DAT535 Project Report

Create a data processing pipeline follows Medallion Architecture

University of Stavanger, Norway

Ainaz Rafiei
Foroozan Ebrahimzadehbadeleh
283597@uis.no
280074@uis.no

ABSTRACT

Data generation has increased significantly globally as a result of rapid technological advancements like the Internet of Things (IoT). Due to the rise in data generation, a particularly potent processing system like Apache Spark was needed. Apache Spark is a widely used tool for large-scale data processing. These processes are fast, memory-efficient, versatile, and resistant to errors. The goal of this project was to use an Apache Spark-deployed data processing pipeline based on the Medallion Architecture to predict UK property values. Our pipeline processes house price data in separate batches requires basic pre-processing (missing data, categorical feature encoding, and quantitative feature normalization), and builds models using low-level programming languages without libraries. We used grid search and cross-validation to increase the performance of Random Forest and Gradient Boosting approaches. Delta Lake, on the other hand, was used to track incremental changes to data. Finally, a complete data set with order and coherency is produced, suitable for producing accurate predictions and being used in applications such as dashboards.

KEYWORDS

Distributed and Parallel Computing, Pipline Processing, Medallion Architecture, Big data, Apache Spark, Resilient Distributed Datasets

1 INTRODUCTION

Due to technological advancements, big data and the Internet of Things have grown in recent years. These advances have considerably increased the demand for scalable and efficient data processing frameworks that can handle large amounts of data while remaining fast and accurate [1]. As a result of this demand, these businesses must implement systems for processing massive amounts of data rapidly and reliably. One of the pillar technologies in the big data ecosystem is Hadoop, an open-source framework for distributed storage and processing of very large datasets using the Hadoop Distributed File System (HDFS) and the MapReduce model, which it uses for batch processing. However, since it has given way to more advanced frameworks, less complex in design, fit for faster execution and better value for real-time analytic needs, its encouragement has become muted over the years [2].

Supervised by Tomasz Wiktorski.

Project in Data Intensive (DAT535), UiS,
2024.

Apache Spark is one of the most popular solutions for processing big data because of its memory-oriented architecture, high-performance computing, and fault resilience [3]. To predict house prices, we had to build an end-to-end pipeline for processing big data, including housing market data.

The UK housing market is affected by factors such as interest rates, levels of income, and supply-demand factors. By analyzing these data, we aim to provide valuable insights for stakeholders, buyers, and sellers to help them understand changes in the price of real estate. [4].

Our goal is to build a scalable pipeline using Apache Spark to predict housing prices based on financial and geographic data. The pipeline uses the Medallion Architecture for data ingestion, cleaning, and analysis [5]. We also trained machine learning models such as Random Forest and Gradient Boosting to boost prediction accuracy, shaping the pipeline into an effective tool for extracting useful insights from UK housing data [4].

Data processing is a crucial component in analyzing large datasets related to housing prices. With Apache Spark, we can preprocess, clean, and analyze this data to find important insights. The most important part of the data pipeline focuses on data cleaning. After cleaning, the data is structured for quick access, enabling the deployment of predictive models that predict housing prices based on key features.

Related Work. Apache Spark has seen widespread adoption in big data analytics, with research highlighting its suitability for tasks that require rapid in-memory data processing and fault tolerance [6, 7]. Apache Spark has become a solution for big data analytics over the years. What makes it special is how it can process data using computer memory, making it much faster than older tools. If something goes wrong, Spark can also fix itself and keep running. It's great for tasks like machine learning and can easily split work across many computers to get things done faster. This is why so many companies now use Spark for their data projects [8]. Existing works such as those by Zaharia et al [2] and other Spark contributors have demonstrated Spark's performance in handling machine learning tasks with distributed data [7].

However, we have focused on the specific application of house price prediction or the adoption of Medallion Architecture within a Spark-based pipeline. Our work builds on these studies by applying Spark's powerful tools to an unstructured pipeline designed for a specialized use case, emphasizing both scalability and predictive accuracy.

With inspiration from earlier Spark projects, we built a system to predict house prices using Spark and the Medallion Architecture.

Our work shows how Spark can be easily adapted to fit specific needs and handle growing amounts of data, even in unique situations.

Here's what we did in our project:

- Created a data pipeline using Medallion Architecture to predict housing prices.
- Used MapReduce algorithms for data cleaning and data preprocessing.
- Generated two machine learning models (Random Forest and Gradient Boosting) and enhanced their accuracy using cross-validation.
- Integrated Delta Lake for efficient incremental data updates, allowing the pipeline to process real-time data changes.

All stages of this project, from data ingestion to data cleaning, data processing, and modeling to evaluation, are explained in a fully documented Jupyter notebook [9].

Disclosure Statement: We used an AI writing tool (Grammarly) to help make our writing clearer, but all the work and ideas are our own. We checked everything carefully to make sure it was accurate.

2 BACKGROUND

Apache Spark is a tool for big data analytics with the capability to handle big data over different systems based in memory processing. Unlike earlier methods, Spark is very flexible for a wide range of analytical operations, as it is great at both batch and real-time data processing. In addition to its ability to apply a variety of data sources and programming languages, Spark has emerged as an outstanding choice for today's data engineering [10].

2.1 Spark

Definition: Apache Spark is an open-source analytics engine built for large-scale data processing. It provides high-level APIs in popular languages like Java, Scala, Python, and R, which makes it easier to build applications across different fields. Spark's architecture supports various types of workloads, including batch processing, real-time analytics, machine learning, and even graph processing [3].

Main features:

- **Speed:** Instead of always reading from the computer's hard drive, Spark keeps data in memory.
- **Ease of Use:** It is also super easy to use. Whether you're comfortable with Python, Java, or other programming languages, you'll find Spark feels natural to work with.
- **Flexibility:** It can work with pretty much any type of data storage you throw at it, whether your data is in Hadoop, Cassandra, HBase, or even Amazon's S3. It's like having a universal adapter that fits all your data plugs.
- **Unified Engine:** Spark is an all-in-one solution. You don't need different tools for different jobs - Spark handles everything from basic data analysis to complex machine-learning tasks under one roof [11].

2.2 Spark application

A Spark application is a distributed program designed to process large datasets across multiple computers. It includes:

- **Driver Process:** The execution coordinators manage the lifecycle, task scheduling, and communication with executor processes.
- **Executor Process:** The distribution agents are responsible for running individual tasks and returning back the communicating results. [10].

Spark applications can be written in languages like Scala, Java, Python, and R and can run on cluster managers such as Spark's standalone manager, Hadoop YARN, or Kubernetes. For this project, we used PySpark, Spark's Python API.

2.3 Data pipelines

A data pipeline includes all of the methods by which data moves through systems. It consists of a series of steps that are carried out in a specific order, with the output of one step acting as the input for the next step.

There are usually three key elements: the source, the data processing steps, and finally, the destination, or "sink." Data can be modified during the transfer process, and some pipelines may be used simply to transform data, with the source system and destination being the same. [12]

Types of data pipelines:

- **Batch Processing Pipelines:** Define the fundamental building blocks that handle large volumes of data at scheduled intervals, like monthly reports or ETL processes. [13]
- **Stream Processing Pipelines:**Handle real-time data so that it can be analyzed right away, which is perfect for use cases like live monitoring or fraud detection. [2]

For this project, we used a batch-processing pipeline using OpenStack which is a cloud platform that provides resources for large-scale data processing.

2.4 Cloud usage in data pipeline

Cloud-base data pipelines clean, transform, and store data using a variety of cloud platform functionality (e.g., analytical functions and data warehouse). These pipelines also have benefits such as the scalability and adaptativeness of cloud infrastructure that allows straightforward scaling up or down to different tasks, and ease of maintenance. However, ensuring smooth pipeline performance isn't without its challenges. Delays in data processing and integration between different systems are issues that are prevalent. Solving these is dependent on a high level of prioritization of ensuring data quality and efficiently managing the flow of the data to avoid performance bottlenecks and prevent the system from slowing down. [3]

2.5 Medallian Architecture

The Medallion Architecture [14] structures data processing into three layers in data lakehouse setting in order to guarantee data quality.:

Layers of Medallion Architecture:

- **Bronze Layer (Raw Data):**
  As the foundation of the Medallion Architecture, it stores and ingests raw data from various sources in its native format.
- **Silver Layer (Cleansed and Conformed Data):**
  For covering the data cleaning, validation, and normalization in intermediate stages, apply this layer which provides a standardized view of data, suitable for detailed analysis.
- **Gold Layer (Curated Business-Level Data):**
  The output layer is based on enhanced data tailored for analytical, reporting, and machine learning and helps deliver insight that is actionable by business use.

2.6 Data abstraction in Spark

Spark offers different abstractions to manage distributed data processing:

- **Resilient Distributed Datasets (RDDs):** RDDs are the building block of Spark, immutable, distributed data containers of objects. They support parallel processing and fault tolerance by allowing the system to reconstruct lost data if there is a failure.
- **DataFrames:** Based on RDDs, DataFrames offers a tabular representation of data in the form of a relational database. They are optimized and provide a higher-level API for both formatted and semi-formatted data.
- **Datasets:** Datasets are powerful typed, immutable collections of objects, combining RDDs and DataFrames, support compile-time type checking and optimize for the manipulation of structured data [15].

3 METHODOLOGY

3.1 Data ingestion

Data ingestion is an important step in preparing the data to be analyzed and predicting. This stage involves several steps as follows.

3.1.1 . Identifying Use Case and Relevant Data

The main goal of this research is to analyze and predict UK residential market prices from housing transaction data. Our dataset contains relevant data about property sales, such as transaction IDs, prices, dates of transfer, types of property, and geographical location info and geographical data.

3.1.2 . Preparing Cloud Setup

In order to achieve effective data processing, we configure a cloud infrastructure based on OpenStack, which offers sufficient scalability for processing large datasets. Specifically, in this environment, we used the Hadoop Distributed file system (HDFS) for long-term data storage with an inherent degree of durability, fault tolerance, and scalability, fault tolerance, and scalability through data replication across multiple DataNodes. Acting as an in-memory compute engine, Spark reads and writes data from HDFS in an optimized way to read data directly from distributed nodes on multiple nodes.

Hadoop was installed by specifying the cluster having one NameNode to manage the metadata and several DataNodes to store data blocks. The cluster was configured with default replication factors in order to deliver both fault tolerance and data locality. Next, Spark was installed and merged with Hadoop to provide distributed data processing by configuring the master node and one or more worker nodes based on YARN (Yet Another Resource Negotiator) for resource management. Therefore, Spark can provision resources in the cluster in a dynamic way, which allows Spark to carry out parallel tasks as they need to.

With this setting, Hadoop and Spark easily integrate with each other, allowing effective data loading, storage, and distributed processing at scale on a cluster containing multiple nodes.

3.1.3 . Modifying Data to Simulate Real-World Scenarios

For more accurately simulating real-world data situation, we changed the dataset so as to look like an unstructured data. Real-world data is noisy and comes with inconsistencies, missing values, and disorder, and this stage enabled us to recapitulate those imperfections.

For the first, we introduced into the Date of Transfer the inconsistencies in the form of the insertion of spaces between characters, mimicking variations that may occur in the field due to the date formatting.

Then, we completed the incorporation of unstructured rows by introducing random characters into various fields and mangling their ordering representing the randomness of real-world data.

We also replicated some rows randomly to simulate typical data pitfalls such as duplicated entries.

Last, we transformed the dataset from structured DataFrame to robust distributed dataset (RDD). This transformation allows effective massively parallel and distributed processing and prepares the data for the next stage of the pipeline. Those changes also transformed a formal dataset to a informal form, paving the way for real-world data application difficulties.

3.2 Data cleaning

This subsection describes the procedure followed in order to clean and preprocess the raw data so that it is usable for analysis and predictive modeling tasks. Data cleaning plays a essential role in improving the quality and accuracy of data, thereby influencing the quality of analysis outputs.

3.2.1 . Identifying preprocessing steps

We first studied the raw data in order to detect quality problems including the existence of duplicate rows, missing values, and the irregularities of the format. According to this analysis, we described the required preprocessing steps.

3.2.2 . Reordering the data

The first step of our data cleaning pipeline was to shuffle the data to present an unstructured format and then to reorder columns such that each row includes the necessary data consistently. In this stage, it is crucial for further processing and analysis, since it enables us to precisely select and use the data necessary for our predictive model.

We provided a function which rearranges the columns in accordance with strucutred criteria and contains checking functions to verify that the data conforms to the expected patterns.

3.2.3 . Removing duplicate rows

After reordering the columns, we proceeded to remove duplicate entries from the dataset. Duplicate rows can skew analysis and lead to incorrect predictions. We implemented a routine using Spark's MapReduce capabilities to identify and eliminate these duplicates efficiently.

Code 1: Pseudo-code: Remove duplicate rows using MapReduce

```
RDD_key_value = RDD_reordered.map(lambda row: (row, 1))
RDD_unique = RDD_key_value.reduceByKey(lambda a, b: a).
    map(lambda x: x[0])
```

3.2.4 . One-hot encoding

After removing duplicate rows, the next step in our data-cleaning process was to apply one-hot encoding to categorical variables. Categorical features, such as the Town/City in our dataset, need to be converted into a numerical format to be suitable for machine learning models. We particularly concentrate on encoding the Property Type, Old/New and Duration, and label encoding on the Town/City field, in the context of our application— UK housing prices condition analysis and prediction.

Code 2: Pseudo-code: One-hot and label encoding

```
function create_category_map(data, category_index):
    category_map = empty dictionary
    for each row in data:
        category = row[category_index]
        if category is not in category_map:
            category_map[category] = next available
                number
    return category_map

function encode_row(row, category_map):
    property_type = row[2]
    encoded_property_type = [1 if property_type == 'S'
        else 0,
                             1 if property_type == 'T'
                                 else 0,
                             1 if property_type == 'F'
                                 else 0]
    old_new = row[3]
    encoded_old_new = [1 if old_new == 'N' else 0]
    town_city = row[4]
    encoded_town_city = category_map[town_city] if
        town_city in category_map else -1
    duration = row[5]
    encoded_duration = [1 if duration == 'F' else 0]
    return combine(encoded_property_type, encoded_old_new
        , encoded_town_city, encoded_duration)

function encode_data(data, category_map):
    encoded_data = []
    for each row in data:
        encoded_row = encode_row(row, category_map)
        encoded_data.append(encoded_row)
    return encoded_data
```

3.2.5 . Normalization of price column

Normalization is an essential preprocessing step, particularly for numerical features, e.g., real property prices. We used Min-Max normalization on the price column to map the prices from 0 to 1. This avoids making the price feature on a comparable level to other features, which may help improve machine learning algorithm performance.

Code 3: Pseudo-code: Normalization of Price Column

```
price_min = RDD_unique.map(lambda row: int(row.split(',')
    [0])).min()
price_max = RDD_unique.map(lambda row: int(row.split(',')
    [0])).max()

def normalize_price(row):
    fields = row.split(',')
    price = int(fields[0])
    normalized_price = (price - price_min) / (price_max -
        price_min)
    fields[0] = str(normalized_price)
    return ','.join(fields)
```

- **Extracting year and month from "Date of Transfer"**
  For the detection of seasonal trends, we derived the year and month from the Date of Transfer. Use this for the capture of temporal characteristics, for example, changes in the housing prices across months and years.

Code 4: Pseudo-code: Extracting Year and Month from Date of Transfer

```
def extract_year_month(row):
    fields = row.split(',')
    date_of_transfer = fields[1]
    year = date_of_transfer.split('-')[0]
    month = date_of_transfer.split('-')[1]
    fields[1] = year
    fields.insert(2, month)
    return ','.join(fields)
```

3.2.6 . Data Profiling

We performed data profiling to compute amount of minimum, amount of maximum, and amount of average of the cleaned dataset. This data is used to shed light on the wider picture of the housing market pricing, and thus to facilitate descriptive analysis and reporting.

Code 5: Pseudo-code: Data Profiling

```
price_RDD = RDD_final.map(lambda row: float(row.split(',
    )[0]))

min_price = price_RDD.min()
max_price = price_RDD.max()
avg_price = price_RDD.reduce(lambda a, b: a + b) /
    price_RDD.count()
```

- **Distribution of property types**
  We also counted the number of occurrences of each type of property to discover the trends and common types in the dataset.

Code 6: Pseudo-code for Property Type Distribution

```
property_type_RDD = RDD_final.map(lambda row: row.
    split(',')[3])

property_type_distribution = property_type_RDD.map(
    lambda x: (x, 1)).reduceByKey(lambda a, b: a +
     b)
```

- **Distribution of towns/cities**
  Similarly, we computed the number of each of the towns/city's all combinations for analysis of location representation and how it is related to pricing.

Code 7: Pseudo-code for Town/City Distribution

```
town_city_RDD = RDD_final.map(lambda row: row.split
    (',')[6])

town_city_distribution = town_city_RDD.map(lambda x
    : (x, 1)).reduceByKey(lambda a, b: a + b)
```

- **Price distribution by year**
  The analysis of price distribution on the time scale enables us to detect trends in housing market. We grouped the prices by year to calculate the average price by year to see long term trends and shifts in the housing market.

Code 8: Pseudo-code for Price Distribution by Year

```
year_price_RDD = RDD_final.map(lambda row: (row.
    split(',')[1], float(row.split(',')[0])))

year_price_distribution = year_price_RDD.mapValues(
    lambda x: (x, 1)) \.reduceByKey(lambda a, b: (
    a[0] + b[0], a[1] + b[1]))

avg_price_per_year = year_price_distribution.
    mapValues(lambda x: x[0] / x[1])
```

- **Repartitioning the RDD**
  In order to best prepare for the future pipeline, we repartitioned the RDD on the basis of the Town/City key, which provided better data distribution for model training.

Code 9: Pseudo-code for Repartitioning the RDD

```
  RDD_keyed_by_city = RDD_final.map(lambda row: (row.
      split(',')[6], row))
RDD_repartitioned = RDD_keyed_by_city.repartition(8)
RDD_repartitioned_cleaned = RDD_repartitioned.map(lambda
    x: x[1])
```

By following these steps, we have a comprehensively processed dataset which is now clean, normalized and ready to analyze and model. Normalized price and extracted date features improve the usability of the dataset for machine learning algorithms.

3.2.7 . Saving clean data as Parquet format

After preprocessing and cleaning the dataset, we archived the cleaned data to Parquet format. Parquet is a compact, columnar storage file format suitable for use by big data processing sets such as Apache Spark. The primary reasons for choosing Parquet:

- **Reduced Storage Space:** Parquet files are built for fast storage, which leads to a smaller storage footprint in general than formats such as CSV or JSON. This is especially relevant when working with big data because it saves data storage costs.
- **Faster Query Performance:** Saving the data in Parquet format, we are able to further accelerate the data processing tasks in the Gold layer of our Medallion Architecture. Thanks to Parquet columnar storage paradigm, query execution and read operations are highly efficient and important for analytics and model training.
- **Schema Evolution:** By supporting complex data types and schema evolution, Parquet is more suitable to deal with evolving data structures through time [16].

3.3 Data serving

According to our use case, the data serving phase consists in the development of a predictive model for predicting housing price and the mapping of processed and cleaned data to concrete informations. In this stage the suitable machine learning algorithms for price prediction are defined and selected and also implemented on the cleaned dataset.

3.3.1 . Business Use Case

The main purpose of this work is to build a model to predict the cost of UK residential houses on the basis of economic, geospatial, and property-tailored variables. This model can be beneficial to a wide range of stakeholders—real estate agents, buyers, sellers, and policymakers— by providing them with insight into using data to inform property investments, valuation, and market trends. Using Spark's distributed processing power, the model can work well with large datasets and is therefore ideally suited for inclusion into real-time dashboards or batch analytics systems.

After predictions are produced they can be delivered to downstream systems or analytics tools for visualization and subsequent analysis. These findings can be embedded in dashboards, reports, or APIs, so that stakeholders can see the current state of the market and take decisions based on the most likely values of housing prices.

3.3.2 . Relevant Algorithms

For a reliable predictive model, we apply two regression algorithms, Random Forest Regressor and Gradient Boosting Trees (GBT), which are both adaptive to non-linear dependence structure in structured data.

(1) **Random Forest Regressor:**
- **Description:** The Random Forest algorithm is an ensemble approach that builds a large number of decision trees and combines the result of the decision trees to increase the robustness and precision of the model. This approach is particularly useful in scenarios with high-dimensional data or when the relationships between features are complex and non-linear.
- **Implementation:** We use Spark's `RandomForestRegressor` with a parameter grid for tuning hyperparameters like `maxDepth`, `numTrees`, and `maxBins`. This helps us optimize the model's performance by finding the best parameter settings through grid search and cross-validation.

Code 10: Pseudo-code: Define and train the Random Forest Regressor

```
rf = RandomForestRegressor(featuresCol='features'
    , labelCol='Price')
rf_pipeline = Pipeline(stages=[rf])
paramGrid = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [5, 10, 15]) \
    .addGrid(rf.numTrees, [20, 50, 100]) \
    .build()

crossval = CrossValidator(estimator=rf_pipeline,
    estimatorParamMaps=paramGrid, evaluator=
    evaluator, numFolds=3)
rf_model = crossval.fit(final_train_data)
```

(2) **Gradient Boosting Trees (GBT):**

- **Description:** Gradient Boosting is another ensemble technique, where the models are built in sequence with each model adjusting errors made by its predecessors. This method is very effective to regressor tasks and is also promising for enhancing prediction performance in structured data.
- **Implementation:** We apply Spark's `GBTRegressor` with cross-validation and grid search, tuning hyperparameters such as `maxDepth`, `maxBins`, and `maxIter` (number of boosting rounds).

Code 11: Pseudo-code: Define and train the Random Forest Regressor

```
gbt = GBTRegressor(featuresCol='features',
     labelCol='Price', maxIter=100)
paramGrid = ParamGridBuilder() \
    .addGrid(gbt.maxDepth, [5, 10, 15]) \
    .addGrid(gbt.maxBins, [32, 64]) \
    .build()

crossval_gbt = CrossValidator(estimator=
     gbt_pipeline, estimatorParamMaps=paramGrid,
     evaluator=evaluator, numFolds=3)
gbt_model = crossval_gbt.fit(final_train_data)
```

Both algorithms were chosen for their ability to handle complex data patterns and make precise predictions. By using grid search and cross-validation, we ensured that each model was fine-tuned for optimal performance in predicting UK housing prices.

3.3.3 . Incremental Data Management

For up-to-date loading of the model with the current data, we implemented incremental data management with Delta Lake. With Delta Lake it is possible to manage and update data in a straightforward way and, as a result, it is perfectly suited to real-time stream of data applications. Through storing data into Delta Lake, we are able to batch incorporate new data into the current data set periodically, without having to reprocess the whole database. This approach ensures that the model can be retrained on fresh data, enabling it to produce current predictions for downstream applications.

- **Setting Up Delta Lake:** The cleaned and processed training data is first written to Delta Lake format for efficient storage of structured data with ACID transaction support. New data is also loaded in Delta format to facilitate efficient merging of new data and data already present.

Code 12: Pseudo-code: Store processed data in Delta Lake

```
DeltaDataPath = "hdfs://namenode:9000/user/ubuntu/
     FinalProject/Delta_trainData"
final_train_data.write.format("delta").mode("
     overwrite").save(DeltaDataPath)
```

- **Incremental Updates:** The latest data is loaded intermittently into storage as a Delta table and matched with the historical data using a given identifier (e.g.,ID). This feature is enabling incremental updates, where newly added records only need to be pasted into an existing dataset, thereby avoiding data repetition and maintaining data accuracy.

Code 13: Pseudo-code: Merge new data into the Delta table if any updates are available

```
if DeltaTable.isDeltaTable(spark, DeltaDataPath):
    delta_table = DeltaTable.forPath(spark,
         DeltaDataPath)
    delta_table.alias("existing").merge(
        new_data_df.alias("new"),
        "existing.ID = new.ID"
    ).whenNotMatchedInsertAll().execute()
```

With Delta Lake, incremental data management, it is simplified data updates and the pipeline is kept efficient. Storing data in Delta format preserves data validity and provides a convenient way to process new records, hence the model is trained on the latest data available at all times.

4 EVALUATION

Evaluation is a key part of measuring the efficiency of our predictive models and guaranteeing they produce good and trustworthy results. During this part, we evaluate the predictive potential of the Random Forest and Gradient Boosting models with important error metrics to determine their accuracy and resilience. Below, we describe our evaluation procedure and present the results achieved.

4.0.1 . Experimental setup

In this work, we benchmarked our results by creating a scalable platform using Apache Spark for distributed data processing and Delta Lake for incremental data management. Our experiments aimed to assess the pipeline's efficiency and accuracy in predicting UK housing prices. Below, we summarize the configuration, data preprocessing, and evaluation metrics employed in the experiments.

4.0.2 . Dataset Description

Our experiments are based on the UK Housing Prices Paid dataset available on Kaggle. This dataset contains detailed information regarding property sales in the United Kingdom and contains the key data points needed by our predictive model. The original dataset includes the following the features:

- **Transaction Unique Identifier:** A unique ID assigned to each transaction.
- **Price:** The sale price of the property.
- **Date of Transfer:** The date when the property was sold or transferred.
- **Property Type:** The type of property, such as detached, semi-detached, or flat.
- **Old/New:** Indicates if the property is new or pre-owned.
- **Duration:** Indicates if the property has a freehold or leasehold status.
- **Town/City, District, County:** Geographical identifiers of the property.
- **PPDCategory Type:** Classification of the property on the basis of transaction type (i.e., standard vs. additional property).
- **Record Status:** The status of the record (e.g., add, delete).

After data cleaning and preprocessing, we extracted a subset of columns having the highest implication on the prediction model construction. The final attributes used are:

- **Price:** Target variable for prediction.
- **Town/City:** Represents the city or town where the property is located.
- **Duration:** Specifies whether the property is freehold or leasehold.
- **Old/New:** Indicates if the property is new or pre-owned.
- **Property Type:** Type of property, such as house or flat.
- **Date of Transfer:** Date of transaction, which was decomposed into year and month for modeling.

These preprocessed features were subsequently transformed, encoded and arranged in a way that would allow the predictive model to forecast housing prices from past record transaction data. [4]

4.0.3 . System Setup and Resource Configuration

The experimental environment was placed on virtual machines with configuration of Apache Spark and Delta Lake, generating a distributed platform that efficiently manages big data. Each virtual machine was configured with:

- **Operating System:** Ubuntu Linux
- **Memory Allocation:** Spark drivers and executors memory were augmented to 6 GB each to accommodate large volume of data and processing needs.
- **CPU Cores:** More CPU cores were assigned to Spark executors for parallel processing, which increased both speed and efficiency of data processing and model training.
- **Cluster Configuration:** Data was cleaned in a Spark cluster environment with a single or multiple large partitions of a minimum 100 partitions ensuring optimized use of memory and minimized data shuffles.

These configurations allowed our pipeline to handle large datasets efficiently, maintaining manageable training and evaluation times.

4.0.4 . Generating Load and Parameter Setup

In order to assess the scalability and efficiency of our model, we created a synthetic load by continuously querying the dataset and running batch jobs to represent real-world workload scenarios. The key parameters configured in our Spark setup included:

- **Memory Configuration:** The driver and executor memory were fixed at 6 GB each, with shuffle partitions set to 300 to well support large data shuffles.
- **Data Repartitioning:** The dataset was split 100 times to better manage memory and reduce data transfer bottlenecks between nodes.

Detailed set up instructions such as environment setup, dependent setup and parameter setup are provided in the project's GitHub repository. The following procedure guarantees reproducibility for any researcher or developer that wishes to reproduce the experiments.

4.0.5 . Define Evaluation Metrics

- **Root Mean Squared Error (RMSE):** RMSE quantifies the average magnitude of prediction error, and assigning higher weights to larger errors. It also offers an indication of how well the model is performing by comparing estimated versus actual prices.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$$

- **Mean Absolute Error (MAE):** MAE reflects the average magnitude of errors contained in a set of predictions, independent of their orientation. Unlike RMSE, MAE gives equal weight to all errors, making it less sensitive to large outliers. A lower MAE indicates that the model's predictions are closer to the actual values on average.

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$$

- **R-squared (R^2):** R^2 indicates the proportion of variance in the dependent variable that is predictable from the independent variables. A higher R^2 value indicates a better fit of the model to the data.

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}$$

The trained model, evaluated using these metrics, achieved an RMSE of X and an R^2 of Y, demonstrating its effectiveness in predicting housing prices. The trained model was subsequently stored in HDFS and made available for further predictions and updates.

4.0.6 . Calculating Errors on Training and Test Data

Using these metrics, we evaluated both models on the training and test datasets. This paradigm allowed us to evaluate the performance of each model on unseen data (test set), and evaluate how good those models are in real-world scenarios.

Code 14: Pseudo-code for calculating evaluation metrics

```
    predictions = model.transform(data)
    rmse = evaluator.evaluate(predictions)
    mae = evaluator.setMetricName('mae').evaluate(
        predictions)
    r2 = evaluator.setMetricName('r2').evaluate(
        predictions)
    print(f"{dataset_name}-RMSE:{rmse},MAE:{mae},R  :{r2}
        ")
    return rmse, mae, r2

# Random Forest
(train_rf_rmse, train_rf_mae, train_rf_r2) =
    calculate_errors(best_rf_model, final_train_data, "
    Random␣Forest-Train")
(test_rf_rmse, test_rf_mae, test_rf_r2) =
    calculate_errors(best_rf_model, final_test_data, "
    Random␣Forest-Test")

# Gradient Boosting
(train_gbt_rmse, train_gbt_mae, train_gbt_r2) =
    calculate_errors(best_gbt_model, final_train_data, "
    Gradient␣Boosting-Train")
(test_gbt_rmse, test_gbt_mae, test_gbt_r2) =
    calculate_errors(best_gbt_model, final_test_data, "
    Gradient␣Boosting-Test")
```

Note: For this evaluation, predictions were generated using a 10 percent sample of the test data. This approach enabled us to not only decrease the computational cost and achieve an efficient result, but also to capture important performance trends. This subset provides a representative evaluation; however, predictions on the entire dataset can be performed as resources allow, enhancing scalability.

5 RESULT

Our experiments validated that the enhanced pipeline is able to process amount large data and produce price predictions. Key observations include:

- **Resource Utilization:**
  Increasing memory and CPU allocation in Spark led to quicker processing time due to reduced training and prediction time. This enhancement is critical to the scalability of such a model to larger data sets.
- **Incremental Data Management with Delta Lake:**
  Delta Lake enabled appropriate handling of the new data so that the model could be re-trained using the most-recent data without having to reprocess the full dataset. This approach ensures that predictions remain current as new data is integrated.
- **Model Performance:**
  Using grid search and cross-validation, we optimized the Random Forest and Gradient Boosting models to enhance prediction accuracy. However, the evaluation metrics—particularly RMSE and MAE—indicate potential for further improvement. The test RMSE values remain relatively high, suggesting some deviation from actual values, while the low R^2 values imply that the models capture only part of the data's relationships. This indicates that additional tuning or feature engineering may further improve model performance.
- **Sample Predictions:**
  Sample predictions from both models demonstrated some alignment with actual prices; however, the low R^2 values and error metrics suggest that model predictions may lack precision on certain data points. This highlights a potential need for further improvements in model architecture, feature selection, or additional data preprocessing.

Normalized Sample Prediction		
Price	RF_Prediction	GBT_Prediction
1.4155	7.7769	6.5862
1.0010	0.0026	0.0028
1.5065	8.0965	6.3830
2.0121	8.8797	6.9225
2.0121	0.0082	0.0143

Table 1: Comparison of Prediction Results

- **Model Comparison:**
  Comparison is done on the basis of the major evaluation metrics, training durations and the generalization capability of the models on unseen data.

Model Comparison			
Model	RMSE	MAE	R^2
Random Forest (Train)	0.003644	0.000885	0.085059
Random Forest (Test)	0.086664	0.000882	0.086664
Gradient Boosting (Train)	0.093022	0.000873	0.093022
Gradient Boosting (Test)	0.096235	0.000869	0.096235

Table 2: Comparison of Evaluation Metrics

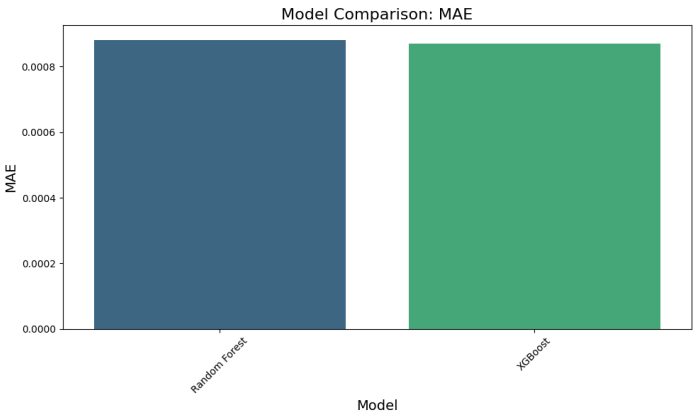

Figure 1: Mean Absolute Error (MAE) results.

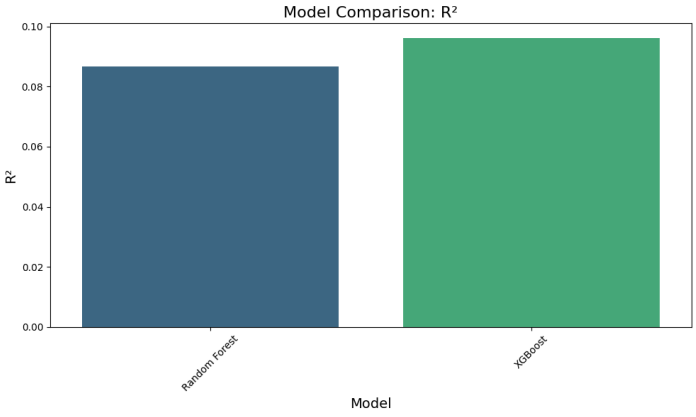

Figure 2: R-Squared (R^2) results.

Based on the evaluation metrics, although the two models have obtained similar results, Random Forest appears to offer a slightly higher tradeoff between error reduction and generalization, and thus it is a realistic choice for the study dataset. However, additional optimization is likely to be required in order to enhance the predictive ability of these models.

- **Sample Prediction Analysis:**
  The resulting metrics and the R^2 values indicate that the models may have limited accuracy for intricate pricing scenarios. Random Forest slightly over Gradient Boosting in

generalization, as evidenced by its slightly lower RMSE and higher R^2 on a test set.

- **Training and Prediction Time:**
  Another consideration is the computational efficiency of each model. In practice:
  - Random Forest: This model is typically more expedient to train than Gradient Boosting as a result of its parallelism and reduced computational load per iteration.
  - Gradient Boosting: This is more computationally expensive since trees are essentially constructed one at a time, where mistakes are corrected as the tree is built. This is a slower train, however it may offer slight advantages in prediction.

  In our experiments, the Random Forest model completed training and predictions faster, which is advantageous for large datasets or applications requiring frequent retraining. Time efficiency ofRandom Forest thereby facilitates its scalability and suitability for real-time/near real-time applications, whereas Gradient Boosting may be more appropriate when accuracy is a constraint instead of one of the limiting factors.

- **Recommendations:**
  Considering evaluation metrics, sample predictions, and computational engagement, we suggest Random Forest as the major model in this project. However, Gradient Boosting might provide minor improvements in some instances but due to its high computational cost to little accuracy, it is a more sensible option to model for scale and generalization.

  Future work could be an additional tuning of both models or testing for other algorithms including neural networks that can improve accuracy but maintain an efficient computational operation. Further, the use of a full dataset (as are beyond the 10 percent subsets used here) for training and evaluation, can bring further understanding of the performance of the model.

  The full implementation including all steps necessary is available in the Jupyter Notebook [9].

6 CHALLENGES

Building this big data processing pipeline from the ground up carried with it many challenges, to test both technical mastery and problem-solving skills. These challenges provided a good amount of teaching opportunities and a point of reference for subsequent progress. Below, we present a brief overview of some of the main difficulties faced during the project.

- **Challenges with Unstructured Data**
  Among the first problems encountered was the searching for and the preparation of an adequately sized dataset according to the requirements of the project because we wished to simulate a less ordered data environment. The selection and alteration of an appropriate dataset took up a lot of time. This experience also highlighted the need to carefully examine data and its structure prior to pipeline development. In many real-world applications, data is required to be transformed or enriched in order to fulfil particular project objectives thereby increasing the pipeline complexity.
- **Data Cleaning Challenges**
  Efficient data cleaning with Spark's MapReduce routines posed another challenge. The data shuffling process, essential to MapReduce, led to inefficiencies and longer processing times. Specifically, our accidental use of functions such as collect(needlessly generated high CPU load as it unknowingly pushed the bulk of the data to the driver node. This experience showed that profiling and fine-tuning each step is critical for achieving maximum resource utilization. We also obtained significant knowledge about how to model data-cleaning pipelines in distributed environments and how to avoid generic errors.
- **Model Performance and Resource Constraints**
  Resource limitations resulted in training and evaluation limitations during our models. We could only use a 10 percent sample of the data for testing and evaluation, which restricted the model's exposure to the full dataset. While this method enabled us to measure performance effectively, it is not possible it may not represent the maximum accuracy that can be obtained using the complete dataset. Furthermore, the hyperparameter tuning for models such as Random Forest and Gradient Boosting was computationally expensive and required us to trade model complexity for the amount of resources available to us. This challenge brought into focus the need for scalable infrastructure for processing big data and complex models.
- **Incremental Data Management**
  Integration of Delta Lake for incremental data management was an area of learning, as we migrated and adapted it to support batch data updates without having to reprocess the whole dataset. Creation of Delta Lake for efficient management of new data entailed technical changes to preserve data integrity and minimize redundancy. Experience highlighted the difficulty involved in maintaining time series large data volumes and underlined the necessity of sophisticated data management tools in big data environments.
- **Lack of Prior Experience**
  Our main goal of this project was to attempt to build a large data pipeline and this project was our first step to creating such a pipeline with a steep learning curve. Challenges, such as efficiency of resources, and incremental data handling, emerged from our very limited experience with these technologies. Overcoming these obstacles required extensive research, experimentation, and a willingness to learn from mistakes. Notwithstanding these challenges, the project has offered us, enormous hands-on experience to improve our skills in distributed computing and big data processing.
- **Troubleshooting Hadoop and Spark Cluster Setup**
  Sometimes when we try to SSH into the remote server, we get a connection timeout error to handle that requires a hard reboot of Spark. Afterward, a "Connection Refused" error appeared when starting Hadoop's DFS. When Hadoop services non-implementation was proved to us, the services were restarted using stop-dfs.sh and start-dfs.sh. Finally, the

NameNode and DataNode were manually started, and write operations were enabled.

7 CONCLUSION

This work was achieved using an Apache Spark scalable data processing pipeline and the Medallion Architecture to make UK housing price predictions. The combination of high technology (e.g., Delta Lake for incremental data update) and ensemble machine learning (Random Forest and Gradient Boosting) allowed us to manage large datasets and drive meaningful insights.

Although the final model demonstrates overfitting and needs to be fine-tuned for better generalization, the project's constraints (time constraints and limited practice with the implementation of machine learning models) limited the final accuracy. Although it has been challenging, the work resulted in establishing a scalable and reliable data processing pipeline.

The project encountered a few challenges, for example, insufficient computing power, lack of experience in building reliable ML, and managing technical problems associated with distributed systems.

Despite these challenges, the project succeeded in building a scalable pipeline, demonstrating the feasibility of Spark-based solutions for big data analytics. It is also important that future work aims to improve feature engineering, hyperparameter optimization, and consideration of different algorithms to improve prediction performance. Extending the analysis to the complete dataset and refitting the pipeline for real-time scenarios would benefit the utility of the tool to stakeholders in the housing market even more.

In summary, this work demonstrates the use of Spark pipelines for big data processing and predictive analysis, and it provides a good starting point for future research on big data analytics.

REFERENCES

[1] Lionel Sujay Vailshery. Number of iot connections worldwide 2022-2033, 2024.
[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
[3] Amazon. What is apache spark?, 2024.
[4] Kaggle. Dataset, 1995.
[5] Fady GA. implemented the medallion architecture using apache spark and apache hdoop, 2024.
[6] Reintech. Utilizing spark's in-memory computing for fast analytics, 2024.
[7] Apache Spark. Apache spark research.
[8] Analytics Insight. Introduction to apache spark for big data processing, 2024.
[9] Foroozan Ebrahimzadehbadeleh and Ainaz Rafiei. Jupyter notebook, 2024. Available at https://github.com/ForoozanE/Create-a-data-processing-pipeline/.
[10] Databricks. What are spark applications, 2024.
[11] Apache Spark. Apache spark™ - unified engine for large-scale data analytics, 2024.
[12] Databricks. What is a data pipeline?, 2024.
[13] Batch Data. What is a batch data pipeline?, 2024.
[14] Databricks. Medallion architecture, 2024.
[15] Apache Spark. Apache spark, 2024.
[16] Apache Spark. documentation for apache parquet, 2024.