IT494
Big Data Processing

Project Report

Group Members

Vishrut Shah - 202001039
Dev Malkan - 202001040
Siddharth Makhija - 202001087

Topic : kNN-IS
An Iterative Spark-based design of the k-Nearest Neighbors classifier for big data

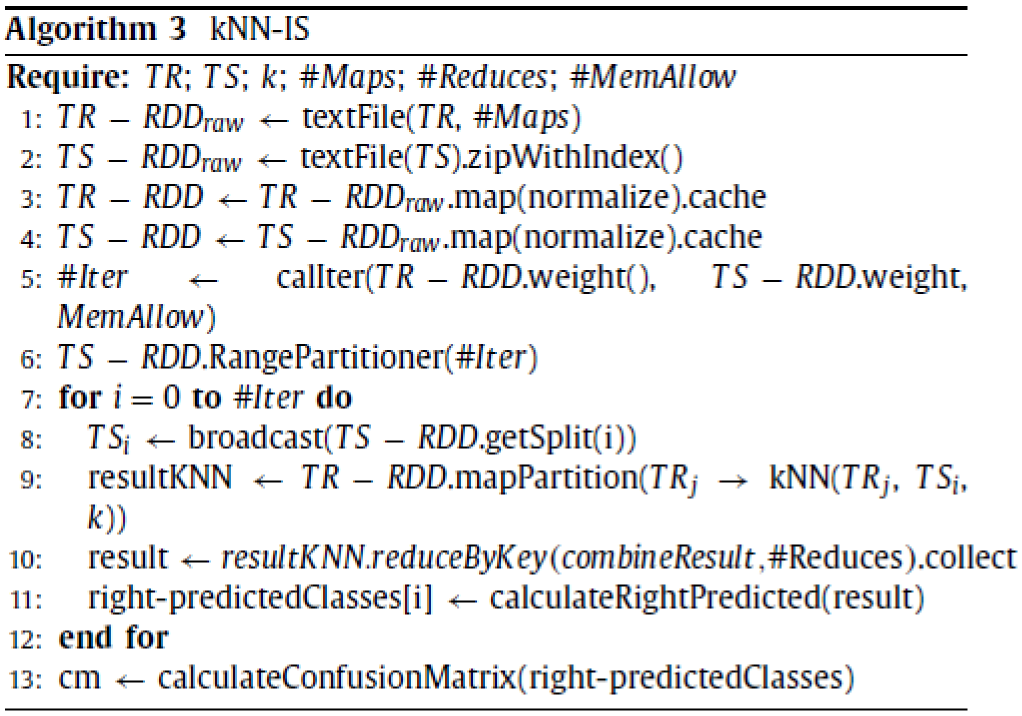
Problem

The k-Nearest Neighbours classifier is a very popular method in data mining as it is easy to implement and still provides good results. But application of this model in big data is not feasible as it takes a huge amount of memory and long runtime.

Solution

A novel Spark-based method for performing an accurate k-nearest neighbour classification is given in the research paper to solve the above problem. This method took advantage of Spark’s in-memory operations to categorise a large number of unknown instances using large training data. The k-nearest neighbours in various training data splits are calculated during the map phase. Subsequently, the list acquired in the map phase is processed by numerous reducers to determine the definitive neighbours. This proposal's main focus is on managing the test set and, where practical, storing it in memory. If not, it is divided into a minimal number of chunks, with a MapReduce applied for each chunk, utilising Spark's caching capabilities to reuse the previously divided training data.

Algorithms Implemented

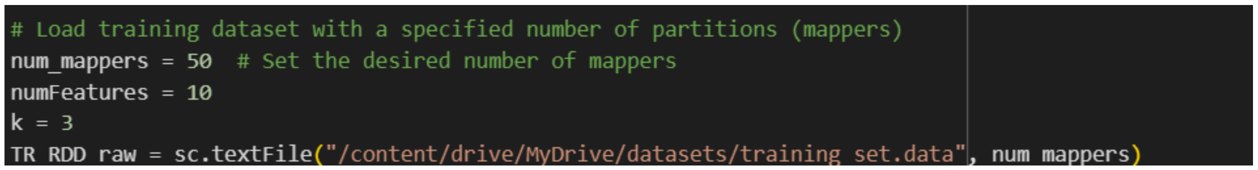


Algorithm 3 kNN-IS

```
Require: $TR$; $TS$; $k$; $\#Maps$; $\#Reduces$; $\#MemAllow$
1: $TR-RDD_{raw}$ ← textFile($TR$, $\#Maps$)
2: $TS-RDD_{raw}$ ← textFile($TS$).zipWithIndex()
3: $TR-RDD$ ← $TR-RDD_{raw}$.map(normalize).cache
4: $TS-RDD$ ← $TS-RDD_{raw}$.map(normalize).cache
5: $\#Iter$ ← callter($TR-RDD$.weight(), $TS-RDD$.weight, $MemAllow$)
6: $TS-RDD$.RangePartitioner($\#Iter$)
7: for $i = 0$ to $\#Iter$ do
8:     $TS_i$ ← broadcast($TS-RDD$.getSplit(i))
9:     resultKNN ← $TR-RDD$.mapPartition($TR_j \rightarrow$ kNN($TR_j$, $TS_i$, $k$))
10:    result ← $resultKNN.reduceByKey(combineResult$,#Reduces).collect
11:    right-predictedClasses[i] ← calculateRightPredicted(result)
12: end for
13: cm ← calculateConfusionMatrix(right-predictedClasses)
```

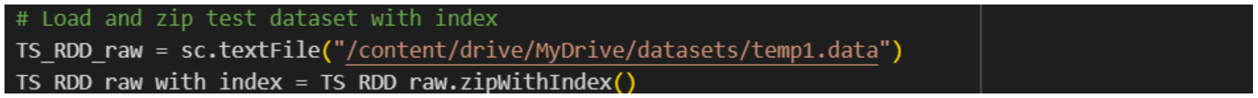
We implemented the algorithm given in the above image in python using its pyspark library.

1) RDD is created for the training dataset by specifying number of maps.



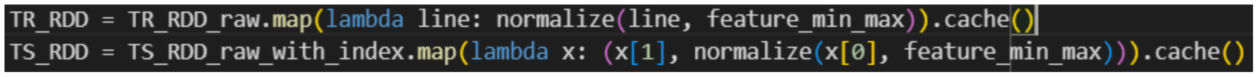
```
# Load training dataset with a specified number of partitions (mappers)
num_mappers = 50  # Set the desired number of mappers
numFeatures = 10
k = 3
TR_RDD_raw = sc.textFile("/content/drive/MyDrive/datasets/training_set.data", num_mappers)
```

2) RDD is created for testing dataset along with a key for every single instance of the dataset using zipWithIndex() function.



```
# Load and zip test dataset with index
TS_RDD_raw = sc.textFile("/content/drive/MyDrive/datasets/temp1.data")
TS_RDD_raw_with_index = TS_RDD_raw.zipWithIndex()
```

3) As Euclidean distance is required in the kNN classifier, we need to normalize the data between the range 0-1. Additionally datasets are cached so that they can be reused in the future.



```
TR_RDD = TR_RDD_raw.map(lambda line: normalize(line, feature_min_max)).cache()
TS_RDD = TS_RDD_raw_with_index.map(lambda x: (x[1], normalize(x[0], feature_min_max))).cache()
```

4) Then we calculate the number of iterations required for the given testing dataset by keeping the memory allowance into consideration.

```
def calIter(tr_weight, ts_weight, mem_allow):
    numIterations = 0;
    weightTrain = (8 * tr_weight * numFeatures) / (num_mappers * 1024.0 * 1024.0)
    weightTest = (8 * ts_weight * numFeatures) / (1024.0 * 1024.0)
    if (weightTrain + weightTest < mem_allow * 1024.0):
          numIterations = 1
    else:
      if (weightTrain >= mem_allow * 1024.0):
        print("Train weight bigger than lim-task. Abort")
        sys.exit(1)

      numIterations = int((1 + (weightTest / ((mem_allow * 1024.0) - weightTrain))))
    return numIterations
```

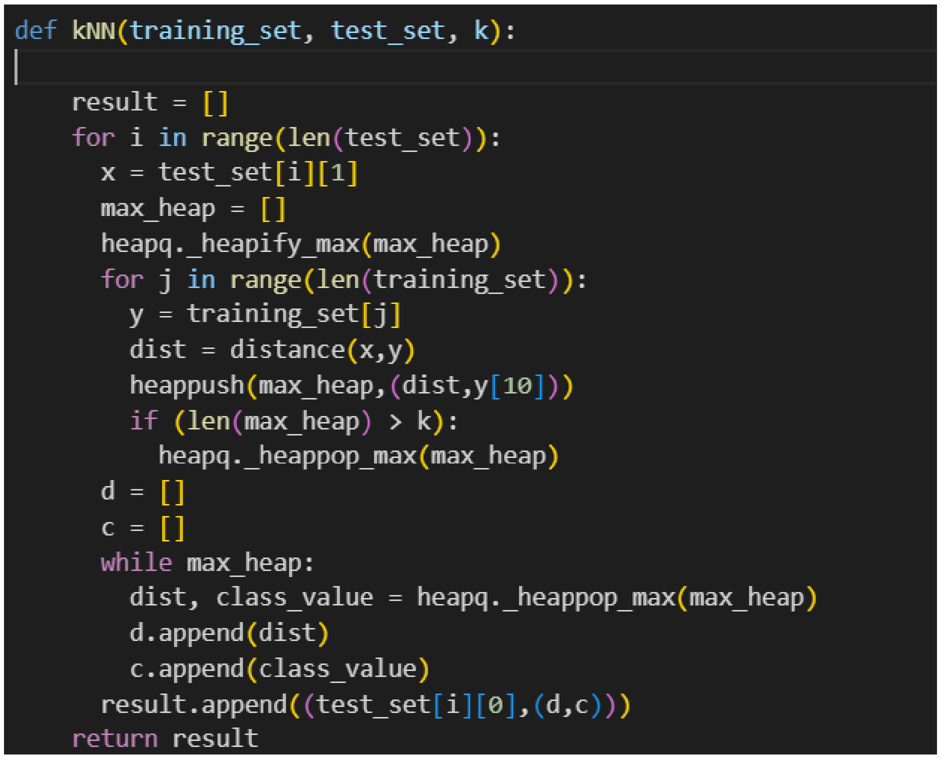
5) partitionBy() function is used to partition the testing dataset into subsets equal to the number of iterations calculated in the previous step.

```
# Range partitioning for TS_RDD
TS_RDD.partitionBy(iterations,partitionFunc=range)
all_partitions = TS_RDD.glom().collect()
```

6) After that, code enters into the for loop in which the subset of that iteration is broadcasted into the main memory using inbuilt broadcast() function.

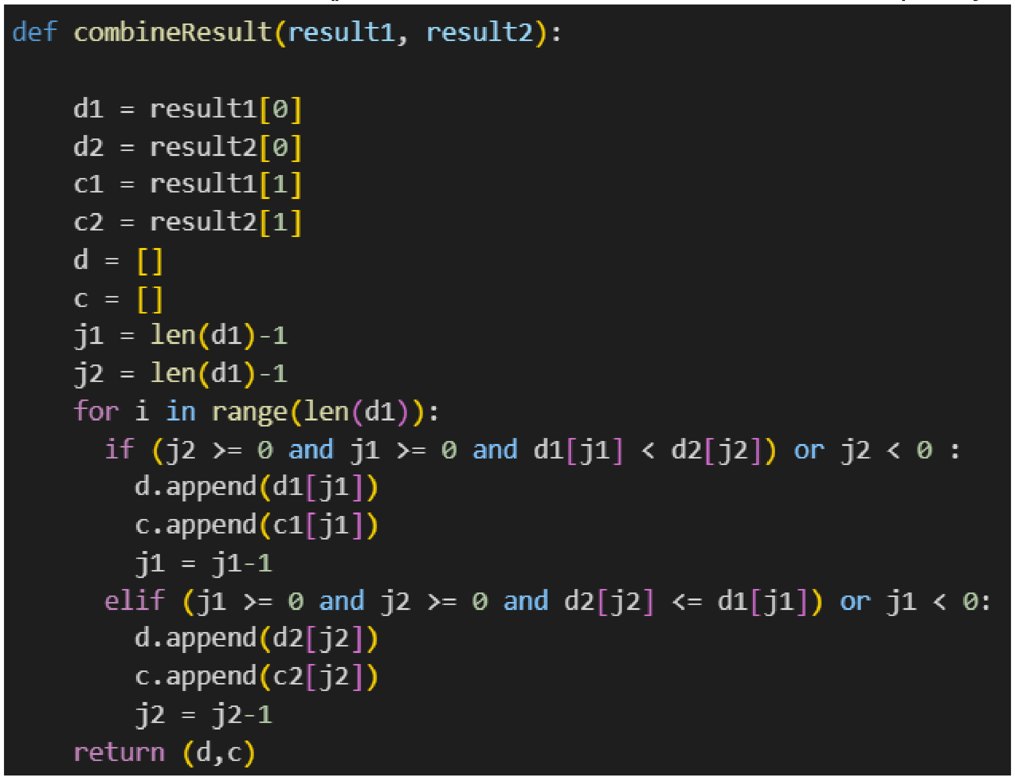
```
for i in range(len(all_partitions)):
    # Broadcast TS_i
    TS_i = spark.sparkContext.broadcast(all_partitions[i])

    # MapPartition to perform kNN
    resultKNN = TR_RDD.mapPartitions(lambda tr_partition: kNN(list(tr_partition), TS_i.value, k))

    # ReduceByKey to combine results
    result = resultKNN.reduceByKey(lambda result1,result2: combineResult(result1,result2)).collect()

    # Calculate right predicted classes for this iteration
    right_predicted_classes.append(calculateRightPredicted(result,TS_i.value))
```

7) Next is the mapping phase in which k nearest neighbours are computed for each partition of training and testing dataset. It returns key-value pairs where the key is the number of the testing instance and the value is list of distance-class. This is done using a custom kNN function. Also the distance list contains k neighbours in sorted order so that the reduce phase becomes faster.



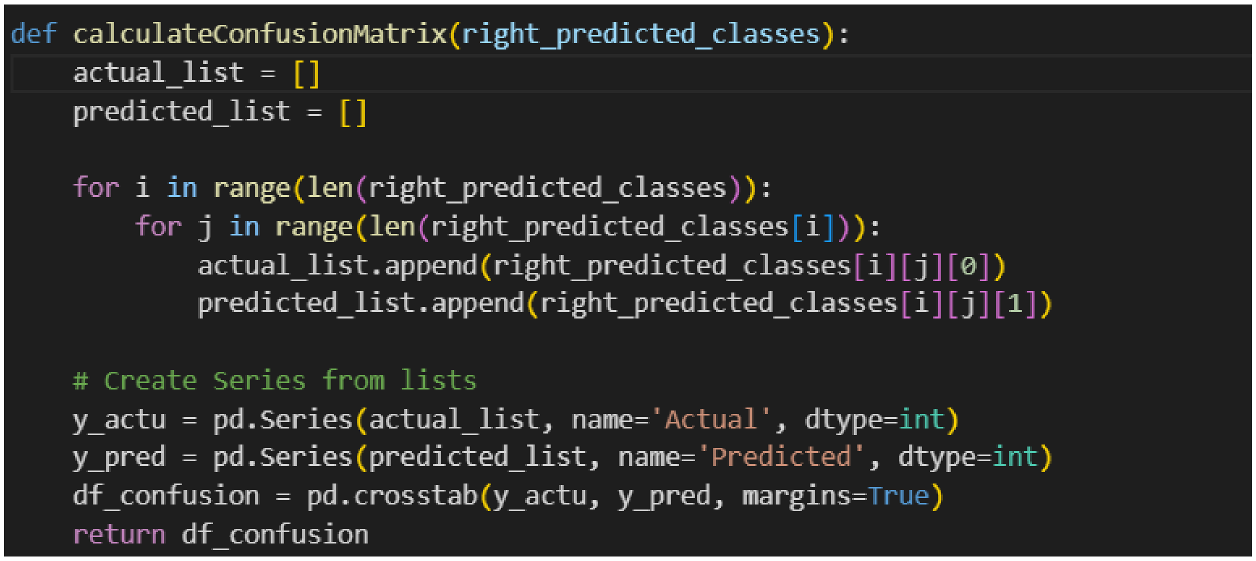
```python
def kNN(training_set, test_set, k):

    result = []
    for i in range(len(test_set)):
      x = test_set[i][1]
      max_heap = []
      heapq._heapify_max(max_heap)
      for j in range(len(training_set)):
        y = training_set[j]
        dist = distance(x,y)
        heappush(max_heap,(dist,y[10]))
        if (len(max_heap) > k):
          heapq._heappop_max(max_heap)
      d = []
      c = []
      while max_heap:
        dist, class_value = heapq._heappop_max(max_heap)
        d.append(dist)
        c.append(class_value)
      result.append((test_set[i][0],(d,c)))
    return result
```

8) Reduce phase outputs the final nearest k neighbours of each testing instance by comparing results received from each of the maps. This is also done using the custom combineResult() function which has the worst time complexity of O(k).



```python
def combineResult(result1, result2):

    d1 = result1[0]
    d2 = result2[0]
    c1 = result1[1]
    c2 = result2[1]
    d = []
    c = []
    j1 = len(d1)-1
    j2 = len(d1)-1
    for i in range(len(d1)):
      if (j2 >= 0 and j1 >= 0 and d1[j1] < d2[j2]) or j2 < 0 :
        d.append(d1[j1])
        c.append(c1[j1])
        j1 = j1-1
      elif (j1 >= 0 and j2 >= 0 and d2[j2] <= d1[j1]) or j1 < 0:
        d.append(d2[j2])
        c.append(c2[j2])
        j2 = j2-1
    return (d,c)
```

9) Finally all the results are used to make a confusion matrix which can help in calculating accuracy of this method.



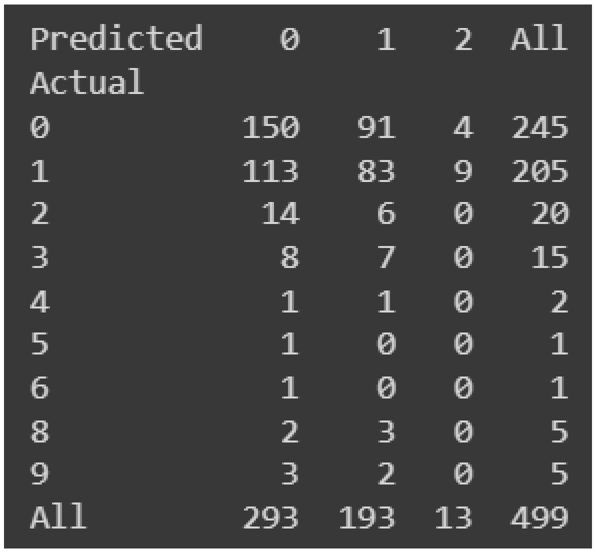
```
def calculateConfusionMatrix(right_predicted_classes):
    actual_list = []
    predicted_list = []

    for i in range(len(right_predicted_classes)):
        for j in range(len(right_predicted_classes[i])):
            actual_list.append(right_predicted_classes[i][j][0])
            predicted_list.append(right_predicted_classes[i][j][1])

    # Create Series from lists
    y_actu = pd.Series(actual_list, name='Actual', dtype=int)
    y_pred = pd.Series(predicted_list, name='Predicted', dtype=int)
    df_confusion = pd.crosstab(y_actu, y_pred, margins=True)
    return df_confusion
```

Testing

Dataset used: We used Poker-hand dataset from UCI Machine Learning repository

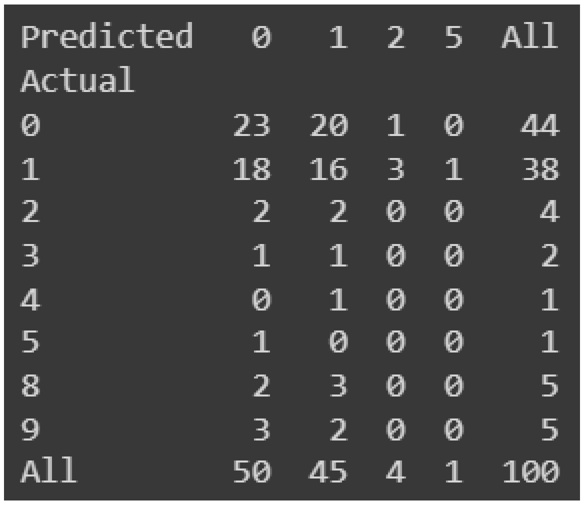
Results: Here are few confusion matrices generated on this dataset using our code

1) Training set - 25000 instances, Test set - 499 instances, k = 5



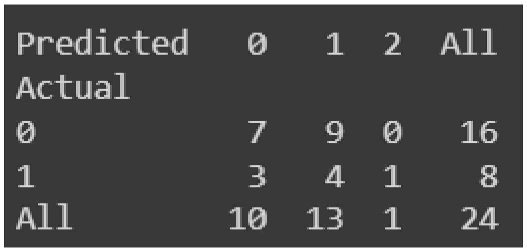
```
Predicted     0    1   2  All
Actual
0           150   91   4  245
1           113   83   9  205
2            14    6   0   20
3             8    7   0   15
4             1    1   0    2
5             1    0   0    1
6             1    0   0    1
8             2    3   0    5
9             3    2   0    5
All         293  193  13  499
```

2) Training set - 25000 instances, Test set - 100 instances, k = 1



```
Predicted   0   1  2  5  All
Actual
0          23  20  1  0   44
1          18  16  3  1   38
2           2   2  0  0    4
3           1   1  0  0    2
4           0   1  0  0    1
5           1   0  0  0    1
8           2   3  0  0    5
9           3   2  0  0    5
All        50  45  4  1  100
```

3) Training set - 2000 instances, Test set - 24 instances, k = 3



```
Predicted   0   1  2  All
Actual
0           7   9  0   16
1           3   4  1    8
All        10  13  1   24
```

References

- Maillo, Jesus, et al. "kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbors classifier for big data." Knowledge-Based Systems 117 (2017): 3-15.
- M. Lichman , UCI Machine Learning Repository, 2013
- https://archive.ics.uci.edu/
- https://github.com/JMailloH/kNN _ IS