DAT535 - Flight Delay Prediction

Predict Southwest Airlines Co. future delay using Spark and hadoop

Eirik Wilhelmsen
Andreas Nese
University of Stavanger, Norway
eirik@thewilhelmsens.no
andreas@nese.no

ABSTRACT

Air travel has become an everyday mode of transportation, but the increase in flight delays can lead to higher costs for the airlines and growing dissatisfaction among passengers. Machine learning models can be used to predict these delays, potentially improving operational efficiency and customer experience. This paper explores three machine learning models (Linear regression, Multivariable linear regression and Random Forest regression) and one deep learning model (Multilayer Perceptron classifier). The models are evaluated using Root Mean Squared Error (RMSE) and R^2. Additionally, we will investigate system optimization by experimenting with different Spark session configurations.

Our baseline had an RMSE of 30.086 and a R^2 of 0.044, with Multivariable Linear Regression being the best-performing model with an RMSE of 4.78 and 0.976. The testing also showed that allocating 3Gb for executing memory, four cores, and three instances gave the best optimization regarding speed and efficiency.

KEYWORDS

Flight delays, Machine Learning, Spark, Hadoop, MLlib, Medallion Architecture

1 INTRODUCTION

1.1 Context and Motivation

With more planes in the skies than ever, ensuring that flights depart on schedule has become increasingly critical. Delays increase airport congestion and create a corollary effect that disrupts connecting flights, increases operating costs, and inconveniences passengers. This can lead to a distrust in the aviation industry. Therefore, a predictive model for flight delays can help airlines and airports optimize the schedule and give passengers updated information. Such models can also help improve and understand where the biggest delays are coming from.

1.2 Research Problem

As the dataset grows, allocating computational resources efficiently to maintain performance becomes more important. This is particularly important in a distributed SPARK environment, where incorrect allocation can lead to significant runtime overhead.

The main problem for this assignment was investigating how to develop machine learning models and examining their efficiency. This meant we had to construct an environment for the models that optimally allocates the correct cores and memory.

In addition, the software must be able to preprocess an unstructured dataset using SPARK MapReduce functionalities efficiently.

Related Work. Many papers have explored different types of predicting models and approaches to try to find an answer to mitigate flight delays. The paper from Esmaeilzadeh et al. used the SVM model to explore the relationships between flights and delays [11]. They used data from the three main airports in New York, focusing on weather, airport ground operations, demand capacity and flow management. Their results showed that factors such as pushback (50%), taxi-out (48%) and ground delay (40%) were highly associated with delays.

The paper from M.Al-Tabbakh et al. focuses on different decision tree classifiers to analyze delays in Eygpt Airlines flights [19]. Their work concluded that REPtree was the most efficient classifier based on accuracy and running time.

In a study by Ye et al., the flight departure delays at Nanjing Lokou airport were predicted using LR, SVM and LightGBM models. The models were trained using data from March 2017 to February 2018. The results showed that for a 1-h forecast horizon, the LightGBM model was the best performing, with an accuracy of 0.8655 and 6.65 min absolute error, a reduction of 1.83 min from previous results [28].

2 BACKGROUND

In this section, we provide an overview of the technologies used in our project: Open Stack, Apache Hadoop, Apache Spark, medallion architecture and MLlib. These tools, including cloud computing, big data processing, and machine learning workflows, form our project's foundation. Each subsection explains their roles, capabilities, and limitations, thus laying the groundwork for understanding how they contribute to the project.

2.1 Medallion Structure

The Medallion Architecture is a widely used design pattern in big data systems, used to organize data and progressively enhance its structure and quality as it flows through distinct layers [20]. The architecture consists of 3 layers: the *bronze*, *silver* and *gold* layer. There are many advantages to using a medallion structure in a data project. The architecture offers a logical progression of data

Supervised by Jayachander Surbiryala and Tomasz Wiktorski.

Project in Computer Science (DAT535), IDE, UiS
2024.

cleaning, allowing the user to recreate any downstream tables from raw sources [21].

2.1.1 Bronze layer. The bronze layer is the starting point of the medallion architecture, and it is where all the incoming data from external sources systems is ingested and stored. The primary purpose of this layer is to provide a good and unchanged record of raw data, thus ensuring traceability, historical accuracy and flexibility for later processing. The table structure in the bronze layer equals the source systems schemas, capturing the data "as-is". In addition, metadata columns are added to capture information such as load data and time and process identifiers [20].

2.1.2 Silver layer. The data are taken from the bronze layer and matched, merged, conformed, and cleaned to make it more usable for analysis. The silver layer is characterized by quality improvement and data structuring [9]. With the focus on data usability and integrity, the silver layer sets up the data for further applications, such as reporting, advanced analytics and machine learning

2.1.3 Gold layer. The gold layer represents the highest stage in the medallion architecture and is where the data are carefully chosen, combined, and optimized for specific use cases. The purpose of the gold layer is to facilitate decision-making, enable the use of predicting models, and improve efficiency [20].

2.2 Open Stack

OpenStack is a cloud operating system that manages computing, storage, and networking resources across a data center, including virtual machines (VMs). It enables the provisioning and management of VMs, which can be used to run applications, store data, and handle networking tasks within a cloud environment. In addition to being used as a SaaS, Open Stack can also be used to provide orchestration, fault management and service management [25].

2.3 Apache Hadoop

Apache Hadoop is an open-source framework that manages the storage and processing of data. Hadoop uses distributed storage and parallel processing to handle big data, to break it down into smaller workloads that can be run simultaneously [15].

2.3.1 Hadoop Distributed File System (HDFS). HDFS is a distributed file system where Hadoop nodes operate on data that can be found on the node's local storage. This removes any network latency, providing high-throughput access to application data [14].

HDFS works with a main master node called *namenode* and multiple worker nodes called *datanodes*. The *namenode* is the node within the cluster that knows what the data contains and where it belongs. The *namenode* can also be used to control access to files and allocate jobs regarding write, read, remove, etc., across the data nodes [3].

The *datanodes* communicate with the *namenode* to see if they are needed to commence and complete a task. *Datanodes* can also communicate with each other and, therefore, collaborate during file operations [15].

2.3.2 Yet Another Resource Negotiator (YARN). YARN is responsible for managing computing resources in clusters and using them to schedule user applications. YARN performs scheduling and resource allocation across the Hadoop system [15]. The fundamental idea of YARN is to split up the functionalities of the resource manager and the job scheduling/monitoring into different daemons. This involves using a global ResourceManager (RM) and an ApplicationMaster (AM) dedicated to each application [1]. YARN's various components can dynamically allocate resources and schedule application processing.

2.3.3 MapReduce. MapReduce simplifies the distributed programming by providing two key processing steps that developers implement, **Map** and **Reduce**. In the mapping stage, the data is divided into chunks for parallel processing, allowing transformation logic to be applied to each individual chunk. After this, the Reduce takes over to handle aggregating data from the Map set [3].

Map:

The data is split into smaller chunks. Hadoop then decides how many mappers it should use based on information such as chunk size and memory available. Each chunk is then assigned to a mapper. The worker nodes apply the map function to their local data and store it, with the master node ensuring that only a single copy of the redundant input data is processed [3]. The takes the inputs as a 'key, value' pair and produces intermediate outputs of 'key, value' pairs as outputs [2].

Reduce:

The reducing part can not start while the mapper is still in process. All the output from the map with the same key is assigned to the same reducer, which then piles the values for that key. Unlike the map function, the reducer function is not mandatory [3].

2.4 Apache Spark

Apache Spark is a distribution processing system used for big data workloads. It uses both in-memory cashing and optimized query execution to analyse queries on data of any size.

With MapReduce, each step requires reading and writing from a disk, and due to the latency of the disk I/O, Spark was created to mitigate these MapReduce limitations [26].

Spark does this by processing the data in memory, thereby reducing the number of steps required in a job and reusing the data across multiple parallel operations. with Spark, only a single step is required: The data is loaded into memory, the operations are performed, and the results are written back, enabling faster execution [26].

Spark consists of Spark Core and a set of libraries. Spark Core provides distributed task transmission, scheduling, and I/O functionality. It uses a Resilient Distributed Dataset (RDD) as its fundamental data type. The RDD is constructed so that its computational complexity is abstracted for the users [10]. Additionally, Spark manages data operations by aggregating data and partitions across the cluster. It processes the data, either transferring it to another data store or using it in an analytical model. Users are not required to specify file destinations or allocate computational resources for storing or retrieving files [10].

2.5 Apache Spark's Machine Learning Library (MLlib)

Apache Spark's Machine Learning Library (MLlib) is designed for simplicity, scalability and easy integration with different tools. MLlib is built on top of Spark and consists of different learning algorithms and utilities, including classification, regression, clustering etc. MLlib integrates with other Spark components, such as Spark SQL and Dataframes, allowing for preprocessing, munging, model training, and prediction making. MLlib can also be used on already trained models to make predictions on Structured Streaming [22].

2.6 Models

We use several predictive models to analyse our task. Each model has its own strengths and weaknesses. The models covered in this project are linear regression (LR), random forest regression (RF), multivariable linear regression (MLR), and multilayer perceptron classifier (MLPC). These models range from traditional statistical models to more advanced models. The following subsections detail each model with its use cases and characteristics.

2.6.1 Linear regression (LR). Linear regression (LR) is a model used for classification problems. It is quite simple by nature and thus provides easy-to-understand formulas for generating predictions. LR analysis predicts a variable's value based on another variable's value [18]. Due to its simplicity, LR is a good model that often serves as the baseline and benchmark for other models [5]. It allows for an easy understanding of the prediction task and can identify issues with the dataset.

2.6.2 Multivarable linear regression . Multivariable linear regression is a statistical technique that uses explanatory variables in an effort to predict the outcome of a response variable [7]. The goal is to model the linear relationships between two or more independent variables and one independent variable [7].

2.6.3 Random Forest Regressor. Random Forest Regressor is a machine learning algorithm that combines many decision trees into one model. RF models are used both for classification and regression problems [12]. RF uses ensemble learning, meaning it is making and training multiple models trained on the same data and then averaging the results of each model. The final prediction is based on the average of all the individual trees' predictions [4]; this aims to find the best predictive results [6].

2.6.4 Multilayer Perception Classifier (MLPC). A multilayer perceptron classifier is an artificial neural network with multiple layers of neurons called nodes. It is a simple neural network model widely used in supervised learning for both classification and regression problems [17]. The model is organized into layers; the input layer is where the data is first placed, and Then the computations are performed in so-called "hidden layers". The last layer is called the *output-layer* and is where the predictions are made [23].

3 YOUR METHOD

3.1 Design

This chapter describes the processes and methods used to set up a distributed cluster system and argues why our proposed design is sufficient. Our method must be able to handle large amounts of data and distribute the work of reading files, writing to files, creating models and evaluating models. We explain how the design utilizes Hadoop and Spark and then present a design that can carry out preprocessing. Furthermore, we discuss machine learning models and present the models we have chosen to implement: linear regression, multi-linear regression, random forest regression and the deep learning model multilayer perceptron classifier. The choice of models gives us a broad perspective on how simple and heavier models react to different allocations of resources.

3.1.1 Cluster Setup.

Distributed Cluster Structure. The computational infrastructure in this assignment is designed using a distributed cluster environment. This environment consists of a master node, which we have called *namenode*, and three workers, *datanode1*, *datanode2*, and *datanode3*. *Namenode's* purpose is to manage the environment's allocated computational resources and structure the workers' tasks. The workers aim to carry out assigned tasks.

Resource Allocation. Using a distributed cluster environment with the *namenode* and *datanodes* division is a sufficient design choice to handle large amounts of data efficiently. Each node in the cluster is configured with 8GB of RAM, which should be sufficient for the nodes to handle parallel processing without overloading the system. Parallel processing is important when we train models. Furthermore, the nodes are given four virtual CPUs (VCPUs), contributing to increased parallel processing efficiency. Finally, each node is allocated 40GB of local disk space to store temporary files and data containing calculations during the task. These specifications ensure sufficient memory capacity and processing power to perform machine learning in an efficient manner.

Hadoop and SPARK. Using Hadoop on our cluster was a natural design choice as Hadoop is specially designed to handle large amounts of data by distributing work tasks on several nodes [15]. Hadoop uses HDFS for intermediate data storage, which we can further design SPARK to exploit. Since we have designed our SPARK method to run in YARN mode, SPARK also manages to allocate resources on the nodes. However, this design choice means that SPARK depends on both HDFS storage and YARN's resource distribution to function effectively and that YARN requires additional resources that we take from SPARK. Despite these dependencies, this design choice was made because it is a known, well-functioning method for handling large amounts of data [14]. Ultimately, this leads to Hadoop and SPARK complementing each other as SPARK efficiently computes while Hadoop takes care of resource allocation.

Workflow Algorithm. The work distribution in the cluster is based on the FIFO (First In, First Out) algorithm. All tasks are distributed to the nodes in the order they are received. Although this creates predictability in the system, it also creates some limitations. For example, a large task early will prevent small, quick tasks from being run.

3.1.2 Preprocessing. Data preprocessing is an important part of the project. We lifted the raw data out of the bronze layer and transformed it into the silver layer. This improvement ensured

easier further use of the data and allowed us, among other things, to select data for training.

Data Validation. We initially had to look at some essential values to validate the data. First, we knew that the dataset had to have the correct amount of columns so no errors were raised later. Further, it made sense that all planes had a *tail_number*. In addition, we planned to remove anomalies, i.e., instances where the delay was unreasonably high. We also discovered that each instance had a cancelled value, which was 0 if the flight left or 1 if the flight was cancelled; it was reasonable to remove cases where the trip was cancelled. In the end, we planned to replace NaN values with 0. Ultimately, these validation steps became essential to use the data further and avoid any further type errors.

Addressing Duplicate Records. Another essential step during preprocessing is to address duplicates. We planned to use Spark's map-reduce built-in function, distinct(), which removes duplicates. The link at [24] will lead to the Spark GitHub repo, where one can see the distinct function. This design choice could cause a bottleneck since distinct is a relatively heavy operation, including ReduceByKey, which involves significant communication between the nodes to allow shuffle operations. In addition, considerable overhead memory is also necessary to keep track of unique values.

Write To Dataframe. When the data is filtered, we want to write to a data frame so that the data is officially in the silver layer. Ideally, this should be a data frame of the parquet type, as parquet is more professional and efficient than, for example, CSV. A dataframe would allow us to handle further data efficiently and clearly, but a header would also be needed. The design, which involves writing to the dataframe, utilizes HDFS, which means that several smaller parts of the file are stored, which increases efficiency.

3.1.3 Design of method to change SPARK benchmarks values. The method for initializing the SPARK Session was planned to be dynamic so we could efficiently test different benchmark values directly in the code. We thus avoided directly modifying the SPARK config file. The plan was to have a Python function that created a new SparkSession using the SparkConf object that received parameters such as driver and executor memory and the number of instances. This made it easy to adapt the cluster's resources for tests. We planned to investigate the SPARK UI to verify the correct initialization of the session. For later needs, this design could be used to distribute the cluster's resources between work tasks that perform differently depending on the resource distribution.

3.1.4 Machine Learning Models. In this sub-chapter, we will explain the machine learning models further, focusing on how we have used them to predict flight delays. We will also discuss why we have used certain models, the complexity of the task, efficiency within Spark and Hadoop and the trade-off between cost and time were the key factors in deciding which models to use.

Linear Regression. Linear regression (LR) is a simple machine learning model often used as the baseline for predicting results [8]. Due to its simplicity, it is easy to implement and can help to understand relationships between the feature variable and target variable [16]. LR is also useful for optimizing model performance and prediction time. Given its efficiency in both training and prediction, linear regression provides a solid foundation for comparison. One could argue that such accessible models could be skipped, and more complex models should be prioritized. Still, we have included linear regression because it prepares us for the more advanced models and offers valuable insights into leveraging SPARK and Hadoop for effective predictions in a distributed system.

Multivariable Linear Regression. Multivariable linear regression (MLR) was included in our design, which allows us to evaluate how the system handles models that take several variables into account simultaneously. This is particularly relevant to the task, as we are focused on creating efficient models in a distributed environment with Spark and Hadoop. MLR makes it possible to see how the model performs with several input functions, which is important when working with large datasets in a distributed system. In addition, MLR is relatively easy to implement, which makes it well-suited for testing in a Hadoop-Spark environment. This gives us a first step in assessing how the system handles increased complexity while maintaining the efficiency necessary for large datasets and distributed computations.

Random Forest Regression. The random forest regression (RF) was selected because it performs well with both linear and non-linear data relationships. The model uses an ensemble learning technique, combining the results from its individual decision trees, thus improving accuracy and minimising overfitting [27]. RF are more computationally heavy than simpler models like LR and MLR, meaning they are better suited to capture intricate patterns in the data while also enabling us to investigate how to optimize it. RF provides a good middle ground between computational cost and performance. It offers a big step up in performance while also being less computationally heavy than deep learning models.

Multilayer Perception Classifier. Multilayer perception classifier is a type of neural network that captures patterns and relationships from the data [23]. The MLPC model was chosen because it is a much heavier model than the existing ones, and we will be better able to investigate optimization. We saw the need to include at least one deep learning method, as this gave us more understanding of how the Spark session's memory and cores allocation affected the time spent by the model. Finally, the model should capture non-linear relationships between data, i.e. how one value affects another. Ultimately, MLPC was a sensible model to include because it primarily gave us an understanding of optimization and gave good prediction results.

3.2 Implementation

3.2.1 Spark Initialization Functionality. As suggested in the design, we implemented a Spark session initialization method that allowed the dynamic creation of a session with given values to distribute the resources in the cluster. We chose to set some attributes to be fixed, such as overhead memory for the executor and the driver, which are 1 GB and 512 MB, respectively, as well as driver memory, which was set to 1GB. Alongside this, we set executor.memory, executor.instances and executor.cores to dynamic values as these were the most interesting values to change. This was an implementation choice that deviated from the design, but during

implementation we realized that the number of cores deployed in the Spark session plays a significant role when optimizing models.

The number of executor instances is the number of workers running on the system. More workers lead to better parallel processing, which is good when reading large files. Too few workers can cause the cluster not to distribute all its resources, while too many can lead to too much resource sharing, which can have a negative impact.

Executor memory is the amount of memory each executor is allocated. This is interesting because our system relies on buffers when, for example, reading files and lines.Finally, we look at the number of executor cores each worker gets. This will allow a worker to execute more parallel tasks again. A session with three instances and four cores means that all available cores (12) are deployed, but a session with four instances and three cores also deploys all cores. Therefore, we have chosen to look more closely at the relationship between these sessions in practice, which will be tested and discussed in the evaluation chapter. Ultimately, adjusting these three variables will lead to what we believe will be the most significant differences when looking at optimization.

While implementing the dynamic function that creates Spark sessions, we found one essential configuration: "spark.master.yarn". This config makes Spark use yarn and distribute jobs on nodes other than the master node. Our proposed solution configures yarn on Spark in the following way:

```
.config("spark.master","yarn")\
```

3.2.2 Data Preprocessing. During the implementation of data preprocessing, we mostly followed our design plan. This meant we read the raw text dataset to hdfs using `hdfs -put raw_data.txt`. The file is read to the Spark session, which uses RDD (Resilient Distributed Datasets) to manage the dataset across nodes. Each available node runs `process_line()` on its assigned line. Further `Process_line()` filters the file as we described in the design. It divides the line into fields so that we can check individual fields. This way, we remove instances where the flight has been cancelled and mark the line as cancelled. Furthermore, we checked whether *tail_number* is present; if not, the line is marked as missing tail. We then remove anomalies by marking lines where *departure_delay* is more than 1999. Finally, we replace NaN values with 0 so that these values can be used further. `Process_line()` returns a tuple where the first element is the line's marking, e.g. valid (if the line is approved), cancelled, missing_tail or anomaly. Only lines marked with valid are retained and then using `.map()` combined with the rest of the line. We then use distinct to remove duplicates. As described in the design, we were worried that distinct would cause a bottleneck, as it is a heavy operation, but it did not cause any problems. This is presumably due to the fact that the size of the data set is only about 600MB.

Ideally, we wanted to use Parquet when writing the filtered data to a data frame. However, writing to dataframes, in general, was very problematic. First, we could write to Parquet, but it took much longer than writing to CSV. Thus, CSV became the type of data frame of choice. We also had problems creating a header for the data frame. We tried to debug with various AI models. The problem was that the header was stored in a separate part in the hdfs file system, so when we were to read the data frame, it took another part without a header, which ultimately raised an error. The solution was to reduce all computed lines to 1 file using `coalsce(1)`. This contradicts multithreaded functionality, but we argue that storing only one file is sufficient when we operate with a dataset of only 600MB. In addition, since there is only one file, the header is placed at the top, and we can easily read the file with the header.

3.3 Implementing Machine Learning Models with Spark MLlib

In this section, we will provide a proposed solution on how to implement the different models using Spark MLlib. We look exclusively at the airline South West Airlines (WN) as they have the most instances in the dataset, where we try to predict *departure_delay*.

Linear Regression. As described earlier, linear regression (LR) looks at the relationship between one feature and the data we want to predict. Since LR is a simple model in relation to computational resources, we created three individual models that look at three different features to predict *departure_delay*. A global step that had to be implemented across all the different LR models was to initialize which columns in the data set we were going to use. To avoid repetitive code, we therefore created a function for it. First, we used the previous instance's *departure_delay* as a feature. This was the most challenging feature to look at since we had to "make" the previous *departure_delay* column. We solved this by importing lag from *pyspark.sql.functions* and creating a new column called *departure_delay_next* which was the previous instance's *departure_delay*. This also required us to delete the top row from the dataset since the first instance of *departure_delay_next* is NaN. Then we used VectorAssembler to name *departure_delay_next* to features so that the model understands that these are features, visualized in the code below.

```
VectorAssembler(inputCols=feature_columns, outputCol="features")
assembler.transform(WN_flights)
```

Listing 1: Spark Code for Feature Transformation

Then we configure the training, and test split to 0.8 and 0.2, respectively, with `seed=12345` (for ability of recreation). Furthermore, we import LinearRegression directly from MLlib and train the model and then evaluate it. The initialisation of LR means the data is now in the golden layer.

Multivariable Linear Regression. Implementing multivariable linear Regression (MLR) also involved initializing all the features, where, in contrast to LR, we used all numerical columns, apart from *departure_delay*, which is again the variable we were supposed to predict. Ideally, we should have included the departure and arrival airports in features, but since these are not numerical values, the model cannot use them. In essence, features are the big difference between LR and MLR, and as a direct consequence of more features, MLR is a heavier model to train.

Random Forest Regression. Random forest regression (RF) uses the same features as MLR so that we can reuse the new data set created through MLR. Like LR, Spark MLlib also has its own Random Forest Regressor model, which takes, among other things, the features column, the desired output, the number of trees and the depth of

the trees. We set the number of trees to 50 so that the model will be heavier than both LR and MLR to train.

Multilayer Perceptron Classifier. We implemented a multilayer perceptron classifier (MLPC) as the only deep learning model. Ideally, we should have had several deep learning models, as these were the most interesting to examine in relation to how the model coped with changes in allocated resources in the Spark session. However, deep learning models were very problematic to implement, and we only had time for one. We argue that MLPC is a good deep learning model to include since it will add the first neural network to our collection of models.

Regarding the implementation, we mainly followed the Spark MLlib documentation [13] and ChatGPT. The documentation helped us with the model, while ChatGPT helped us implement the bucketizer that divides *departure_delay* into smaller categories. This is mainly done so that the model performs better and is more accurate. Furthermore, we have said that hidden layers 1, 2 and 3 should have 32, 16 and 8 nodes, respectively. Initially, 64, 32 and 16 were used, but this caused some problems with insufficient memory being allocated. Our model uses sigmoid in the hidden layers and softmax in the output layer, as we have not stated anything else [13]. Furthermore, we split training and test as is done in the other models and convert the categorical predictions into numerical ones. We have used categorical predictions since it creates a more accurate model and reduces noise.

4 EXPERIMENTAL EVALUATION

To evaluate our proposed solution, we chose to look at two things in particular. The first is how well the models perform in terms of generating accurate predictions. In this evaluation methodology, we chose to include Root Mean Squared Error (RMSE), R^2-score, and accuracy as these are arguably the most used evaluation metrics, in addition to the fact that we have experience with them from before. The second evaluation methodology is to create different Spark Sessions with different distributions of available resources to see how the models perform in relation to processing time and efficiency.

4.1 Experimental setup

This part of the project describes the experimental setup. We begin by explaining the evaluation methodologies chosen and why and how they have been implemented. Next, we describe the various pre-defined Spark tests we will run on all models to see which type of resource distribution is most efficient. Next, we describe Spark history servers, the chosen solution for observing the results of the different Spark sessions.

4.1.1 Evaluation Metrics: Model Accuracy. The chosen evaluation metrics to determine the accuracy of the models are Root Mean Squared Error (RMSE) and R^2-score. These are two evaluation metrics we have experience with in the past. RMSE tells something about how much the model, on average, misses for each instance. R^2 looks at the variation in the target data; the closer to 1, the better. Finally, we have included accuracy for MLPC since it puts the predictions into categories, so we measure the number of times it chooses the correct category. Other evaluation metrics that could have been included are Mean Absolute Error (MAE) or F1-score, which are not included because our focus was not on whether the models predict well.

When it came to implementing the evaluation methodology, we calculated using Spark MLlib's predefined module, RegressionEvaluator. This was very simple and was the same for all models except the MLPC model. The code for evaluating RMSE and R^2 is:

```
rmse = RegressionEvaluator(labelCol="DEPARTURE_DELAY",
      predictionCol="prediction", metricName="rmse").evaluate(
      predictions)
```

```
r2 = RegressionEvaluator(labelCol="DEPARTURE_DELAY",
      predictionCol="prediction", metricName="r2").evaluate(
      predictions)
```

Here, we define the target as *departure_delay*, compare it with the model's prediction and set the metric to rmse and r2. To calculate the accuracy of the MLPC model, we used MultiClassificationEvaluator, which used much of the same procedure as the other models. We set the target variable against the model's prediction and set accuracy as the evaluation metric.

4.1.2 Evaluation Metrics: System Optimization. When optimizing the code, we defined four Spark session tests in advance. Each session was pre-defined with a different allocation of resources. We started with test_1, in which we gave little resources and gradually gave the tests more resources. Finally, test_4, which we gave too many resources, i.e. more dedicated memory and cores than the cluster has available. The purpose of the various tests is to see how the system reacts to primarily the amount of allocated memory. What happens if the Spark session runs out of memory? And what happens if it gets too much?

To be more specific, we have included a table below showing the allocation of resources for the four tests.

	Test_1	Test_2	Test_3	Test_4
Num driver instances	3	4	3	6
Driver memory	1GB	3GB	3GB	5GB
Driver cores	1	3	4	3

Table 1: Tables of tests with different Spark session resource distribution

To ensure a fair testing ground, each model is run ten times per test, and each run's executing time is recorded. After ten runs, each model's fastest and slowest times are removed. This is to eliminate any outliers or anomalies. The remaining eight runs are then averaged to measure the model's execution performance and efficiency

4.1.3 Spark History Server. The Spark history server is an important tool for monitoring and accessing information about the spark jobs. This allowed us to debug and optimize Spark jobs by rewiring the output from the logs and fine-tuning our project. We utilized SSH-based port forwarding to enable access to the services running on our VM from our local machines. This was achieved by configuring a SOCKS5 proxy that routes network traffic through the VM.

The Spark history server gave us valuable details into the performance of our Spark sessions. The SSH port forwarding mentioned above enabled event logging and configuration access so that we could analyse the job execution metrics in real time. Using the Spark history server, we could see how memory and amount of cores was allocated on the different nodes. This helped us understand what tasks were the most memory intensive, and we could optimize resource allocation. We also tracked each job's execution time, which helped us pinpoint potential bottlenecks that could affect overall performance. The spark history can be accessed via *http://<namenode-ip>:18080*, where <namenode-ip> is the private IP address of namenode.

4.2 Results

In this chapter, we describe the results produced by the models and the results we obtained from the individual Spark session.

4.2.1 Model Prediction Accuracy. As mentioned in the implementation, we created three different independent linear regression models. The three models looked at the relationship between the previous flight's departure delay and departure delay, the relationship between scheduled departure and departure delay, and the relationship between scheduled arrival and departure delay. The best model was the one that used scheduled departure as a feature. It had an R^2 score and RMSE of 30.086 and 0.044, respectively. This means that the model missed an average of 30 minutes for each predicted flight delay. Multilinear regression, however, with significantly more features, performed better. It got RMSE and R^2 of 4.78 and 0.9756, respectively. Note that R^2 should ideally be as close to 1 as possible, so these results were good. In addition, we see that it misses by an average of 4.78 minutes. Random forest regression has the same features as MLR but got relatively worse results, 12.694 and 0.83, respectively. The multilayer perception classifier model performs even worse than RF with RMSE, R^2 and accuracy of 21.35, 0.52, and 0.966, respectively. Below is a table that provides a clear insight into all the models' results.

	LR (baseline)	MLR	RF	MLPC
RMSE	30.086	4.78	12.694	21.35
R^2	0.044	0.976	0.83	0.52
Accuracy	-	-	-	0.966

Table 2: Accuracy between the different models

Note that accuracy on MLPC was calculated based on how many instances the model was able to put into the correct category.

4.2.2 Optimalization Results. This section describes how each model reflects changes in how the cluster's resources are allocated in the Spark session. Each model has its own graph, where the y-axis describes the number of seconds the model or process spent on *test_n* marked on the x-axis.

Below is the graph of how long preprocessing took with respect to each test.

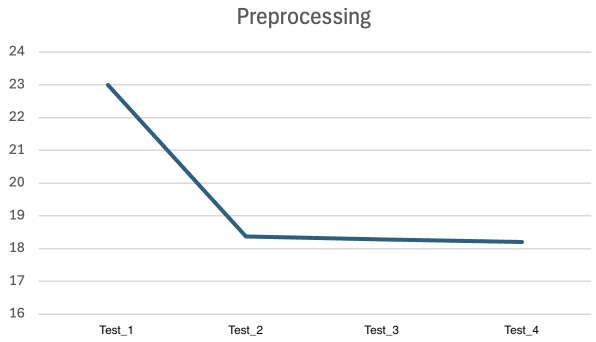

Figure 1: Processing time of data preprocessing with respect to tests

The preprocessing process is the only process where test_4 was the best way to allocate resources. Test_1 was the worst.

Below is the graph of how long linear regression took with respect to each tests.

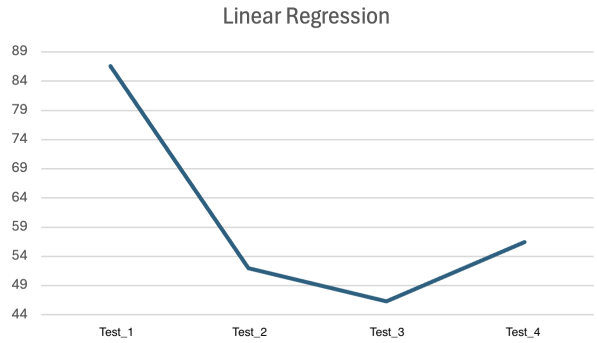

Figure 2: Processing time of linear regression with respect to tests

We see that linear regression responded worst to test_1's resource allocation. However, here there is a greater difference between test_2, test_3 and test_4, with test_3 being the best. With test_3's resource allocation, linear regression took about 47 seconds to make three predictions.

Below, you can see the average time multivariable linear regression took with respect to the tests.

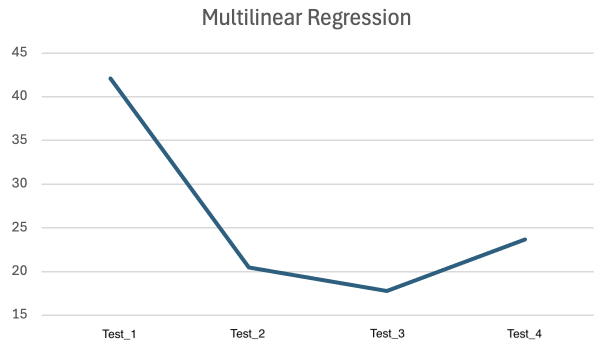

Figure 3: Processing time of multivariable linear regression with respect to tests

Again, test_3 is the best test where we see that multivariable linear regression took an average of about 18 seconds to generate predictions.

Below, you can see the average time Random Forest Regression took with respect to the tests.

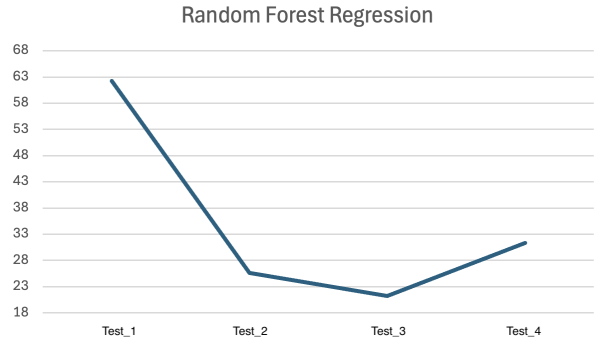

Figure 4: Processing time of random forest regression with respect to tests

Again, test_3 is the best test, where random forest regression takes an average of 22 seconds to generate predictions. This means that MLR, which is better in accuracy, is also faster than RF.

Below, you can see the average time multilinear perception classifier took with respect to the tests.

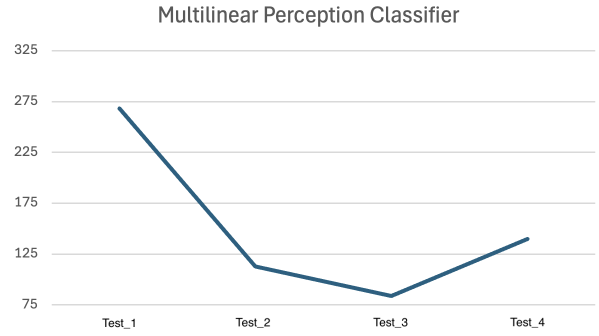

Figure 5: Processing time of multilayer perception classifier with respect to tests

Finally, test_3 is the best on the single deep learning model. One can see on the y-axis that the time MLPC takes is significantly longer than the others. One can also see that the difference in time between test_1 and test_2 is almost 200 seconds, which indicates a successful optimization.

4.2.3 *Discussion.*

Model Prediction Accuracy. Regarding how good our models were at predicting flight delays, we realized that the linear regression model that only looked at the relationship between one feature and departure delay was insufficient. Furthermore, we were positively surprised by the accuracy of MLR, which turned out to be the best model. MLR performed better than RF, although they have the same features. This can be explained by the fact that the variables in the dataset have an approximately linear relationship to each other. Another reason could be that we need to use more trees in the RF model or that the trees need to be deeper. Surprisingly, the MLPC gave even worse results than RF. Again, we argue that this may be due to a linear relationship in the dataset. However, the model predicts categorically, and since accuracy is almost equal to 1, the model manages to put the predictions in nearly all correct categories. Still, when these values are made numerically again, they become poor.

Optimalization. Through testing, we better understood the relationship between allocated resources and performance optimization. In test_1, we allocated insufficient memory (512 MB) and only one core per executor. This setup led to poor performance across all models, as reflected in the benchmark results, which was expected. Observing the Spark UI, we noticed that no jobs required more than approximately 600 MB of executor memory, which is why test_1 was poor resource allocation, as instances had to share workloads since they had not been given enough memory.

The Spark observation made us realize there was a bottleneck regarding the number of cores in the cluster. The maximum number of workers we can allocate while ensuring each has at least one core is 12 (the cluster has 12 cores). For the cluster's RAM to be exceeded, each worker must be allocated over 2 GB of memory (12 * 2 = 24) - an amount they can receive but do not fully utilize in practice, as no job requires more than 600MB.

The fact that there was a bottleneck in the number of available cores was something we anticipated, and that is why we added test_2 and test_3. These tests look at the ratio between instances and cores in practice. Test_2 has four instances and three cores, and test_3 has three instances and four cores. This means that test_2 should be better at small tasks because it has more instances, but since it has few cores, it can make heavier jobs take longer. Test_3 allows each process to run more things at the same time, and that should be able to handle larger amounts of data, but since there are fewer processes, parallelism is minimized.

In practice, the two tests make a marginal difference between all system processes. However, test_3 is best overall, with the exception of data preprocessing. This means that our system is characterized by larger and heavier tasks, and test_3's allocation of 4 cores per instance makes the instances work faster.

It was somewhat surprising that test_4's resource allocation was the most efficient method for data preprocessing. The point of the

test was to illustrate that when we allocated too many resources, the efficiency would be lowered since unavailable resources would try to be allocated but fail. We set 6 instances and three cores, meaning that only four instances get cores before there are no more available cores (12). We thought that preprocessing held sufficient performance because the remaining two instances without cores could perform I/O operations. However, according to the Spark history server, only three instances were initialized because we allocated too much memory (5GB), so test_4 was initialized with three instances, four cores, and 5GB of memory. Thus, test_4 and test_2 are similar except for memory allocation, which is also not used up. We remain unsure why the data preprocessing process was so efficient with test_4's resource allocation method.

5 CONCLUSION

This thesis investigated how different machine learning models and resource allocation optimization in a distributed cluster can be used to predict flight delays for Southwest Airlines. The results show that balancing resources such as memory and cores in Spark significantly impacts the performance of both computing and model execution. The test_3 configuration (three instances, four cores per instance) delivered the best results for most processes, highlighting the importance of carefully tuned resource usage in distributed environments.

Regarding the model's accuracy, multivariable linear regression (MLR) was the most accurate, with the lowest RMSE and highest R^2 value, reflecting the model's ability to effectively handle linear relationships in the dataset. Random forest and multilayer perceptron classifier performed less well, partly due to the linear nature of the dataset and resource allocation challenges for heavier models such as MLPC.

Furthermore, the project demonstrated how optimizing Spark configurations can significantly improve performance. This includes using Spark UI and Spark history server to identify bottlenecks, such as insufficient memory allocation in test_1, and understanding the relationship between the number of cores and instances in different workloads.

In conclusion, this project has demonstrated that using distributed tools such as Spark and Hadoop provides challenges and opportunities for handling large data sets. By adapting resource usage to specific requirements of both data and models, predictive analytics for flight delays can be improved, which can contribute to better operational efficiency and customer satisfaction.

5.0.1 Future work. Future work includes dynamically changing the distribution of resources before each individual process to increase efficiency. In addition, one can find an explanation for why test_4 was best at data preprocessing. This thesis has yet to look closely at changing the allocated driver memory, which can be looked at in more detail. Finally, we have yet to look so much at optimizing our models regarding accuracy. For example, one can make the airport column a numerical value so Spark MLlib can use this data.

6 ACKNOWLEDGMENTS

- ChatGPT and Grammarly has been used in this project to help with reformulations and enhance the readability of the thesis.
- ChatGPT and Copilot have been used during the development.

7 GITHUB REPOSITORY

The Github repository can be found at: https://github.com/EirikWilhelmsen/DAT535-flight_delay_2024/tree/main

REFERENCES

[1] [n. d.]. Apache Hadoop 3.4.1 – Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[2] [n. d.]. MapReduce 101: What It Is & How to Get Started. https://www.talend.com/resources/what-is-mapreduce/
[3] 2021. What is MapReduce? https://www.databricks.com/glossary/mapreduce
[4] AnalytixLabs. 2023. Random Forest Regression — How it Helps in Predictive Analytics? https://medium.com/@byanalytixlabs/random-forest-regression-how-it-helps-in-predictive-analytics-01c31897c1d4
[5] Baseline. [n. d.]. Baseline Model - an overview | ScienceDirect Topics. https://www.sciencedirect.com/topics/computer-science/baseline-model
[6] Nima Beheshti. 2022. Random Forest Regression. https://towardsdatascience.com/random-forest-regression-5f605132d19d
[7] Rebecca Bevans. 2020. Multiple Linear Regression | A Quick Guide (Examples). https://www.scribbr.com/statistics/multiple-linear-regression/
[8] Bluechip. 2023. Top 8 Machine Learning algorithms – Bluechip AI Asia, AI Development Company. http://www.bluechipai.asia/category/top-8-machine-learning-algorithms/
[9] ChaosGenius. 2024. Medallion Architecture 101—Inside Bronze, Silver & Gold Layers. https://www.chaosgenius.io/blog/medallion-architecture/
[10] DatabricksSpark. 2019. Apache Spark. https://www.databricks.com/glossary/what-is-apache-spark
[11] Ehsan Esmaeilzadeh and Seyedmirsajad Mokhtarimousavi. 2020. Machine Learning Approach for Flight Departure Delay Prediction and Analysis. *Transportation Research Record* 2674, 8 (Aug. 2020), 145–159. https://doi.org/10.1177/0361198120930014 Publisher: SAGE Publications Inc.
[12] GeekForGeek. 2019. Random Forest Regression in Python. https://www.geeksforgeeks.org/random-forest-regression-in-python/ Section: AI-ML-DS.
[13] Guide. [n. d.]. Apache Spark. https://spark.apache.org/docs/3.5.3/ml-classification-regression.html#multilayer-perceptron-classifier
[14] HadoopDataBricks. 2021. What is Hadoop Distributed File System (HDFS). https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs
[15] HadoopGoogle. [n. d.]. What is Hadoop and What is it Used For? https://cloud.google.com/learn/what-is-hadoop
[16] Hwang. [n. d.]. A new simple iterative reconstruction algorithm for SPECT transmission measurement - Hwang - 2005 - Medical Physics - Wiley Online Library. https://aapm.onlinelibrary.wiley.com/doi/full/10.1118/1.1944288
[17] Stephen Jeske and MarketMuse. [n. d.]. What is a Multilayer Perceptron (MLP) - Multilayer Perceptron (MLP) Definition from MarketMuse Blog. https://blog.marketmuse.com/glossary/multilayer-percpetron-definition/
[18] IBM LinearRegression. 2021. What Is Linear Regression? | IBM. https://www.ibm.com/topics/linear-regression
[19] Shahinaz M. Al-Tabbakh, Hanaa M. Mohamed, and Zahed H. El. 2018. Machine Learning Techniques for Analysis of Egyptian Flight Delay. *International Journal of Data Mining & Knowledge Management Process* 8, 3 (May 2018), 01–14. https://doi.org/10.5121/ijdkp.2018.8301
[20] Medallion. 2022. What is a Medallion Architecture? https://www.databricks.com/glossary/medallion-architecture
[21] MedallionDataWiki. [n. d.]. Medallion Architecture. https://dataengineering.wiki/Concepts/Medallion+Architecture
[22] MLlib. 2018. Machine Learning Library (MLlib). https://www.databricks.com/glossary/what-is-machine-learning-library
[23] MLP. [n. d.]. Multilayer Perceptrons in Machine Learning: A Comprehensive Guide. https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning
[24] Mridul Muralidharan. [n. d.]. spark/core/src/main/scala/org/apache/spark/rdd/RDD.scala at master · apache/spark. https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala
[25] OPENSTACK. [n. d.]. Open Source Cloud Computing Platform Software. https://www.openstack.org/software/
[26] What is Spark. [n. d.]. What is Spark? - Introduction to Apache Spark and Analytics - AWS. https://aws.amazon.com/what-is/apache-spark/

[27] Mohit Uniyal. 2024. Ensemble Learning: A Comprehensive Guide. https://www.appliedaicourse.com/blog/ensemble-learning/

[28] Bojia Ye, Bo Liu, Yong Tian, and Lili Wan. 2020. A Methodology for Predicting Aggregate Flight Departure Delays in Airports Based on Supervised Learning. *Sustainability* 12, 7 (Jan. 2020), 2749. https://doi.org/10.3390/su12072749 Number: 7 Publisher: Multidisciplinary Digital Publishing Institute.