with master and worker nodes, thus there is no need to set up several virtual machines, install and set up a HDFS-file system and an Apache Spark cluster. Amazon EMR offers out of the box compatibility with Apache Spark and a Hadoop HDFS-compatible file system called EMRFS (EMR File System). The SageMaker Spark-library in its turn allows connecting an Apache Spark-application with SageMaker. An issue with a solution using MongoDB Atlas, Amazon SageMaker and Amazon EMR is that three separate virtual machine clusters will be active at the same time, which results in a waste of money, data and network IO (see figure 2).

A solution for minimizing waste of data and network IO would be to centralize the whole application solution to a single virtual machine cluster. A MongoDB-server could be set up on the Spark master-node, thus keeping compatibility with the app and making the transformation of semi-structured
data to structured data a local network-operation. This way the data won’t have to leave the ecosystem during analyzing and processing, thus reducing network IO and transfer time. But since the machine learning algorithms will be written in SageMaker, some processing will still occur inside of SageMaker’s eco-system. A solution for this would be to preprocess the data using Spark and create the machine learning models with SageMaker, thus cutting costs since the on-demand hour price is more expensive with SageMaker than Apache Spark (see figure 9). MongoDB can be connected to Apache Spark through the MongoDB Connector for Spark, but the connector uses the aggregation functionality of MongoDB rather than Spark Pipelines for filtering and querying data, which may result in some lost benefits of using Apache Spark.  

Another solution for integrating the system would be to keep the whole data system within the AWS-ecosystem by using Spark for both preprocessing and machine learning. An issue with this approach is that the machine learning engineer does all the pre-processing within Pandas DataFrame’s, thus all pre-processing has to be rewritten for Apache Spark. Another issue with this approach is that the SageMaker Spark-connector doesn’t include all of SageMaker’s functionality, thus putting restriction on the work of the machine learning engineer. A solution for this would be to use the Spark machine learning library MLlib and thus getting rid of the SageMaker notebooks all together (see figure 10), but this will interfere with the work of the machine learning engineer.

There exists the possibility to create a Data Lake on MongoDB Atlas and connect it to a S3 bucket. But in this case the data need to be moved from a database to the MongoDB Data Lake daily, alternatively using the MongoDB Data Lake as the main database of Aline. But using MongoDB as the user database is a requirement from the company. Therefor this solution might not improve upon the solution of moving data from MongoDB to S3.

After getting in contact with Aline’s machine learning engineer and getting access to Aline’s GitHub-repo’s and receiving an AWS and MongoDB account the prototypes in figure 9 and figure 10 were both considered impractical. Therefor a third infrastructure is proposed where as little intervention as possible is carried out in the current system architecture. By using middleware both MongoDB Atlas and the SageMaker setup can be kept intact.
Figure 8: On-Demand Pricing for Amazon SageMaker vs Amazon EC2. Price is valid from 31 March 2022, source: aws.amazon.com/
Figure 9: A proposition of a ML-infrastructure using Amazon EMR for storage and preprocessing and connect to SageMaker through a Spark SageMaker connector.

Figure 10: Another proposition keeping the whole ML-infrastructure within the Amazon EMR-ecosystem.
8.3 Early implementation of transferring data

8.3.1 Design and implementation

To avoid huge transfers daily (the database is expected to eventually reach terabyte’s), an idea was that only material uploaded to MongoDB within a recent time interval would be imported and then be exported to Amazon S3. Each batch of material could then be appended to a CSV-file in an Amazon S3 bucket, however Amazon S3 doesn’t allow modifying files, files can only be created, removed or overwritten [12]. So a solution of storing each document as a JSON-file in a folder in an Amazon S3 bucket was thought of, with each file having the size of a couple of kilobytes. This way the most recent material could be imported from MongoDB and appended to a folder at Amazon S3. Since there are some latency of uploading objects to S3 the implementation turned out to be slow, but significant improvements could be reached using parallel-processing using the Python-library threaded, which allows for multi-threaded program designs. The imported data was divided into chunks and then delegated to each own thread.

The program was first implemented on local disk on a personal laptop and then moved to AWS Lambda. Both because of lack of previous experience with AWS Lambda and to make sure the software worked as designed before passing it on to the cloud. However, reading multiple JSON’s from S3 turned out to be incredibly time consuming, therefor the implementation had to be rewritten storing the whole MongoDB collection to a CSV-file in a S3 bucket. Thus losing the ability of only transferring newly generated material.

8.3.2 Results

In early development all testing of moving material was done on a private laptop with the current specification:

- CPU: Intel i7 8550u (4 cores, 8 threads, max frequency of 4 ghz)
- RAM: 8 gb ddr3
- Storage: 250 gb SSD with M.2-connector
- OS: Fedora 26 (a linux distribution from Red Hat Inc)

The method of measuring import and export-times was the Python Time-library, thus measuring the time before invoking a function and measuring
the time when the function is completed. Importing a collection of 1602 documents from MongoDB into memory took about 1 second. Exporting them to an Amazon S3 bucket as separate JSON-files took about 865 seconds which was considered unacceptable. Exporting the same amount of documents as a JSON file took 81 seconds, which is 10X faster. To increase performance the Python-library *Threaded* was used for enabling multi-threaded processing support. Making use of 10 threads decreased the duration of exporting to separate JSON files down to an average of 47.452 seconds, which is an incredible improvement from 865 seconds. Thus the solution of writing separate JSON-files instead of a single CSV-file turned out not only to be more convenient for the task at hand, but also faster even when the entire collection was transferred. Multiple configurations can be seen in the figure 11.

A fixed value for the amount of threads wouldn’t comply with the principles of scalability and maintainability and too simply divide the number of documents with a fixed value - for example 10 - would result in 10 000 documents spawning 1000 threads. A steep increase of threads being used could be prevented using logarithms. \( \log_2(n) \times \log_{10}(n) \) was considered a decent formula for testing, thus resulting in 9 threads for moving 42 documents, 34 threads for moving 1 600 documents and 59 threads for moving 16 000 documents. The conclusion of the formula comes from exporting 1600 documents with various number of threads seen in figure ??.

However, reading JSON’s from S3 with the BOTO3-library *GetObject*-method proved to be very slow, taking about one second per JSON-file. According to *Best practices design patterns: optimizing Amazon S3 performance* each *prefix* (directory) is supposed to be able to handle 5,500 get-operation requests per second [2], but using a multi-threaded solution didn’t speed up the process enough to be of an acceptable level. Other BOTO3-methods was used for trying to improve the performance of fetching JSON-files using iterators and paginators to list all S3 file-locations and using the *get*-method to fetch the files manually. A Python library called AWS Data Wrangler which connects Pandas DataFrames to AWS was used to try decreasing the duration of reading data, but to no success. Another consideration was using *Amazon Glue* which can crawl S3-buckets for data and merge it into several file formats, for example CSV-files or Apache Parquete which is another structured-data format that can be heavily compressed for reducing network transfer and is compatible with Pandas. But this would mean importing data from MongoDB as JSON-files and then using secondary software to convert this data to CSV, and one must also consider that all AWS cloud services
charges for processing time, which mean that using all these steps cannot be justified for reducing the time and cost of transferring data.

Moving data within an AWS data center, from an application running in the cloud to cloud storage would in theory improve performance, but in reality, moving 1620 objects using AWS Lambda turned out much slower than fetching and storing data from MongoDB to S3 with a personal laptop at home. According to AWS documentation, the number of vCPUs are in proportion to the memory used, where 1,769 megabytes of memory provides the equivalence of the full usage of one vCPU [3]. Thus the slow-down when moving from Laptop to Lambda is most probably due to the CPU on my personal computer being much faster than the vCPU on the default settings of Lambda.

With the use of 512 megabytes the export time from the application to S3 took about 53.748 seconds. See figure 12 for other configurations. This leads to the conclusion that the process of importing and exporting data is CPU-bound rather than IO-bound when taking the advantage of parallel processing. Exporting all data at once to a CSV-file gave surprising results when running the application in Lambda. Export time, which both included wrapping all data into a Pandas DataFrame and exporting it to CSV using AWS Data Wrangler resulted in a duration of 0.44 seconds using a 3538 megabytes setup (using two vCPUs). Fetching JSON-objects from S3 was much faster than using AWS than the personal laptop, but still too slow for production. According to the documentation the average latency of fetching an object is 250 milliseconds, testing proved this to be accurate [2]. These results led to a re-design of the transferring system with the use of a single CSV-file instead of multiple JSON-files.
Figure 11: The time duration of exporting 1600 documents from MongoDB to S3 using a personal laptop. The outermost right bar shows data being exported to a single CSV-file using a single thread, while the left bars show data exported to multiple JSON-files using various numbers of threads.
Figure 12: The time duration while performing transfers from MongoDB to S3 in comparison to the size of memory chosen using AWS Lambda. The size of the memory determines how many virtual CPU cores are used. The out most right bar shows data exported to a single CSV-file using one thread while the left bars shows data exported to multiple JSON-files.
9 Appendix B: Code Examples

Listing 1: A loop in the Supply_datasets_for_ML-application which takes chunks of imported data and sends an export job to a separate thread.

```python
# max_size is the maximum documents allowed in a chunk
# threads is a list of executed thread jobs
# docs is the list of documents for the current chunk
# n is a counter for number of documents in the current batch
for doc in cursor:
    if n < max_size:
        docs.append(doc)
        n += 1
    else:
        # When max_size has been reached, write chunk to file
        # 1. Create and execute thread
        full_path = 's3://' + bucket + path + name + str(file_n) + '.csv'
        thread = threading.Thread(target=
            export_documents_thread, args=(docs, full_path))
        thread.start()
        threads.append(thread)
        # Go on with a new chunk
        docs = []  # This will redirect the list pointer to a
            # new memory location
        n = 0
        file_n += 1
        print("Exporting chunk", file_n)
```

Listing 2: The main function in the recommendations_preprocess-application.

```python
# Import data
prefix = "data/learners/"
bucket = "{bucket-name is hidden in this report}"

# Connect clients
s3_client = boto3.client("s3")
lambda_client = boto3.client('lambda')
```
# List number of chunks
response = s3_client.list_objects_v2(Bucket=bucket, Prefix=
          → prefix)
files = response.get("Contents")
number_of_chunks = len(files) - 1

if number_of_chunks < 1:
    raise "Zero files in the 'learners'-folder!"

# Pre-process all chunks of data
for chunk in range(number_of_chunks):
    # Get a JSON containing which chunk the child process is
    → gonna pre-process
    input_load = { "chunk": chunk }

    # Make a 'worker' pre-process this chunk
    lambda_client.invoke(
        FunctionName = 'arn:aws:{bucket-name-is-hidden}:
        → function:recommendations_preprocess_worker',
        InvocationType = 'Event',
        Payload = json.dumps(input_load)
    )

return {
    'statusCode': 200
}

---

Listing 3: The main function in the recommendations_refresh-application.

# Import data
path = "s3://{bucket-name-is-hidden-in-this-report}/data/
learners = wr.s3.read_csv(path=path + "learners/", use_threads=
      → True)["_id", "bookmarks"]

# Get pre-processed data
dataset = wr.s3.read_csv(path=path + "preprocessed/",
      → use_threads=True)
dataset.set_index("user", inplace=True)

# Make data into correct formats
data = dataset.to_numpy()
labels = dataset.index.to_numpy()

# Make predictions
payload = np2csv(data)
response = SAGEMAKER_RUNTIME.invoke_endpoint(EndpointName=ENDPOINT_NAME, ContentType="text/csv", Body=payload)
results = json.loads(response["Body"].read().decode())

# Generate cluster data
generate_clusters(dataset, learners, results, path)

return {
    'statusCode': 200
}

References


