Big Data and Machine Learning Applications

Portfolio Assessment

Student ID 1: 2089114
Student ID 2: 2275122

Course: MSc Data Analytics
Year: 2025

Total Word count: 3,948

Table of Contents

Big Data and Machine Learning Applications Lab Book

Word count: 1998

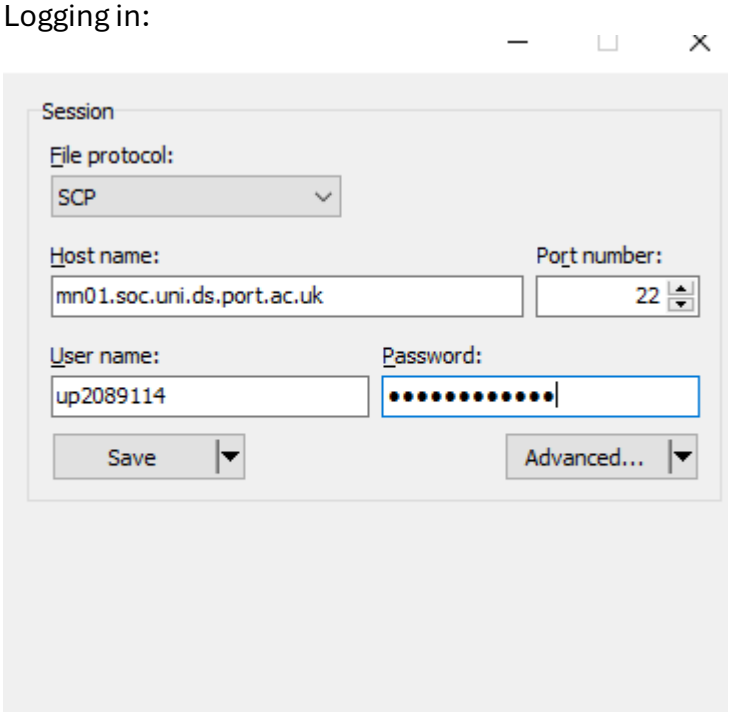
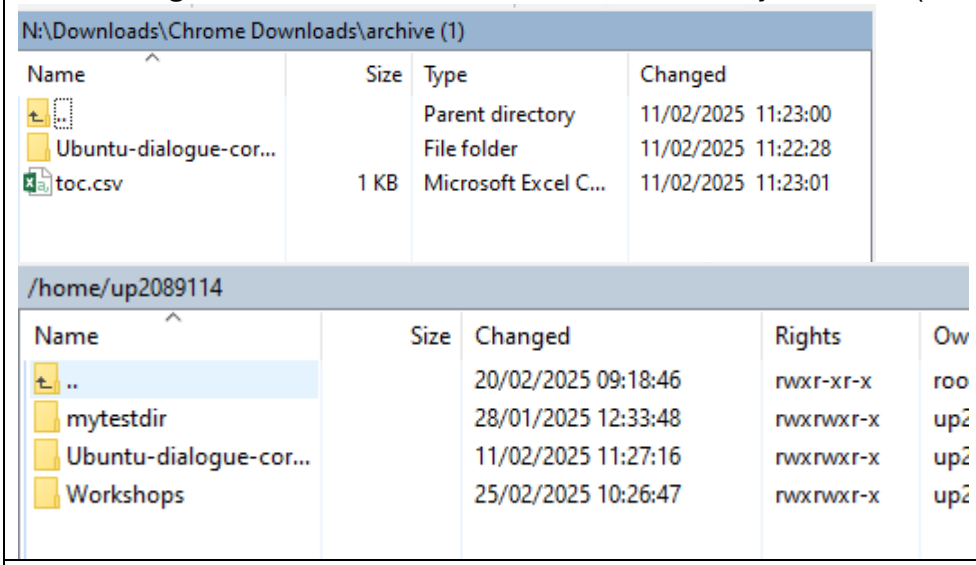
Week 1: Exploring the Hadoop Cluster

Objective: to be able to navigate the cluster, performing basic actions.

Part 1: Accessing cluster

Table 1.

Cluster navigation

Getting Data to the Cluster
Logging in:  Transferring files from local machine to home directory in cluster (HDFS) 
Listing contents of HDFS directory
Command: `hdfs dfs -ls`

Output:

```
[up2089114@mn01 ~]$ hdfs dfs -ls
Found 7 items
drwxr-xr-x   - up2089114 up2089114          0 2025-02-12 13:33 Ubuntu-dialogue-corpus
-rw-r--r--   3 up2089114 up2089114    3399671 2025-01-28 12:34 netflix_titles.csv
drwxr-xr-x   - up2089114 up2089114          0 2025-02-16 21:17 out
drwxr-xr-x   - up2089114 up2089114          0 2025-02-16 21:35 out2
drwxr-xr-x   - up2089114 up2089114          0 2025-02-16 21:55 out3
drwxr-xr-x   - up2089114 up2089114          0 2025-02-16 22:12 out4
drwxr-xr-x   - up2089114 up2089114          0 2025-02-11 09:52 tmp
```

Part 2: MapReduce Job Execution

Table 2.

MapReduce script and output

Command: `yarn jar $HADOOP_PREFIX/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.9.2.jar wordcount Ubuntu-dialogue-corpus out5`

```
[up2089114@mn01 ~]$ hdfs dfs -cat /user/up2089114/out5/part-r-00000 | head -20
@       2
@"      3
@$@@Y@@(?.f'@5J@#@@/8`7@I@@n3?a&?L:@^P@p@t?@)i@@k:@?u[Xh@Dd)?@X@2@?@)@@o@`@?Ab@-@?@?@@@`@@
d#wS@@]@        2
@ACTION 4
@X@@'JS/@@C~@@D9@@@@@@@yX@J7Ξ@hkU@+@@@@K@@|!5;@L@@@@"   3
@echo   1
        5
"OOT    2
"       1
"       12
,       12
/       2
11440   1
ActionParsni>   1
Bradley>2
DonkeyHote>     4
Dr_Willis>      1
Julien_>1
LordOfTim>      2
Sanky>  2
```

This job processed text from the Ubuntu-dialogue-corpus directory, counting the number of word occurrences for each word in the Ubuntu file, and stored the results in out5 on HDFS.

Initially, it was overwhelming to interact with Hadoop and HDFS given it is a new environment and unfamiliar. Nevertheless, uploading data and performing basic MapReduce job made the approach more understandable. This foundational experience bettered confidence in navigating the cluster and established the framework for further investigation.

Week 2: More on MapReduce

Objective: Run basic MapReduce jobs on small data and across the cluster

Part 1: Writing Map and Reduce Functions in Python

Table 1.

Index of files made

MRJob Number	Purpose	File name made for MR Job function
1	Count number of characters, lines and words in	mr_unix_wc.py
2	Word count – number of occurrences for each word in a poem (small scale)	"mr_word_count.py"
3	Word count on larger scale, on the cluster ('Ubuntu-dialogue corpus' imported into HDFS home directory (see Week 1) **(one reducer)**	Rewritten under "mr_word_count.py"
4	Word count on cluster **(5 reducers)**	"mr_word_coun1.py"
5	Word count on cluster– adapting initial word count model using a comma as the separator	"mr_word_count2.py"
6	Word count on cluster – using 'findall' function	"mr_word_count3.py"

Table 2.

MRJobs and output

MRJob 1

```
from mrjob.job import MRJob

class MRUnixWC(MRJob):

    def mapper(self, key, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRUnixWC.run()
```

```
$ python mr_unix_wc.py poem.txt
```

```
"chars" 1267
"lines" 42
"words" 242
Removing temp directory /tmp/mr_unix_wc.up2089114.20250211.094426.434318...
```

Characters = 1267
Lines = 42
Words = 242

MRJob 2:

```
python mr_word_count2.py -r hadoop poem.txt
```

```
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, key, value):
        # key: line number - not used here
        # value: contents of one line
        for word in value.split() :
            yield word, 1               # output a pair: word and count

    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordCount.run()
```

Section of output:

```
"moon," 3
"moon." 1
"my"    1
"next"  1
"nose," 3
"nose." 1
"note." 1
"of"    8
"on"    3
"one"   1
```

Note: remaining screenshots of outputs cannot be located as I cannot access mn01.soc server

MRJob 3:

```
$ python mr_word_count.py -r hadoop -o hdfs:///user/dbc/out
hdfs:///user/dbc/Ubuntu-dialogue-corpus
```

Takes a very long time to run

Produces a single file in out directory in HDFS home directory:

```
[up2089114@mn01 ~]$ hdfs dfs -cat /user/up2089114/out/part-00000| head -20
```

Output produced a lot of jargon and unrecognisable words

MRJob 4:

```
$ python mr_word_count1.py -r hadoop -o hdfs:///user/up2089114/out -D
mapred.reduce.tasks=5 hdfs:///user/up2089114/Ubuntu-dialogue-corpus
```

Much faster than one reducer

Produces 5 output files:

```
[up2089114@mn01 ~]$ hdfs dfs -ls out1
Found 6 items
-rw-r--r--   3 up2089114 up2089114          0 2025-02-11 12:11 out1/_SUCCESS
-rw-r--r--   3 up2089114 up2089114  157693443 2025-02-11 12:11 out1/part-00000
-rw-r--r--   3 up2089114 up2089114  157698393 2025-02-11 12:11 out1/part-00001
-rw-r--r--   3 up2089114 up2089114  157452853 2025-02-11 12:10 out1/part-00002
-rw-r--r--   3 up2089114 up2089114  157285455 2025-02-11 12:10 out1/part-00003
-rw-r--r--   3 up2089114 up2089114  157723824 2025-02-11 12:10 out1/part-00004
```

Output in each file still produced a lot of jargon and unrecognisable words

MRJob 5:

```
for word in value.split() :
```

This part in the original word count model, changed to:

```
for word in value.split(",") :
```

```
$ python mr_word_count2.py -r hadoop -o hdfs:///user/up2089114/out -D
mapred.reduce.tasks=5 hdfs:///user/up2089114/Ubuntu-dialogue-corpus
```

Similar speed as MrJob 4, also produces 5 out files, output still contained unrecognisable words – no improvement

MRJob 6:

```
from mrjob.job import MRJob
import re

wordRE = re.compile(r"[a-zA-Z]+")

class MRWordCount(MRJob):
    def mapper(self, key, value):
        # key: line number - not used here
        # value: contents of one line
        for word in wordRE.findall(value) :
            yield word, 1                # output a pair: word and count
    def reducer(self, key, values):
        yield key, sum(values)


if __name__ == '__main__':
    MRWordCount.run()
```

Produced 5 outputs too, all still contained unrecognisable words

This lab centered on writing MapReduce functions in Python using MRJob. We created multiple versions of word count models, adapting them to run on the Hadoop cluster. All word count programs produced unrecognisable words, even though we attempted to fix this using improvements such as more reducers, a comma separator (separates

each word or token with a comma to improve accuracy) and the ‘findall’ function (extracts only real words, ignoring punctuation, numbers or symbols – produce meaningful output). Despite these efforts, results remained ‘noisy,’ we also encountered an issue when reusing the ‘out’ directory, since it was already being used for the single reducer model, hence, job 4 was producing a single output file instead of 4. We resolved this by simply renaming the out file ‘out1.’ Through this, I developed practical debugging skills in HDFS. We also had to create new files for each job using the nano editor, the instructions did not specify this for job 2-5, giving a sense of achievement that a new basic skill has been acquired.

Week 3: Going Further with MapReduce

Objective: to be able to use MapReduce to calculate Page Ranking and attempt at improving algorithms to produce more meaningful output.

Part 1: Single iteration and continuous iterable PageRank algorithms

Table 1.

Single iteration and output

File Name: mr_page_rank1.py (single iteration)

```
from mrjob.job import MRJob

N = 11

class MRPageRank(MRJob):

    def mapper(self, key, line):
        # key: line number – not used here
        # line: contents of one line
        fields = line.split()
        pageRank = float(fields [1])  # current PageRank of this page
        outLinks = fields [2:]        # outgoing links of this page

        prContrib = pageRank / len(outLinks)
        for mId in outLinks :
            yield mId, prContrib

    def reducer(self, mId, values):
        s = sum(values)
        mPageRank = 0.85 * s + 0.15 / N
        yield mId, mPageRank

if __name__ == '__main__':
    MRPageRank.run()
```

File Name: web-graph.txt

Inserted Text:

```
A  0.090909  A  B  C  D  E  F  G  H  I  J  K
B  0.090909  C
C  0.090909  B
D  0.090909  A  B
E  0.090909  B  D  F
F  0.090909  B  E
G  0.090909  B  E
H  0.090909  B  E
I  0.090909  B  E
J  0.090909  E
K  0.090909  E
```

Script: `python mr_page_rank1.py web-graph.txt`

Output:

```
[up2089114@mn01 ~]$ nano mr_page_rank1.py
[up2089114@mn01 ~]$ nano web-graph.txt
[up2089114@mn01 ~]$ python mr_page_rank1.py web-graph.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Running step 1 of 1...
Creating temp directory /tmp/mr_page_rank1.up2089114.20250218.121246.327218
job output is in /tmp/mr_page_rank1.up2089114.20250218.121246.327218/output
Streaming final output from /tmp/mr_page_rank1.up2089114.20250218.121246.327218/output...
"A"     0.059297475
"B"     0.316872975
"C"     0.0979338
"D"     0.04641869999999999
"E"     0.32975175
"F"     0.04641869999999999
"G"     0.02066115
"H"     0.02066115
"I"     0.02066115
"J"     0.02066115
"K"     0.02066115
Removing temp directory /tmp/mr_page_rank1.up2089114.20250218.121246.327218...
```

Page E is the most important page with highest rank, Page G-K the least.

Table 2.

Iterable job + output

File name: mr_page_rank2.py (iterable version)

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.protocol import RawValueProtocol

N = 11  # Nodes in our graph
NITER = 50

class MRPageRank(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol
        # Default for input; use here for output so we can "input our output"

    def mapper(self, key, value):
        # key: line number – not used here
        # value: contents of one line
        fields = value.split()
        myId = fields [0]             # ID of this page
        pageRank = float(fields [1])  # current PageRank of this page
        outLinks = fields [2:]        # outgoing links of this page

        yield myId, outLinks

        prContrib = pageRank / len(outLinks)
        for mId in outLinks :
            yield mId, prContrib

    def reducer(self, mId, values):
        s = 0
        for val in values :
            if(isinstance(val, list)) :
                mOutLinks = val  # From *first* yield in mapper above
            else :
                s += float(val)

        mPageRank = s * 0.85 + 0.15 / N
        yield None, mId + " " + str(mPageRank) + " " + " ".join(mOutLinks)
            # NB: key value irrelevant with RawValueProtocol output

    def steps(self) :
        # Repeat the same map-reduce phase NITER times
        step = MRStep(mapper=self.mapper, reducer=self.reducer)
        return [step] * NITER

if __name__ == '__main__':
    MRPageRank.run()
```

Script: `$ python mr_page_rank2.py web-graph.txt`

```
Running step 42 of 50...
Running step 43 of 50...
Running step 44 of 50...
Running step 45 of 50...
Running step 46 of 50...
Running step 47 of 50...
Running step 48 of 50...
Running step 49 of 50...
Running step 50 of 50...
job output is in /tmp/mr_page_rank2.up2089114.20250223.154150.138731/output
Streaming final output from /tmp/mr_page_rank2.up2089114.20250223.154150.138731/output...
A 0.03278149315934767 A B C D E F G H I J K
B 0.3843697808389126 C
C 0.3429414531872355 B
D 0.03908709209997012 A B
E 0.08088569323450431 B D F
F 0.03908709209997012 B E
G 0.01616947901685893 B E
H 0.01616947901685893 B E
I 0.01616947901685893 B E
J 0.01616947901685893 E
K 0.01616947901685893 E
:
```

Explanation: shows page outlinks, values gradually improved.

Changing NITER to 1:

File name: mr_page_rank3.py

Output:

```
Running step 1 of 1...
Creating temp directory /tmp/mr_page_rank3.up2089114.20250223.154814.251358
job output is in /tmp/mr_page_rank3.up2089114.20250223.154814.251358/output
Streaming final output from /tmp/mr_page_rank3.up2089114.20250223.154814.251358/output...
A 0.059297475 A B C D E F G H I J K
B 0.316872975 C
C 0.0979338 B
D 0.04641869999999999 A B
E 0.32975175 B D F
F 0.04641869999999999 B E
G 0.02066115 B E
H 0.02066115 B E
I 0.02066115 B E
J 0.02066115 E
K 0.02066115 E
```

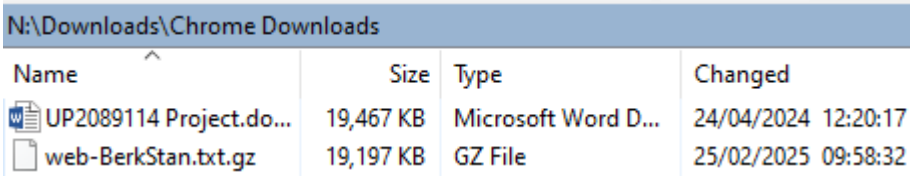
Same results as single iteration script but now matches format of ‘web-graph.txt’ file.

Part 2. Applying to large web graph

Table 3.

MRJob on larger dataset + improvements

Uploading dataset to HDFS (web-Berkstan.txt.gz)



/home/up2089114

Name	Size	Changed	Rights	Owner
mr_page_rank1.py	1 KB	18/02/2025 11:31:48	rw-rw-r--	up2089...
mr_page_rank2.py	2 KB	23/02/2025 15:39:52	rw-rw-r--	up2089...
mr_page_rank3.py	2 KB	23/02/2025 15:47:15	rw-rw-r--	up2089...
mr_unix_wc.py	1 KB	04/02/2025 11:19:09	rw-rw-r--	up2089...
mr_word_count.py	1 KB	16/02/2025 20:46:01	rw-rw-r--	up2089...
mr_word_count2.py	1 KB	16/02/2025 21:59:28	rw-rw-r--	up2089...
mr_word_count3.py	1 KB	16/02/2025 22:02:18	rw-rw-r--	up2089...
netflix_titles.csv	3,320 KB	28/01/2025 12:31:36	rw-rw-r--	up2089...
new-file-name	0 KB	25/02/2025 10:17:16	rw-rw-r--	up2089...
output.txt	1 KB	25/02/2025 10:18:06	rw-rw-r--	up2089...
poem.txt	2 KB	04/02/2025 11:28:47	rw-rw-r--	up2089...
poem.txt.save	2 KB	12/02/2025 13:29:41	rw-rw-r--	up2089...
poem.txt.save.1	2 KB	12/02/2025 13:31:36	rw-rw-r--	up2089...
python	1 KB	18/02/2025 11:31:40	rw-rw-r--	up2089...
web-BerkStan.txt.gz	19,197 KB	25/02/2025 09:58:32	rw-r--r--	up2089...

Creating graph file (web-Wikipedia.txt)

```
  GNU nano 2.3.1               File: web-Wikipedia.txt

# A little Web graph
2  3
3  2
4  1
4  2
5  2
5  4
5  6
6  2
6  5
7  2
7  5
8  2
8  5
9  2
9  5
10  5
11  5
```

Creating MapReduce Preprocessing Program

File name: mr_preprocess.py

Script:

```
  GNU nano 2.3.1                File: mr_preprocess.py

from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol

N = 11
pageRank = 1 / N

class MRPreprocess(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol
        # Default for input; use here for output so we can "input our output"

    def mapper(self, key, value):
        # key: line number - not used here
        # value: contents of one line

        # Ignore blank lines and comments
        if len(value) == 0 or value [0] == '#':
            return

        fields = value.split()

        fromNode = fields [0]
        toNode   = fields [1]

        yield fromNode, toNode

    def reducer(self, mId, values):
        mOutLinks = values
        yield None, mId + " " + str(pageRank) + " " + " ".join(mOutLinks)

if __name__ == '__main__':
    MRPreprocess.run()
```

The reducer creates outgoing links for each node, outputting the node's ID and its initial PageRank.

Running the preprocess program

`python mr_preprocess.py web-Wikipedia.txt`

Output:

```
[up2089114@mn01 ~]$ python mr_preprocess.py web-Wikipedia.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Running step 1 of 1...
Creating temp directory /tmp/mr_preprocess.up2089114.20250225.124256.628021
job output is in /tmp/mr_preprocess.up2089114.20250225.124256.628021/output
Streaming final output from /tmp/mr_preprocess.up2089114.20250225.124256.628021/output...
10 0.09090909090909091 5
11 0.09090909090909091 5
2 0.09090909090909091 3
3 0.09090909090909091 2
4 0.09090909090909091 1 2
5 0.09090909090909091 2 4 6
6 0.09090909090909091 2 5
7 0.09090909090909091 2 5
8 0.09090909090909091 2 5
9 0.09090909090909091 2 5
```

Node 1, previously known as node A is missing, since it has no outgoing link, it did not appear in the output.

Modifying preprocess program to show Node 1.

Creating new file: `mr_preprocess.py2`

Script:

```
  GNU nano 2.3.1                File: mr_preprocess.py2

from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol

N = 11
pageRank = 1 / N

class MRPreprocess(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol
        # Default for input; use here for output so we can "input our output"

    def mapper(self, key, value):
        # key: line number - not used here
        # value: contents of one line

        # Ignore blank lines and comments
        if len(value) == 0 or value [0] == '#':
            return

        fields = value.split()
        fromNode = fields [0]
        toNode   = fields [1]

        yield fromNode, toNode
        yield toNode, None

    def reducer(self, mId, values):
        mOutLinks = [] ;
        for val in values :
            if not (val is None) :
                mOutLinks.append(val)
        yield None, mId + " " + str(pageRank) + " " + " ".join(mOutLinks)

if __name__ == '__main__':
    MRPreprocess.run()
```

Explanation: To account for node 1, the output will include those with and without outgoing links.

Command: `python mr_preprocess.py2 web-Wikipedia.txt`

Output:

```
1 0.09090909090909091
10 0.09090909090909091 5
11 0.09090909090909091 5
2 0.09090909090909091 3
3 0.09090909090909091 2
4 0.09090909090909091 1 2
5 0.09090909090909091 2 4 6
6 0.09090909090909091 2 5
7 0.09090909090909091 2 5
8 0.09090909090909091 2 5
9 0.09090909090909091 2 5
```

Now includes node 1

Combining preprocessing MapReduce job with previous Iterative PageRank script

File name: `mr_page_rank_big.py`

Script:

```
from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.protocol import RawValueProtocol

NITER = 50
N = 11
initPageRank = 1 / N

class MRPageRank(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol
        # Default for input; use here for output so we can "input our output"

    def preMapper(self, key, value):

        if len(value) == 0 or value [0] == '#':
            return

        fields = value.split()
        fromNode = fields [0]
        toNode   = fields [1]

        yield toNode, None
        yield fromNode, toNode

    def preReducer(self, mId, values):

        outLinks = []
        for val in values :
            if not (val is None) :
                outLinks.append(val)

        yield None, mId + " " + str(initPageRank) + " " + " ".join(outLinks)


    def mapper(self, key, value):

        fields = value.split()
        myId     = fields [0]         # ID of this page
        pageRank = float(fields [1])  # current pagerank of this page
        outLinks = fields [2:]        # outgoing links of this page

        yield myId, outLinks

        nOutLinks = len(outLinks)
        if(nOutLinks > 0) :
            prContrib = pageRank / nOutLinks
            for mId in outLinks :
                yield mId, prContrib


    def reducer(self, mId, values):
        s = 0
        for val in values :
            if(isinstance(val, list)) :
                mOutLinks = val  # From "outLinks" yield in mapper above
            else :
                s += float(val)

        mPageRank = s * 0.85 + 0.15 / N
        yield None, mId + " " + str(mPageRank) + " " + " ".join(mOutLinks)
            # NB: key value irrelevant with RawValueProtocol output

    def steps(self) :
        # preprocessing, then N times "main" MR
        pre   = MRStep(mapper=self.preMapper, reducer=self.preReducer)
        step  = MRStep(mapper=self.mapper, reducer=self.reducer)
        return [pre] + [step] * NITER

if __name__ == '__main__':
    MRPageRank.run()
```

Output:

```
1 0.027645934714267257
10 0.013636363636363636 5
11 0.013636363636363636 5
2 0.3241496975987292 3
3 0.2892207429507734 2
4 0.03296369665389087 1 2
5 0.06821411653244908 2 4 6
6 0.03296369665389087 2 5
7 0.013636363636363636 2 5
8 0.013636363636363636 2 5
9 0.013636363636363636 2 5
```

Explanation: Results are significantly different to expected values, due to insufficient treatment of 'dangling nodes.'

Treating dangling nodes

File name: mr_page_rank_big2.py

Script:

```
  GNU nano 2.3.1              File: mr_page_rank_big2.py                      Modified

from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.protocol import RawValueProtocol

NITER = 50
N = 11
initPageRank = 1 / N

class MRPageRank(MRJob):

    OUTPUT_PROTOCOL = RawValueProtocol
        # Default for input; use here for output so we can "input our output"

    def preMapper(self, key, value):

        if len(value) == 0 or value [0] == '#':
            return

        fields = value.split()
        fromNode = fields [0]
        toNode   = fields [1]

        yield toNode, None
        yield fromNode, toNode

    def preReducer(self, mId, values):

        outLinks = []
        for val in values :
            if not (val is None) :
                outLinks.append(val)

        yield None, mId + " " + str(initPageRank) + " " + " ".join(outLinks)


    def intermapper(self, key, line):

        fields = line.split()
        myId  = fields [0]      # ID of this page

        yield myId, line        # forward directly to interreducer for output

        if(len(fields) == 2) :  # dangling node
            pageRank = float(fields [1])   # current pagerank of this page
            yield "accDangling", pageRank

    def interreducer(self, mId, values):

        if mId == "accDangling" :
            s = sum(values)
            yield None, "accDangling " + str(s)
        else :
            # Should only receive one value, from first yield in intermapper
            line = next(values)
            yield None, line


    def mapper(self, key, value):

        fields = value.split()
        myId     = fields [0]         # ID of this page
        pageRank = float(fields [1])  # current pagerank of this page

        if myId == "accDangling" :
            p = pageRank / N
            for i in range(N) :
                yield str(i + 1), p   # to be shared amongst *all N* nodes
```

```
        else :
            outLinks = fields [2:]    # outgoing links of this page

            yield myId, outLinks

            nOutLinks = len(outLinks)
            if(nOutLinks > 0) :
                prContrib = pageRank / nOutLinks
                for mId in outLinks :
                    yield mId, prContrib

    def reducer(self, mId, values):
        s = 0
        for val in values :
            if(isinstance(val, list)) :
                mOutLinks = val  # From "outLinks" yield in mapper above
            else :
                s += float(val)

        mPageRank = s * 0.85 + 0.15 / N
        yield None, mId + " " + str(mPageRank) + " " + " ".join(mOutLinks)


    def steps(self) :
        # preprocessing, then N times "inter" MR followed by "main" MR
        pre   = MRStep(mapper=self.preMapper, reducer=self.preReducer)
        inter = MRStep(mapper=self.intermapper, reducer=self.interreducer)
        step  = MRStep(mapper=self.mapper, reducer=self.reducer)
        return [pre] + [inter, step] * NITER

if __name__ == '__main__':
    MRPageRank.run()
```

Output:

```
1 0.03278149315934767
10 0.01616947901685893 5
11 0.01616947901685893 5
2 0.3843697809528769 3
3 0.3429414533690357 2
4 0.03908709209997011 1 2
5 0.08088569323450433 2 4 6
6 0.03908709209997011 2 5
7 0.01616947901685893 2 5
8 0.01616947901685893 2 5
9 0.01616947901685893 2 5
```

Produces same values for PageRank obtained when we applied iterable version to ‘web-graph.txt’

This week’s lab helped me understand how the PageRank algorithm works using MRJob and Hadoop. We began by running a simple version that calculated rank in one iteration, which was very quick but not accurate. Followed this with an iterable version, providing more accurate results by updating the ranks step by step. I also learned how to change a large web graph file into the format needed for PageRank to run properly. The code was experimented with on a larger dataset and we learned how changing settings like number of reducers can affect how long the job takes. Overall, I learned how to build and improve MapReduce jobs and process real-world data. Key skill learned – writing scalable code that works well with big data in Hadoop.

Week 4: Getting Started with Apache Spark

Objective: to be able to use Spark Basics and SQL queries, as well as the Spark Machine Learning Library to perform a classification task using multiple classifiers.

Part 1: Spark Basics and SQL Queries on Flight Data

Viewing and counting data

```
flightData2015.show()
flightData2015.count()
```

```
+-----------------+-----------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-----------------+-----+
|    United States|          Romania|   15|
|    United States|          Croatia|    1|
|    United States|          Ireland|  344|
|            Egypt|    United States|   15|
|    United States|            India|   62|
|    United States|        Singapore|    1|
|    United States|          Grenada|   62|
|       Costa Rica|    United States|  588|
|          Senegal|    United States|   40|
|          Moldova|    United States|    1|
|    United States|     Sint Maarten|  325|
|    United States| Marshall Islands|   39|
|           Guyana|    United States|   64|
|            Malta|    United States|    1|
|         Anguilla|    United States|   41|
|          Bolivia|    United States|   30|
|    United States|         Paraguay|    6|
|          Algeria|    United States|    4|
|Turks and Caicos ...|    United States|  230|
|    United States|        Gibraltar|    1|
+-----------------+-----------------+-----+
only showing top 20 rows
```

Summary of total incoming flights per country for 2015

```
flightData2015.select("DEST_COUNTRY_NAME", "count")
.groupBy("DEST_COUNTRY_NAME").sum("count").show(5)
```

```
+-----------------+----------+
|DEST_COUNTRY_NAME|sum(count)|
+-----------------+----------+
|         Anguilla|        41|
|           Russia|       176|
|         Paraguay|        60|
|          Senegal|        40|
|           Sweden|       118|
+-----------------+----------+
only showing top 5 rows
```

`"groupBy"` groups together rows based on the value of destination name

This query calculates total number of routes into each country

```
>>> spark.sql("""
...       SELECT DEST_COUNTRY_NAME, sum(count)
...       FROM flight_data_2015
...       GROUP BY DEST_COUNTRY_NAME
...       """).show(10)
+--------------------+----------+
|   DEST_COUNTRY_NAME|sum(count)|
+--------------------+----------+
|             Moldova|         1|
|             Bolivia|        30|
|             Algeria|         4|
|Turks and Caicos ...|       230|
|            Pakistan|        12|
|    Marshall Islands|        42|
|            Suriname|         1|
|              Panama|       510|
|         New Zealand|       111|
|             Liberia|         2|
+--------------------+----------+
only showing top 10 rows
```

Retrieving the top 5 destinations:

```
>>> high5Sql = spark.sql("""
... SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
... FROM flight_data_2015
... GROUP BY DEST_COUNTRY_NAME
... ORDER BY sum(count) DESC
... LIMIT 5
... """)
>>> high5Sql.show()
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

Part 2: Word Count using Spark API

```
>>> from pyspark.sql.functions import split, explode
>>>
>>> wordCounts = textFile.select(explode(split(textFile.value, "\\s+")).
alias("word")).groupBy("word").count()
>>> wordCounts.show()
+--------------------+-----+
|                word|count|
+--------------------+-----+
|              online|    1|
|              graphs|    1|
|          ["Parallel|    1|
|          ["Building|    1|
|              thread|    1|
|       documentation|    3|
|            command,|    2|
|         abbreviated|    1|
|            overview|    1|
|                rich|    1|
|                 set|    2|
|         -DskipTests|    1|
|                name|    1|
|page](http://spar...|    1|
|        ["Specifying|    1|
|              stream|    1|
|                run:|    1|
|                 not|    1|
|            programs|    2|
|               tests|    2|
+--------------------+-----+
only showing top 20 rows
```

Splits words into its own row and counts using 'explode'/'split' , and calculates its number of occurrences with 'groupby.'

Part 3: Spark Machine Learning Library

Building regression models to predict house prices in Melbourne from numerous features, see Table 1.

Data Preparation:

```
from pyspark.ml.feature import VectorAssembler

### used the vector assembler to transform our dataset ####
vectorAssembler = VectorAssembler().setInputCols(["Rooms", "Bathroom",
                                                  "Landsize", "Lattitude",
                                                  "Longtitude"]).setOutputCol("features")
```

Vector Assembler combines multiple single feature columns into one vector column for machine learning tasks.

Table 1.

Model Building

Model Used	Purpose
Random Forest 1 (maxdepth=5)	Baseline model
Random Forest 2 (maxdepth=10)	Comparison to baseline to see if increasing max depth increases model accuracy
Random Forest 3 (maxdepth=20)	
Decision Tree (maxdepth=20)	Effect of other regression models on prediction accuracy, relative to Random Forest
Linear Regression (maxIter=100)	

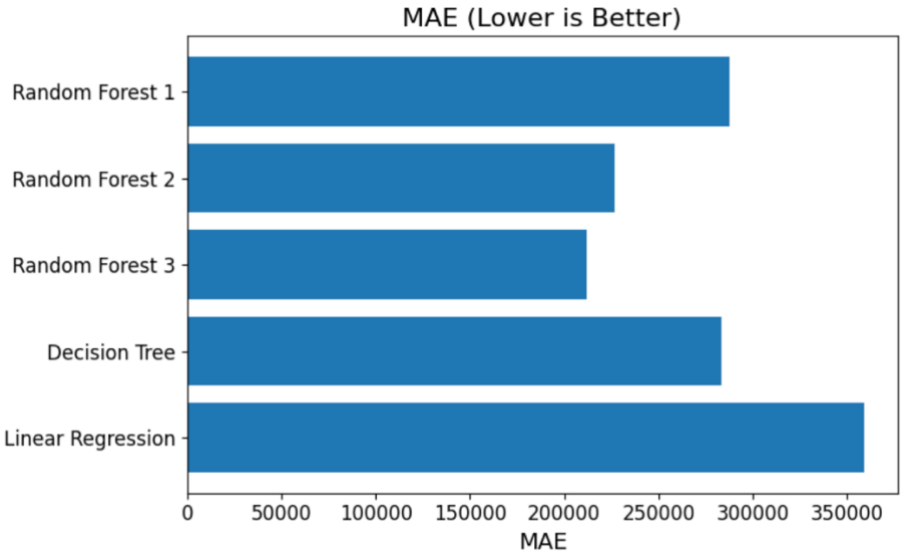
```
from pyspark.ml.regression import RandomForestRegressor, DecisionTreeRegressor, LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

models = {
    "Random Forest (maxDepth=10)": RandomForestRegressor(labelCol="Price", featuresCol="features", maxDepth=10),
    "Random Forest (maxDepth=20)": RandomForestRegressor(labelCol="Price", featuresCol="features", maxDepth=20),
    "Decision Tree (maxDepth=20)": DecisionTreeRegressor(labelCol="Price", featuresCol="features", maxDepth=20),
    "Linear Regression": LinearRegression(labelCol="Price", featuresCol="features", maxIter=100)}

# Metrics to evaluate
metrics = ["mae", "rmse", "r2"]

for name, model in models.items():
    print(f"\n Results for: {name}")
    fitted = model.fit(trainingData)
    predictions = fitted.transform(testData)

    for metric in metrics:
        evaluator = RegressionEvaluator(labelCol="Price", predictionCol="prediction", metricName=metric)
        score = evaluator.evaluate(predictions)
        print(f"{metric.upper()}: {score:.4f}")
```

Results were evaluated on its Mean Absolute Error (MAE), Root Mean Squared Error (RMSE) and Adjusted R^2 Value (R^2), see table below.

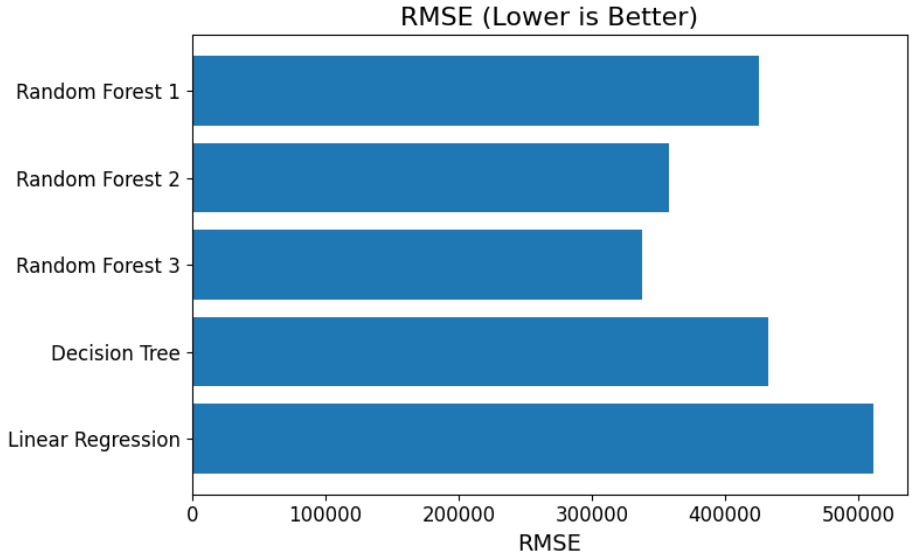
Table 2.

Metric Evaluation of Regression Models.

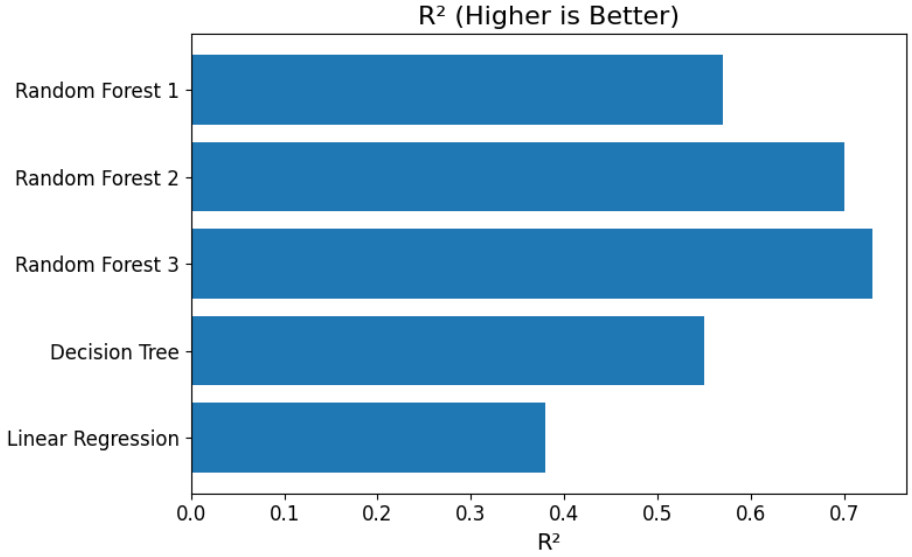
	MAE	RMSE	R^2
Random Forest 1	287637.71	425463.55	0.57
Random Forest 2	226422.30	357810.57	0.70
Random Forest 3	211673.60	338204.24	0.73
Decision Tree	283193.44	432862.54	0.55
Linear Regression	358847.93	511232.89	0.38



Random Forest with max depth of 20 has the lowest average prediction error, its predictions were closer to actual house price values than Linear Regression which deviated the most from true values.



Random Forest 3 again showed the lowest error, indicating its predictions were closer to actual house price predictions on average. Once again, Linear Regression performed the worst.



Random Forest can explain for most of the variation in the data, while Linear Regression can explain the least. Decision Tree and Random Forest 1 performed relatively weak, while Random Forest 2 performs well too.

This week's lab involved Spark streaming basics, generating SQL Queries and building machine learning models to predict house prices. I considered the machine learning section to be of most importance, given part 1 and 2 of this lab book entry was not entirely new information.

I learned how different tree depths impact model complexity, where increasing it led to reduced error and better explanation for variance.

A key skill developed was critically interpreting evaluation metrics across multiple models to choose the best performer.

Week 5: More Spark and Recommender Systems

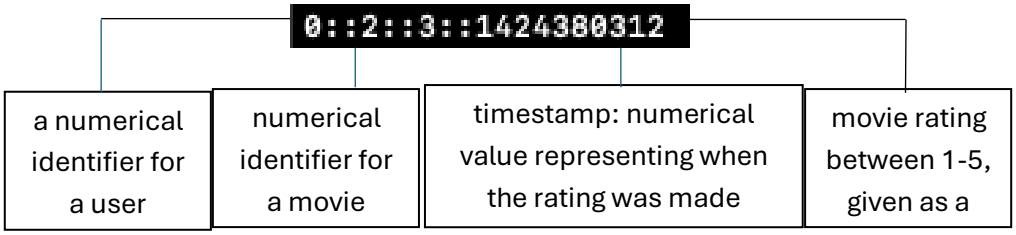
Objective: To explore running Alternating Least Squares (ALS) and Recommender Systems on a Hadoop cluster, using increasing data set sizes and analysing/comparing performance across all dataset sizes.

Part 1: Data Preparation

Using Alternating Least Squares (ALS) in Spark

Dataset used: Sample of MovieLens Rating

```
ls: cannot access home: No such file or directory
[up2089114@mn01 ~]$ head /home/spark/spark-2.4.0-bin-hadoop2.7/data/m
llib/als/sample_movielens_ratings.txt
0::2::3::1424380312
0::3::1::1424380312
0::5::2::1424380312
0::9::4::1424380312
0::11::1::1424380312
0::12::2::1424380312
0::15::1::1424380312
0::17::1::1424380312
0::19::1::1424380312
0::21::1::1424380312
```



Raw Data

```
>>> lines = spark.read.text("file:///home/spark/spark-2.4.0-bin-hadoop2
.7/data/mllib/als/sample_movielens_ratings.txt")
>>> lines = spark.read.text("file:///home/spark/spark-2.4.0-bin-hadoop2
.7/data/mllib/als/sample_movielens_ratings.txt").show()
+--------------------+
|               value|
+--------------------+
| 0::2::3::1424380312|
| 0::3::1::1424380312|
| 0::5::2::1424380312|
| 0::9::4::1424380312|
|0::11::1::1424380312|
|0::12::2::1424380312|
|0::15::1::1424380312|
|0::17::1::1424380312|
|0::19::1::1424380312|
|0::21::1::1424380312|
|0::23::1::1424380312|
|0::26::3::1424380312|
|0::27::1::1424380312|
|0::28::1::1424380312|
|0::29::1::1424380312|
|0::30::1::1424380312|
|0::31::1::1424380312|
|0::34::1::1424380312|
|0::37::1::1424380312|
|0::41::2::1424380312|
+--------------------+
only showing top 20 rows
```

Splitting values into 3 columns:

```
>>> ratings = parts.select(parts.value.getItem(0).cast("int").alias("userId"),
...                        parts.value.getItem(1).cast("int").alias("movieId"),
...                        parts.value.getItem(2).cast("float").alias("rating"))
>>> ratings.show()
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     0|      2|   3.0|
|     0|      3|   1.0|
|     0|      5|   2.0|
|     0|      9|   4.0|
|     0|     11|   1.0|
|     0|     12|   2.0|
|     0|     15|   1.0|
|     0|     17|   1.0|
|     0|     19|   1.0|
|     0|     21|   1.0|
|     0|     23|   1.0|
|     0|     26|   3.0|
|     0|     27|   1.0|
|     0|     28|   1.0|
|     0|     29|   1.0|
|     0|     30|   1.0|
|     0|     31|   1.0|
|     0|     34|   1.0|
|     0|     37|   1.0|
|     0|     41|   2.0|
+------+-------+------+
only showing top 20 rows
```

Data Splitting for Training and Testing

```
>>> (training, test) = ratings.randomSplit([0.8, 0.2])
>>> ratings.count()
1501
>>> training.count()
1211
>>> test.count()
290
```

Movie ratings (n=1501), 80% for training set (n = 1211), 20% for testing set (n = 298)

Part 2: Model training using ALS

```
>>> from pyspark.ml.recommendation import ALS
>>> als = ALS(maxIter=10, regParam=0.05, coldStartStrategy="drop",
...                 userCol="userId", itemCol="movieId", ratingCol="rating")
model = als.fit(training)
```

Part 3: Model Predictions and Evaluation

```
>>> predictions = model.transform(test)
>>> predictions.show()
+------+-------+------+----------+
|userId|movieId|rating|prediction|
+------+-------+------+----------+
|    12|     31|   4.0|0.97753894|
|     5|     31|   1.0| 0.3213985|
|    24|     31|   1.0|  0.938265|
|    14|     31|   3.0|  2.030851|
|    18|     31|   1.0| 0.5258906|
|    16|     85|   5.0| 2.0570328|
|    20|     85|   2.0| 2.2807631|
|     7|     85|   4.0| 2.4466128|
|     2|     85|   1.0|0.78372115|
|    22|     65|   1.0| 1.5043877|
|    19|     65|   1.0| 1.9095733|
|    15|     65|   2.0|  1.893177|
|    19|     53|   2.0|0.20482689|
|    20|     78|   1.0|0.89674884|
|    24|     78|   1.0| 1.0073581|
|    11|     78|   1.0|  1.078367|
|    19|     81|   1.0| 1.2363504|
|    27|     28|   1.0| 2.5775597|
|     5|     28|   1.0| 1.9174035|
|    17|     28|   1.0| 1.5543244|
+------+-------+------+----------+
only showing top 20 rows
```

Produces the original data frame, with the 'prediction' column added.

Root Mean Squared Error evaluation

```
>>> from pyspark.ml.evaluation import RegressionEvaluator
>>> evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating"
, predictionCol="prediction")
>>> rmse = evaluator.evaluate(predictions)

>>> print("Root-mean-squared error = " + str(rmse))
Root-mean-squared error = 1.035794793719454
```

Model's predictions deviate from correct predictions by 1.04 units.

Part 4. Generating Recommendations

Extracting three unique users for recommendations:

```
>>> users = ratings.select("userID").distinct().limit(3)
>>> users.show(5)
+------+
|userID|
+------+
|    28|
|    26|
|    27|
+------+
```

Generating top 10 movie recommendations for these users:

```
>>> userSubsetRecs = model.recommendForUserSubset(users, 10)
>>> userSubsetRecs.show()
[Stage 251:==============================>                    (60 + 24

+------+--------------------+
|userId|     recommendations|
+------+--------------------+
|    28|[[12, 4.4512305],...|
|    26|[[22, 4.9007645],...|
|    27|[[32, 3.7997298],...|
+------+--------------------+
```

Displaying the recommendations:

```
>>> userSubsetRecs.show(truncate=False)
+------+-------------------------------------------------------------------
---------------------------------------------------------------------------
------------------------------+
|userId|recommendations

                              |
+------+-------------------------------------------------------------------
---------------------------------------------------------------------------
------------------------------+
|28    |[[12, 4.4512305], [81, 4.4137244], [92, 4.385125], [2, 4.076856]
, [89, 3.8213549], [49, 3.6394162], [57, 2.872593], [24, 2.7482543], [19
, 2.7315733], [0, 2.6785471]] |
|26    |[[22, 4.9007645], [88, 4.779039], [30, 4.7274055], [24, 4.549536
], [23, 4.5204782], [7, 4.4612646], [51, 4.3172812], [74, 4.2905416], [7
5, 4.222791], [32, 3.6319034]]|
|27    |[[32, 3.7997298], [30, 3.7180722], [18, 3.6860855], [23, 3.48264
7], [34, 3.2653432], [8, 3.25289], [48, 3.218635], [79, 3.1129777], [27,
 3.006318], [75, 2.9118595]]  |
+------+-------------------------------------------------------------------
---------------------------------------------------------------------------
------------------------------+
```

(userID) (movieID) (rating)

```
|28    |[[12, 4.4512305]
```

Part 5. Running ALS Job Across the cluster with increasing size of datasets

ALS on sample of 10,000 reviews

Preparing the script (combining part 1-3), see comments in code for explanation

```
  GNU nano 2.3.1                        File: als_example.py

from pyspark.sql import SparkSession

from pyspark.sql.functions import split

from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS

import time


spark = SparkSession.builder.appName("ALSExample").getOrCreate()

startTime = time.time()

lines = spark.read.text("spark-examples/sample_movielens_ratings.txt")

parts = lines.withColumn("value", split("value", "::"))
ratings = parts.select(parts.value.getItem(0).cast("int").alias("userId"),
                       parts.value.getItem(1).cast("int").alias("movieId"),
                       parts.value.getItem(2).cast("float").alias("rating"))

(training, test) = ratings.randomSplit([0.8, 0.2])

# Build the recommendation model using ALS on the training data
# Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics
als = ALS(maxIter=10, regParam=0.05, coldStartStrategy="drop",
          userCol="userId", itemCol="movieId", ratingCol="rating")
model = als.fit(training)

# Evaluate the model by computing the RMSE on the test data
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-squared error = " + str(rmse))

# Generate top 10 movie recommendations for a specified set of users
users = ratings.select(als.getUserCol()).distinct().limit(3)
userSubsetRecs = model.recommendForUserSubset(users, 10)

userSubsetRecs.show(truncate=False)

endTime = time.time()

print("Elapsed time = " + str(endTime - startTime))

spark.stop()
```

Transferring Data to HDFS and running the Spark Job

```
[up2089114@mn01 ~]$ hdfs dfs -mkdir spark-examples
[up2089114@mn01 ~]$ hdfs dfs -put /home/spark/spark-2.4.0-bin-hadoop2.7/data/mllib/als/sample_
movielens_ratings.txt spark-examples
[up2089114@mn01 ~]$ hdfs dfs -ls spark-examples
Found 1 items
-rw-r--r--   3 up2089114 up2089114      32363 2025-03-15 12:48 spark-examples/sample_movielens
_ratings.txt
[up2089114@mn01 ~]$ spark-submit --master yarn als_example.py
```

Spark job submitted to hadoop cluster using Yarn.

Results:

```
Root-mean-squared error = 1.1726371193003717
+------+--------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
|userId|recommendations
                                                                                |
+------+--------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
|28    |[[12, 4.2894006], [49, 4.0032625], [89, 3.6920469], [40, 3.5058086], [82, 3.326188], [
62, 3.2882934], [68, 3.0087614], [19, 2.891514], [23, 2.8361933], [0, 2.7308445]]|
|26    |[[88, 4.9480186], [94, 4.918556], [22, 4.8452935], [24, 4.7961273], [23, 4.7099686], [
7, 4.4340687], [32, 4.4098587], [51, 4.085772], [90, 4.0006094], [92, 3.941093]] |
|27    |[[23, 3.6800866], [27, 3.597342], [18, 3.463696], [32, 3.4254656], [66, 3.0015788], [8
0, 2.8413393], [48, 2.827481], [44, 2.726789], [49, 2.6823905], [50, 2.6762161]] |
+------+--------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+

Elapsed time = 21.098276138305664
```

ALS on 100,000 data entries

Copying and Modifying the exisiting ALS script (see comments in screenshot)

```
[up2089114@mn01 ~]$ cp als_example.py als_ml-100k.py
[up2089114@mn01 ~]$ nano als-ml-100k.py
```

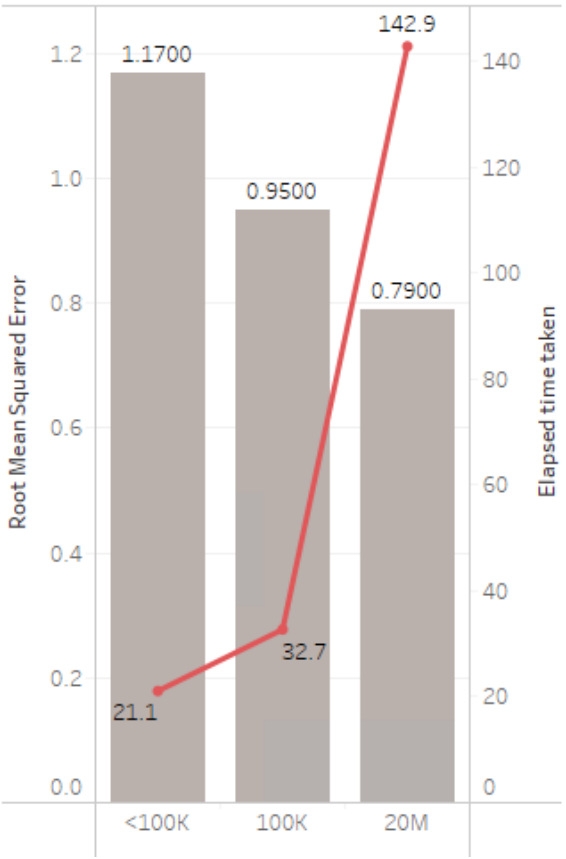
```
##### Changing display name to ALS_MovieLens_100k #######
spark = SparkSession.builder.appName("ALS_MovieLens_100K").getOrCreate()

startTime = time.time()

##### Changing file path to u.data #####
lines = spark.read.text("spark-examples/u.data")

##### Implementing a tab separator ######
parts = lines.withColumn("value", split("value", "\t"))
ratings = parts.select(parts.value.getItem(0).cast("int").alias("userId"),
                       parts.value.getItem(1).cast("int").alias("movieId"),
                       parts.value.getItem(2).cast("float").alias("rating"))
```

Running ALS on MovieLens dataset with 20 million data entries

Copying and Modifying existing ALS script

```
##### Changing display name to ALS_MovieLens_20M #######
spark = SparkSession.builder.appName("ALS_MovieLens_20M").getOrCreate()

startTime = time.time()

##### Changing file path to u.data #####
lines = spark.read.text("spark-examples/u.data")

##### Removing this data loading code ######
##### parts = lines.withColumn("value", split("value", "\t"))
##### ratings = parts.select(parts.value.getItem(0).cast("int").alias("userId"),
#####                        parts.value.getItem(1).cast("int").alias("movieId"),
#####                        parts.value.getItem(2).cast("float").alias("rating"))

##### Replacing data loading code with: ######
ratings = spark.read.option("inferSchema", "true").option("header", "true").csv("spark-examples$
```

Comparison of elapsed time taken and RMSE of ALS across varying dataset sizes



The larger the dataset, the lower the error, although, the larger the dataset, the longer processing.

This week's lab explored the ALS algorithm in Spark for movie recommendations. We began by parsing the MovieLens dataset into structured userID, movieID, and rating columns. Then, data was split into 80% training and 20% testing sets. The model was evaluated using RMSE, which measures how far predicted ratings deviate from actual ones, this reflects the accuracy of the model as higher RMSE (more deviations) reflects a lower accuracy. We then generated top 10 movie recommendations from 3 users. The experiments were scaled to three bigger dataset sizes: small sample of 100,000, 100,000 and 20 million entries. As dataset size increased, RMSE reduced (accuracy improved), demonstrating larger datasets help the ALS model generalise better in predicting user preferences. However, training time increased significantly as size increased – highlighting a common trade-off in big data, where higher accuracy is accompanied by longer computation.

The key skill developed was the ability to scale and evaluate models using Spark, while critically comparing performance across multiple dataset sizes. This week clearly demonstrated how systems such as ALS allow efficient model training at scale – essential in real-world recommendation engines.

Week 6: Introduction to Deep Learning

Objective: perform transfer learning and compare between augmented and non-augmented models. To also use transfer learning to classify images and evaluate model performance.

Part 1: Dog Breed Classification using Pretrained ResNet50 model

Transfer learning utilises pretrained models to solve new tasks, saving both time and data; it's beneficial for improving classification accuracy, reducing overfitting and making deep learning accessible with limited resources.

Figure 1a.

Transfer Learning Script

```
import numpy as np
from tensorflow.python.keras.applications.resnet50 import preprocess_input
from tensorflow.python.keras.preprocessing.image import load_img, img_to_array

image_size = 224

def read_and_prep_images(img_paths, img_height=image_size, img_width=image_size):
    imgs = [load_img(img_path, target_size=(img_height, img_width)) for img_path in img_paths]
    img_array = np.array([img_to_array(img) for img in imgs])
    output = preprocess_input(img_array)
    return(output)
```

```
from tensorflow.python.keras.applications import ResNet50

my_model = ResNet50(weights='/home/kaggle/resnet50/resnet50_weights_tf_dim_ordering_tf_kernels.h5')

test_data = read_and_prep_images(img_paths)
preds = my_model.predict(test_data)
```

```
from tensorflow.python.keras.applications.imagenet_utils import decode_predictions
from IPython.display import Image, display

#most_likely_labels = decode_predictions(preds, top=3, class_list_path='/home/kaggle/resnet50/imagenet_class_index
most_likely_labels = decode_predictions(preds, top=3)

for i, img_path in enumerate(img_paths):
  display(Image(img_paths[i]))
  print(most_likely_labels[i])
```

Figure 1b

Flowchart simplifying transfer learning process

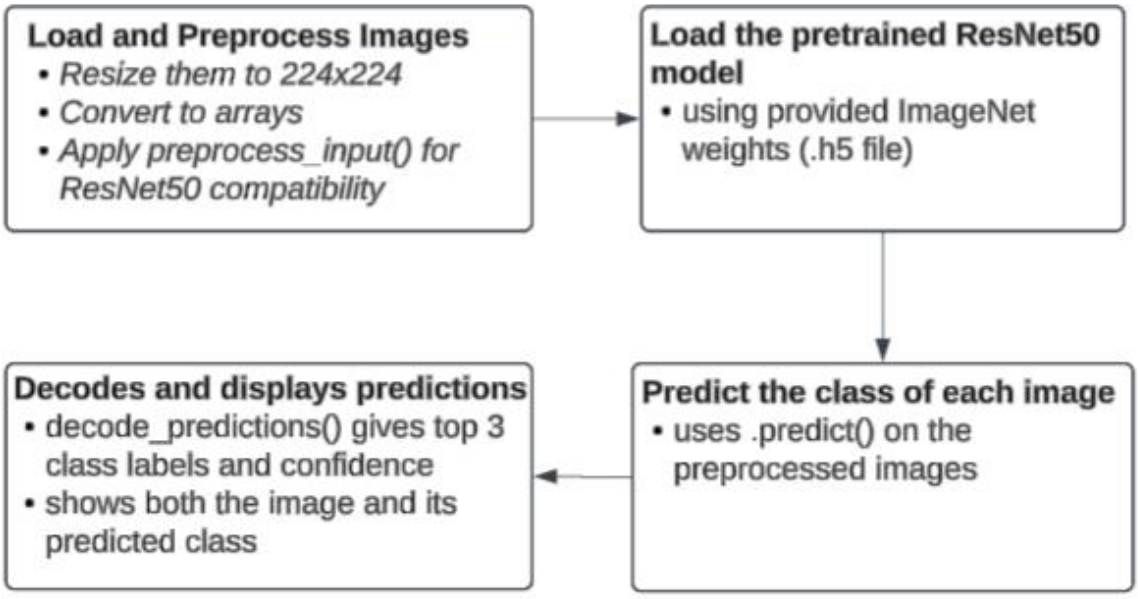


Table 2.

Image Classification Performance

Image	Top Prediction	Confidence	Other Predictions	Interpretation
	Japanese Spaniel	49.1%	Pekinese (41.4%) Blenheim Spaniel (3.4%)	Close prediction between Japanese Spaniel and Pekinese due to similar features like fur and facial shape. Suggests models limitations/sensitivity to minor breed differences
	Beagle	90.1%	English Foxhound (3.2%) Walker Hound (0.8%)	High prediction accuracy. The model correctly classified the breed with strong confidence. ResNet50 is robust for well-known dog breeds.
	Dingo	99.7%	Malinois (0.1%) Ibizan Hound (0.06%)	Excellent, almost near-perfect, certainty in identifying both dogs as dingoes.

Part 3: Transfer Learning - classifying images as from a rural or urban area

Prediction performance via transfer learning was compared between a model without augmentation, see Figure 2a, and with augmentation (Figure 2b).

Figure 2a.

No augmentation

```
from tensorflow.python.keras.applications import ResNet50
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, GlobalAveragePooling2D

# num_classes is the number of categories your model chooses between for each prediction
num_classes = 2
resnet_weights_path = '/home/kaggle/resnet50/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5'
    # Note this is *different* to weights file above.

model_no_aug = Sequential()
model_no_aug.add(ResNet50(include_top=False, pooling='avg', weights=resnet_weights_path))
model_no_aug.add(Dense(num_classes, activation='softmax'))

# We don't want to train the first layers - ResNet50 is already trained
model_no_aug.layers[0].trainable = False
```

```
my_new_model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
from tensorflow.python.keras.applications.resnet50 import preprocess_input
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator

image_size = 224
data_generator = ImageDataGenerator(preprocessing_function=preprocess_input)

train_generator = data_generator.flow_from_directory(
        '/home/kaggle/rural_and_urban_photos/train',
        target_size=(image_size, image_size),
        batch_size=24,
        class_mode='categorical')

validation_generator = data_generator.flow_from_directory(
        '/home/kaggle/rural_and_urban_photos/val',
        target_size=(image_size, image_size),
        class_mode='categorical')

my_new_model.fit_generator(
        train_generator,
        steps_per_epoch=3,
        validation_data=validation_generator,
        validation_steps=1)
```

Figure 2b.

With augmentation

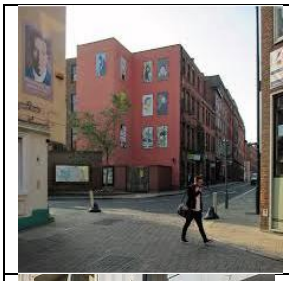
```
from tensorflow.python.keras.applications.resnet50 import preprocess_input
from tensorflow.python.keras.preprocessing.image import ImageDataGenerator

image_size = 224

data_generator_with_aug = ImageDataGenerator(preprocessing_function=preprocess_input,
                                    horizontal_flip=True,
                                    width_shift_range = 0.2,
                                    height_shift_range = 0.2)

train_generator = data_generator_with_aug.flow_from_directory(
        '/home/kaggle/rural_and_urban_photos/train',
        target_size=(image_size, image_size),
        batch_size=24,
        class_mode='categorical')

data_generator_no_aug = ImageDataGenerator(preprocessing_function=preprocess_input)
validation_generator = data_generator_no_aug.flow_from_directory(
        '/home/kaggle/rural_and_urban_photos/val',
        target_size=(image_size, image_size),
        class_mode='categorical')

my_new_model.fit_generator(
        train_generator,
        steps_per_epoch=3,
        epochs=2,
        validation_data=validation_generator,
        validation_steps=1)
```

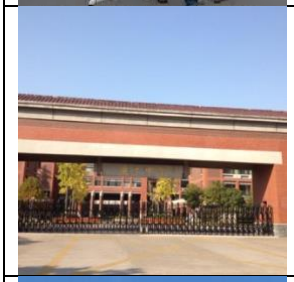
```
preds = model_aug.predict(test_data)
class_indices = validation_generator.class_indices
print(class_indices)

for i, img_path in enumerate(img_paths):
    display(Image(img_path))
    print(preds[i])
```

Table 3.

Summary of output:

Image	Rural (No Aug)	Urban (No Aug)	Rural (Aug)	Urban (Aug)	Interpretation
	0.61	0.39	0.33	0.67	Incorrect prediction without augmentation, but model confidently predicted urban with augmentation.

	0.05	0.95	0.01	0.99	Both models confidently identify photo as urban
	0.04	0.96	0.01	0.99	Both models confidently identify photo as urban
	0.37	0.63	0.15	0.85	Both models confidently identify photo as urban, model with augmentation more confident.
	0.12	0 .88	0.01	0.99	With augmentation, almost 100% certain image is urban.
	0.18	0.82	0.05	0.95	Both models strongly predicted urban.
	0.41	0.59	0.14	0.86	Without augmentation is very uncertain, but with augmentation correctly predicted with great confidence.

	0.32	0.68	0.05	0.95	With augmentation predicted class with more certainty
	0.31	0.69	0.21	0.79	With augmentation predicted class with more certainty
	0.47	0.53	0.15	0.85	With augmentation predicted class with more certainty
	0.68	0.32	0.45	0.55	Initially classified as rural, adding augmentation introduced confusion, made the model less certain
	0.89	0.11	0.94	0.06	Both models strongly predicted image is rural.
	0.62	0.38	0.65	0.35	Both images predicted image as rural, but certainty is low.

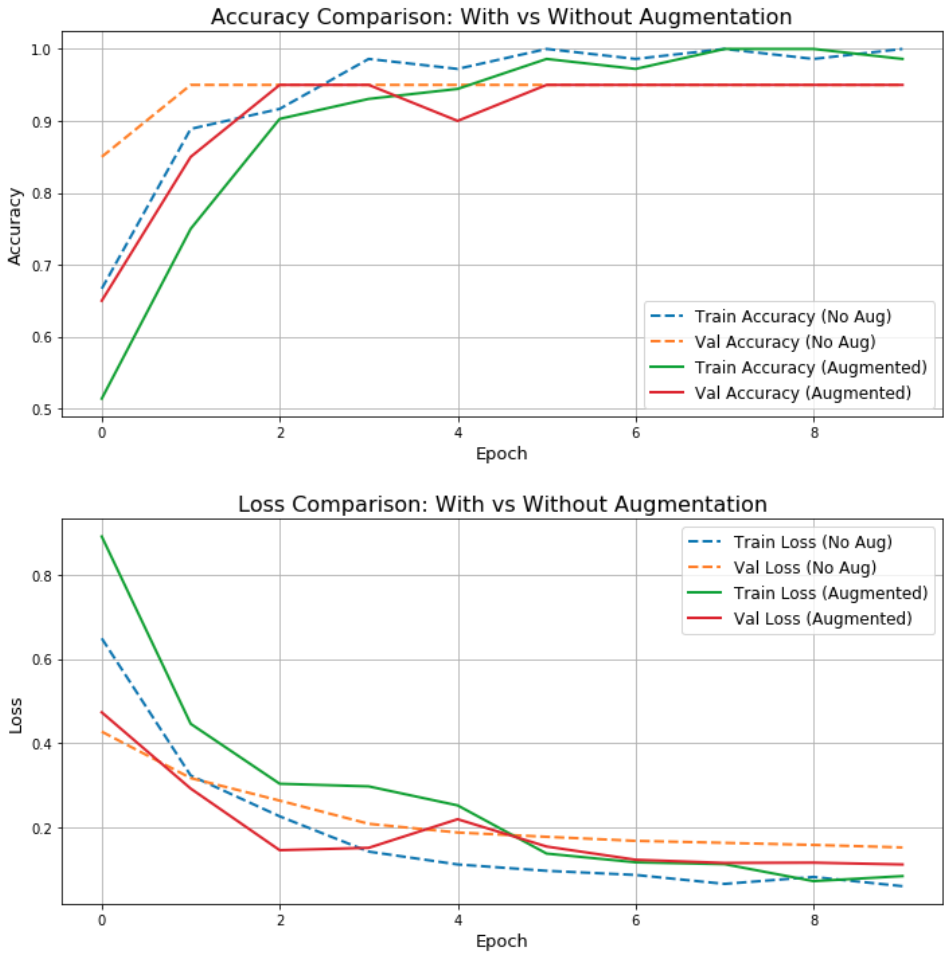
	0.99	0.01	0.99	0.01	Both models almost perfectly predicted image is rural.
	0.72	0.28	0.85	0.15	Both models strong predict image is from a rural area, with augmentation more certain.
	0.77	0.23	0.92	0.08	Both models strong predict image is from a rural area, with augmentation more certain.
	0.97	0.03	0.99	0.01	Both models very strongly predicted image is rural.
	0.89	0.11	0.97	0.03	Both models strongly predicted image is rural, model with augmentation slightly outperforms.
	1.00	0.00	1.00	0.00	Perfect classification from both models that the image is from a rural area.

	0.94	0.06	0.93	0.07	Both models predicted rural, without augmentation outperforms with augmentation by 0.1.

The model with augmentation performed the best with most classification accuracy however it shows mild overfitting – training loss drops while validation loss plateaus. With augmentation, training is harder, but validation performance improves, suggesting better generalisation, see Figure 3.

Figure 3.

Accuracy Loss Plot for both models



Augmentation model performed well on distinct urban/rural area but struggled with ambiguous images, using a ResNet50 allowed higher accuracy. Future work could

explore how to minimise the risk of overfitting with the augmented model, given its practical value.

This lab allowed me to gain experience in understanding the effect of image classification using pretrained ResNet50 for transfer learning, thus adding to my skillset.

Week 7: Building Convolutional Neural Networks

Objective: Build CNNs from scratch and compare the performance of different versions of a CNN with model enhancements on an image classification task.

Part 1: Building CNN's

Figure 1.

Basic model layer + keras visual

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, Conv2D, Dropout

model = Sequential()
model.add(Conv2D(20, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(img_rows, img_cols, 1)))
model.add(Conv2D(20, kernel_size=(3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer='adam',
              metrics=['accuracy'])
```

Figure 2.

Improved model layer + keras visual

```
model = Sequential()
model.add(Conv2D(30, kernel_size=(3, 3),
                 strides=2,
                 activation='relu',
                 input_shape=(img_rows, img_cols, 1)))
model.add(Dropout(0.5))
model.add(Conv2D(30, kernel_size=(3, 3), strides=2, activation='relu'))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer='adam',
              metrics=['accuracy'])
model.fit(x, y,
          batch_size=128,
          epochs=2,
          validation_split = 0.2)
```

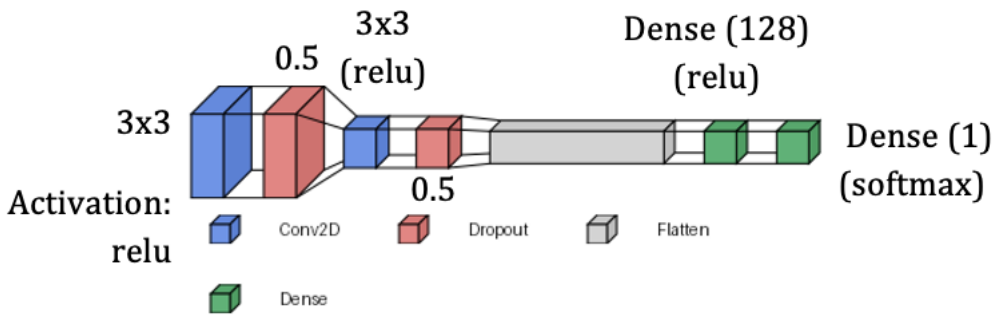


Table 1.

Summary of basic and improved model

Feature	Basic Model	Improved Model
Filters (features)	20 per layer	30 per layer (shrinks image)
Image Reduction	No	Yes (uses stride)
Layers	2 conv --> Flatten --> 2 Dense	Conv --> dropout (prevent overfitting) --> conv --> Flatten --> 2 dense

Part 2: Applying the improved model to an Astronomical Image Set

Objective: classify galaxy images into three classes:

1. Smooth
2. Disk
3. Artefact

The accuracy of each classification was compared across different versions of the improved model:

Table 2.

Model versions

Model Version	Conv Layers	Kernel Size	Dropout?	Extras
V1 (complex)	3	5x5	No	
V2 (simplified)	2	3x3	No	
V3 (Dropout)	2	3x3	0.5	
V4 (augmented)	2	3x3	0.5	Horizontal + Vertical Flip

Table 3.

Results

Image	V2	V1	V3	V4
	**Smooth** *0.101*	**Smooth** *0.459*	**Smooth** *0.225*	**Smooth** *0.181*
	Disk *0.896*	**Disk** *0.540*	**Disk** *0.765*	**Disk** *0.806*
	Artefact *0.003*	**Artefact** *0.001*	**Artefact** *0.001*	**Artefact** *0.013*
	**Smooth** *0.092*	**Smooth** *0.067*	**Smooth** *0.103*	**Smooth** *0.058*
	Disk *0.908*	**Disk** *0.933*	**Disk** *0.896*	**Disk** *0.942*
	Artefact *0.000*	**Artefact** *0.000*	**Artefact** *0.000*	**Artefact** *0.000*
	**Smooth** *0.718*	**Smooth** *0.628*	**Smooth** *0.544*	**Smooth** *0.791*
	Disk *0.282*	**Disk** *0.372*	**Disk** *0.456*	**Disk** *0.209*
	Artefact *0.000*	**Artefact** *0.000*	**Artefact** *0.000*	**Artefact** *0.000*
	**Smooth** *0.070*	**Smooth** *0.119*	**Smooth** *0.131*	**Smooth** *0.084*
	Disk *0.930*	**Disk** *0.881*	**Disk** *0.869*	**Disk** *0.916*
	Artefact *0.000*	**Artefact** *0.000*	**Artefact** *0.000*	**Artefact** *0.000*
	Smooth	**Smooth**	**Smooth**	**Smooth**

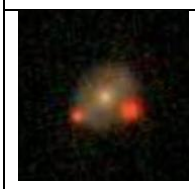
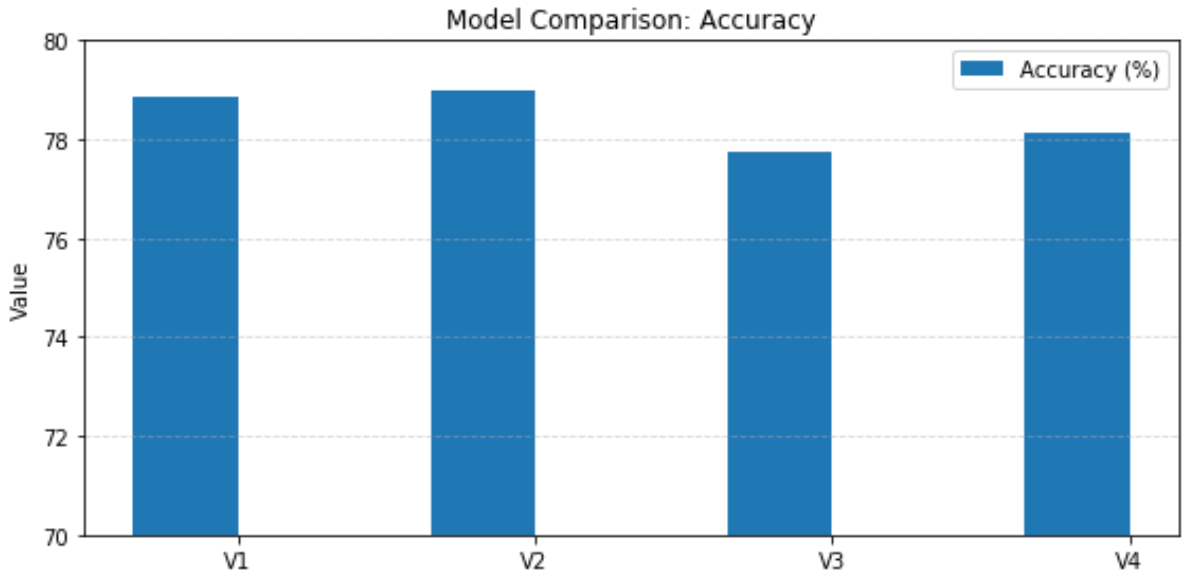
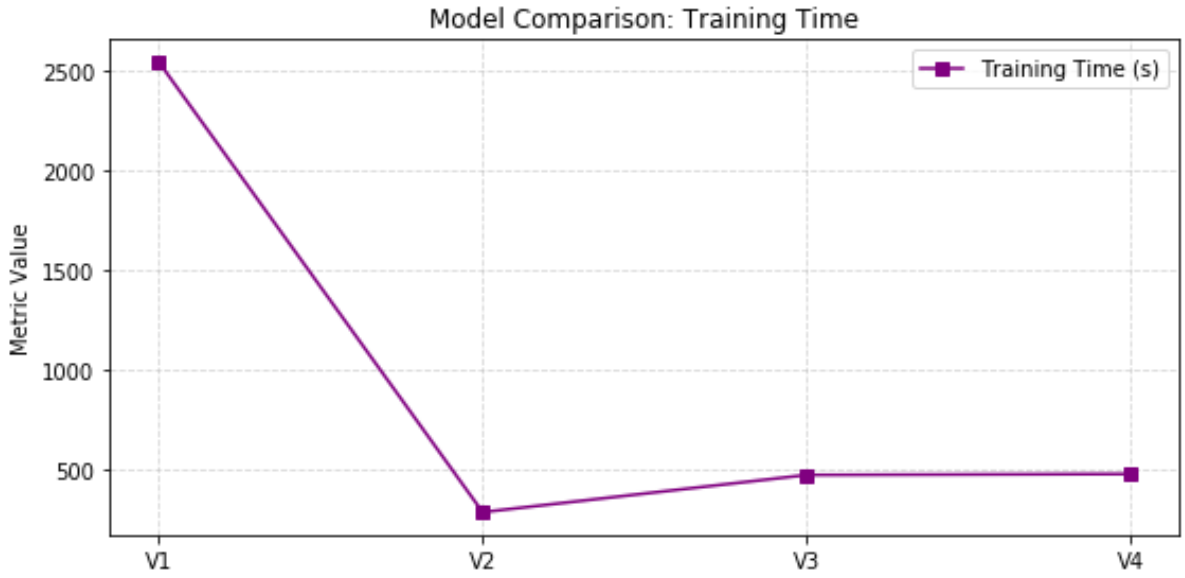
	0.195	0.305	0.198	0.416
	Disk 0.805	Disk 0.695	Disk 0.802	Disk 0.584
	Artefact 0.000	Artefact 0.000	Artefact 0.000	Artefact 0.000
	Smooth 0.085	Smooth 0.209	Smooth 0.210	Smooth 0.217
	Disk 0.915	Disk 0.790	Disk 0.789	Disk 0.782
	Artefact 0.000	Artefact 0.001	Artefact 0.001	Artefact 0.001
	Smooth 0.531	Smooth 0.739	Smooth 0.536	Smooth 0.581
	Disk 0.469	Disk 0.261	Disk 0.463	Disk 0.418
	Artefact 0.000	Artefact 0.000	Artefact 0.001	Artefact 0.001
	Smooth 0.056	Smooth 0.104	Smooth 0.149	Smooth 0.065
	Disk 0.944	Disk 0.896	Disk 0.851	Disk 0.935
	Artefact 0.000	Artefact 0.000	Artefact 0.000	Artefact 0.000
	Smooth 0.299	Smooth 0.669	Smooth 0.482	Smooth 0.358
	Disk 0.701	Disk 0.330	Disk 0.517	Disk 0.641
	Artefact 0.000	Artefact 0.001	Artefact 0.001	Artefact 0.001

Figure 3.

Accuracy and train time comparison





Convolutional networks are crucial in big data applications, where image-based classification is required at scale. This experiment compared different model versions showing that removing dropout and reducing kernel size yielded a higher accuracy and quicker training time, as it preserved all features during training. However, V3 underperformed, perhaps due to excessive dropout, which overly constrained the model and hindered learning. This emphasises the importance of model architecture and choice of model tuning when handling large data. I found this week's lab incredibly useful, it was very intriguing to see how architecture is important in model performance, where small changes can greatly impact accuracy. A key skill developed was being able to critically compare model performance through structured/systematic experimentation and evaluation, which will be immensely useful for the mini project and in future.

Week 8: Spark Streaming Basics

Objective: Perform basic actions in spark streaming and applying it to two live activity datasets.

Streaming data arrives continuously, consisting of real-time updates where the program is continuously running and the output keeps changing.

Part 1: Applying basics to streaming dataframe

Table 1.

Spark streaming basics

Loading data

`>>> streamingDataFrame = (spark.readStream.schema(staticSchema).option("maxFilesPerTrigger", 1)`
`... .format("csv").option("header", "true")`
`... .load("file:///home/spark/Spark-The-Definitive-Guide-master/data/retail-data/by-day/*.csv"))`

Running Query

`>>> purchaseByCustomerPerHour = (streamingDataFrame.selectExpr("CustomerId", "(UnitPrice * Quantity) as total_cost", "InvoiceDate")`
`... .groupBy(col("CustomerId"), window(col("InvoiceDate"), "1 day")).sum("total_cost"))`
`>>> purchaseQuery = (purchaseByCustomerPerHour.writeStream.format("memory")`
`... .queryName("customer_purchases").outputMode("complete").start())`

Sample of output:

```
-------------------------------------------
Batch: 0
-------------------------------------------
+----------+--------------------+--------------------+
|CustomerId|              window|    sum(total_cost)|
+----------+--------------------+--------------------+
|   12709.0|[2011-06-02 01:00...| 1225.2599999999995|
|   16133.0|[2011-06-02 01:00...|-31.150000000000002|
|   17531.0|[2011-06-02 01:00...| 296.70000000000005|
|   13854.0|[2011-06-02 01:00...|               -2.95|
|   14221.0|[2011-06-02 01:00...|  789.8499999999999|
|   15838.0|[2011-06-02 01:00...|             2105.98|
|   16729.0|[2011-06-02 01:00...|              292.74|
|   15194.0|[2011-06-02 01:00...|               -3.25|
|   14323.0|[2011-06-02 01:00...|-17.700000000000003|
|   12431.0|[2011-06-02 01:00...|               -23.0|
|   14903.0|[2011-06-02 01:00...|               -18.4|
|   16566.0|[2011-06-02 01:00...|               -0.65|
|   14088.0|[2011-06-02 01:00...|  7809.929999999998|
|   15078.0|[2011-06-02 01:00...| 345.70000000000005|
|   12971.0|[2011-06-02 01:00...|              221.12|
|   17375.0|[2011-06-02 01:00...|               -27.6|
|   13534.0|[2011-06-02 01:00...|              -13.04|
|   13408.0|[2011-06-02 01:00...|               397.8|
|   13334.0|[2011-06-02 01:00...| 1071.4000000000003|
|   15184.0|[2011-06-02 01:00...| 235.23000000000002|
+----------+--------------------+--------------------+
only showing top 20 rows

-------------------------------------------
Batch: 1
-------------------------------------------
+----------+--------------------+--------------------+
|CustomerId|              window|    sum(total_cost)|
+----------+--------------------+--------------------+
|   14016.0|[2011-05-16 01:00...| 1133.3300000000002|
|   12709.0|[2011-06-02 01:00...| 1225.2599999999995|
|   15125.0|[2011-05-16 01:00...| 2942.8400000000006|
|   14211.0|[2011-05-16 01:00...| 150.39999999999998|
|   15910.0|[2011-05-16 01:00...| 120.82000000000002|
|   18102.0|[2011-05-16 01:00...|             8895.66|
|   16133.0|[2011-06-02 01:00...|-31.150000000000002|
|   17531.0|[2011-06-02 01:00...| 296.70000000000005|
|   13854.0|[2011-06-02 01:00...|               -2.95|
|   14221.0|[2011-06-02 01:00...|  789.8499999999999|
|   15838.0|[2011-06-02 01:00...|             2105.98|
|   16729.0|[2011-06-02 01:00...|              292.74|
|   17841.0|[2011-05-16 01:00...|  463.3299999999998|
|   13408.0|[2011-05-16 01:00...|             1879.52|
|   15194.0|[2011-06-02 01:00...|               -3.25|
|   12674.0|[2011-05-16 01:00...|              586.74|
|   14323.0|[2011-06-02 01:00...|-17.700000000000003|
|   13319.0|[2011-05-16 01:00...|  732.9199999999998|
|   17898.0|[2011-05-16 01:00...|  99.83999999999999|
|   12431.0|[2011-06-02 01:00...|               -23.0|
+----------+--------------------+--------------------+
only showing top 20 rows
```

Part 2: Applying Spark Streaming to live retail data

Table 2.

Results

Importing dataset

```
streamingDataFrame = (spark.readStream.schema(staticSchema).option
 ("maxFilesPerTrigger", 1)
.format("csv").option("header", "true")
.load("/content/Spark-The-Definitive-Guide/data/retail-data/by-day/*.csv"))
```

Query: Groups purchase per customer per day and sums total cost

```
purchaseByCustomerPerHour = (streamingDataFrame.selectExpr("CustomerId", "(UnitPrice * Quantity)
as total_cost", "InvoiceDate")
.groupBy(col("CustomerId"), window(col("InvoiceDate"), "1 day")).sum("total_cost"))
```

Streaming activation: Writes aggregated purchase data to memory as a named streaming table

```
purchaseQuery =  (purchaseByCustomerPerHour.writeStream.format("memory")
.queryName("customer_purchases").outputMode("complete").start())
```

Produces customer ID and total cost, changing in value for every time the code is ran.

Part 3: Spark Streaming on dataset about mobile activity sensor data

Table 3.

Results

Dataset:

Arrival_Time	Creation_Time	Device	Index	Model	User	gt	x	y	z
1424686735090	1424686733090638193	nexus4_1	18	nexus4	g	stand	3.356934E-4	-5.645752E-4	-0.018814087
1424686735292	1424688581345918092	nexus4_2	66	nexus4	g	stand	-0.005722046	0.029083252	0.005569458
1424686735500	1424686733498505625	nexus4_1	99	nexus4	g	stand	0.0078125	-0.017654419	0.010025024
1424686735691	1424688581745026978	nexus4_2	145	nexus4	g	stand	-3.814697E-4	0.0184021	-0.013656616
1424686735890	1424688581945252808	nexus4_2	185	nexus4	g	stand	-3.814697E-4	-0.031799316	-0.00831604
1424686736094	1424686734097840342	nexus4_1	218	nexus4	g	stand	-7.324219E-4	-0.013381958	0.01109314
1424686736294	1424688582347932252	nexus4_2	265	nexus4	g	stand	-0.005722046	0.015197754	0.022659302
1424686736495	1424688582549592408	nexus4_2	305	nexus4	g	stand	-3.814697E-4	0.0087890625	0.0034332275
1424686736697	1424688582750703248	nexus4_2	345	nexus4	g	stand	0.002822876	-0.008300781	-0.015792847
1424686736898	1424688582952241334	nexus4_2	385	nexus4	g	stand	6.866455E-4	-0.008300781	0.004501343
1424686737100	1424686735109928643	nexus4_1	418	nexus4	g	stand	0.003540039	-0.010177612	-0.026290894
1424686737300	1424688583355164918	nexus4_2	465	nexus4	g	stand	0.002822876	0.0045166016	-0.014724731
1424686737505	1424686735512935017	nexus4_1	498	nexus4	g	stand	0.0024719238	-0.010177612	-0.017745972
1424686737707	1424686735709254597	nexus4_1	537	nexus4	g	stand	-0.0028686523	-0.003768921	0.020706177
1424686737908	1424686735915675495	nexus4_1	578	nexus4	g	stand	-0.0028686523	0.026138306	0.007888794
1424686738109	1424688584160372793	nexus4_2	625	nexus4	g	stand	-3.814697E-4	2.441406E-4	0.033340454
1424686738326	1424688584381747305	nexus4_2	661	nexus4	g	stand	0.0017547607	0.019470215	-0.011520386
1424686738529	1424686736534938191	nexus4_1	701	nexus4	g	stand	0.0024719238	-0.033676147	0.0068206787
1424686738744	1424688584799723655	nexus4_2	744	nexus4	g	stand	-3.814697E-4	-0.002960205	-0.027542114
1424686738935	1424686736943477009	nexus4_1	782	nexus4	g	stand	-0.009277344	-0.009109497	-0.0690155

Creating the streaming dataframe:

```
streaming = (spark.readStream.schema(static.schema).option("maxFilesPerTrigger", 1)
.json("/content/Spark-The-Definitive-Guide/data/activity-data/*.json"))
```

Processing transformation – involving grouping records by activity and counting them:

```
activityCounts = streaming.groupBy("gt").count()
```

Initiating process

```
activityQuery=activityCounts.writeStream.queryName("activity_counts").
format("memory").outputMode("complete").start()
```

Showing output table in a loop – changes over time

```
from time import sleep

for x in range(5) :
  spark.sql("SELECT * FROM activity_counts").show()
  sleep(1)
```

```
+----------+-----+
|        gt|count|
+----------+-----+
|       sit|12309|
|     stand|11384|
|stairsdown| 9365|
|      walk|13256|
|  stairsup|10452|
|      null|10449|
|      bike|10796|
+----------+-----+

+----------+-----+
|        gt|count|
+----------+-----+
|       sit|24619|
|     stand|22769|
|stairsdown|18729|
|      walk|26512|
|  stairsup|20905|
|      null|20896|
|      bike|21593|
+----------+-----+

+----------+-----+
|        gt|count|
+----------+-----+
|       sit|36929|
|     stand|34154|
|stairsdown|28094|
|      walk|39768|
|  stairsup|31357|
|      null|31343|
|      bike|32390|
+----------+-----+
```

This week focused on applying Apache Spark Streaming to two real-time datasets: retail transactions and mobile activity data. CSV files were read incrementally to construct the streaming dataframe for retail dataset. Data was grouped and results were stored as a memory table using writeStream, allowing continuous SQL queries to display changing customer totals.

JSON data was imported and categorised by the gt activity label in the mobile activity dataset. Counts were displayed using SQL queries inside a lop, showing an updated value for gt activity with every batch.

This week was beneficial in providing me with basic skills in Apache Spark Streaming.

Week 9: Introduction to Recurrent Neural Networks

Objective: to be able to build RNN's from scratch and enhance their model architecture in classification tasks.

Part 1a: Classification with Dense ANN

The model architecture of Dense ANN was experimented with to investigate the impact of structural tuning on classification performance.

Table 1

Model versions built

Version	Function
V1	Simple model
V2	Increasing embedding
V3	Increasing dense layers
V4	Changing optimiser to Adam
V5	Combining all enhancements

Table 2

Function, Script and Output of each model

Function	Script
Loading the IMDB Dataset	`from keras.datasets import imdb` `(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)`

Vectorise the sequences	`import numpy as np`
	
	`def vectorize_sequences(sequences, dimension=10000):`
	`  #Creates an all-zero matrix`
	`  results = np.zeros((len(sequences), dimension))`
	`  #Set approperiate indices of the tensor result to 1s`
	`  for i, sequence in enumerate(sequences):`
	`  results[i, sequence] = 1.`
	`  return results`
	
	`#Turn Python lists into NumPy tensors`
	`x_train = vectorize_sequences(train_data)`
	`x_test = vectorize_sequences(test_data)`

Prepare Labels (0 = negative, 1 = positive)	`y_train = np.asarray(train_labels).astype('float32')` `y_test = np.asarray(test_labels).astype('float32')`

Define a Dense ANN Model	`from keras import models`
	`from keras import layers`
	
	`model = models.Sequential()`
	`model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))`
	`model.add(layers.Dense(16, activation='relu'))`
	`model.add(layers.Dense(1, activation='sigmoid'))`

Compile the model	`model.compile(optimizer='rmsprop',`
	`loss='binary_crossentropy',`
	`metrics=['accuracy'])`

Train the model on training data	`#Setting aside a validation set`
	`x_val = x_train[:10000]`
	`partial_x_train = x_train[10000:]`
	
	`y_val = y_train[:10000]`
	`partial_y_train = y_train[10000:]`
	
	`#Training the model`
	`model.fit(partial_x_train,`
	`partial_y_train,`
	`epochs=20,`
	`batch_size=512,`
	`validation_data=(x_val, y_val))`

Evaluate on test data	`results = model.evaluate(x_test, y_test)`

Make predictions	`from prettytable import PrettyTable`
	`#Prediction`
	`preds = model.predict(x_test [0:10])`
	
	`t = PrettyTable(['Real', 'Predicted'])`
	`for i, pred in enumerate(preds):`
	`t.add_row([y_test [i], round(preds [i,0],2)])`
	`print(t)`

Output	+------+-----------+ | Real | Predicted | +------+-----------+ | 0.0 | 0.01 | | 1.0 | 1.0 | | 1.0 | 0.82 | | 0.0 | 0.93 | | 1.0 | 0.99 | | 1.0 | 1.0 | | 1.0 | 1.0 | | 0.0 | 0.0 | | 0.0 | 0.99 | | 1.0 | 1.0 | +------+-----------+
Accuracy and Loss	`accuracy: 0.8556 - loss: 0.6170`

Training Time	Training Time: 44.883241176605225 seconds

Increasing embedding size (V2)	`model = models.Sequential()`
	`model.add(layers.Embedding(max_features, 128))  # Increased Embedding size to 128`
	`model.add(layers.Flatten())`
	`model.add(layers.Dense(16, activation='relu'))`
	`model.add(layers.Dense(16, activation='relu'))`
	`model.add(layers.Dense(1, activation='sigmoid'))`

Metrics	+------+-----------+ | Real | Predicted | +------+-----------+ | 0.0 | 0.35 | | 1.0 | 0.54 | | 1.0 | 0.01 | | 0.0 | 0.18 | | 1.0 | 0.65 | | 1.0 | 0.27 | | 1.0 | 0.24 | | 0.0 | 0.12 | | 0.0 | 0.24 | | 1.0 | 0.62 | +------+-----------+ Training Time: 550.96 seconds Test Accuracy: 70.27%

Adding more dense layers (V3)	`# Build model`
	`model = models.Sequential()`
	`model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))`
	`model.add(layers.Dense(64, activation='relu'))`
	`model.add(layers.Dense(32, activation='relu'))  # Added extra Dense layer`
	`model.add(layers.Dense(1, activation='sigmoid'))`

Metrics	+------+-----------+ | Real | Predicted | +------+-----------+ | 0.0 | 0.02 | | 1.0 | 1.0 | | 1.0 | 0.0 | | 0.0 | 0.99 | | 1.0 | 1.0 | | 1.0 | 0.98 | | 1.0 | 1.0 | | 0.0 | 0.0 | | 0.0 | 0.99 | | 1.0 | 1.0 | +------+-----------+ Training Time: 166.58 seconds Test Accuracy: 81.76%
Changing the optimizer (Adam) (V4)	`model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])`

<table>
<tr><td>Metrics</td><td>

Real	Predicted
0.0	0.0
1.0	1.0
1.0	0.62
0.0	0.34
1.0	1.0
1.0	0.99
1.0	1.0
0.0	0.0
0.0	1.0
1.0	1.0

Training Time: 82.30 seconds
Test Accuracy: 86.26%
</td></tr>
<tr><td>Combining all (V5)</td><td>

```
model = models.Sequential()
model.add(layers.Embedding(max_features, 128))  # Increased Embedding size to 128
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(32, activation='relu'))  # Added extra Dense layer
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

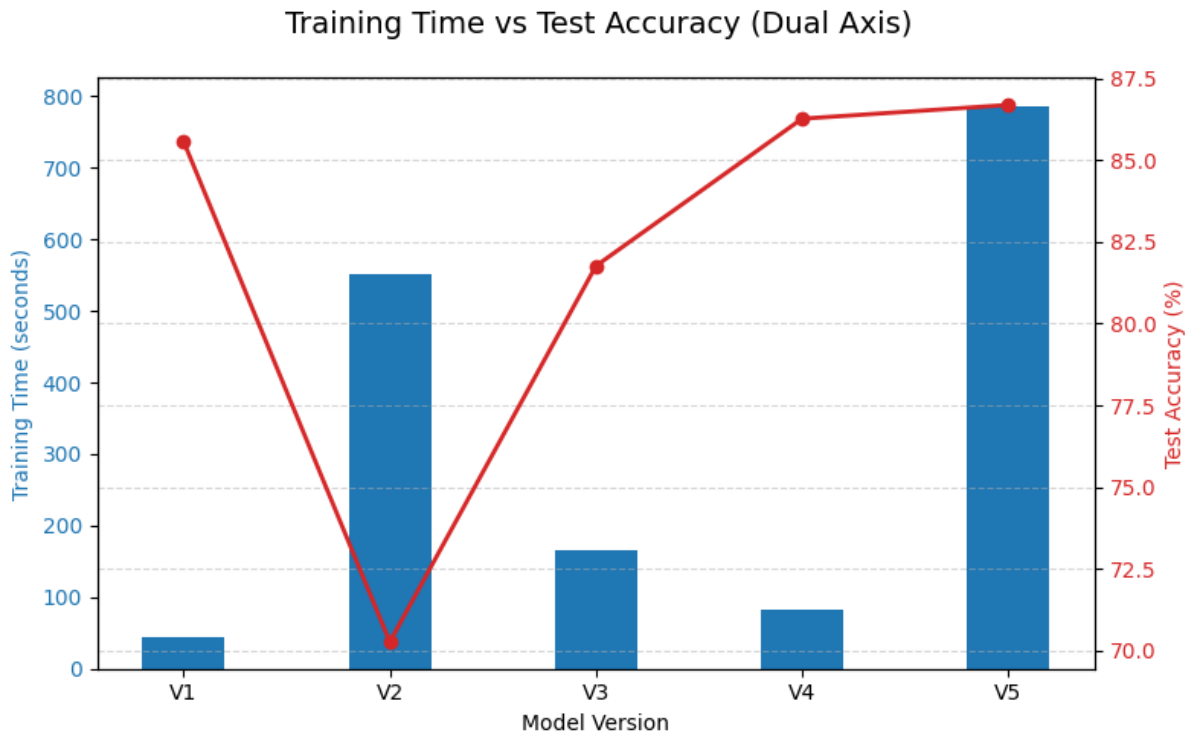
</td></tr>
<tr><td>Metrics</td><td>

Real	Predicted
0.0	0.0
1.0	1.0
1.0	0.26
0.0	0.25
1.0	1.0
1.0	1.0
1.0	0.98
0.0	0.0
0.0	0.99
1.0	1.0

Test Accuracy: 86.68%
Training Time: 786.01 seconds
</td></tr>
</table>

Figure 1.

Comparison of model performance across all versions



V4 (Adam optimizer) gave the best balance of speed and accuracy. V5 achieved the highest accuracy but was slowest. V2 increased time but lowered accuracy. V1 was fastest, V3 moderately improved results.

Using the Adam Optimiser offered the best trade-off between speed and accuracy. V5 achieved the highest accuracy but was the slowest. V2 increased time but lowered accuracy, increasing embedding hurt the model. The baseline model with no enhancements was the fastest and V3 moderately improved results. Model architecture is important in producing reliable classification results,

Part 2: Classification with simple RNN and LSTM

Table 2.

Function, script and output of RNN and LSTM models

Function	Script

Load and preprocess dataset (padding – all reviews exactly 500 words long)	`from keras.datasets import imdb`
	`from keras.utils import pad_sequences`
	
	`max_features = 10000`
	`maxlen = 500`
	`batch_size = 32`
	
	`#Load data and pad it`
	`(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)`
	`input_train = pad_sequences(input_train, maxlen=maxlen)`
	`input_test = pad_sequences(input_test, maxlen=maxlen)`

Building basic SimpleRNN model	```from keras.layers import Embedding, SimpleRNN, Dense
	
	#Define mode tl
	model = models.Sequential()
	model.add(Embedding(max_features, 32))
	model.add(SimpleRNN(32))
	model.add(Dense(1, activation='sigmoid'))
	
	model.compile(optimizer='rmsprop',
	loss='binary_crossentropy',
	metrics=['acc'])
	
	#Train the model
	model.fit(input_train, y_train,
	epochs=15,
	batch_size=128,
	validation_split=0.2)```

Evaluating the model	```results = model.evaluate(input_test, y_test)```

Make predictions	```from prettytable import PrettyTable
	
	# Assuming 'input_test' is your test data from cell 'ipython-input-1-80a2ba7c5744'
	preds = model.predict(input_test[0:10]) # Changed 'test_data' to 'input_test'
	t = PrettyTable(['Real', 'Predicted'])
	for i, pred in enumerate(preds):
	t.add_row([y_test[i], round(pred[0], 2)])
	print(t)```

Output	```+------+-----------+ | Real | Predicted | +------+-----------+ | 0 | 0.57 | | 1 | 0.62 | | 1 | 0.67 | | 0 | 0.48 | | 1 | 0.67 | | 1 | 0.38 | | 1 | 0.49 | | 0 | 0.34 | | 0 | 0.48 | | 1 | 0.51 | +------+-----------+```
Metrics	```Training Time: 589.25 seconds Test Accuracy: 82.18%```

<table>
<tr><td>Increasing recurrent layers</td><td>

```
# Define model
model = models.Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
```

</td></tr>
<tr><td>Output</td><td>

```
+------+-----------+
| Real | Predicted |
+------+-----------+
|  0   |    0.0    |
|  1   |    1.0    |
|  1   |    0.0    |
|  0   |    0.0    |
|  1   |    1.0    |
|  1   |    1.0    |
|  1   |    0.99   |
|  0   |    0.0    |
|  0   |    0.04   |
|  1   |    0.25   |
+------+-----------+
```

</td></tr>
<tr><td>Metrics</td><td>

```
Training Time: 1974.71 seconds
Test Accuracy: 75.10%
```

</td></tr>
<tr><td colspan="2">Adding more recurrent layers harmed the accuracy of the model – it made the model deeper not smarter</td></tr>
</table>

<table>
<tr><td>Building Model with LSTM</td><td>

```
from keras.layers import LSTM

#Define LSTM network
model = models.Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

#Train the model
model.fit(input_train, y_train,
          epochs=20,
          batch_size=128,
          validation_split=0.2)
```

</td></tr>
<tr><td>Output</td><td>

```
+------+-----------+
| Real | Predicted |
+------+-----------+
|  0   |    0.04   |
|  1   |    1.0    |
|  1   |    0.01   |
|  0   |    0.0    |
|  1   |    1.0    |
|  1   |    0.87   |
|  1   |    0.93   |
|  0   |    0.0    |
|  0   |    1.0    |
|  1   |    1.0    |
+------+-----------+
```

</td></tr>
<tr><td>Metrics</td><td>

```
Training Time: 1556.41 seconds
 Test Accuracy: 83.21%
```

</td></tr>
<tr><td colspan="2">LSTM boosted accuracy, marginally outperforming RNN with one layer</td></tr>
</table>

Further improved model: increased embedding size, adam optomiser, more dense layers.	```# Define Improved LSTM model
	model = models.Sequential()
	#increased embedding size
	model.add(Embedding(max_features, 128, input_length=maxlen))
	#increased LSTM units
	model.add(LSTM(64))
	# extra dense layer
	model.add(Dense(64, activation='relu'))
	#add dropout to prevent overfitting
	model.add(Dropout(0.5))
	#another dense layer
	model.add(Dense(32, activation='relu'))
	model.add(Dense(1, activation='sigmoid'))
	
	# changed optimiser to adam
	model.compile(optimizer='adam',
	loss='binary_crossentropy',
	metrics=['acc'])```

output	+------+-----------+ | Real | Predicted | +------+-----------+ | 0.0 | 0.02 | | 1.0 | 1.0 | | 1.0 | 0.44 | | 0.0 | 0.97 | | 1.0 | 1.0 | | 1.0 | 0.98 | | 1.0 | 1.0 | | 0.0 | 0.0 | | 0.0 | 0.99 | | 1.0 | 0.97 | +------+-----------+
metrics	Training time: 1313.56 seconds Accuracy: 86.12%
LSTM with structural tuning enhanced model performance	

Table 3.

Function, script and output of CNN models

<table>
<tr><th>Function</th><th>Script</th></tr>
<tr><td>Build model</td><td>

```
model = models.Sequential()
model.add(Embedding(max_features, 128, input_length=maxlen))
model.add(layers.Conv1D(32, 7, activation='relu'))            # Convolution layer
model.add(layers.MaxPooling1D(5))                             # MaxPooling to downsample
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())                        # Global pooling to flatten
model.add(layers.Dense(1, activation='sigmoid'))              # Output layer

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
```

</td></tr>
<tr><td>Output</td><td>

Real	Predicted
0	0.04
1	1.0
1	0.01
0	0.0
1	1.0
1	0.87
1	0.93
0	0.0
0	1.0
1	1.0

</td></tr>
<tr><td>Metrics</td><td>

```
Training Time: 1556.41 seconds
Test Accuracy: 83.21%
```

</td></tr>
<tr><td>Enhancing model</td><td>

```
model = models.Sequential()
model.add(layers.Embedding(max_features, 256, input_length=maxlen))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

</td></tr>
<tr><td>Metrics</td><td>

```
Training Time: 909.66 seconds
Test Accuracy: 86.43%
```

</td></tr>
<tr><td colspan="2">Enhancing model and architecture improved accuracy</td></tr>
</table>

This week highlighted the importance of model architecture on deep learning neural networks, manipulating the tuning of the models allowed us to grasp a better, valuable

understanding of the purpose of model parameters and its importance. Adjusting parameters such as embedding size, number of dense layers and using an Adam optimizer can impact accuracy, which is incredibly useful for scalable big data applications in real-world settings, such as sentiment analysis on movie reviews, social media datasets etc.

Big Data and Machine Learning Applications Project

Word Count: 1,988

Project Introduction & Background

Sentiment analysis, integral to natural language processing, involves determining the emotional tone behind textual data, with greatest application to analysing movie reviews on platforms like IMDb. Traditional methods such as Naïve Bayes or K-Nearest Neighbour don't always capture the complexity of human language (Olalere, 2024). Businesses that rely on sentiment analysis (e.g. streaming services and review platforms) require scalable, accurate models to process large volumes of text; hence, deep learning models have emerged as a powerful alternative to traditional models. Dense Artificial Neural Networks capture basic patterns in text, while Convolutional Neural Networks can identify local patterns in text by detecting key n-gram features in text efficiently. Recurrent Neural Networks, especially Long Short-Term Memory networks capture flow and context of language over time. Deep learning models have showcased superiority in accuracy to traditional methods in sentiment analysis of movie reviews (Kazemi & Pa, 2022).

Efficient model architectures affect performances of deep learning models (Juraev et al. 2024). The proper architecture balances accuracy, training time, and generalization (Pang, B., & Lee, L. 2008). Therefore, this study will systematically evaluate and compare the effectiveness of the proposed deep learning models by applying architectural enhancements to identify the optimal architecture for binary sentiment classification in movie reviews – addressing these research questions:

1. *Which neural network architecture (ANN, RNN, LSTM, CNN) performs best for binary sentiment classification of movie reviews?*
2. *How do architectural improvements (embedding size, optimizer choice and additional dense layers) impact performance?*
3. *Which neural network and model improvement is the most computationally expensive (i.e. largest training time)*
4. *Can pre-trained word embeddings like GloVe improve performance and reduce overfitting in deep learning models for sentiment analysis*

Data preparation

The IMDB dataset, sourced from Kaggle, contains about 1,444,000 movie reviews which came pre-labelled as positive (2) or negative (1), and was relabelled as (0), (1) for binary classification. It also required further data preparation, see Figure 1. Due to CPU limitations, we sampled 10% of this data to 144,000 reviews. After removing null values, this sample reduced to 138,500.

Figure 1.

Data preprocessing timeline

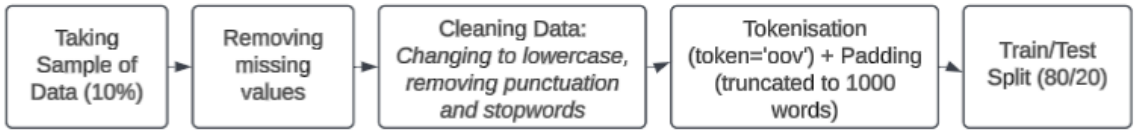


Table 1.

Before and after data preprocessing

Example	Original	Cleaned (lowercase, no punctuation + stopwords)	Tokenised + Padded
1	"over the years this movie has ignorantly used over and over as a tool..."	"years movie ignorantly used tool..."	50, 8, 1, 66, 884, 961, 163, 6722..... 0 0 0 0 0 0 0 0 0
2	"The advice he gives to children in this book..."	"advice gives children book..."	717, 376, 313, 2, 1293.....536 201 154 259
3	"Great album from a wonderful composer and a great band.."	"great album wonderful composer..."	7, 19, 168, 5711, 7... 0 0 0]
4	"When Wulfgar is the most interesting character in a book..."	"wulfgar interesting character book..."	1, 139, 190, 2, 296, 2687..... 0 0 0 0
5	"I usually like Belkin products. I think they have good products..."	"usually like belkin products think good products..."	534, 4, 5665, 511, 30, 5, 511,....., 0, 0, 0, 0, 0

Tokenisation converts words into a unique number using a vocabulary index – models can process text as numerical input. Padding ensures all the sequences are the same length for training, truncating reviews to 1000 words. Those less than 1000 are filled with 0's.

Model Building

Table 2 summarises the models developed. A total of 20 models were processed, covering baseline architectures, individual improvements and combined enhancements. Improvements were tested individually such as increasing embedding size, adding dense layers and changing optimisers, to isolate the individual impact of each. These were then combined into a final version to investigate their collective effect on model performance.

Table 2.

Summary of model experimentation

	Dense Artificial Neural Network (ANN)	Recurrent Neural Network (RNN)	Long-Short Term Memory (LSTM)	Convolutional Neural Network (CNN)
Version 1	Baseline Model (no improvements)			
Version 2	Baseline + Increase Embedding			
Version 3	Baseline + More Dense Layers			
Version 4	Baseline + Adam Optimizer			
Version 5	Enhanced model (all improvements combined)			

Parameters and tuning

Using CNN as an example, the following Keras-style block visuals illustrate each enhancement alongside its key parameters. Structural parameters and tuning strategies were kept consistent across all 20 models to avoid introducing any bias to model performance. The only variation was the replacement of the convolutional layer with either a simple RNN, LSTM or a dense layer (for ANN), dependent on the deep learning model being tested, see Table 3 and 4.

Table 3.

Purpose of baseline parameters

Baseline Parameter	Purpose
1000x64 Embedding	Takes up to 1000 words per review and converts it into a list of 64 numbers that describe the meaning of the word
0.5 dropout	Randomly drops 50% of neurons during training to reduce overfitting
RMSprop optimiser	Stabilize and speed up training
Output layer (Dense 1: Activation – sigmoid)	Produces probability the sentiment is between 0 and 1

Table 4.

Purpose of enhancements

Enhanced Parameter	Purpose
1000x128 Embedding	Doubles size of vector to better capture word meaning – 128 list of numbers produced now
Adam Optimiser	More adaptive and faster convergence than RMSprop
Dense 64: Relu	Adds learnable layers to extract more complex patterns
Dense 32: Relu	Further refines features, betters the depth of learning
Dropout between dense layers	Reduces overfitting more effectively by regularising the added complexity

Figure 2a.

Baseline Model

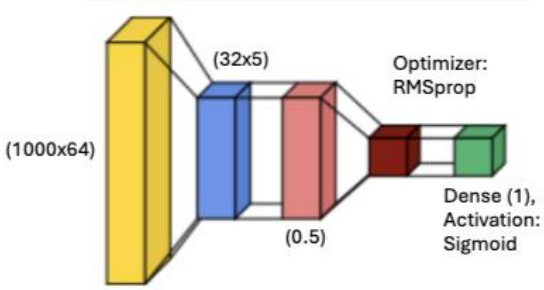


Figure 2b.

Baseline + Increased embedding

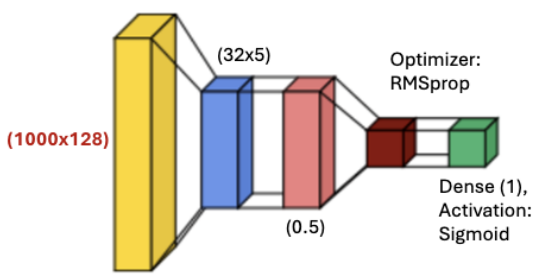


Figure 2c.

Baseline + Increased dense layers

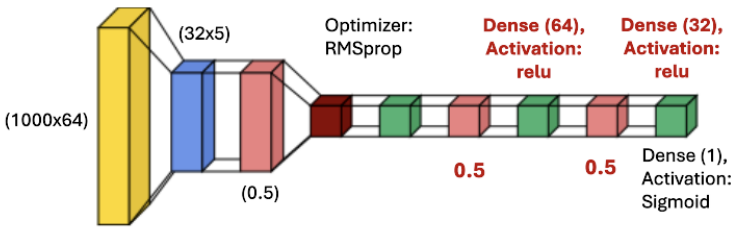


Figure 2d.

Baseline + Adam Optimiser

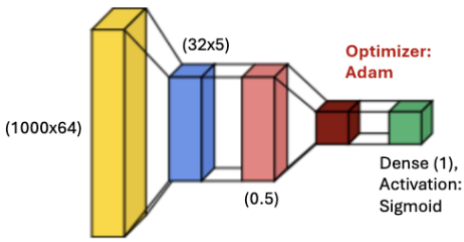
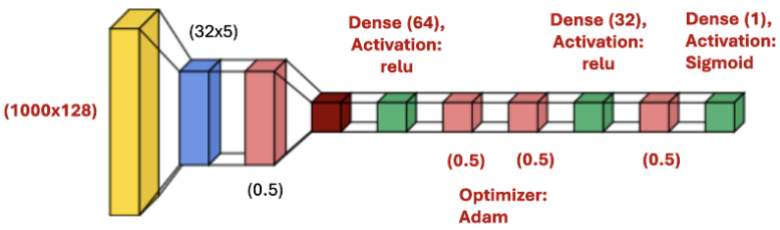


Figure 2e.

Enhanced model



The enhanced model explores the cumulative impact of all tuning efforts. We expect it to perform the best, leveraging deeper feature learning, richer word embeddings and improved learning rates.

Further information relevant to the model building and evaluation is summarised in the table below.

Parameter	Evaluation metrics
5 epochs, batch size 64 – due to CPU limitations	• Accuracy (% of correct predictions) • Training time • Precision (% of predicted positives that are correct) • Recall (% of actual positives correctly identified) • F1-score (balance of precision and recall) • ***Overfit gap*** – we calculated this because it is more practical than including 20 loss/accuracy plots.
Binary Crossentropy for loss function	
Validation Split – 10,000 samples sliced from training data	

The overfit gap calculates the difference between the training loss/validation loss of the final epochs (seen in the red dot in Figure 3):

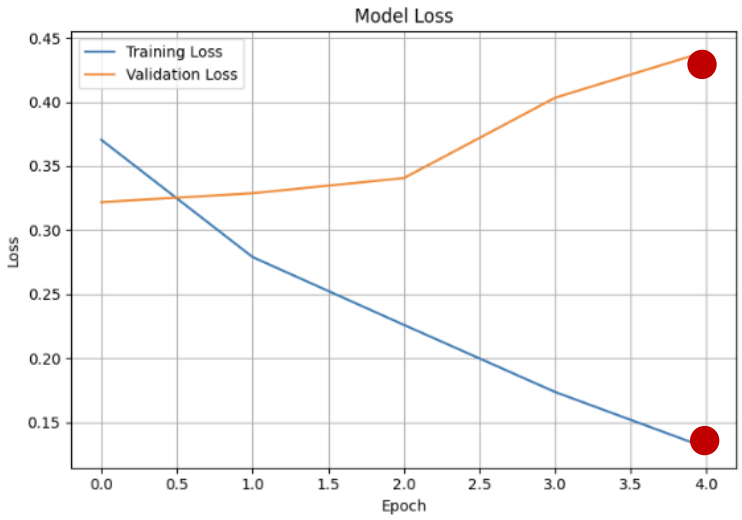
$$Overfit/Underfit\ Gap = Validation\ Loss\ (last\ epoch) - Training\ Loss\ (last\ epoch)$$

The overfit gap is:

- Likely underfitting if < 0
- Good generalisation if ≈ 0
- Likely overfitting if > 0

Figure 3

Example of training/validation loss – from enhanced LSTM model



Example using Figure 3:

$$Overfit/Underfit\ Gap$$
$$= \ 0.43 - 0.11$$
$$= \mathbf{0.32}\ (slight\ overfitting)$$

Results

Individual Model Comparison

Dense ANN

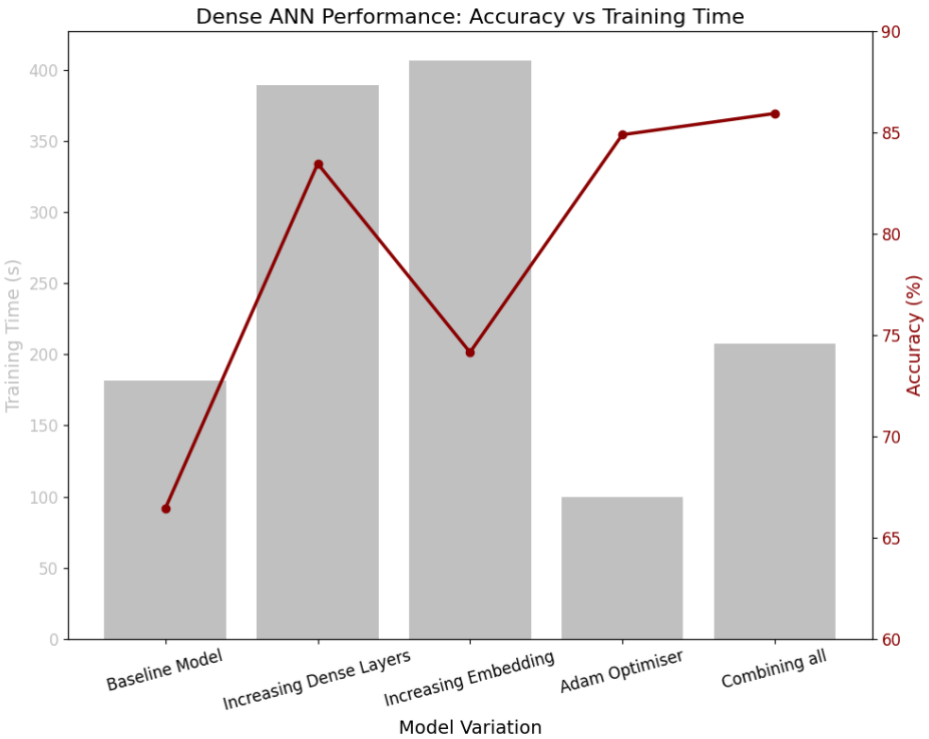
Table 5a.

Key evaluation metrics

	Accuracy	Training Time (s)	Overfit Gap
Baseline Model	**66.44%**	181.64	0.00
Increasing Dense Layers	83.47%	389.89	0.00
Increasing Embedding	74.15%	**407.02**	0.00
Adam Optimiser	84.90%	**99.92**	0.01
Enhanced model	**85.95%**	207.57	0.00

Figure 4.

Bar chart comparing accuracy and training time per model version



Baseline model produced the worst accuracy, while enhanced model produced the best. Training time increases with higher embedding, but quicker for an Adam optimiser.

Table 5b:

Further evaluation metrics

	Precision (Positive)	Precision (Negative)	Recall (Positive)	Recall (Negative)	F1-Score (Positive)	F1-Score (Negative)
Baseline ANN	0.54	0.68	0.88	0.26	0.67	0.37
Improved ANN	0.88	0.83	0.84	0.87	0.86	0.85

Individual enhancements to the model increased accuracy, though, the enhanced model combining all improvements produced the best results, as expected. All models showed strong generalisation, given minimal overfit gaps. Significant improvement to precision, recall and f1-score indicates better classification of true positives and stronger balance between precision and recall.

RNN

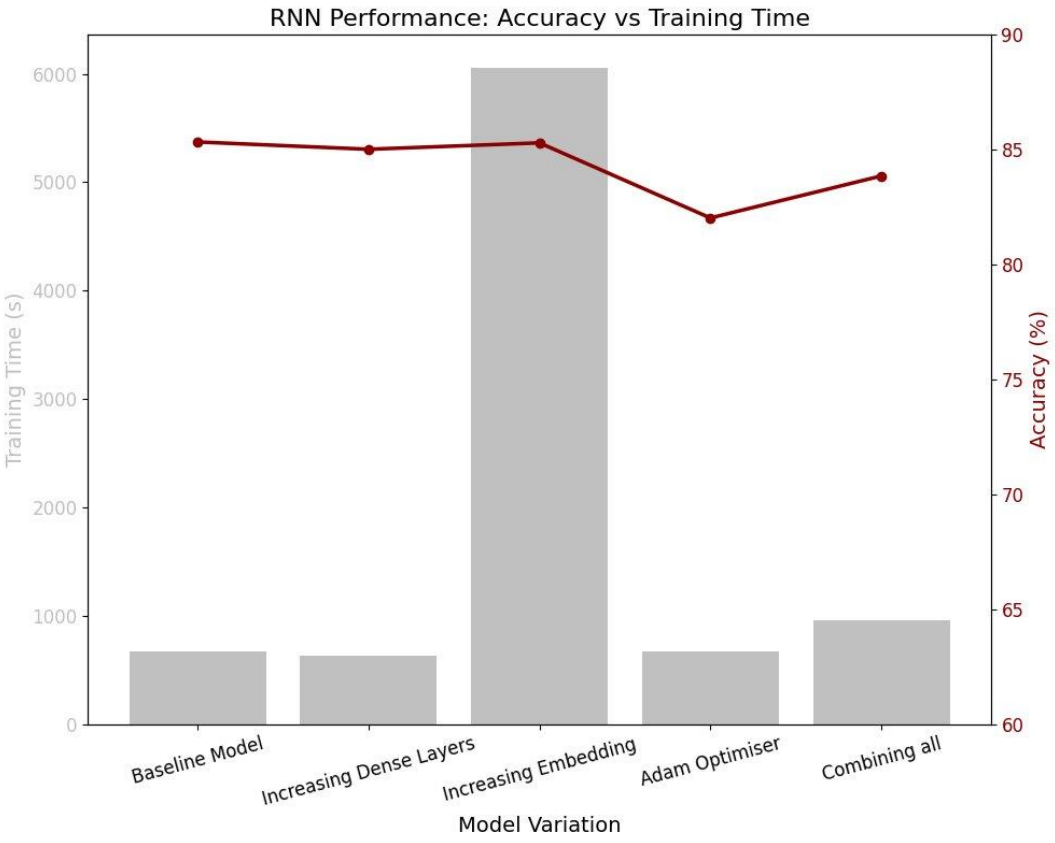
Table 6a.

Key evaluation metrics

	Accuracy	Training Time (s)	Overfit Gap
Baseline Model	**85.34%**	672.75	**0.11**
Increasing Dense Layers	85.02%	**634.34**	0.15
Increasing Embedding	85.30%	**6055.99**	0.19
Adam Optimiser	**82.03%**	669.54	**0.39**
Enhanced model	83.86%	958.59	0.25

Figure 5.

Bar chart comparing accuracy and training time per model version



Baseline model outperformed all models with architectural improvements, only marginally. Increasing embedding, again, undertook longest training time. Model overfitted with each improvement, substantially most for adapting the Adam optimiser

Table 6b:

Further evaluation metrics

	Precision (Positive)	Precision (Negative)	Recall (Positive)	Recall (Negative)	F1-Score (Positive)	F1-Score (Negative)
Baseline RNN	0.83	0.88	0.89	0.82	0.86	0.85
Enhanced RNN	0.83	0.88	0.89	0.82	0.86	0.85

Both Baseline and Enhanced model performed identically across all metrics, achieving high precision and recall; few false positives and most instances were correctly predicted, respectively. F1-score confirms great balance between precision and recall for both classes. The improvements made to the enhanced model did not significantly impact model performance, suggesting the baseline was already optimal.

LSTM

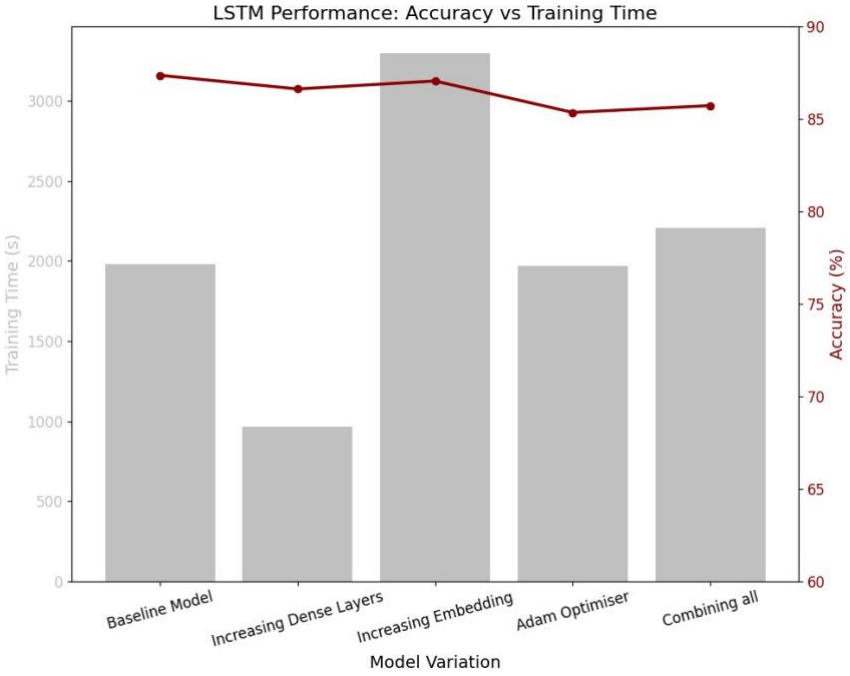
Table 7a.

Key evaluation metrics

	Accuracy	Training Time (s)	Overfit Gap
Baseline Model	**87.37%**	1978.69	**0.08**
Increasing Dense Layers	86.63%	**966.60**	0.11
Increasing Embedding	87.06%	**3300.57**	0.10
Adam Optimiser	**85.36%**	1971.41	0.23
Enhanced model	85.73%	2206.71	**0.31**

Figure 6.

Bar chart comparing accuracy and training time per model version



Baseline model performed the best, while changing to Adam optimiser worsened accuracy. Again, increasing embedding featured longer training time. Enhanced model overfitted data the most.

Table 7b:

Further evaluation metrics

	Precision (Positive)	Precision (Negative)	Recall (Positive)	Recall (Negative)	F1-Score (Positive)	F1-Score (Negative)
Baseline LSTM	0.83	0.88	0.89	0.82	0.86	0.85
Enhanced LSTM	0.83	0.88	0.89	0.82	0.86	0.85

The same interpretation from RNN performance is applied to this in terms of metrics, the baseline model was already optimal.

CNN

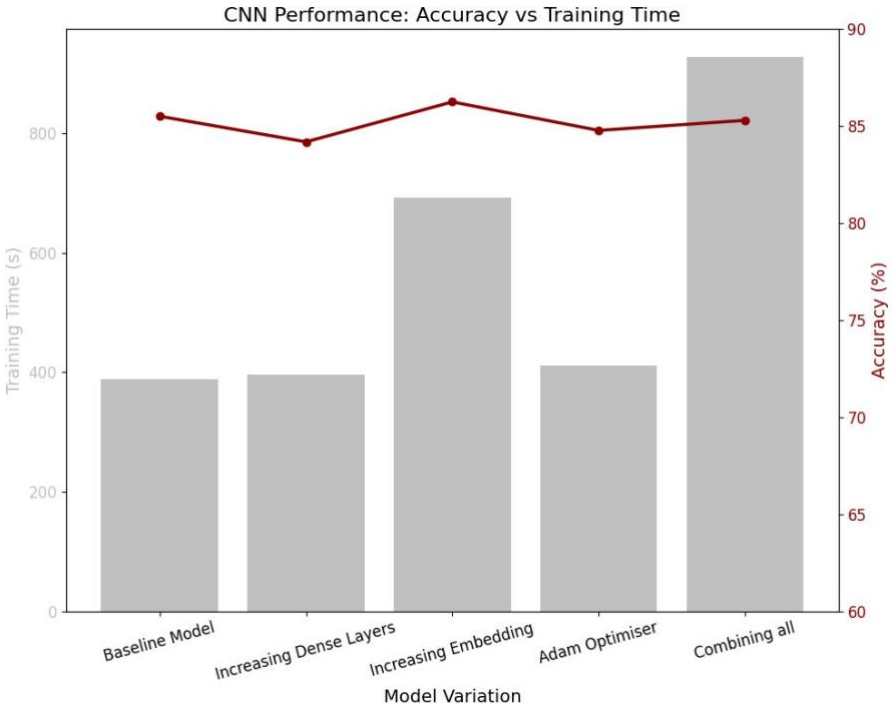
Table 8a.

Key evaluation metrics

	Accuracy	Training Time (s)	Overfit Gap
Baseline Model	85.51%	**389.06**	**0.13**
Increasing Dense Layers	**84.19%**	396.17	0.41
Increasing Embedding	**86.25%**	**693.71**	0.18
Adam Optimiser	84.78%	411.30	0.42
Enhanced model	85.30%	928.58	**0.62**

Figure 7.

Bar chart comparing accuracy and training time per model version



Increasing embedding slightly increased accuracy but showcased longest training time while increasing dense layers showed most decline in accuracy. Very high overfitting score for the enhanced model.

Table 8b:

Further evaluation metrics

	Precision (Positive)	Precision (Negative)	Recall (Positive)	Recall (Negative)	F1-Score (Positive)	F1-Score (Negative)
Baseline CNN	0.83	0.88	0.82	0.89	0.85	0.86
Enhanced CNN	0.65	0.50	0.02	0.99	0.04	0.66

The enhanced CNN model performed significantly worse for identifying positive sentiment across all metrics, particularly recall and F1-score. Both values dropped heavily, reflecting the overfitting of this model.

Discussion 1

Overall, architectural enhancements failed to yield meaningful improvements in RNN, LSTM or CNN models. This implies the baseline versions of each deep learning model were already well-tuned for the task. While increasing embedding is expected to capture richer word embeddings, it simply added noise to the model, leading to increased training time and higher overfitting gaps. This outcome was unexpected and highlighted a trade-off between complexity and performance. Adding dense layers made the model deeper but not smarter, while the Adam optimiser had no room to improve due to optimal performance from RMSProp optimiser and diverging too quickly – such enhancements didn't improve performance where models were already highly effective, such as in LSTM and RNN. The strong generalisation of their baselines indicates less need for tuning. However, ANN benefitted significantly from all enhancements, producing its best accuracy when all changes were combined. Therefore, simpler architectures (ANN) may benefit from targeted tuning more than the other models; hence, careful selection of architecture-specific improvements is critical, otherwise unnecessary complexity increases the risk of overfitting and computational cost without accuracy improvement.

Further Improvement

To address the overfitting observed in the previous models, we incorporated pretrained GloVe embeddings (Global Vectors for Word Representation). This technique turns words into numerical vectors based on their meaning, learned from word occurrence patterns in large text datasets. These models were evaluated using the same metrics previously, with a vocabulary size limited to 1000 words and embedding dimension of 100, these parameters were selected as they optimise GloVe performance while

minimising the risk of overfitting. GloVe was applied to the enhanced version of each model, since these tended to overfit the most.

Table 9.

GloVe-embedding evaluation metrics

	Precision (Positive)	Precision (Negative)	Recall (Positive)	Recall (Negative)	F1-Score (Positive)	F1-Score (Negative)	Overfit Gap	Accuracy
GloVe ANN	0.77	0.77	0.78	0.76	0.77	0.77	0.00	76.93%
GloVe RNN	0.79	0.68	0.60	0.84	0.68	0.75	0.01	76.56%
GloVe LSTM	0.84	0.88	0.89	0.83	0.86	0.85	0.04	85.67%
GloVe CNN	0.78	0.88	0.89	0.75	0.84	0.81	0.13	82.30%

Glove LSTM had the best metrics overall, highlighting strong balanced predictions. Glove CNN also performed well but overfitted the most. GloVe ANN and RNN underperformed, predicting fewer true positives and negatives.

Overall Model Comparison

Figure 8a.

Bar chart comparing precision scores across all models

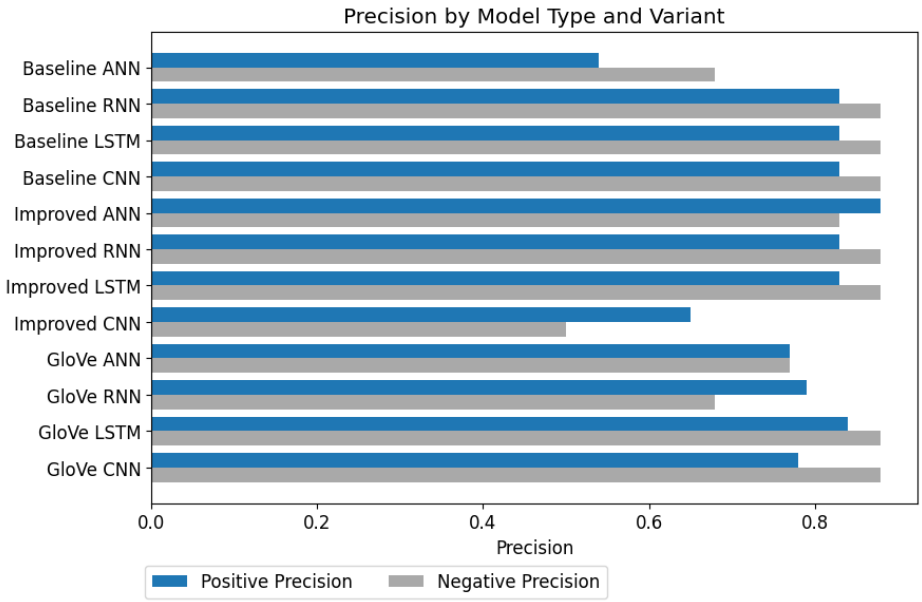


Figure 8b.

Precision scores for all models

	Precision (Positive)	Precision (Negative)
Baseline ANN	0.54	0.68
Baseline RNN	0.83	0.88
Baseline LSTM	0.83	0.88
Baseline CNN	0.83	0.88
Enhanced ANN	0.88	0.83
Enhanced RNN	0.83	0.88
Enhanced LSTM	0.83	0.88
Enhanced CNN	0.65	0.50
GloVe ANN	0.77	0.77
GloVe RNN	0.79	0.68
GloVe LSTM	0.84	0.88
GloVe CNN	0.78	0.88

LSTM consistently had highest precision across all its models on average, reliably predicting true positives. RNN performed well as a baseline and enhanced model, also reliably predicting true positives. ANN showed greatest precision gain. GloVe boosted precision in LSTM but not RNN.

Figure 9a.

Bar chart comparing recall scores across all models

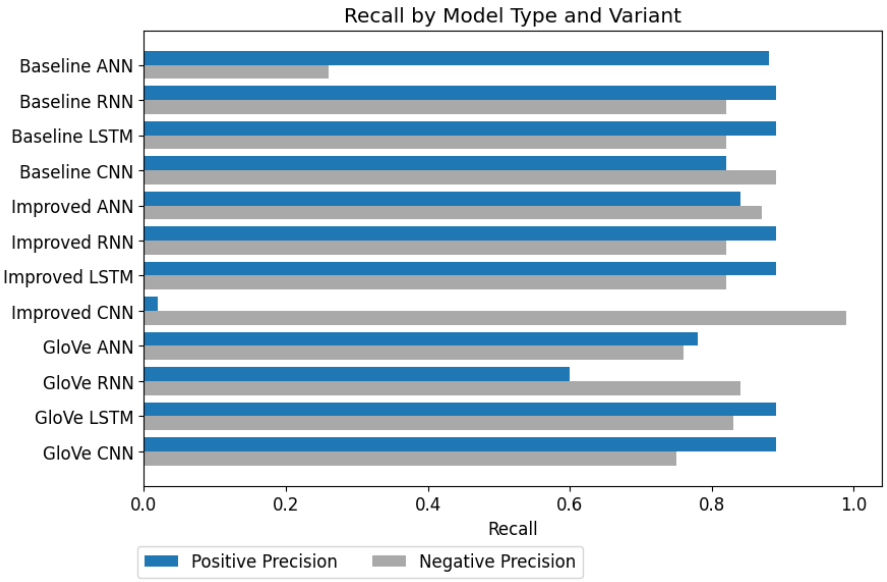


Figure 9b.

Recall scores for all models

	Recall (Positive)	Recall (Negative)
Baseline ANN	0.88	0.26
Baseline RNN	0.89	0.82
Baseline LSTM	0.89	0.82
Baseline CNN	0.82	0.89
Enhanced ANN	0.84	0.87
Enhanced RNN	0.89	0.82
Enhanced LSTM	0.89	0.82
Enhanced CNN	0.02	0.99
GloVe ANN	0.78	0.76
GloVe RNN	0.60	0.84
GloVe LSTM	0.89	0.83
GloVe CNN	0.89	0.75

LSTM also achieved consistently the best recall, identifying a large proportion of true positives. RNN performed well in its baseline and improved models. ANN gained the most improvement. Enhanced CNN recalled negatives well but failed to detect positives.

Figure 10a.

Bar chart comparing F1-scores across all models

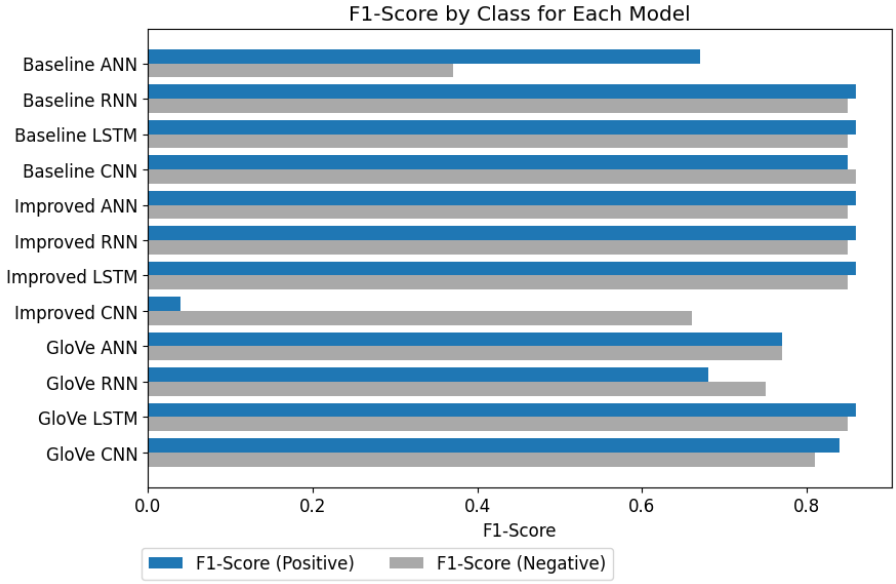


Figure 10b.

F1-scores for all models

	F1-Score (Positive)	F1-Score (Negative)
Baseline ANN	0.67	0.37
Baseline RNN	0.86	0.85
Baseline LSTM	0.86	0.85
Baseline CNN	0.85	0.86
Enhanced ANN	0.86	0.85
Enhanced RNN	0.86	0.85
Enhanced LSTM	0.86	0.85
Enhanced CNN	0.04	0.66
GloVe ANN	0.77	0.77
GloVe RNN	0.68	0.75
GloVe LSTM	0.86	0.85
GloVe CNN	0.84	0.81

LSTM was the most consistent in high F1-scores across all models for both classes, showing strongest balance between precision and recall. RNN maintained great balance for its baseline and enhanced version but declined when paired with GloVe. Enhanced CNN underperformed for positive sentiment classification. ANN showed greatest improvement post-enhancement.

Figure 11a.

Bar chart comparing overfit gap across all models

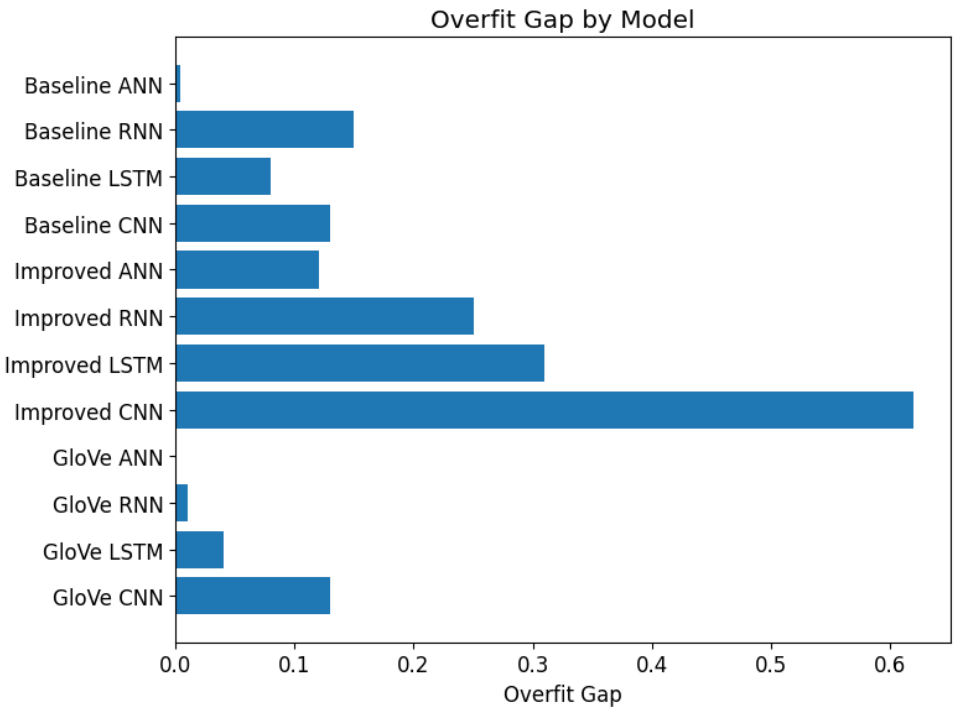


Figure 11b.

Overfit gap for all models

	Overfit Gap
Baseline ANN	0.00
Baseline RNN	0.15
Baseline LSTM	0.08
Baseline CNN	0.13
Improved ANN	0.12
Improved RNN	0.25
Improved LSTM	0.31
Improved CNN	0.62
GloVe ANN	0.00
GloVe RNN	0.01
GloVe LSTM	0.04
GloVe CNN	0.13

For all deep learning models, enhancing the model increased overfitting, but the GloVe model reduced this overfitting. Enhanced CNN overfitted the most, while Baseline and GloVe ANN overfit the least.

Figure 12a.

Bar chart comparing precision scores across all models

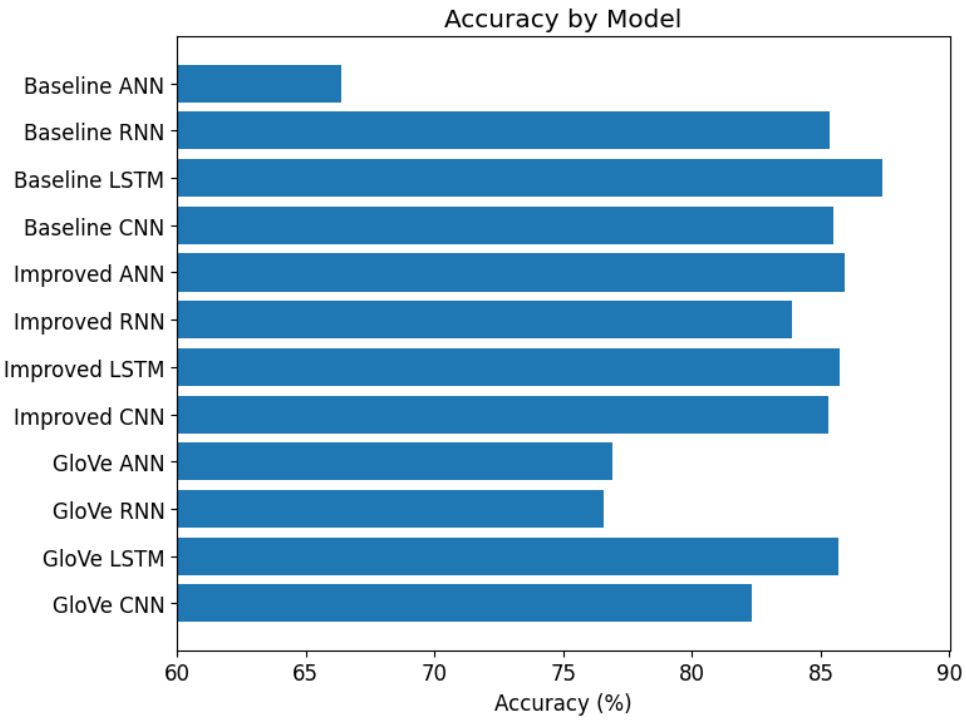


Figure 12b.

Accuracy for all models

	Accuracy
Baseline ANN	66.4%
Baseline RNN	85.34%
Baseline LSTM	87.37%
Baseline CNN	85.51%
Improved ANN	85.95%
Improved RNN	83.86%
Improved LSTM	85.73%
Improved CNN	85.30%
GloVe ANN	76.93%
GloVe RNN	76.56%
GloVe LSTM	85.67%
GloVe CNN	82.30%

High accuracy was consistent for LSTM and CNN across all three versions, RNN performed well for its baseline and enhanced version, but accuracy declined post-GloVe. ANN showed greatest improvement overall.

Discussion 2

LSTM consistently outperformed other deep learning models across all metrics, achieving the highest precision, recall, F1-score and accuracy for baseline, enhanced and GloVe models. It predicted true positives and maintained strong generalisation with minimal overfitting. RNN performed strongly across all 3 versions, finding good balance between precision and recall, but performance suffered when paired with GloVe embeddings. CNN showed high accuracy and balance in baseline and GloVe models, but it significantly overfitted and failed to identify positive sentiment reliably in its enhanced model. ANN showed greatest improvement post-enhancement, highlighting the value of architectural improvements even when paired with GloVe models.

GloVe reduced overfitting across all models but did not consistently boost accuracy, with ANN, RNN and CNN showing a large decline. This highlights a trade-off between overfitting and accuracy, as each GloVe model reduced overfitting to near perfect generalisation, at the expense of lower accuracy.

Conclusion

To address our earlier research questions:

1. ***Which neural network architecture performs best for binary sentiment classification?***
   LSTM performs best overall as a baseline model, improved model and pre-trained GloVe model - marginally outperforming RNN and CNN across all model versions.

2. **How do architectural improvements impact performance?**
   Impact varies by model:
   - ANN improves dramatically, RNN's performance worsened, LSTM and CNN stayed stable but CNN overfit significantly more.
   - Improvements like larger embeddings and extra dense layers increased overfitting and training time

3. **Which model is the most computationally expensive?**
   All LSTM models and RNN with increased embedding had the longest training times.

4. **Can pre-trained GloVe embeddings improve performance and reduce overfitting?**
   While GloVe did not always boost accuracy (like it did for ANN), it consistently produced lower overfit gaps, near perfect generalisation.

Therefore, businesses should utilise improved LSTM as it offers high accuracy and generalises well. With GloVe, LSTM becomes even more stable and generalises even better – more scalable and dependable for large sentiment tasks. Since a pre-trained model was the most valuable, future research should focus on improving the model architecture of this as well as repeating this experiment on a GPU with greater epochs and batch size or using other pre-training models such as Bidirectional LSTM or a Bert-Transformer.

References

Juraev, F., Abuhamad, M., Woo, S. S., Thiruvathukal, G. K., & Abuhmed, T. (2024). Impact of architectural modifications on deep learning adversarial robustness. *arXiv preprint arXiv:2405.01934*.

Olalere, A. (2024). Implementation and Comparison of Deep Learning with Naïve Bayes for Language Processing.

Pa, M., & Kazemi, A. (2022). DL based analysis of movie reviews. *arXiv preprint arXiv:2210.10726*.

Pang, B., & Lee, L. (2008). Opinion mining and sentiment analysis. Foundations and Trends in Information Retrieval, 2(1–2), 1–135. https://doi.org/10.1561/1500000011

Pennington, J. (n.d.). *GLOVE: Global Vectors for Word Representation*. https://nlp.stanford.edu/projects/glove/