HW - 3 DIC
CSE - 587

Samya Bijal Shah
50604267

1 Task 1.1: Edit Distance Computations

1.1 Introduction and Description

Edit distance measures the minimum number of operations required to transform one string into another. Here, three different implementations for computing pairwise edit distances are compared:

- **For-loop**: Nested loops.
- **Multi-process**: Multiprocessing.
- **Spark**: Distributed computation using spark.

1.2 Methodology

```
python edit_dist.py --csv_dir /path/to/csv --num_sentences n
```

The table below shows the execution times: 1.

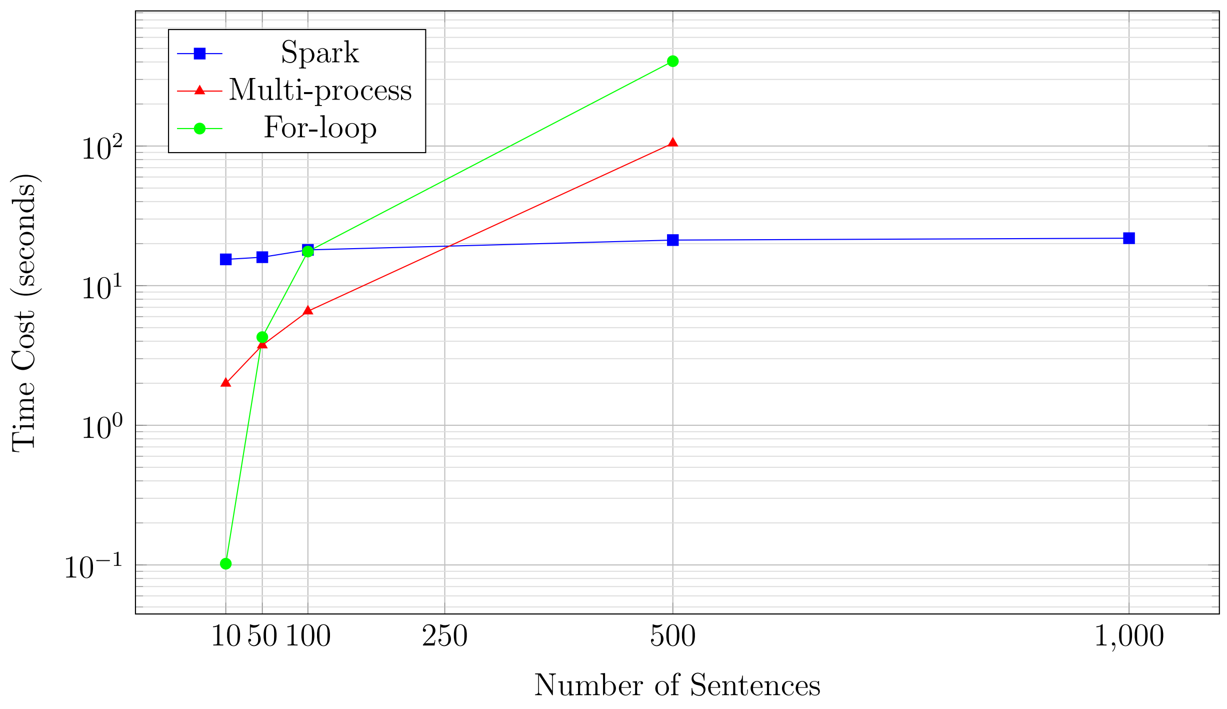
1.3 Results

1.3.1 Table

Table 1: Execution Time (seconds) for Edit Distance Implementations

Number of Sentences	Spark	Multi-process	For-loop
10	15.424	1.994	0.102
50	15.991	3.748	4.280
100	18.038	6.563	17.554
500	21.206	104.717	405.334
1000	21.874	–	–

1.3.2 Performance Trends



1.4 Conclusion

The for-loop has scalability issues and quadratic time complexity, hence the slowest for larger datasets. In contrast, the multi-process and Spark-based methods leverage parallelism, with Spark much more efficient on larger scales due to optimized distributed processing.

2 Task 1.2: MLP Inference

2.1 Introduction and Description

Inference performance for machine learning models, including MLP, is crucial in real-world applications and big data analytics. This task evaluates the inference phase of an MLP classifier using two different implementations:

- **Spark-based**: Spark's distrubuted framework.
- **Non-Spark**: Normal Way.

2.2 Methodology

The `MLP.py` script was executed using the following command structure:

```
python MLP.py --n_input n --hidden_dim d --hidden_layer l
```

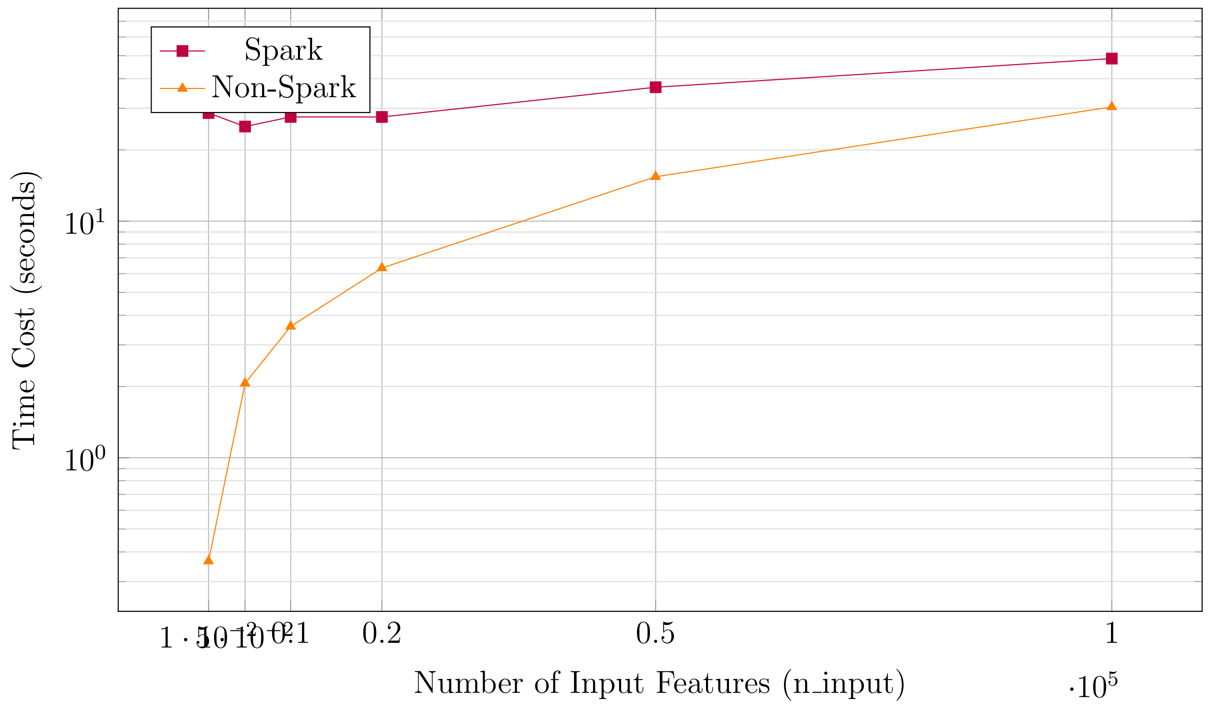
2.3 Results

2.3.1 Execution Times

Table 2: Execution Time

n_input	Spark	Non-Spark
1000	28.637	0.366
5000	25.108	2.064
10000	27.572	3.590
20000	27.560	6.339
50000	36.807	15.405
100000	48.649	30.393

2.3.2 Performance Trends



2.4 Conclusion

Non-Spark MLP is better than Spark: it executes faster and scales linearly for all input sizes. Spark has much higher overhead that restricts its efficiency, making it less suitable for smaller data sets, while perhaps scalable for larger ones.

3 Task 1.3: Flock Move Simulation using Spark

3.1 Introduction and Description

In this task, the provided simulation code (`bird.py`) was updated to utilize Spark for parallel processing of position updates for all birds. A separate Python script, `bird_spark.py`, was developed to implement the Spark-based version. Both Spark and non-Spark implementations were tested with varying numbers of birds to evaluate performance differences.

3.2 Methodology

The `bird_spark.py` script was executed using the following command structure:

```
python bird_spark.py
```

This script was used with different numbers of birds. These are respectively 200, 1000, 5000, 10000, and the maximum number of each simulation is 500 frames. Once the simulation was done, it printed the following format, that is, the average time-cost per frame:

3.3 Results

3.3.1 Execution Times

Table 3: Average Time Cost per Frame (seconds) for Flock Move Simulation Implementations

Number of Birds	Spark	Non-Spark
200	0.3932	0.0178
1000	0.4390	0.1354
5000	1.6353	1.6229

3.4 Conclusion

Thus, for smaller sizes of flocks, the plain non-Spark implementation outperforms Spark due to its higher overhead. While increasing the size of the flock allows Spark's parallelism to reduce the gap, the non-Spark version maintains a slight performance advantage even at larger scales.