Spark – wprowadzenie

W ramach tego tutorialu zapoznamy się ze sposobami tworzenia i uruchamiania aplikacji na platformie Spark.

W ramach różnorodnych projektów przeprowadzane są analizy statystyczne dzieł znanych twórców literatury takich jak chociażby Shakespeare. Wyniki tych analiz publikowane są następnie na stronach WWW. Przykładowo:

- `https://www.opensourceshakespeare.org/stats/`
- `http://www.shicho.net/38/stats/38statistics.php`
- `https://www.sporcle.com/games/Sforzando/most-common-words-in-shakespeare/results`

My zajmiemy się podobnym problemem, ale w stosunku do opowieści dotyczących Sherlocka Holmes'a pisarza Arthura Conan Doyle'a

Dane wykorzystane w ramach tutorialu pochodzą ze strony `https://sherlock-holm.es/ascii`

Tutorial będzie obejmował następujące części:

- Pobranie źródłowych danych i załadowanie ich do systemu plików HDFS
- Dokonanie analizy zawartości tych danych za pomocą powłoki *pyspark*, która jest środowiskiem REPL dla języka Python. W ten sposób dokonamy prototypowania naszego programu.
- Napisanie programu w Pythonie wykorzystując do tego celu PyCharm IDE i uruchomienie go lokalnie
- Uruchomienie tak przetestowanego programu na klastrze (np. w chmurze)

Pobranie i załadowanie danych źródłowych

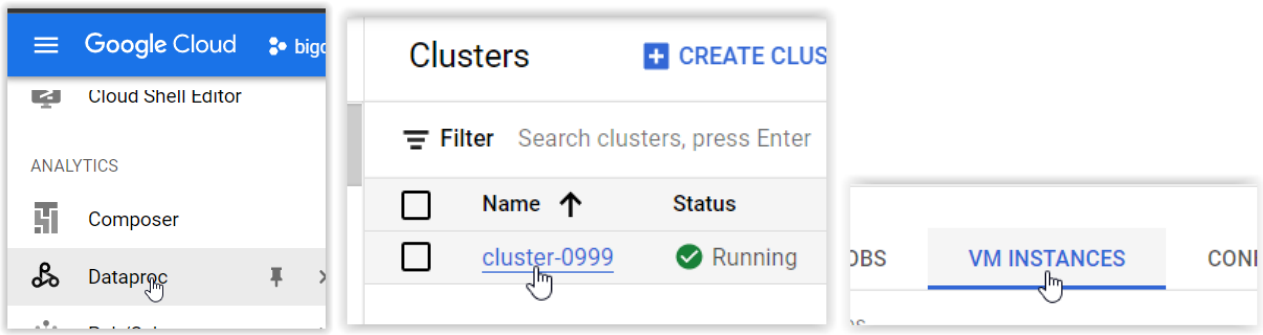
1. W naszych ćwiczeniach ze Sparka będziemy korzystali z klastra dostępnego w ramach środowiska *Dataproc* w ramach *Google Cloud Platform*.

2. Wejdź na stronę konsoli platformy *Google Cloud Platform* `https://console.cloud.google.com/`

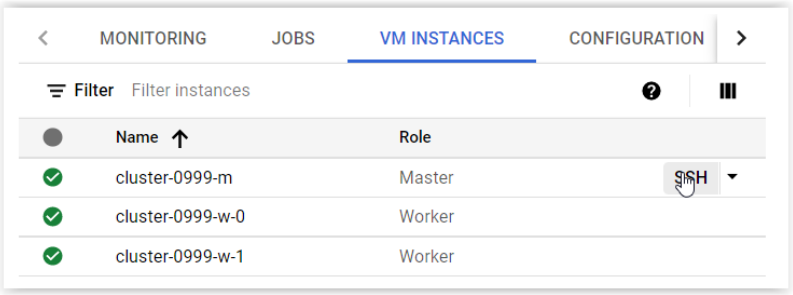
3. Otwórz *Cloud Shell*, a następnie uruchom klaster *Dataproc* za pomocą poniższego polecenia.

```
gcloud dataproc clusters create ${CLUSTER_NAME} \
  --enable-component-gateway --bucket ${BUCKET_NAME} \
  --region ${REGION} --subnet default --zone ${ZONE} \
  --master-machine-type n1-standard-4 --master-boot-disk-size 50 \
  --num-workers 2 --worker-machine-type n1-standard-2 --worker-boot-disk-size 50 \
  --image-version 2.1-debian11 --optional-components JUPYTER \
  --project ${PROJECT_ID} --max-age=3h
```

4. Otwórz za pomocą menu i pozycji *Dataproc* listę klastrów. Wejdź do szczegółów Twojego klastra. Przejdź na zakładkę z listą wirtualnych maszyn, które go tworzą.



5. Uruchom terminal SSH dla węzła master we uruchomionym przed klastrze



6. Korzystając z terminala SSH pobierz dokument zawierający komplet dzieł Arthura Conan Doyle'a dotyczący Sherlocka Holmes'a

```
wget https://sherlock-holm.es/stories/plain-text/cano.txt
```

Jeśli powyższe polecenie nie działa, możesz ten dokument uzyskać ze standardowego miejsca

```
wget https://jankiewicz.pl/bigdata/spark/cano.txt
```

7. Spark może działać w różnych trybach, w tym na klastrze Hadoop np. jako aplikacja YARN. Aby każdy węzeł klastra miał dostęp do źródłowych danych, nie mogą one pozostać w lokalnym systemie plików maszyny master. Musimy je skopiować do miejsca dostępnego z każdego węzła klastra, np. do systemu plików HDFS. Za pomocą poniższego utworzysz swój katalog domowy (o ile nie istnieje) w systemie plików HDFS.

```
hadoop fs -mkdir -p .
```

8. Załaduj do systemu HDFS nasz plik ze źródłowymi danymi.

```
hadoop fs -copyFromLocal cano.txt .
```

9. Sprawdź czy plik znalazł się na swoim miejscu

```
hadoop fs -ls
```

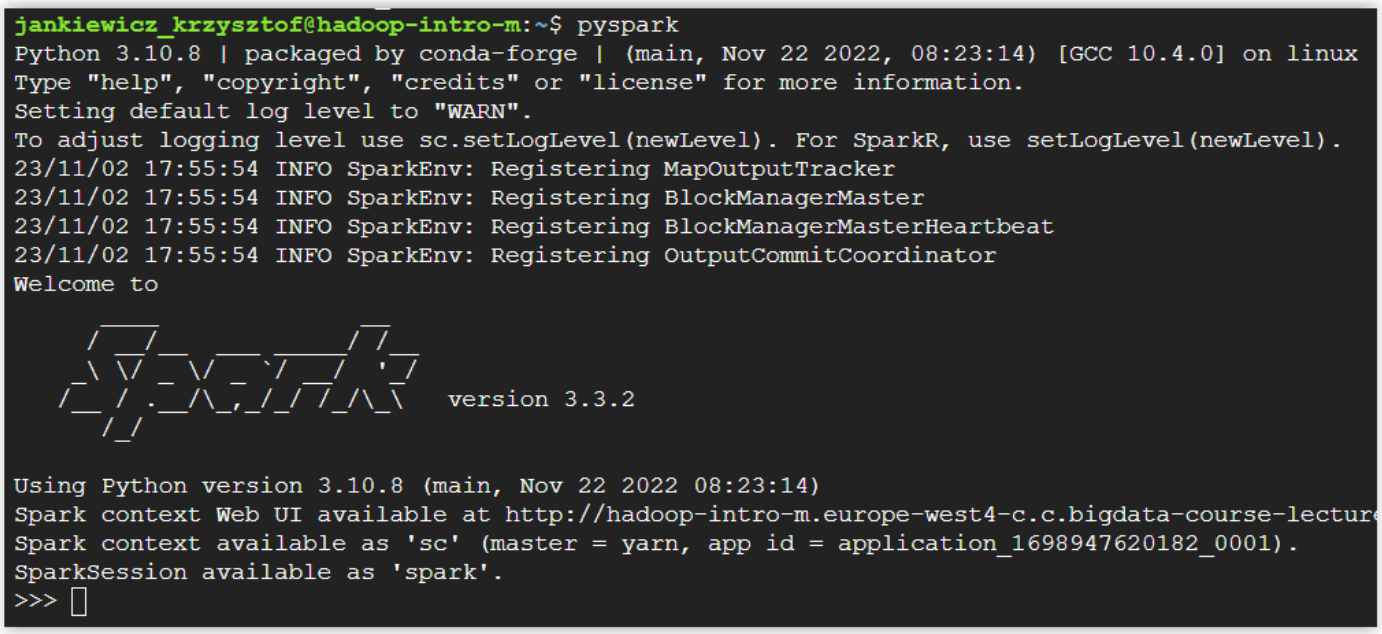
```
jankiewicz_krzysztof@hadoop-intro-m:~$ hadoop fs -ls
Found 1 items
-rw-r--r--   2 jankiewicz_krzysztof hadoop    3868223 2023-11-02 17:55 cano.txt
```

Analiza liczby słów przy wykorzystaniu *pyspark*

Zapoznamy się teraz z narzędziem *pyspark*, dzięki któremu dokonamy prototypowania naszej aplikacji zliczającej słowa, stosując w ten sposób z techniki programowania sterowanego eksperymentami.

10. Uruchom powłokę `pyspark`.

```
pyspark
```



Zwróć uwagę (zapamiętaj lub zapisz) na:

- wersję Sparka,
- wersję Pythona,
- fakt utworzenia zmiennych kontekstu,
- a także na tryb uruchomienia klastra Sparka.

Jak widzisz, Spark został uruchomiony jako aplikacja YARN. Jeśli chcesz, możesz wyjść z aplikacji *pyspark* za pomocą polecenia `quit()`, a następnie uruchomić ją ponownie uruchamiając Sparka lokalnie w wątkach JVM.

```
pyspark --master local[2]
```

W przypadku ograniczonych zasobów i uruchamiania po prostu różnych prostych testów na małym zbiorze danych, oraz w przypadku uruchamiania Sparka bez dostępu do klastra, takie rozwiązanie jest bardzo dobrym pomysłem.

11. Na początku wczytamy zawartość załadowanego pliku do zmiennej `cano`.

```
cano = sc.textFile("cano.txt")
```

W związku z tym, że w ramach tego warsztatu będziemy korzystali z RDD API warto w razie potrzeby zaglądać do dokumentacji celem zapoznania się z dostępnymi w tym API metodami

```
https://spark.apache.org/docs/latest/api/python/reference/pyspark.html#rdd-apis
```

12. W przypadku korzystania z języka Scala doszłoby automatycznie do określenia typu utworzonej zmiennej (typy w Scali są wyznaczane na etapie kompilacji). Dla przykładu dowiedzielibyśmy się, że zmienna `cano` jest typu `RDD[String]` – czyli jest „rozproszoną kolekcją" ciągów znaków (linii ze źródłowego pliku).
    Tu niestety albo wykorzystamy naszą wiedzę na temat poszczególnych obiektów, metod i funkcji, albo skorzystamy z funkcji `type` której przekażemy odpowiednie argumenty dokonując inspekcji przetwarzanych obiektów. Dla przykładu:

```
type(cano)
type(cano.take(1))
type(cano.first())
```

```
>>> cano = sc.textFile("cano.txt")
>>> type(cano)
<class 'pyspark.rdd.RDD'>
>>> type(cano.take(1))
<class 'list'>
>>> type(cano.first())
<class 'str'>
```

    Dwa ostatnie polecenia wykorzystują akcje, a to oznacza uruchomienie zadania w klastrze Sparka.

13. Następnie utworzymy zmienną `canoTokenized` zawierającą zbiór słów (`String`) – każde słowo będzie oddzielnym elementem kolekcji obiektów RDD - `RDD[String]`.

```
import re
canoTokenized = cano.flatMap(lambda line: re.split(r'\W+', line))
```

14. Aby policzyć wystąpienia pogrupujemy nasz RDD w oparciu o zawartość tych znajdujących się w nim elementów (słów).
    Jest to rozwiązanie fatalne ze względu na wydajność. Końcowy efekt można uzyskać w znacznie lepszy sposób – jeśli będziemy omawiali szczegółowo RDD API, wówczas będziemy o tym dyskutowali nieco później.

```
canoWordGroups = canoTokenized.groupBy(lambda word: word)
```

15. Sprawdźmy z czym mamy do czynienia

```
f = canoWordGroups.first()
type(f)
len(f)
type(f[0])
type(f[1])
```

```
>>> f = canoWordGroups.first()
>>> type(f)
<class 'tuple'>
>>> len(f)
2
>>> type(f[0])
<class 'str'>
>>> type(f[1])
<class 'pyspark.resultiterable.ResultIterable'>
```

    A zatem mamy do czynienia z RDD krotek (`tuple`), a dokładnie par, w którym każda para jest grupą. Pierwszy element tej pary jest identyfikatorem grupy (słowem) – wyrażeniem, po którym dokonywaliśmy grupowania. Drugim elementem każdej pary jest obiekt `ResultIterable` zawierający wszystkie pogrupowane elementy (te same słowa – wszystkie ich wystąpienia w grupowanym zbiorze danych).

16. A zatem pozostaje nam zliczenie, dla każdej grupy, elementów umieszczonych w drugim elemencie każdej z par. Zaglądnijmy jakie metody oferuje klasa `ResultIterable`
`https://github.com/apache/spark/blob/master/python/pyspark/resultiterable.py`
A zatem:

```
canoWordCounts = canoWordGroups.map(lambda pair: (pair[0], len(pair[1])))
```

17. W zmiennej `canoWordCounts` mamy potrzebne nam wyniki.
    a. Spróbujmy się na początek dowiedzieć jakie 10 słów występowały najczęściej.

```
topWordList = canoWordCounts.sortBy(lambda pair: pair[1], False).take(10)
for wordPair in topWordList:
    print(wordPair)
```

    b. Następnie, dowiedzmy się ile razy pojawiło się słowo `Watson`

```
canoWordCounts.filter(lambda pair: pair[0] == "Watson").collect()
```

    c. A ile razy słowo `Moriarty`

```
canoWordCounts.filter(lambda pair: pair[0] == "Moriarty").collect()
```

    d. Samodzielnie postaraj się wyznaczyć 10 najczęstszych słów, których długość przekracza 4 znaki. Nie wiesz jak to zrobić? Zaglądnij do RDD API.

18. Na zakończenie pobierz historię wykonywanych poleceń. Będzie przydatna kiedy będziemy chcieli zrobić z niej ostateczną wersję naszej aplikacji.

```
import readline
for i in range(readline.get_current_history_length()):
    print (readline.get_history_item(i + 1))
```

19. Zamknij pracę z programem `pyspark` za pomocą instrukcji `quit()`

Powyższe prototypowanie programu wcale nie musi być dokonywane na klastrze Hadoopa. Uruchomić Sparka opartego na wątkach w JVM można prawie w każdym środowisku.

PyCharm, uruchamianie programu lokalnie

Mając przetestowane nasze koncepcje, postaramy się teraz zebrać zdobyte doświadczenia do napisania programu, który dla zadanego wzorca, będzie podawał 10 najczęstszych słów, które do tego wzorca pasują.

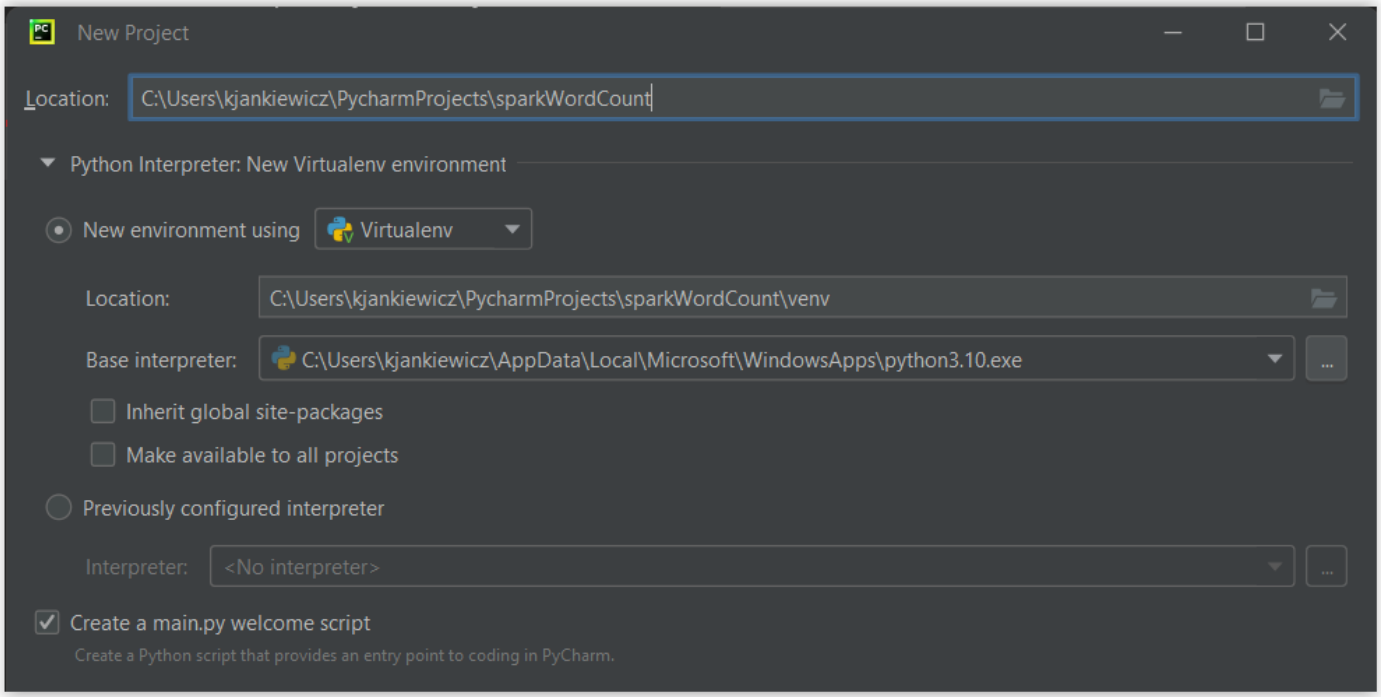
Nie będziemy jednak przygotowywali tego programu na klastrze Sparka. Proces tworzenia oprogramowania może być długotrwały. Błędy, poprawki, wielokrotne uruchamianie kolejnych wersji programu na komercyjnym klastrze mogłoby nas narazić na niepotrzebne koszty.
Dlatego przygotujemy nasz program i uruchomimy go lokalnie, na "przykładowym" zbiorze danych (który z reguły powinien być reprezentatywnym w kontekście zawartości (nie koniecznie rozmiaru) podzbiorem danych docelowych.

Na początku określmy co powinniśmy mieć już zainstalowane:

- JDK w wersji 11 (np. `https://adoptium.net/temurin/releases`)
- PyCharm w wersji Community (`https://www.jetbrains.com/pycharm/download/`)
- Python (np. 3.10)
- Spark w wersji 3.3.3 (`https://spark.apache.org/downloads.html`)
- winutlis dla wersji Hadoop 3.3.5 (`https://github.com/cdarlint/winutils`)

Wersję Pythona oraz Sparka jaką potrzebujemy, aby być kompatybilnym z klastrem Dataproc, można było zobaczyć przy uruchomieniu `pyspark`

20. Uruchom PyCharm. Utwórz nowy projekt o nazwie `sparkWordCount`.



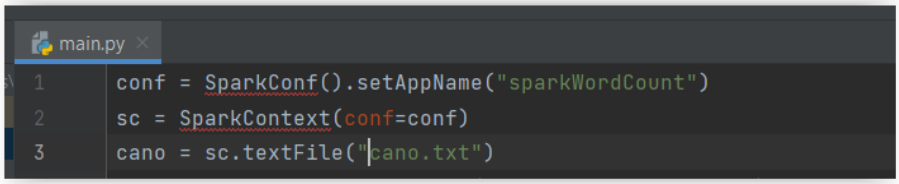
21. Wymień całą zawartość pliku `main.py` na poniższy kod zgodny z naszym prototypem aplikacji

```
cano = sc.textFile("cano.txt")
canoTokenized = cano.flatMap(lambda line: re.split(r'\W+', line))
canoWordGroups = canoTokenized.groupBy(lambda word: word)
canoWordCounts = canoWordGroups.map(lambda pair: (pair[0], len(pair[1])))
```

22. W naszym programie brakuje kilku elementów:
    a. Utworzenie obiektu kontekstu
    b. Obsługi pierwszego parametru wskazującego źródłowe dane
    c. Obsługi drugiego parametru dostarczającego poszukiwany wzorzec

23. Zacznijmy od kontekstu. Dodaj przed pierwszą linią następujący kod tworzący obiekt kontekstu.

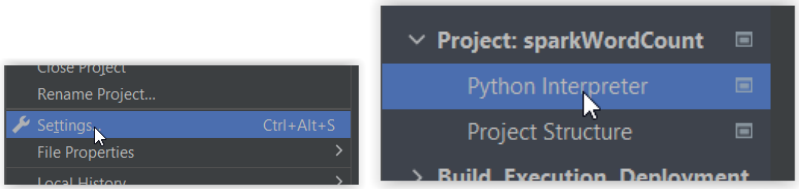
```
conf = SparkConf().setAppName("sparkWordCount")
sc = SparkContext(conf=conf)
```



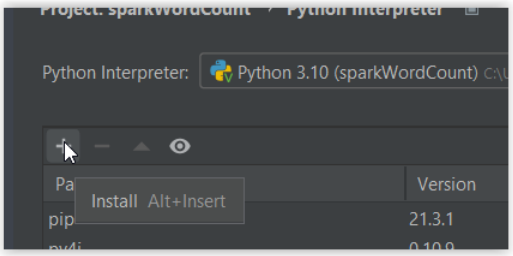
24. Jak widać ewidentnie brakuje nam klas pochodzących z pakietów `pyspark`. Dodaj zatem kolejną linię importującą brakujące elementy układanki

```
from pyspark import SparkConf, SparkContext
```

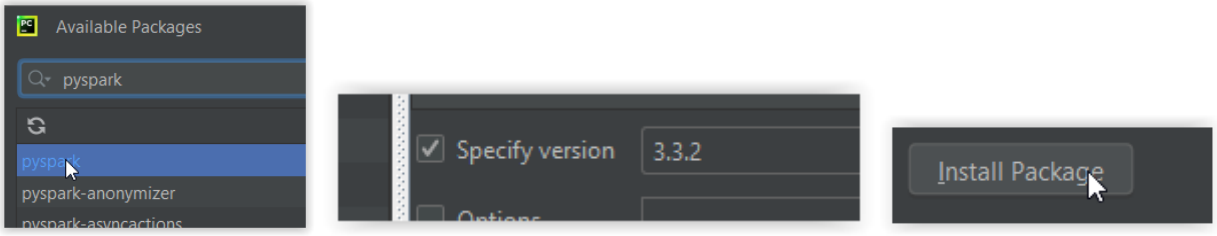
25. Ponownie mamy problem. Tym razem ze względu na brakujący pakiet `pyspark`, w którym nasze klasy występują. Aby uniknąć problemów, wersja `pysparka` powinna być maksymalnie zbliżona (do pierwszego miejsca po przecinku) do wersji Sparka, z której korzystamy. Aby to zagwarantować z menu *File* wybierz pozycję *Settings*. Następnie wybierz ustawienia dotyczące interpreterów Pythona



26. Wybierz przycisk + aby zainstalować brakujące pakiety



27. Znajdź wśród dostępnych pakietów pyspark, koniecznie skoryguj wersję, a następnie wybierz przycisk *Install Package*



28. Zamknij listę pakietów do zainstalowania. Zamknij ustawienia. Problemy z importem powinny po zainstalowaniu obu pakietów zaniknąć.

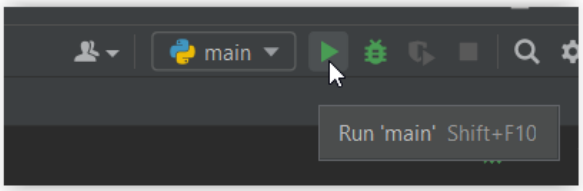
29. Powróć do edycji naszego programu. W drugiej linii kodu dodaj jeszcze brakujący import

```
import re
```

30. Pobierz także plik `cano.txt` na lokalny dysk, a następnie podmień wskazanie na ten plik, aby odpowiadało ono rzeczywistej lokalizacji. Dla przykładu

```
cano = sc.textFile("C:\\Users\\kjankiewicz\\Downloads\\cano.txt")
```

31. Uruchom naszą aplikację



32. Jeśli wszystko poszło dobrze to… no właśnie nie ma żadnych wyników.

33. To oczywiste. Nie ma żadnej akcji (użyliśmy tylko transformacji). Ponadto brak parametryzacji naszego programu. Zacznijmy od drugiego parametru. Na początku dodaj kolejny import oraz sprawdzenie parametrów i przypisanie ich do zmiennych

```
import sys

if len(sys.argv) < 2:
    print("Wymagane dwa parametry: dane źródłowe i szukany wzorzec")
    exit(0)

sourceData = sys.argv[1]
pattern = sys.argv[2]
```

34. Dodaj teraz obsługę zmiennej `pattern`. Poniżej znajdziesz funkcję anonimową, z której możesz skorzystać. Zastanów się jakiej metody obiektu RDD użyć, oraz gdzie ją dodać.

```
lambda word: bool(re.match(pattern, word))
```

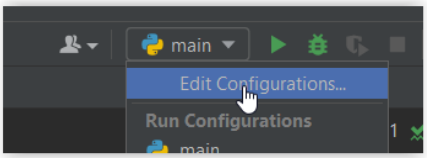
35. Podmień ścieżkę do pliku `cano.txt` na pierwszą z dodanych zmiennych

```
cano = sc.textFile(sourceData)
```

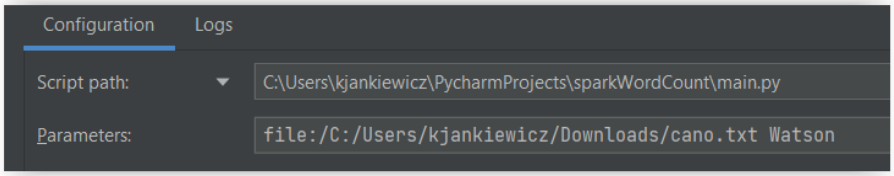
36. Użyjmy teraz akcji, aby móc zobaczyć jakieś rezultaty.

```
topWordList = canoWordCounts.sortBy(lambda pair: pair[1], False).take(10)
for wordPair in topWordList:
    print(wordPair)
```

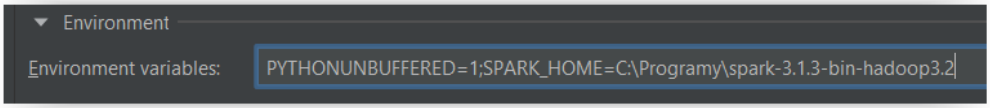
37. Aby uruchomić nasz program z parametrami dokonaj edycji konfiguracji uruchamiającej naszą aplikację



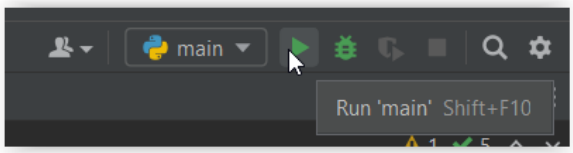
38. Wprowadź oba parametry naszego programu. Nie zatwierdzaj jeszcze zmian.

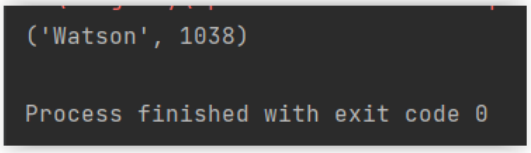


39. Aby można było uruchamiać programy Sparka lokalnie musimy mieć ustawioną zmienną środowiskową SPARK_HOME wskazującą katalog domowy Sparka. Jeśli masz ją już ustawioną w systemie, nic nie musisz robić. Jeśli nie, zlokalizuj katalog, w którym został zainstalowany/rozpakowany Spark, a następnie zdefiniuj odpowiednią zmienną w konfiguracji uruchamiającej nasz program, dla przykładu: `SPARK_HOME=C:\Programy\spark-3.1.3-bin-hadoop3.2`



40. Uruchom nasz program ponownie

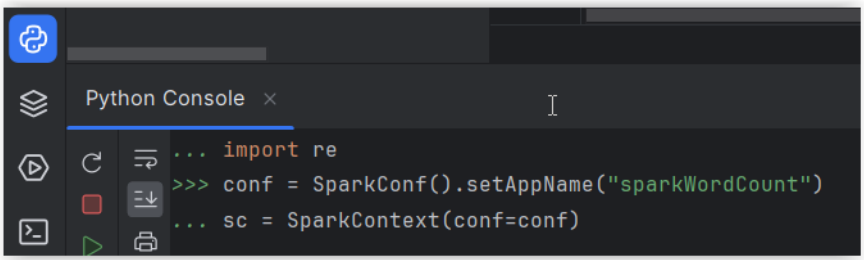




41. Zmień drugi parametr na `a.*a` i uruchom program jeszcze raz. Jeśli wygląda to tak jak obok, to chyba nasz program działa poprawnie.

```
('away', 537)
('again', 480)
('against', 377)
('always', 307)
('already', 270)
('appeared', 169)
('affair', 124)
('appearance', 111)
('afterwards', 104)
('aware', 101)
```

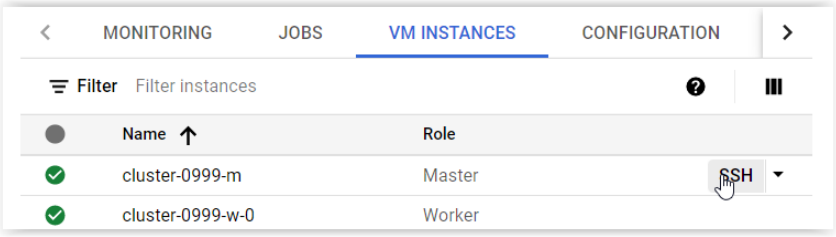
42. Zanim przejdziemy dalej, zwróć uwagę, że prototypowanie swojego programu możesz także wykorzystując konsolę Pythona w *PyCharm*.



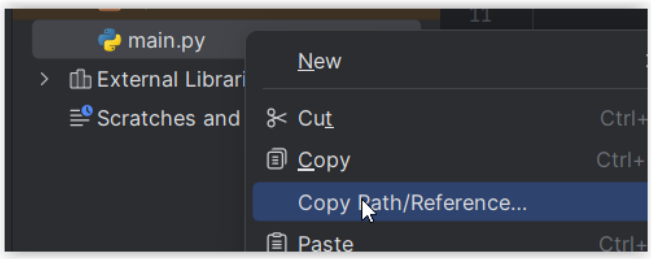
Uruchomienie programu zdalnie w klastrze

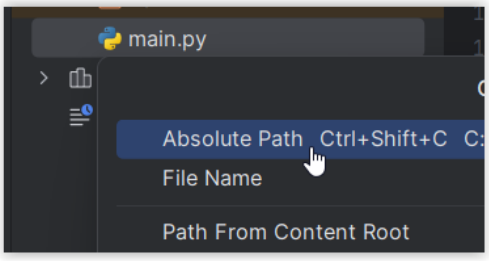
Tak przygotowany program możemy uruchomić w klastrze. Po tych wszystkich wykonanych przez nas działaniach istnieje duża szansa, że uruchomienie programu i tam zakończy się powodzeniem.

43. Aby uruchomić nasz program na naszym klastrze musimy go tam dostarczyć. Można do tego celu wykorzystać funkcjonalność terminala SSH maszyny master naszego klastra.
    Otwórz korzystając z konsoli GCP terminal SSH do maszyny master

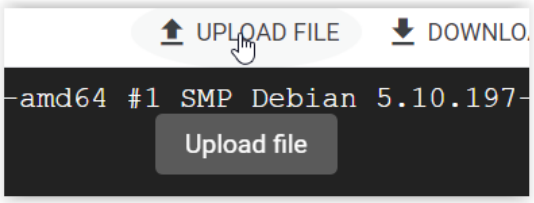


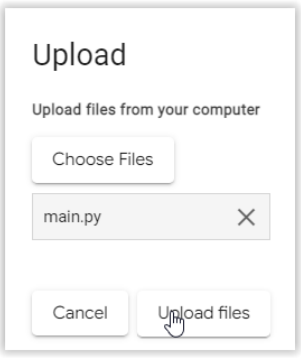
44. Skopiuj ścieżkę prowadzącą do Twojego pliku



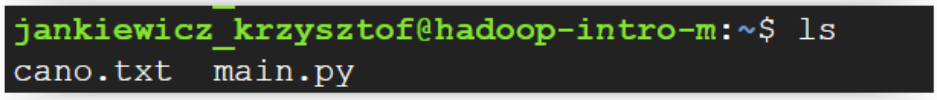


45. A następnie korzystając z opcji *Upload File*, dostępnej w terminalu SSH, załaduj plik `main.py` do systemu plików na serwerze master naszego klastra.





46. Sprawdź czy jest on dostępny w Twoim katalogu domowym



47. Korzystając z poniższego polecenia uruchomimy nasz program

```
spark-submit --master yarn \
   --driver-memory 3g --executor-memory 3g \
   --executor-cores 1 --num-executors 2 \
   main.py cano.txt b.*b
```

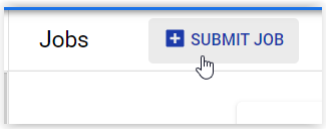
48. Jeśli wszystko pójdzie dobrze, wyniki pojawią się w ciągu kilkunastu sekund.

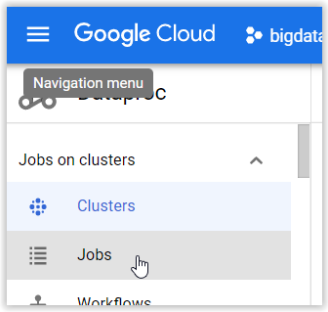
```
23/11/03 07:57:49 INFO FileInputFormat: Total input files to process : 1
('baby', 18)
('baboon', 6)
('busybody', 5)
('brambles', 3)
('bribed', 3)
('bulbous', 2)
('bramble', 2)
('buckboards', 2)
('bob', 2)
('babbled', 1)
```

49. Oczywiście „na produkcji" nie zawsze korzysta się z narzędzia `spark-submit` do wykonywania aplikacji Sparka. Często wykorzystywane są one w bardziej złożonych definicjach przepływów danych orkiestrowanych przez narzędzia takie jak *Apache Airflow*, *Oozie* czy *Workflow* (na platformie GCP).
W celu uruchomienia programu Sparka korzysta się także czasami z interfejsów zewnętrznych narzędzi, np. dostępnych na platformach chmurowych. Na zakończenie uruchomimy nasz program korzystając z interfejsu konsoli GCP.
W terminalu SSH przekopiuj swój plik `main.py` do katalogu `/tmp` a także plik `cano.txt` do systemu plików HDFS do katalogu `/tmp` (zbieżność nazw przypadkowa). Dzięki temu nie będziemy mieli problemu z wskazywaniem ścieżek do tych plików.

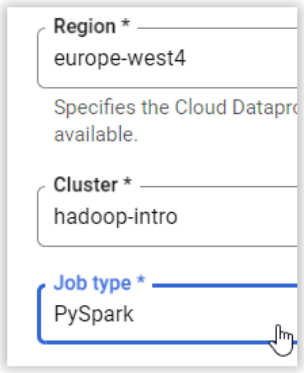
```
cp main.py /tmp/
hadoop fs -copyFromLocal cano.txt /tmp/
```

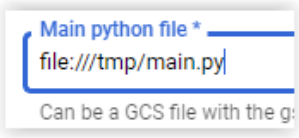
50. W menu nawigacji na stronie dotyczącej klastrów *Dataproc* wybierz pozycję *Jobs*, a następnie przycisk *Submit Job*.

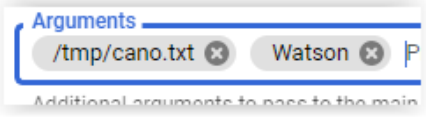




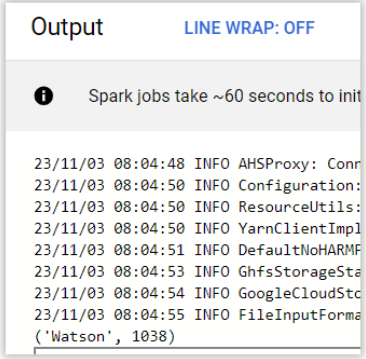
51. Ustaw właściwy region, wskaż Twój klaster, jako miejsce uruchomienia a także typ zadania. Jako główny plik zawierający skrypt pythona wskaż Twój program `file:///tmp/main.py`. Jako argumenty wywołania programu podaj nazwę pliku `/tmp/cano.txt` ze źródłowymi danymi i szukany wzorzec np. `Watson`. Uwaga: po wprowadzeniu wartości każdego argumentu użyj klawisza Enter aby oddzielić ją od wartości następnej. Zakończ definiowanie zadania za pomocą przycisku *Submit*.







52. Po chwili nasz wynik pojawi się w interfejsie narzędzia.



To kończy nasze pierwsze kroki w tworzeniu i uruchamianiu programów Sparka w języku Python.