Anagram Detection: Detect Anagrams (Words with The Same Letters but In a Different Order) From a Dataset of Words.

Abstract

Anagrams are fascinating linguistic puzzles where words or phrases can be rearranged to form other valid words or phrases. In this project, I explore an efficient approach to detect anagrams within a dataset of words. The goal is to identify pairs of words that share the same set of letters but differ in their arrangement.

Anagram:

An **anagram** is a word or phrase formed by rearranging the letters of another word or phrase. For example, “listen” and “silent” are anagrams because they use the same letters but in a different order.

Uses of Anagrams:

- **Word Games and Puzzles**:
  - Anagrams are popular in word games like **Scrabble**, **crossword puzzles**, and **Jumble**.
  - They challenge players to rearrange letters to form new words.
- **Cryptography**:
  - Historically, anagrams were used in cryptography to encode secret messages.
  - By rearranging letters, hidden meanings could be communicated.
- **Text Analysis and NLP**:
  - In natural language processing (NLP), anagrams help analyze patterns in text.
  - Detecting anagrams in a large corpus can reveal linguistic relationships.
- **Humour and Creativity**:
  - Writers use anagrams to add wit and playfulness to their work.
  - Clever anagrams create unexpected connections and commentary.

Real-Time Applications

- **Pseudonyms and Character Names**:
    - Authors create pseudonyms or character names using anagrams of their own names.
    - Anagrams add depth and intrigue to fictional characters.
- **Cryptography and Security**:
    - Modern cryptography techniques use anagrams for secure communication.
    - Anagrams can encode sensitive information.
- **Pattern Recognition**:
    - Anagram detection algorithms are used to recognize patterns in various contexts.
    - For example, identifying anagrams in product names or domain names.
- **Wordplay in Marketing and Branding**:
    - Companies use anagrams to create memorable slogans or brand names.
    - Anagrams add a layer of creativity and uniqueness.

Detecting Anagrams Using Spark

1. **Resilient Distributed Datasets (RDDs)**:
    - RDDs are the fundamental data structures in Spark. They represent distributed collections of data that can be processed in parallel across a cluster.
    - In our case, we create an RDD from a dataset of words containing potential anagrams.
2. **Key-Value Pairs**:
    - We convert the RDD into key-value pairs, where the key represents the sorted letters of each word, and the value is the word itself.
    - For example, if we have the word “listen,” the key would be “eilnst” (sorted letters), and the value would be “listen.”
3. **Grouping Anagrams**:
    - We use the reduceByKey transformation to group anagrams together based on their sorted keys.
    - The result is a collection of key-value pairs where the key represents the sorted letters, and the value is a list of anagrams.
4. **Example Output**:
    - For instance, we might get the following anagram groups:
        - Anagrams for ‘aadeemqrsu’: [‘masquerade’, ‘squaremade’]
        - Anagrams for ‘aadins’: [‘dianas’, ‘naiads’]
        - Anagrams for ‘aaginnostt’: [‘antagonist’, ‘stagnation’]

Resilient Distributed Datasets (RDDs):

Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs:

- ***PARALLELIZE METHOD:***

  Parallelizing an existing collection in your driver program.

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

- ***TEXTFILE METHOD:***

  Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

```
distFile = sc.textFile("data.txt")
```

Solution:

We implement the above approach in spark RDD'S techniques. Our solution efficiently identifies anagrams within both large and small datasets.

PARALLELIZE METHOD:

An **anagram detection** process using **Apache Spark**.

1. **Creating a SparkContext**:

   - We start by creating a Sparkcentext, The parallelize method is a function provided by the SparkContext class.
   - This allows us to work with distributed data using Spark.

```
# 1--parallelize method
import pyspark
from pyspark import SparkContext
spark = SparkContext.getOrCreate()
```

2. **Defining the List of Words**:
   - We define a list of words called a_words. These words represent our input data.

```
a_words =  [  "masquerade", "squaremade", "dianas", "naiads",
              "antagonist", "stagnation", "bates", "beast",
              "beats", "thanks", "meteor", "lentis", "night",
              "thing", "desserts", "stressed", "listen","estab"
              "silent", "remote", "ignth", "ereomt", "skanth"  ]
```

3. **Creating an RDD (Resilient Distributed Dataset)**:

   - We convert the a_words list into an RDD using spark.sparkContext.parallelize(a_words).

   - RDDs are the fundamental data structure in Spark, allowing parallel processing.

```
1  a_words_rdd = spark.parallelize(a_words)
2  a_words_rdd.collect()
```

▸ (1) Spark Jobs

```
'squaremade',
'dianas',
'naiads',
'antagonist',
'stagnation',
'bates',
'beast',
'beats',
'thanks',
'meteor',
'lentis',
'night',
'thing',
'desserts',
'stressed',
'listen',
'estabsilent',
'remote',
'ignth',
```

4. **Mapping to Key-Value Pairs**:

   - We transform each word in the RDD into a key-value pair.
   - The key is the sorted letters of the word (e.g., “aadinss” for “naiads”).
   - The value is a list containing the original word (e.g., [“naiads”]).

```
a_words_sort = a_words_rdd.map(lambda word: ("".join(sorted([letter for letter in word])), [word]))
a_words_sort.collect()
```

```
▸ (1) Spark Jobs
('aadeemqrsu', ['squaremade']),
('aadins', ['dianas']),
('aadins', ['naiads']),
('aaginnostt', ['antagonist']),
('aaginnostt', ['stagnation']),
('abest', ['bates']),
('abest', ['beast']),
('abest', ['beats']),
('ahknst', ['thanks']),
('eemort', ['meteor']),
('eilnst', ['lentis']),
('ghint', ['night']),
('ghint', ['thing']),
('deerssst', ['desserts']),
('deerssst', ['stressed']),
('eilnst', ['listen']),
('abeeilnsstt', ['estabsilent']),
('eemort', ['remote']),
```

5. **Reducing by Key to Group Anagrams**:
   - We use reduceByKey to group anagrams together.
   - For each key (sorted letters), we concatenate the corresponding word lists.

```
a_words_reduce = a_words_sort.reduceByKey(lambda word1, word2: word1 + word2)
a_words_reduce.collect()
```

```
Out[68]: [('aaginnostt', ['antagonist', 'stagnation']),
 ('ahknst', ['thanks', 'skanth']),
 ('aadins', ['dianas', 'naiads']),
 ('aadeemqrsu', ['masquerade', 'squaremade']),
 ('abest', ['bates', 'beast', 'beats']),
 ('eemort', ['meteor', 'remote', 'ereomt']),
 ('eilnst', ['lentis', 'listen']),
 ('ghint', ['night', 'thing', 'ignth']),
 ('deerssst', ['desserts', 'stressed']),
 ('abeeilnsstt', ['estabsilent'])]
```

6. **Printing the Result**:
   - The final result is a collection of anagram groups.
   - We print each key (sorted letters) along with its anagrams.

```
for key, anagrams in sorted(a_words_reduce.collect()):
    print(f"{key}: {anagrams}")
```

```
aadeemqrsu: ['masquerade', 'squaremade']
aadins: ['dianas', 'naiads']
aaginnostt: ['antagonist', 'stagnation']
abeeilnsstt: ['estabsilent']
abest: ['bates', 'beast', 'beats']
ahknst: ['thanks', 'skanth']
deerssst: ['desserts', 'stressed']
eemort: ['meteor', 'remote', 'ereomt']
eilnst: ['lentis', 'listen']
ghint: ['night', 'thing', 'ignth']
```

TEXTFILE METHOD:

1. **Creating a SparkContext**:

We start by creating a Sparkcentext, The parallelize method is a function provided by the SparkContext class.

- This allows us to work with distributed data using Spark.

```
# 2--Textfile method
import pyspark
from pyspark import SparkContext
spark = SparkContext.getOrCreate()
```

2. **Read the Text File**:

- The code starts by reading a text file named “words2-2.txt” located in the specified path.
- The textFile method from the Spark context is used to read the file, and the resulting RDD (Resilient Distributed Dataset) is stored in the variable a_words2.

Python

```
1  a_words2 = spark.textFile("/FileStore/tables/words2-3.txt")
2  a_words2.collect()
```

▸ (1) Spark Jobs

```
'The cat chased the rats and found them hiding in the star',
'The silent night echoed with the melody of lentis star',
'His disrespectful remarks angered her, but when he continued to provoke her she became truly enraged',
'The gardener used a brush to clear the dirt from the shrub',
'The only thing he could see in the darkness of the night was the faint glow of distant stars',
'She caught ten fish with her net at the lake yesterday',
'He used a pin to fasten the paper being careful not to nip his fingers',
'She was determined to finish the race even though she felt under timed due to the unexpected delay',
'The ancient evergreen tree stood tall and serene unaware of the revenage stirring in the shadows',
'During their conversation about wildlife they discussed the importance of conservation efforts to protect endangered species',
"As she gazed out at the serene landscape she couldn't help but feel a sneer of contempt curling on her lips",
'While the astronomer studied the constellations he encountered a mysterious legend of an ancient moonstarrer who navigated the cosmos',
'He wiped the crumbs from his beard after enjoying a hearty meal of warm bread',
'After a long break, the baker returned to the kitchen to resume baking delicious treats',
'As she applied lipstick she felt her fingers slip causing a smudge on her lips',
'In the race for success its important to also take care of your well being',
'The angel watched over the earth from a heavenly angle offering guidance and protection to those below',
'The astronaut took careful paces as he floated through the vast expanse of space',
'As he measured the length of the board he noticed a small insect crawling along his chin causing him to flinch an inch',
```

3. **Split Words**:

- The flatMap transformation is applied to the RDD a_words2.
- Inside the flatMap function, each line of the text file is split into individual words using the space character as the delimiter.
- The resulting RDD, a_words_rdd2, contains all the words from the file.

```
1  a_words_rdd2=a_words2.flatMap(lambda x:x.split(" "))
2  a_words_rdd2.collect()
```

▸ (1) Spark Jobs

```
'length',
'of',
'the',
'board',
'he',
'noticed',
'a',
'small',
'insect',
'crawling',
'along',
'his',
'chin',
'causing',
'him',
'to',
'flinch',
'an',
```

4. **Sort Letters**:

- For each word in a_words_rdd2, the code sorts its letters alphabetically.
- The map transformation is used to create a new RDD, a_words_sort1, where each element is a tuple.
- The first element of the tuple is the sorted letters (formed by joining the letters), and the second element is a list containing the original word.
- This step prepares the data for grouping anagrams.

```
a_words_sort2 = a_words_rdd2.map(lambda word: ("".join(sorted([letter for letter in word])), [word]))
a_words_sort2.collect()
```

▸ (1) Spark Jobs

```
('btu', ['but']),
('ehnw', ['when']),
('eh', ['he']),
('cdeinnotu', ['continued']),
('ot', ['to']),
('ekooprv', ['provoke']),
('ehr', ['her']),
('ehs', ['she']),
('abceem', ['became']),
('lrtuy', ['truly']),
('adeegnr', ['enraged']),
('Teh', ['The']),
('adeegnrr', ['gardener']),
('desu', ['used']),
('a', ['a']),
('bhrsu', ['brush']),
('ot', ['to']),
('acelr', ['clear']),
('eht', ['the']),
```

5. **Group Anagrams**:

- The reduceByKey transformation is applied to a_words_sort2.
- It groups the words with the same sorted letters (anagrams) together.
- The resulting RDD, a_words_reduce2, contains pairs where the key is the sorted letters, and the value is a list of anagrams.

```
a_words_reduce2 = a_words_sort2.reduceByKey(lambda word1, word2: word1 + word2)
a_words_reduce2.collect()
```

(1) Spark Jobs

```
('aerstt', ['treats']),
('adeilpp', ['applied']),
('acginsu', ['causing', 'causing']),
('degmsu', ['smudge']),
('In', ['In']),
('for', ['for']),
('oruy', ['your']),
('aegln', ['angel', 'angle']),
('eorv', ['over']),
('acdeginu', ['guidance']),
('ehost', ['those']),
('aanorsttu', ['astronaut']),
('aceps', ['paces', 'space']),
('adeflot', ['floated']),
('ghhortu', ['through']),
('aeenpsx', ['expanse']),
('abdor', ['board']),
('cdeinot', ['noticed']),
('ceinst', ['insect']),
('aglno', ['along']),
```

6. **Print Anagrams**:
   - The code iterates through the key-value pairs in a_words_reduce2.
   - For each key (sorted letters), it prints the anagrams associated with that key.
   - The output displays anagrams grouped by their sorted letters.

```
for key, anagrams in sorted(a_words_reduce2.collect()):
    print(f"{key}: {anagrams}")

```

(1) Spark Jobs

```
ehost: ['those']
ehr: ['her', 'her', 'her', 'her', 'her']
ehs: ['she', 'she', 'she', 'she', 'she', 'she']
eht: ['the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the',
'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the', 'the']
ehty: ['they', 'they']
eilnst: ['silent', 'listen', 'silent', 'lentis']
eimorsstuy: ['mysterious']
ekooprv: ['provoke']
ellw: ['well']
ent: ['ten', 'net']
eorv: ['over']
ety: ['yet']
fhiins: ['finish']
fhis: ['fish']
fmor: ['from', 'from', 'from']
fo: ['of', 'of', 'of', 'of', 'of', 'of', 'of', 'of', 'of', 'of', 'of', 'of']
for: ['for']
```

Anagram-Inspired Password Generator

Anagrams involve rearranging letters to create new words or phrases. In our password generator, we'll use this concept to create unique and memorable passwords. Here's how we can do it:

- **Word-Based Passwords**:
  - Instead of random characters, let's generate passwords using real words.
  - We'll select random words from a predefined list to create memorable passphrases.
- **Customizable Length**:
  - Users can choose the number of words in their password.
  - Longer passwords are generally more secure.
- **Avoid Ambiguity**:
  - We'll ensure that the selected words are clear and unambiguous.
  - No room for misinterpretation!

Conditions in code:

1. **Input**:
    - You prompt the user to input a name (presumably their own).
2. **Processing**:
    - You convert the input name into a list of characters.
    - You randomly select an uppercase character from the list.
    - You generate two random numeric digits.
    - You randomly choose a special character from the set ["!", "@", "#", "$"].
    - You construct the password by replacing the randomly chosen uppercase letter with its uppercase version, followed by the special character and the numeric digits.
3. **Example**:
    - Suppose the user inputs the name “alice”.
    - The uppercase letter randomly chosen from “alice” could be “a”.
    - Two random numeric digits are generated (e.g., “2” and “4”).
    - A special character is randomly chosen (e.g., “@”).
    - The password could be “Ali@24”.
4. **Permutations**:
    - You create a set of all possible permutations of the characters in the password.
    - For example, if the password is “Ali@24”, the permutations might include “Ali24@”, “A2li@4”, “2@lAi4”, etc.
5. **Output**:
    - You print the set of possible passwords (permutations).
    - You also display the total number of unique passwords generated.

```python
import itertools
from itertools import permutations
import random
def generate_password(name):
  if len(name) < 4:
    print("Enter the correct name")
  else:
    name = name[:4]
    listing=list(name)
    character =["!","@","#","$"]
    upper_case = random.choice(listing)
    numeric=""
    for i in range(2):
        num=random.choice([str(i) for i in range(5)])
        numeric +=num
    special_character =random.choice(character)
    password=(name.replace(upper_case,upper_case.upper()))+special_character+numeric
    return password
name = input()
result= generate_password(name)
if result:
  possible_password=set([''.join(prem) for prem in permutations(result)])
  print(possible_password)
  print(name,len(list(possible_password)))
```

Aju

```
Enter the correct name
```

Nasrin

```
'$2sNAr4', '42ArN$s', '$2As4Nr', 'rAs24$N', 'r$N2A4s', 'N2r4sA$', '2NAr4$s', 'r$AN24s', 'NrsA42$', '4A$2Nrs', '2rAsN4$', '4N2rsA$', '$As4rN
2', 's4AN2r$', '24AsN$r', 'r24sNA$', '$4A2Nrs', 'A$4Ns2r', 'r2$sA4N', 's42$ArN', 's$42ArN', 'r2A4Ns$', 'A2Nsr$4', 'r4$s2NA', 's2rA4$N', '$sAN
42r', '4AsN$r2', '2rN$sA4', 'As2N4$r', 'NAs42$r', 'AN2s$r4', 'Asr2$4N', 'r2s$AN4', 'A$2srN4', '2rsNA$4', 'AN$s24r', '2rNs$4A', '$r4sA2N', '4A
Nsr2$', 'srNA2$4', 'rs$A24N', 'A42Nr$s', '$ArN2s4', '$4NAr2s', 'r2s4$NA', 's2NA$r4', '24NrA$s', 'sNr4$A2', 'A4r2s$N', 'A42$srN', '4s$ArN2',
'24A$Nsr', 'NA2r4$s', 'A2$Nr4s', '$s2Nr4A', 'sr2N4$A', '$sr42AN', 'N4rsA2$', '4A$sN2r', '2NsAr4$', '4r2N$sA', 'srA2$4N', 'rNA4$2s', 'r$2N4A
s', 's24A$Nr', '2s4r$NA', '4s2r$NA', 'As42$rN', 'r4$2ANs', '4Nsr2$A', 'r4A2$Ns', '2sr$N4A', '2sNA$r4', 'NA2$rs4', 'ArN2$s4', 'ANs$42r', '4As
$2Nr', 'r$42sAN', 'sA$N42r', 'N24$Asr', 'rA$2N4s', 'sA$42Nr', '4NAr$s2', '4sNA2$r', 'rsNA24$', 'As$2rN4', 'r24N$sA', 'r$4sAN2', '$srN2A4', '2
NrA4s$', '4$rsN2A', '2NAr$4s', '$sAr4N2', '2s4A$Nr', 'sr$A42N', 'A4Nr$2s', '$s42NrA', 'r2s4NA$', '4A2s$Nr', 's2AN4$r', 'srN$24A', '$NA2sr4',
'$24ArNs', 's2N$4Ar', 'rN4$As2', 'N2r$4As', 'N2$r4As', 'r24A$sN', 'Ar24N$s', 'sNrA4$2', 's24$ArN', '$2ArsN4', 'sN4A$r2', 'Ns2rA4$', 'sA$24r
N', 'rN24A$s', '$N24rsA', 'Ns4A2r$', 'A4N$sr2', 's2N4Ar$', '$A4r2sN', 'sANr4$2', '$42sANr', '4$rA2Ns', '2$A4rsN', 'sN4A$2r', 'ANs24r$', '2sNr
$A4', 'Nr2A$s4', '4$2AsrN', 'N24$srA', 'ANrs$24', '24A$srN', '4rsN$2A', '$4NrA2s', '24sNAr$', 'NAs4r$2', 'rA$2Ns4', 'AsN2r4$', '$NsAr42', 'sN
r2A4$', '24sNr$A', '4N$r2sA', 'sN2rA4$', '4r2$AsN', '4As2$rN', 'A4sr$N2', 's$N4Ar2', 's4A2Nr$', 's$42NrA', 'N2$sr4A', 'N$sr24A', '4rsN2A$',
'ArN$2s4', 'rA$sN24', 'A4srN$2', 'AsrN42$', 'r4sN2$A', '4AN2$sr', 'sN2Ar$4', 'ArNs4$2', 'N$2rsA4', 'r2s4N$A', '42$sNAr', '2rNsA$4', 'A$sN4r
2', 's$ANr24', 'sN2rA$4', '4ANs$r2', 'r42AN$s', 'AN4$rs2', '42N$Ars', 'N$sr42A', 'A24$sNr', 'AN2rs4$', 'As$2Nr4', '$Ar2s4N', 'A2rs$N4', '2Nrs
A4$', '4s$2NrA', '4$Ars2N', '2N4s$rA', '$ArNs42', 's$N24rA', 'A4r$sN2', 'sA2$4Nr', '42s$NrA', '$4Nr2sA', 'A$2r4Ns', 'N2s4rA$', 'rNsA$24', '4r
s$A2N', '2sr4$AN', 'sNA4$2r', 'A2Nsr4$', 's2r4$AN', 'A$r42sN', '$2Asr4N', 'N$sA24r', 'sN4rA2$', 'sNr4$2A', 'A42srN$', 'Ar42Ns$', '2s$4NAr',
'A2$N4rs', 'Ars$2N4', 's$rN4A2', '2$rAs4N', '24$NArs', 'rN42s$A', '42s$rAN', 'Ars$4N2', '$N2Ars4', '$srN4A2', 'rs2N4A$', '$A2N4rs', 'Ns2r4A
$', '4$2ANrs', '$AN2sr4', 's2AN$4r', 'rA$Ns42', '2N$r4sA', '4NAr$2s', '$sr24AN', '$s2ArN4', 'N2As$4r', '2$NrAs4', 'NA$2rs4', '$NAsr24', '$r2N
As4', 'rA$s2N4', 'rA$42Ns', 'A4$N2rs', '2$4ANrs', '4$rsA2N', '24s$NrA', '4rA2Ns$', '4rA2s$N', '4ArN2$s', '4Nr$2sA', 's4$2NAr', 'AN4r2s$', '$s
2AN4r', 's$4Ar2N', '$4NA2sr'}
Nasrin 5040
```

Results

Upon testing our solution with various word datasets, we achieve accurate anagram detection. The algorithm scales well, making it suitable for real-world applications.

Conclusion

Detecting anagrams is not only intellectually stimulating but also practically useful. Our project provides an elegant solution to this classic problem, demonstrating the power of data structures and algorithms in natural language processing.