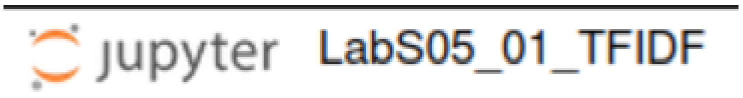
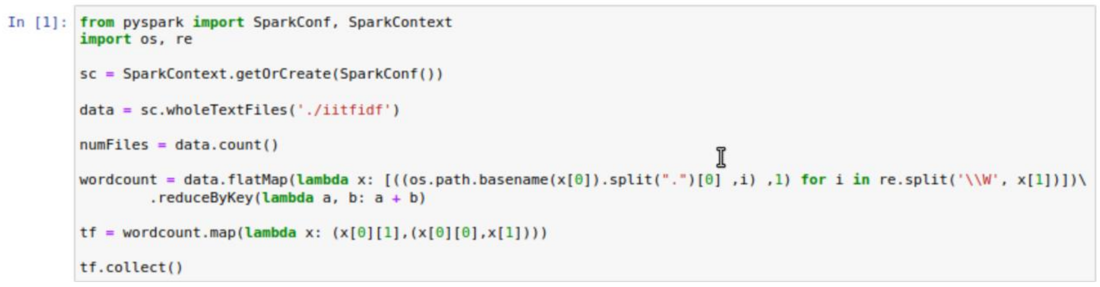
Basic

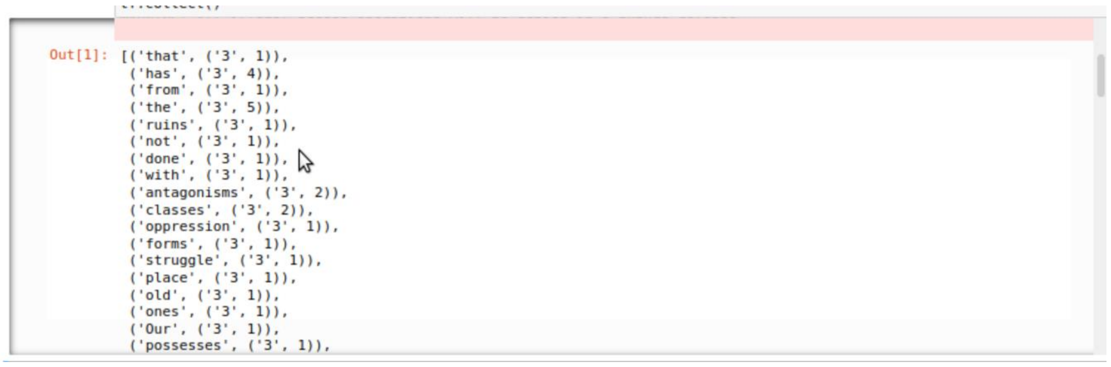


LabS05_01_TFIDF



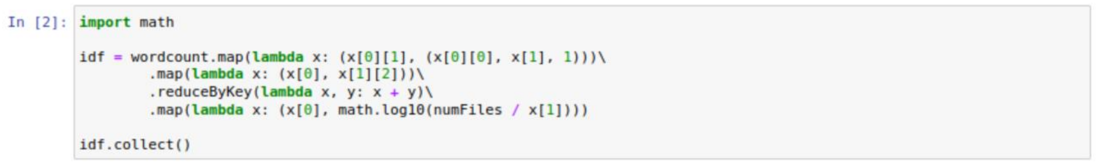
```
In [1]: from pyspark import SparkConf, SparkContext
        import os, re

        sc = SparkContext.getOrCreate(SparkConf())

        data = sc.wholeTextFiles('./iitfidf')

        numFiles = data.count()

        wordcount = data.flatMap(lambda x: [((os.path.basename(x[0]).split(".")[0] ,i) ,1) for i in re.split('\\W', x[1])])\
                .reduceByKey(lambda a, b: a + b)

        tf = wordcount.map(lambda x: (x[0][1],(x[0][0],x[1])))

        tf.collect()
```

Calculate the term frequency of the each word in documents



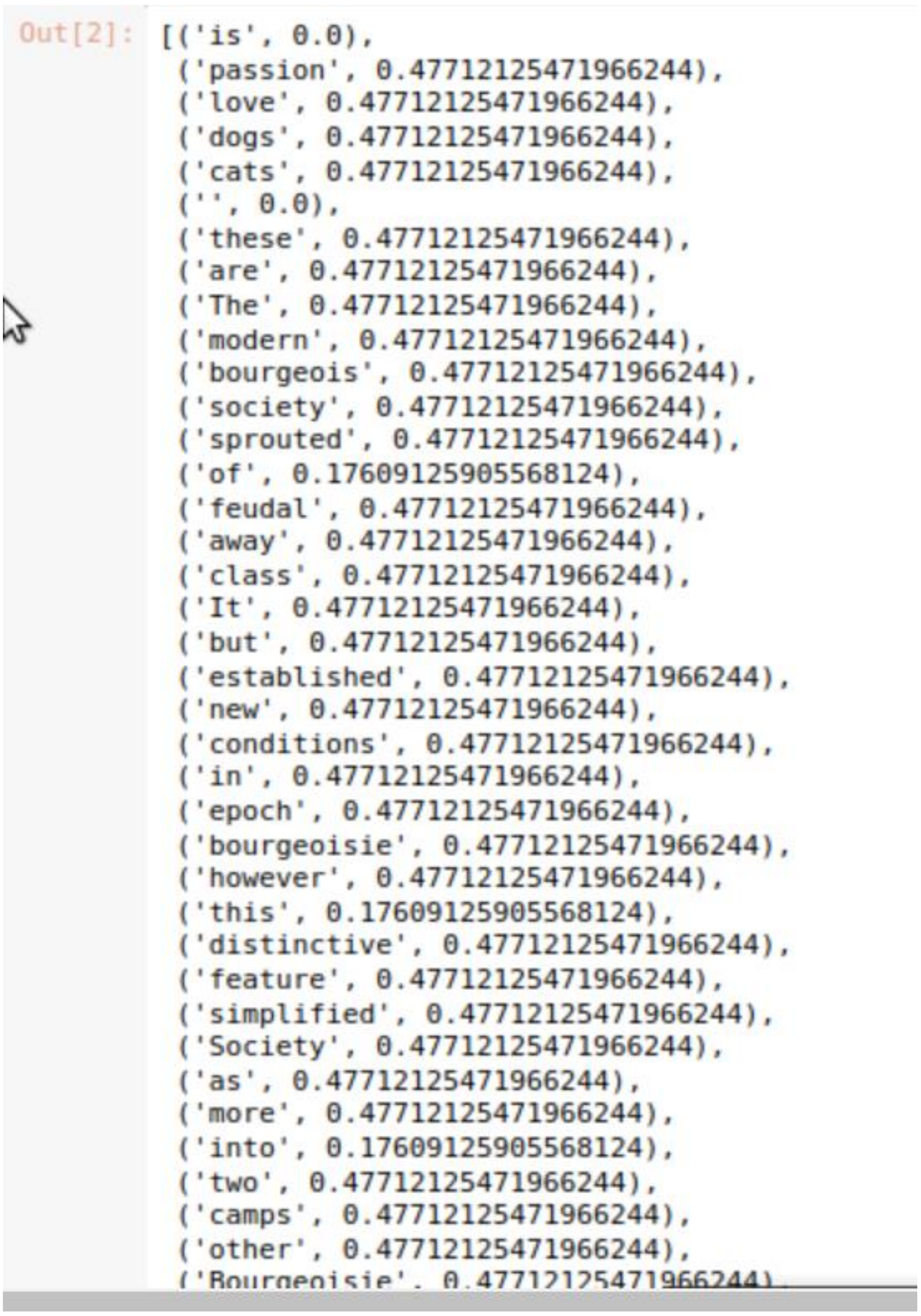
```
Out[1]: [('that', ('3', 1)),
         ('has', ('3', 4)),
         ('from', ('3', 1)),
         ('the', ('3', 5)),
         ('ruins', ('3', 1)),
         ('not', ('3', 1)),
         ('done', ('3', 1)),
         ('with', ('3', 1)),
         ('antagonisms', ('3', 2)),
         ('classes', ('3', 2)),
         ('oppression', ('3', 1)),
         ('forms', ('3', 1)),
         ('struggle', ('3', 1)),
         ('place', ('3', 1)),
         ('old', ('3', 1)),
         ('ones', ('3', 1)),
         ('Our', ('3', 1)),
         ('possesses', ('3', 1)),
```

The term frequency output



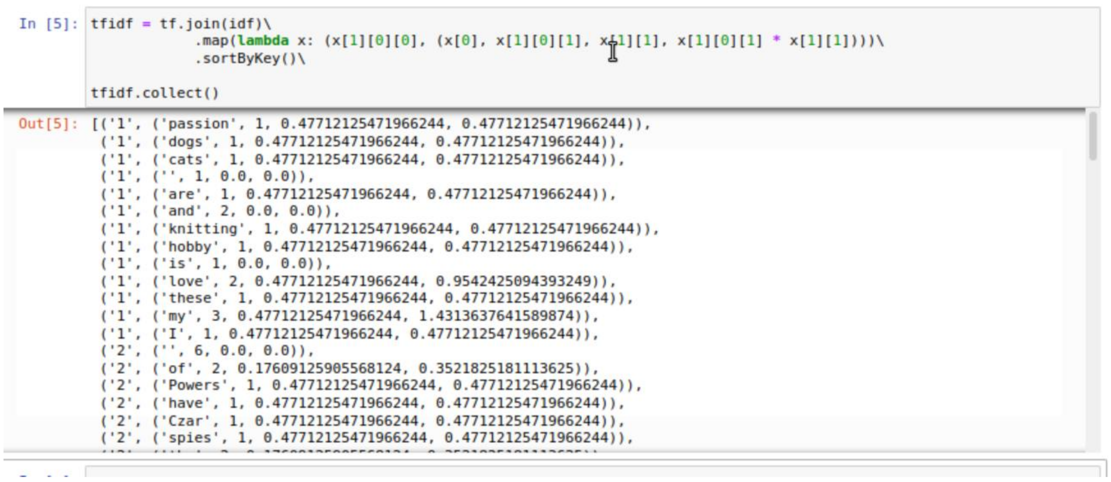
```
In [2]: import math

        idf = wordcount.map(lambda x: (x[0][1], (x[0][0], x[1], 1)))\
                .map(lambda x: (x[0], x[1][2]))\
                .reduceByKey(lambda x, y: x + y)\
                .map(lambda x: (x[0], math.log10(numFiles / x[1])))

        idf.collect()
```

Calculate the inverted document frequency



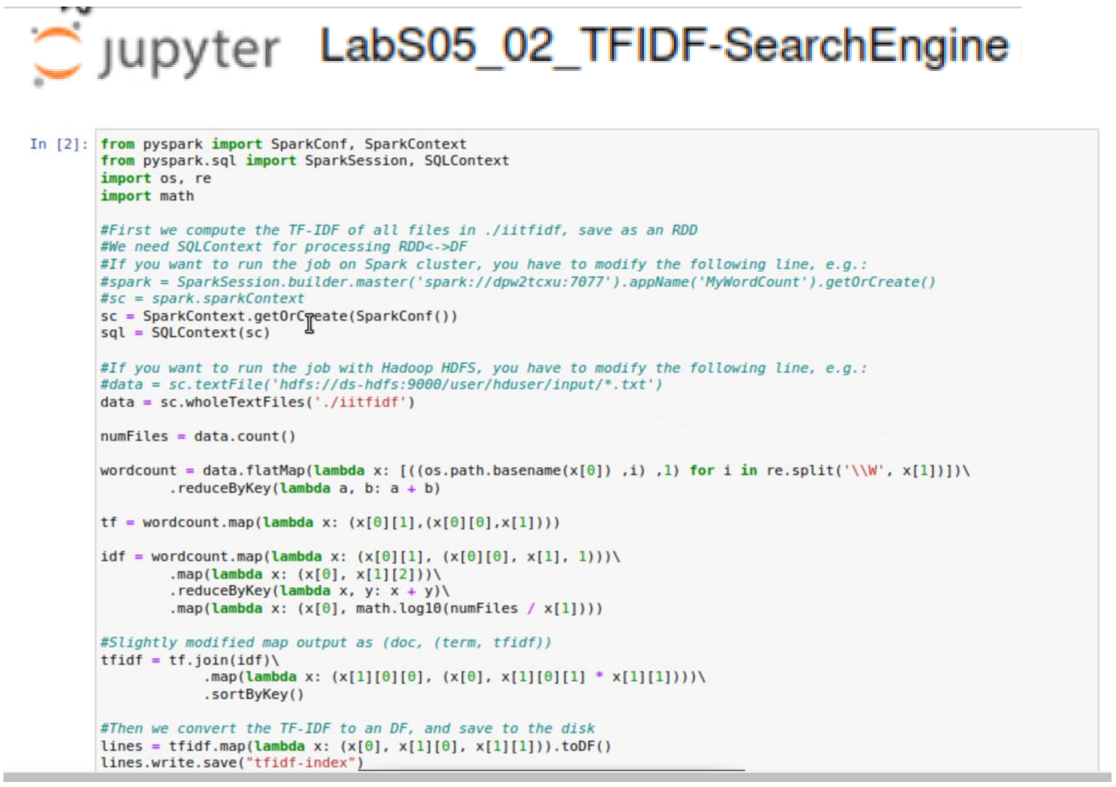
```
Out[2]: [('is', 0.0),
         ('passion', 0.47712125471966244),
         ('love', 0.47712125471966244),
         ('dogs', 0.47712125471966244),
         ('cats', 0.47712125471966244),
         ('', 0.0),
         ('these', 0.47712125471966244),
         ('are', 0.47712125471966244),
         ('The', 0.47712125471966244),
         ('modern', 0.47712125471966244),
         ('bourgeois', 0.47712125471966244),
         ('society', 0.47712125471966244),
         ('sprouted', 0.47712125471966244),
         ('of', 0.17609125905568124),
         ('feudal', 0.47712125471966244),
         ('away', 0.47712125471966244),
         ('class', 0.47712125471966244),
         ('It', 0.47712125471966244),
         ('but', 0.47712125471966244),
         ('established', 0.47712125471966244),
         ('new', 0.47712125471966244),
         ('conditions', 0.47712125471966244),
         ('in', 0.47712125471966244),
         ('epoch', 0.47712125471966244),
         ('bourgeoisie', 0.47712125471966244),
         ('however', 0.47712125471966244),
         ('this', 0.17609125905568124),
         ('distinctive', 0.47712125471966244),
         ('feature', 0.47712125471966244),
         ('simplified', 0.47712125471966244),
         ('Society', 0.47712125471966244),
         ('as', 0.47712125471966244),
         ('more', 0.47712125471966244),
         ('into', 0.17609125905568124),
         ('two', 0.47712125471966244),
         ('camps', 0.47712125471966244),
         ('other', 0.47712125471966244),
         ('Bourgeoisie', 0.47712125471966244),
```

The output of the inverted document frequency



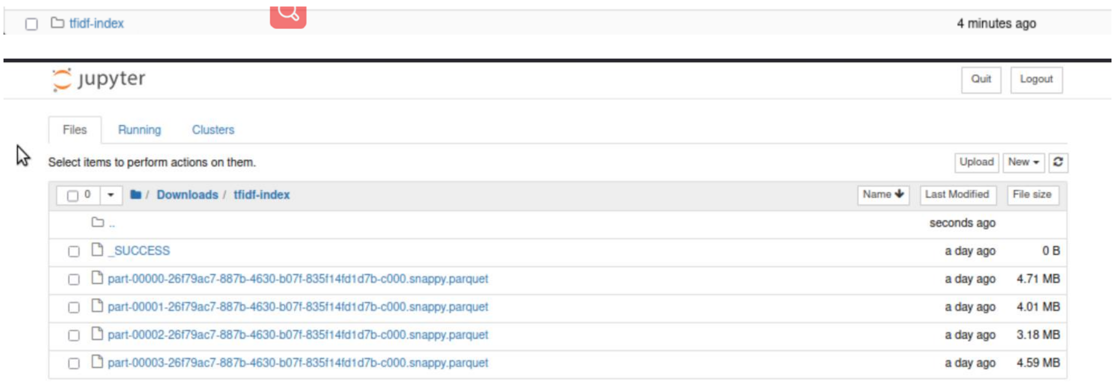
```
In [5]: tfidf = tf.join(idf)\
                    .map(lambda x: (x[1][0][0], (x[0], x[1][0][1], x[1][1], x[1][0][1] * x[1][1])))\
                    .sortByKey()\

        tfidf.collect()

Out[5]: [('1', ('passion', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('dogs', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('cats', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('', 1, 0.0, 0.0)),
         ('1', ('are', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('and', 2, 0.0, 0.0)),
         ('1', ('knitting', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('hobby', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('is', 1, 0.0, 0.0)),
         ('1', ('love', 2, 0.47712125471966244, 0.9542425094393249)),
         ('1', ('these', 1, 0.47712125471966244, 0.47712125471966244)),
         ('1', ('my', 3, 0.47712125471966244, 1.4313637641589874)),
         ('1', ('I', 1, 0.47712125471966244, 0.47712125471966244)),
         ('2', ('', 6, 0.0, 0.0)),
         ('2', ('of', 2, 0.17609125905568124, 0.3521825181113625)),
         ('2', ('Powers', 1, 0.47712125471966244, 0.47712125471966244)),
         ('2', ('have', 1, 0.47712125471966244, 0.47712125471966244)),
         ('2', ('Czar', 1, 0.47712125471966244, 0.47712125471966244)),
         ('2', ('spies', 1, 0.47712125471966244, 0.47712125471966244)),
```

Computing TF-IDF and show the result



```
In [2]: from pyspark import SparkConf, SparkContext
        from pyspark.sql import SparkSession, SQLContext
        import os, re
        import math

        #First we compute the TF-IDF of all files in ./iitfidf, save as an RDD
        #We need SQLContext for processing RDD<->DF
        #If you want to run the job on Spark cluster, you have to modify the following line, e.g.:
        #spark = SparkSession.builder.master('spark://dpw2tcxu:7077').appName('MyWordCount').getOrCreate()
        #sc = spark.sparkContext
        sc = SparkContext.getOrCreate(SparkConf())
        sql = SQLContext(sc)

        #If you want to run the job with Hadoop HDFS, you have to modify the following line, e.g.:
        #data = sc.textFile('hdfs://ds-hdfs:9000/user/hduser/input/*.txt')
        data = sc.wholeTextFiles('./iitfidf')

        numFiles = data.count()

        wordcount = data.flatMap(lambda x: [((os.path.basename(x[0]) ,i) ,1) for i in re.split('\\W', x[1])])\
                .reduceByKey(lambda a, b: a + b)

        tf = wordcount.map(lambda x: (x[0][1],(x[0][0],x[1])))

        idf = wordcount.map(lambda x: (x[0][1], (x[0][0], x[1], 1)))\
                .map(lambda x: (x[0], x[1][2]))\
                .reduceByKey(lambda x, y: x + y)\
                .map(lambda x: (x[0], math.log10(numFiles / x[1])))

        #Slightly modified map output as (doc, (term, tfidf))
        tfidf = tf.join(idf)\
                    .map(lambda x: (x[1][0][0], (x[0], x[1][0][1] * x[1][1])))\
                    .sortByKey()

        #Then we convert the TF-IDF to an DF, and save to the disk
        lines = tfidf.map(lambda x: (x[0], x[1][0], x[1][1])).toDF()
        lines.write.save("tfidf-index")
```

Based on the given English documents, calculate the tf-idf and save in a directory.

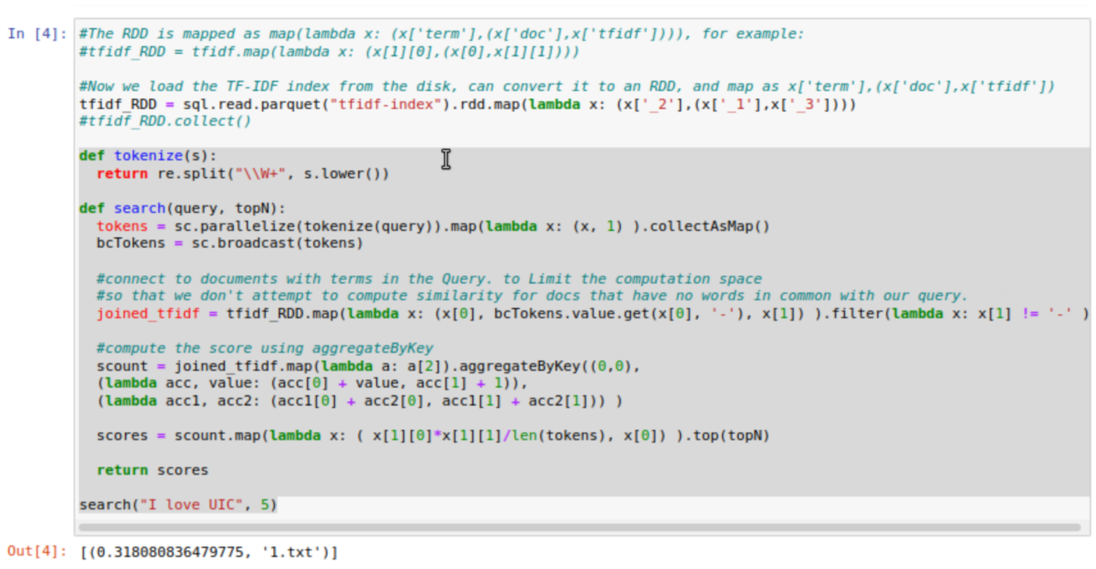


Select items to perform actions on them.

/ Downloads / tfidf-index

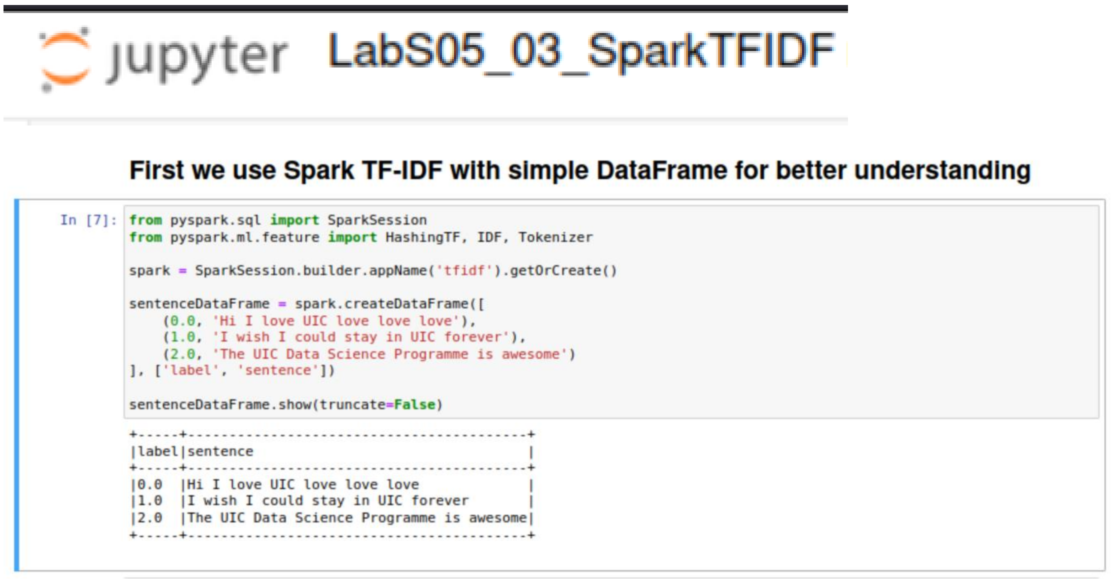
..	seconds ago	
_SUCCESS	a day ago	0 B
part-00000-26f79ac7-887b-4630-b07f-835f14fd1d7b-c000.snappy.parquet	a day ago	4.71 MB
part-00001-26f79ac7-887b-4630-b07f-835f14fd1d7b-c000.snappy.parquet	a day ago	4.01 MB
part-00002-26f79ac7-887b-4630-b07f-835f14fd1d7b-c000.snappy.parquet	a day ago	3.18 MB
part-00003-26f79ac7-887b-4630-b07f-835f14fd1d7b-c000.snappy.parquet	a day ago	4.59 MB

Here is the files that generated by the codes



```
In [4]: #The RDD is mapped as map(lambda x: (x['term'],(x['doc'],x['tfidf']))), for example:
        #tfidf_RDD = tfidf.map(lambda x: (x[1][0],(x[0],x[1][1])))

        #Now we load the TF-IDF index from the disk, can convert it to an RDD, and map as x['term'],(x['doc'],x['tfidf'])
        tfidf_RDD = sql.read.parquet("tfidf-index").rdd.map(lambda x: (x['_2'],(x['_1'],x['_3'])))
        #tfidf_RDD.collect()

        def tokenize(s):
          return re.split("\\W+", s.lower())

        def search(query, topN):
          tokens = sc.parallelize(tokenize(query)).map(lambda x: (x, 1) ).collectAsMap()
          bcTokens = sc.broadcast(tokens)

          #connect to documents with terms in the Query. to Limit the computation space
          #so that we don't attempt to compute similarity for docs that have no words in common with our query.
          joined_tfidf = tfidf_RDD.map(lambda x: (x[0], bcTokens.value.get(x[0], '-'), x[1]) ).filter(lambda x: x[1] != '-' )

          #compute the score using aggregateByKey
          scount = joined_tfidf.map(lambda a: a[2]).aggregateByKey((0,0),
          (lambda acc, value: (acc[0] + value, acc[1] + 1)),
          (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])) )

          scores = scount.map(lambda x: ( x[1][0]*x[1][1]/len(tokens), x[0]) ).top(topN)

          return scores

        search("I love UIC", 5)

Out[4]: [(0.318080836479775, '1.txt')]
```

Search specific key words based on the generated tf-idf and return the most similar document and its score.

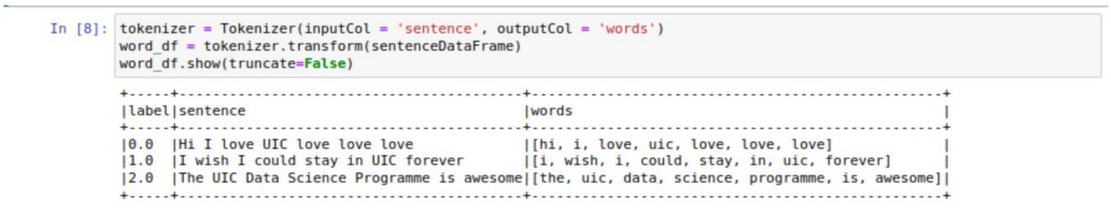


jupyter LabS05_03_SparkTFIDF

First we use Spark TF-IDF with simple DataFrame for better understanding

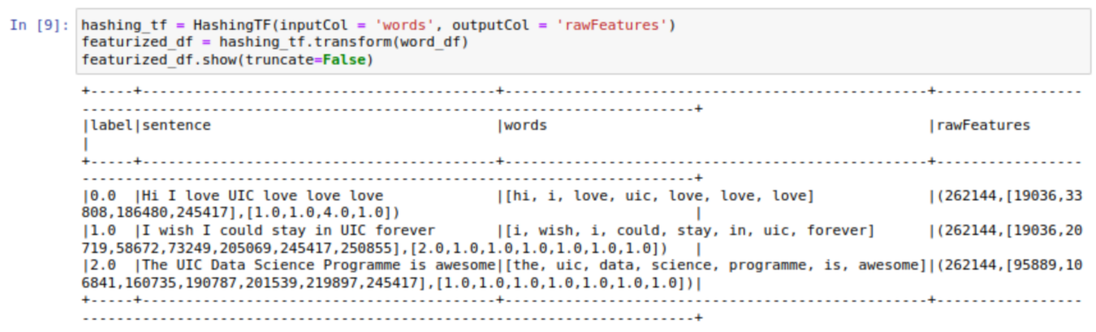
```
In [7]: from pyspark.sql import SparkSession
        from pyspark.ml.feature import HashingTF, IDF, Tokenizer

        spark = SparkSession.builder.appName('tfidf').getOrCreate()

        sentenceDataFrame = spark.createDataFrame([
            (0.0, 'Hi I love UIC love love love'),
            (1.0, 'I wish I could stay in UIC forever'),
            (2.0, 'The UIC Data Science Programme is awesome')
        ], ['label', 'sentence'])

        sentenceDataFrame.show(truncate=False)

        +-----+-----------------------------------------+
        |label|sentence                                 |
        +-----+-----------------------------------------+
        |0.0  |Hi I love UIC love love love             |
        |1.0  |I wish I could stay in UIC forever       |
        |2.0  |The UIC Data Science Programme is awesome|
        +-----+-----------------------------------------+
```

Use Spark TF-IDF with simple data frame



```
In [8]: tokenizer = Tokenizer(inputCol = 'sentence', outputCol = 'words')
        word_df = tokenizer.transform(sentenceDataFrame)
        word_df.show(truncate=False)

        +-----+-----------------------------------------+------------------------------------------------+
        |label|sentence                                 |words                                           |
        +-----+-----------------------------------------+------------------------------------------------+
        |0.0  |Hi I love UIC love love love             |[hi, i, love, uic, love, love, love]            |
        |1.0  |I wish I could stay in UIC forever       |[i, wish, i, could, stay, in, uic, forever]     |
        |2.0  |The UIC Data Science Programme is awesome|[the, uic, data, science, programme, is, awesome]|
        +-----+-----------------------------------------+------------------------------------------------+
```

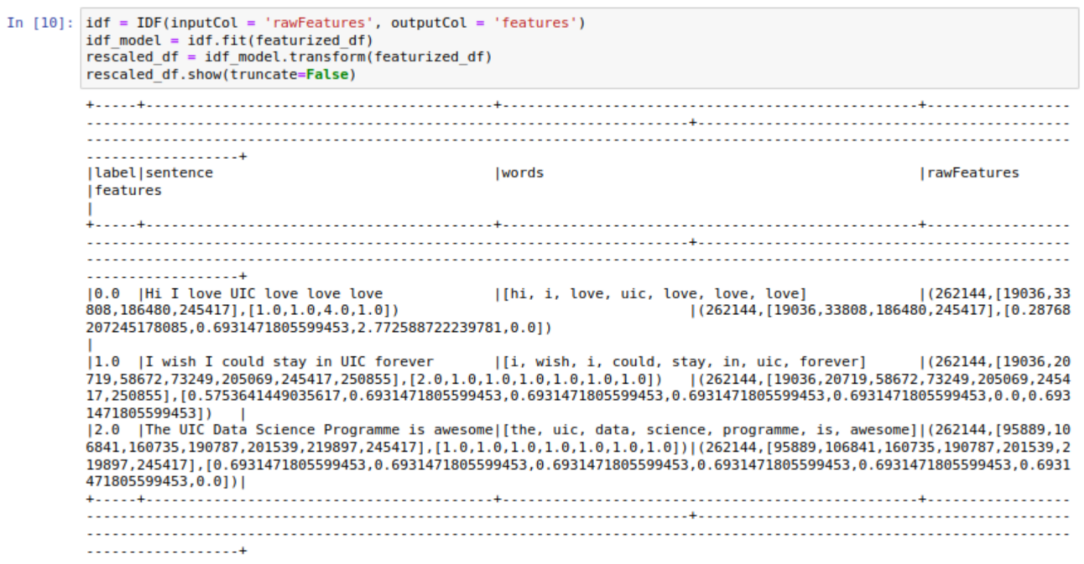
Tokenize the word



```
In [9]: hashing_tf = HashingTF(inputCol = 'words', outputCol = 'rawFeatures')
        featurized_df = hashing_tf.transform(word_df)
        featurized_df.show(truncate=False)
```

```
+-----+-----------------------------------------+-------------------------------------------------+-----------------
--------------------------------------------------------------------------+
|label|sentence                                 |words                                            |rawFeatures
|
+-----+-----------------------------------------+-------------------------------------------------+-----------------
--------------------------------------------------------------------------+
|0.0  |Hi I love UIC love love love             |[hi, i, love, uic, love, love, love]             |(262144,[19036,33
808,186480,245417],[1.0,1.0,4.0,1.0])                                     |
|1.0  |I wish I could stay in UIC forever       |[i, wish, i, could, stay, in, uic, forever]      |(262144,[19036,20
719,58672,73249,205069,245417,250855],[2.0,1.0,1.0,1.0,1.0,1.0,1.0])      |
|2.0  |The UIC Data Science Programme is awesome|[the, uic, data, science, programme, is, awesome]|(262144,[95889,10
6841,160735,190787,201539,219897,245417],[1.0,1.0,1.0,1.0,1.0,1.0,1.0])|
+-----+-----------------------------------------+-------------------------------------------------+-----------------
--------------------------------------------------------------------------+
```

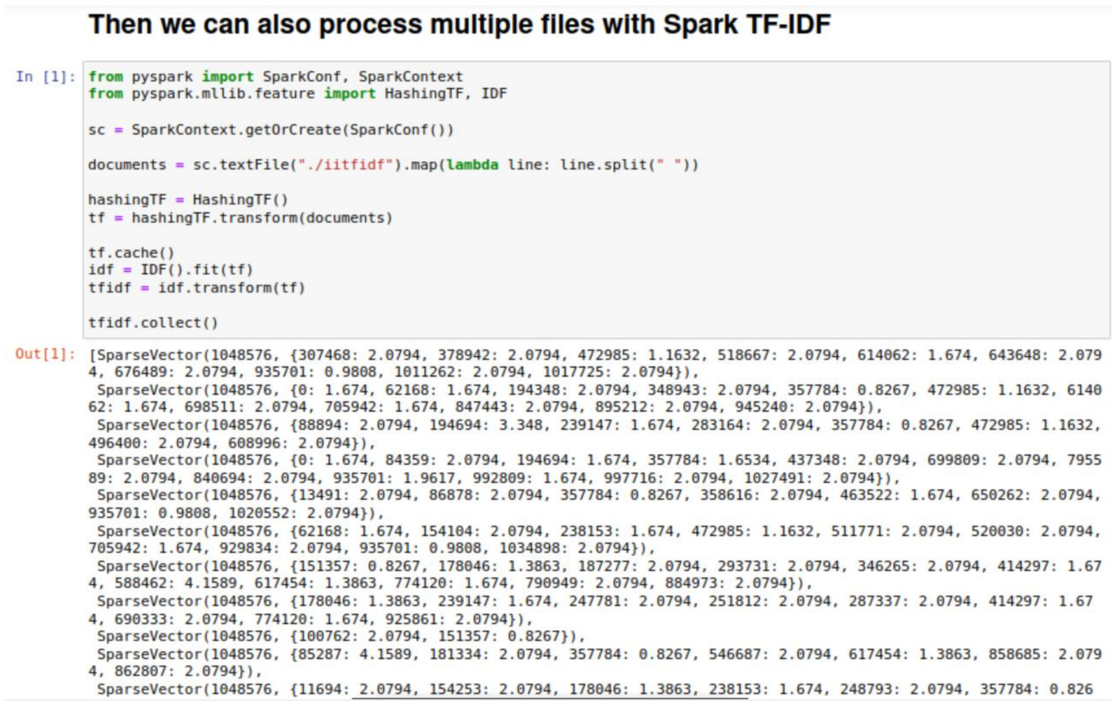
Encode the words and calculate term frequency



```
In [10]: idf = IDF(inputCol = 'rawFeatures', outputCol = 'features')
         idf_model = idf.fit(featurized_df)
         rescaled_df = idf_model.transform(featurized_df)
         rescaled_df.show(truncate=False)
```

```
+-----+-----------------------------------------+-------------------------------------------------+-----------------
-------------------------------------------------------------------------+------------------------------------------
------------------------------------------------------------------------------------------------------------------
-----------------+
|label|sentence                                 |words                                            |rawFeatures
|features
|
+-----+-----------------------------------------+-------------------------------------------------+-----------------
-------------------------------------------------------------------------+------------------------------------------
------------------------------------------------------------------------------------------------------------------
-----------------+
|0.0  |Hi I love UIC love love love             |[hi, i, love, uic, love, love, love]             |(262144,[19036,33
808,186480,245417],[1.0,1.0,4.0,1.0])                                    |(262144,[19036,33808,186480,245417],[0.28768
207245178085,0.6931471805599453,2.772588722239781,0.0])
|
|1.0  |I wish I could stay in UIC forever       |[i, wish, i, could, stay, in, uic, forever]      |(262144,[19036,20
719,58672,73249,205069,245417,250855],[2.0,1.0,1.0,1.0,1.0,1.0,1.0])     |(262144,[19036,20719,58672,73249,205069,2454
17,250855],[0.5753641449035617,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.0,0.693
1471805599453])    |
|2.0  |The UIC Data Science Programme is awesome|[the, uic, data, science, programme, is, awesome]|(262144,[95889,10
6841,160735,190787,201539,219897,245417],[1.0,1.0,1.0,1.0,1.0,1.0,1.0])|(262144,[95889,106841,160735,190787,201539,2
19897,245417],[0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931471805599453,0.6931
471805599453,0.0])|
+-----+-----------------------------------------+-------------------------------------------------+-----------------
-------------------------------------------------------------------------+------------------------------------------
------------------------------------------------------------------------------------------------------------------
-----------------+
```

Calculate the inverted term frequency



Then we can also process multiple files with Spark TF-IDF

```
In [1]: from pyspark import SparkConf, SparkContext
        from pyspark.mllib.feature import HashingTF, IDF

        sc = SparkContext.getOrCreate(SparkConf())

        documents = sc.textFile("./iitfidf").map(lambda line: line.split(" "))

        hashingTF = HashingTF()
        tf = hashingTF.transform(documents)

        tf.cache()
        idf = IDF().fit(tf)
        tfidf = idf.transform(tf)

        tfidf.collect()
```

```
Out[1]: [SparseVector(1048576, {307468: 2.0794, 378942: 2.0794, 472985: 1.1632, 518667: 2.0794, 614062: 1.674, 643648: 2.079
        4, 676489: 2.0794, 935701: 0.9808, 1011262: 2.0794, 1017725: 2.0794}),
         SparseVector(1048576, {0: 1.674, 62168: 1.674, 194348: 2.0794, 348943: 2.0794, 357784: 0.8267, 472985: 1.1632, 6140
        62: 1.674, 698511: 2.0794, 705942: 1.674, 847443: 2.0794, 895212: 2.0794, 945240: 2.0794}),
         SparseVector(1048576, {88894: 2.0794, 194694: 3.348, 239147: 1.674, 283164: 2.0794, 357784: 0.8267, 472985: 1.1632,
        496400: 2.0794, 608996: 2.0794}),
         SparseVector(1048576, {0: 1.674, 84359: 2.0794, 194694: 1.674, 357784: 1.6534, 437348: 2.0794, 699809: 2.0794, 7955
        89: 2.0794, 840694: 2.0794, 935701: 1.9617, 992809: 1.674, 997716: 2.0794, 1027491: 2.0794}),
         SparseVector(1048576, {13491: 2.0794, 86878: 2.0794, 357784: 0.8267, 358616: 2.0794, 463522: 1.674, 650262: 2.0794,
        935701: 0.9808, 1020552: 2.0794}),
         SparseVector(1048576, {62168: 1.674, 154104: 2.0794, 238153: 1.674, 472985: 1.1632, 511771: 2.0794, 520030: 2.0794,
        705942: 1.674, 929834: 2.0794, 935701: 0.9808, 1034898: 2.0794}),
         SparseVector(1048576, {151357: 0.8267, 178046: 1.3863, 187277: 2.0794, 293731: 2.0794, 346265: 2.0794, 414297: 1.67
        4, 588462: 4.1589, 617454: 1.3863, 774120: 1.674, 790949: 2.0794, 884973: 2.0794}),
         SparseVector(1048576, {178046: 1.3863, 239147: 1.674, 247781: 2.0794, 251812: 2.0794, 287337: 2.0794, 414297: 1.67
        4, 690333: 2.0794, 774120: 1.674, 925861: 2.0794}),
         SparseVector(1048576, {100762: 2.0794, 151357: 0.8267}),
         SparseVector(1048576, {85287: 4.1589, 181334: 2.0794, 357784: 0.8267, 546687: 2.0794, 617454: 1.3863, 858685: 2.079
        4, 862807: 2.0794}),
         SparseVector(1048576, {11694: 2.0794, 154253: 2.0794, 178046: 1.3863, 238153: 1.674, 248793: 2.0794, 357784: 0.826
```

Process multiple files with Spark TF-IDF