References:

https://sparkbyexamples.com/pyspark-tutorial/

using parallelize()

SparkContext has several functions to use with RDDs. For example, it's `parallelize()` method is used to create an RDD from a list.

```
Copy

# Create RDD from parallelize
dataList = [("Java", 20000), ("Python", 100000), ("Scala", 3000)]
rdd=spark.sparkContext.parallelize(dataList)
```

using textFile()

RDD can also be created from a text file using `textFile()` function of the SparkContext.

```
# Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/test.txt")
```

Once you have an RDD, you can perform transformation and action operations. Any operation you perform on RDD runs in parallel.

Stanford notes: https://stanford.edu/~rezab/dao/notes/

https://medium.com/@rezandry/find-most-relevance-text-data-using-pyspark-with-tf-idf-a4269a13e59

https://datascience-enthusiast.com/Python/text_analysis.html

Is this a different corpus? Must be separated by a return not tab.

In [16]:

```
data = spark.sparkContext.textFile("/Users/iris/data.txt")
# # convert to lowercase before splitting
# split_rdd = data.flatMap(lambda term: term.split())
print(data.collect())
```

```
['10022814 growth inhibition dis_breast_cancer_dis cell grb2 downregulation correlate inactivation mitogen-activated
protein kinase gene_egfr+_gene erbb2 cell  increased dis_breast_cancer_dis growth associate increase expression gene_
epidermal_growth_factor_gene receptor gene_egfr+_gene erbb2 receptor tyrosine kinase rtks  upon activation rtks trans
mit oncogenic signal binding growth factor receptor bind protein-2 grb2 turn bind sos activate ras-raf-mek-mitogen-ac
tivated protein map kinase pathway  grb2 important transformation fibroblast gene_egfr+_gene erbb2 whether grb2 impor
tant proliferation dis_breast_cancer_dis cell express rtks unclear  use liposome deliver nuclease-resistant antisense
oligodeoxynucleotide oligos specific grb2 mrna dis_breast_cancer_dis cell  grb2 protein downregulation inhibit dis_br
east_cancer_dis cell growth degree growth inhibition dependent upon activation and-or endogenou level rtks  grb2 inhi
bition lead map kinase inactivation gene_egfr+_gene erbb2 dis_breast_cancer_dis cell suggest different pathway use ge
ne_egfr+_gene erbb2 regulate dis_breast_cancer_dis growth  ', '10022831 role gene_alphavbeta3_gene integrin activatio
n vascular endothelial growth factor receptor-2  interaction between integrin gene_alphavbeta3_gene extracellular mat
rix crucial endothelial cell sprout capillary angiogenesis  furthermore integrin-mediated outside-in signal co-operat
e growth factor receptor promote cell proliferation motility  determine potential regulation angiogenic inducer recep
tor integrin system investigate interaction between gene_alphavbeta3_gene integrin tyrosine kinase vascular endotheli
al growth factor receptor-2 vegfr-2 human endothelial cell  report tyrosine-phosphorylated vegfr-2 co-immunoprecipita
```

REFERENCES:

Before we process the data, we need to do pre-processing the data to get the partial data from dataset.

```
kecilRawData = rawData.map(lambda x: x.lower())
fields = kecilRawData.map(lambda x: x.split("\t"))
documents = fields.map(lambda x: x[2].split(" "))
documentId = fields.map(lambda x: x[0])
```

That code process all data into lowercase, then split the data using regex \t because the data separated by tab (.tsv), you need adjust what like dataset that you used, like if you use .csv , you can used comma (,) to split the data. And then, we need save the document id to identity which document belong to.

We can create hashingTF using HashingTF, and set the fixed-length feature vectors with 100000, actually the value can adjust as the feature vectors that will used. And then, we can use the result of hashingTF to transform the document into tf.

```
hashingTF = HashingTF(100000)
tf = hashingTF.transform(documents)
```

Text Analysis and Entity Resolution

(1d) Amazon record with the most tokens

Which Amazon record has the biggest number of tokens?

In other words, you want to sort the records and get the one with the largest count of tokens.

```
# TODO: Replace <FILL IN> with appropriate code
def findBiggestRecord(vendorRDD):
    """ Find and return the record with the largest number of tokens
    Args:
        vendorRDD (RDD of (recordId, tokens)): input Pair Tuple of record ID and tokens
    Returns:
        list: a list of 1 Pair Tuple of record ID and tokens
    """
    return vendorRDD.sortBy(lambda (id, tokens): -len(tokens)).take(1)

biggestRecordAmazon = findBiggestRecord(amazonRecToToken)
print 'The Amazon record with ID "%s" has the most tokens (%s)' % (biggestRecordAmazon[0][0],
                                                                   len(biggestRecordAmazon[0][1]))
```

```
The Amazon record with ID "b000o24l3q" has the most tokens (1547)
```

Pre-processing:

```
my_para = data.map(lambda x: tuple(x.split('\n')))
```

✓ 0.3s

```
#split_list = my_para.map(lambda x: list(x[0].split()))
split_list = my_para.map(lambda x: tuple(x[0].split()))
#split_list2 = split_list.filter(lambda x: x[0] is not None)


doc_id = split_list.zipWithIndex().keyBy(lambda x: x[1])

# unpacking 2 tuple into 1 singular tuple to compute
doc_id = doc_id.map(lambda x: tuple((x[0], *x[1])))

# re-packing the tuple
doc_id = doc_id.map(lambda x: tuple((x[0], x[1])))
doc_id.take(3)
```

✓ 0.8s

I do loads of preprocessing to this data like splitting into a tab. Inserting document ID through zipWithIndex() function. I had to pack and unpack the RDD to convert for the transformation.
The above code transforms into: *('doc_id', 'term')* from *corpus.*

```
# separate into key-value pairs
# ('doc_id','term') --> (( 'doc_id', 'token'), 1 )
kv_mapper = doc_id.flatMap(lambda x: tuple( [ ((x[0], i),1) for i in x[1]]))
```

✓ 0.2s

```
kv_mapper.take(5)
```

✓ 0.7s

```
[((0, '10022814'), 1),
 ((0, 'growth'), 1),
 ((0, 'inhibition'), 1),
 ((0, 'dis_breast_cancer_dis'), 1),
 ((0, 'cell'), 1)]
```

I make a tuple of key-value pairs for the term. Where it is mapped to: *((‘doc_id’, ‘term’), 1)*.

The flatmap eradicates the partitioning so we result in list of tuples.

```
# do the term-count
# (('doc_id', 'token'), 1) --> (('doc_id', 'token'), [1++])
reduce_1 = kv_mapper.reduceByKey(lambda x, y: x+y)


# = kv_mapper.map(lambda x: x[1]).sum()
reduce_1.take(4)
```

✓ 0.4s

```
[((0, '10022814'), 1),
 ((0, 'growth'), 4),
 ((0, 'inhibition'), 1),
 ((0, 'dis_breast_cancer_dis'), 2)]
```

```
# tf = reducer_2.map(lambda x: (x[0], (len(doc_id1)/x[1])))
# tf.take(5)
```

✓ 0.2s

```
# mapping for IDF start
# (('doc_id', 'token'), tf) --> ('token', ('doc_id', tf))

# tf = reducer_2.map(lambda x: (x[0], (len(doc_id1)/x[1])))

shuffle = reduce_1.map(lambda x: (x[0][1], (x[0][0], x[1])))
shuffle.take(3)
```

✓ 0.8s

```
[('10022814', (0, 1)), ('growth', (0, 4)), ('inhibition', (0, 1))]
```

First, I add the number of occurrences of each term in the document by reducing the key i mapped earlier.

```
# tokenise for idf
# (('doc_id', 'token'), tf) --> ('token', ('doc_id', tf, 1))
mapper_2 = reduce_1.map(lambda x: (x[0][1], (x[0][0], x[1],1)))
mapper_2.take(2)
```

✓ 0.8s

```
[('10022814', (0, 1, 1)), ('growth', (0, 4, 1))]
```

```
# only take term and token
# ('token', ('doc_id', tf,1)) --> ('token', 1)
mapper_3 = mapper_2.map(lambda x: (x[0], x[1][2]))
mapper_3.take(3)
```

✓ 0.7s

```
[('10022814', 1), ('growth', 1), ('inhibition', 1)]
```

```
# counting by token
# ('token', 1) --> ('token', [1++])
reducer_2 = mapper_3.reduceByKey(lambda x, y: x+y)
reducer_2.take(3)
```

✓ 0.1s

```
[('10022814', 1), ('growth', 5), ('inhibition', 3)]
```

I make my way through the RDD so I can end up with word count and term. In mapper_3, I map the RDD to grab only the required elements of the tuple. So I can apply the term-frequency formula in the next step.

```
doc_id2 = doc_id.map(lambda x: x[1]).collect()
```

✓ 0.8s

```
tf = reducer_2.map(lambda x: (x[0], x[1], (x[1]/len(doc_id2[0]))))
tf.take(5)
```

✓ 1.1s

```
[('10022814', 1, 0.008771929824561403),
 ('growth', 5731, 50.271929824561404),
 ('inhibition', 2070, 18.157894736842106),
 ('dis_breast_cancer_dis', 437, 3.8333333333333335),
 ('cell', 7026, 61.63157894736842)]
```

```
# idf = tf.map(lambda x: (x[0], math.log10(len(doc_id1)/x[1])))

idf = tf.map(lambda x: (x[0], math.log10(len(doc_id2)/x[1])))


idf.take(3)
```

✓ 0.9s

```
[('10022814', 4.029140179764322),
 ('growth', 0.2709097713065723),
 ('inhibition', 0.7131698343074041)]
```

Since RDD are non-iterable, I cleverly went around it by casting it as a list; once it is passed through a mapper, it will recast itself as an RDD.

Variable ‘reducer_2’ has form *(‘term’, ‘count’)*
We send it through the tf mapper to return the term-frequency to receive:
$wf = term/total\ doc\ word\ count$

```
# ('doc_id', 'everything_else') --> (('doc_id'), ('token', 'tf', 'idf', 'tf-idf'))
# tf_idf = joint_my.map(lambda x: (x[1][0][0], ( x[0], x[1][0][1], x[1][1], x[1][0][1] * x[1][1])) ).sortByKey()
tf_idf = joint_my.map(lambda x: (x[0], ( x[1][0] * x[1][1])) )
tf_idf.collect()
```

✓ 0.3s

```
Output exceeds the size limit. Open the full output data in a text editor
[('10022814', 0.0),
 ('growth', 1.1183520069376303),
 ('downregulation', 0.0),
 ('erbb2', 0.0),
 ('increased', 0.0),
 ('increase', 1.2041199826559248),
 ('expression', 1.0375350005115247),
 ('gene_epidermal_growth_factor_gene', 1.2041199826559248),
 ('receptor', 1.0375350005115247),
 ('rtks', 0.0),
 ('upon', 1.2041199826559248),
 ('activation', 1.2723233459190997),
 ('transmit', 0.0),
 ('binding', 0.0),
 ('factor', 1.1183520069376303),
 ('bind', 0.0),
 ('protein-2', 0.0),
 ('turn', 0.0),
 ('interaction', 1.2041199826559248),
 ('matrix', 0.0),
```

I now compute the simple multiplication of tf and idf by mapping the value into single RDD.