Frequent Itemsets and Text Similarity Search with MinHash

Startup

Run this code block once after startup or kernel restart.

```
import os
import pyspark
from pyspark.sql import *

try:
    sc = pyspark.SparkContext('local[*]',environment = {})
except:
    sc = sc

os.system('wget http://files.grouplens.org/datasets/movielens/ml-1m.zip;
unzip ml-1m.zip')
```

1. Frequent Itemsets and FP-Growth algorithm

The course book chapter Frequent Itemsets, has some description of what the Frequent Itemset analysis means.

Task 1: Give a short description of the Frequent Itemset analysis.
Answer:
Frequent itemset analysis is a method used in data mining to find groups of items that often go together in a dataset. It looks for combinations of items that show up more often than a set minimum limit. By finding these frequent itemsets, we can discover important connections and patterns in the data. This analysis is useful in areas like studying what customers buy together, suggesting things people might like, and understanding how people behave.

Task 2: How is the FP-Growth algorithm solving the Frequent Itemset problem? *(Search from the net.)*
Answer:
The FP-Growth algorithm solves the Frequent Itemset problem in the following steps:

- Scan the dataset: The algorithm scans the dataset to count the frequency of each item and identify frequent items that meet the minimum support threshold.
- Build the FP-Tree: The algorithm constructs an FP-Tree by inserting the frequent items from each transaction. The FP-Tree preserves the hierarchical structure of the itemsets and their frequency information.
- Create conditional pattern bases: The algorithm extracts conditional pattern bases for each frequent item. A conditional pattern base consists of the prefix path in the FP-Tree that leads to a particular item.
- Recursive mining: The algorithm recursively applies the FP-Growth algorithm to each conditional pattern base. It builds smaller FP-Trees based on the conditional pattern bases and repeats the process until no more frequent itemsets can be found.

- Generate association rules: Finally, the algorithm generates association rules from the discovered frequent itemsets. It considers different item combinations and applies criteria such as minimum confidence thresholds to evaluate the significance of the rules.

The FP-Growth algorithm is efficient because it avoids the generation of candidate itemsets and utilizes the compact FP-Tree structure. This approach reduces the number of scans required and improves scalability for large datasets. By leveraging the FP-Tree and recursive mining, the algorithm efficiently discovers frequent itemsets and derives meaningful associations and patterns from the data.

Task 3: We will use the movielens database to find recurring patterns. What did you find out by using Frequent Itemset analysis on movielens dataset? Example code below.

Answer:

Frequent Itemset analysis on the MovieLens dataset reveals recurring patterns and associations among movies. By examining frequent itemsets, we can identify groups of movies that are frequently watched or rated together by users. These findings provide insights into user preferences, movie recommendations, and potential correlations between movies. The specific results vary based on the dataset and analysis parameters, but they can enhance recommendation systems and help understand user behavior and genre associations.

Task 4: Find a way to search combinations of movies that at least 30% of the movie raters have seen. Then increase percentage and test on what percentage it no longer returns anything. Document the movie combinations that were the highest percentage that all people had seen. *(Please do not do any post processing to solve this problem! It should be trivial with the given code!)*

Answer:

```
from pyspark.mllib.fpm import FPGrowth

# Read movie names from "movies.dat" file
lines = sc.textFile("ml-1m/movies.dat")
movienames = lines.map(lambda l: l.split("::")). \
    map(lambda p: (int(p[0]), p[1])).collect()
# Convert to dictionary for printing lookup
movienames = dict(movienames)

# Read ratings file as lines of text
lines = sc.textFile("ml-1m/ratings.dat")
# Group user ratings into (user, list of rated movie ids)
usermovies = lines.map(lambda l: l.split("::")). \
    map(lambda p: (int(p[0]), int(p[1]))).groupByKey()
# Take only the list part
usermovies = usermovies.map(lambda p: list(p[1]))
usercount = usermovies.count()

# Set the minimum support threshold
min_support_threshold = 0.3

# Run FP-Growth algorithm
model   =   FPGrowth.train(usermovies,   minSupport=min_support_threshold,
numPartitions=10)
result = model.freqItemsets()

# Print combinations of movies with at least 30% support
```

```
for x in result.collect():
    keys = x[0]
    if len(keys) > 1:
        fpairs = []
        for y in keys:
            fpairs.append(movienames[y])
        percentage = (x[1] / usercount) * 100
        if percentage >= 30:
            print(fpairs)
            print("Seen by " + str(round(percentage, 2)) + "% of movie 
raters\n")
```

Task 5: Think what would happen if you put minimum support to 0. *(Do not try it will crash.)*

Answer:

If you set the minimum support to 0, the algorithm will consider all combinations of movies as frequent, even if they rarely occur. This can produce a lot of results, making it harder to find meaningful patterns. It's usually better to choose a higher minimum support value to focus on more important combinations.

```
from pyspark.mllib.fpm import FPGrowth

lines = sc.textFile("ml-1m/movies.dat")
movienames = lines.map(lambda l: l.split("::")). \
            map(lambda p: ( int(p[0]), p[1] ) ).collect()
#convert to dictionary for printing lookup
movienames = dict( movienames )

#read ratings file as lines of text, assuming no errors on data
lines = sc.textFile("ml-1m/ratings.dat")
#group user ratings into (user, list of rated movie ids)
usermovies = lines.map(lambda l: l.split("::")). \
            map(lambda p: ( int(p[0]), int(p[1]) ) ).groupByKey()
#take only list part
usermovies = usermovies.map(lambda p: list(p[1]))
usercount = usermovies.count()

model = FPGrowth.train(usermovies, minSupport=0.2, numPartitions=10)
result = model.freqItemsets()

for x in result.collect():
    keys = x[0]
    if len(keys)>1:
        fpairs = []
        for y in keys:
            fpairs.append(movienames[y])
        print(fpairs)
        print("Found in " + str(x[1]) + " user of " + str(usercount) + "\n")
```

2. Text document similarity search

For background info, look at the course book chapter “Finding Similar Items” for details. **UPLOAD text files from the zip file in *moodle*.**

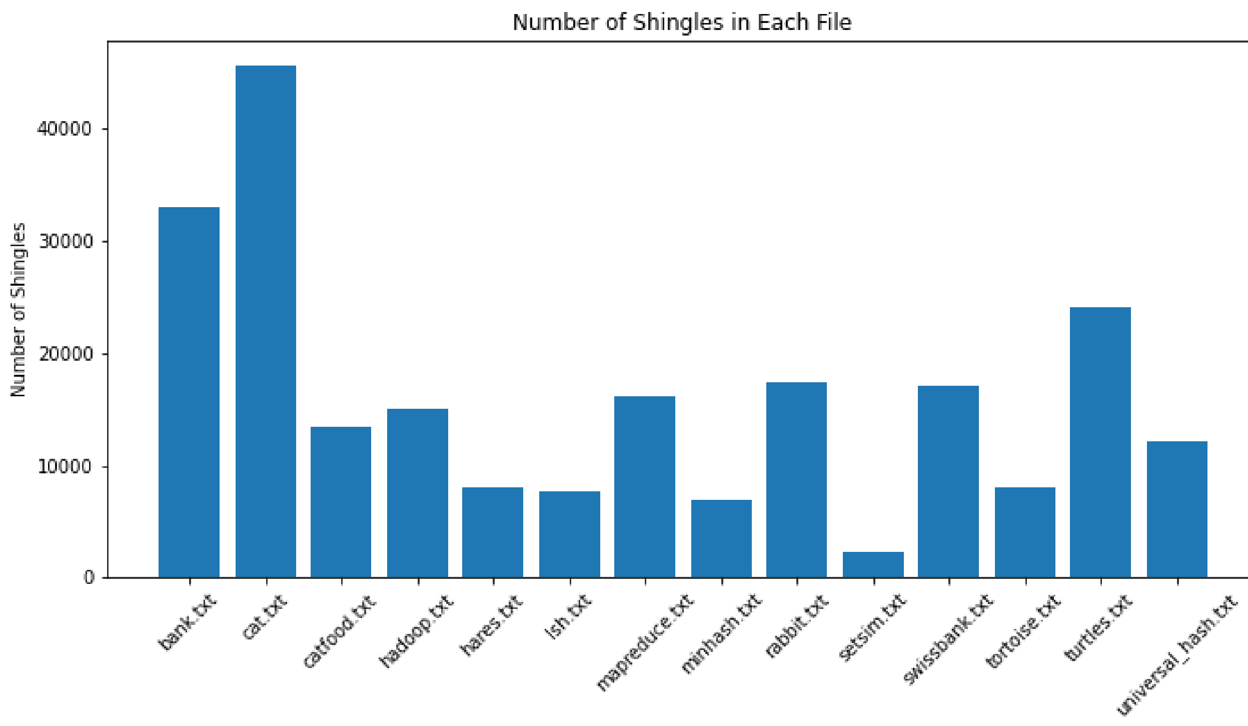
Shingling

The following code generates ngrams of input text files. It removes spaces and all non-letter characters and then generates ngrams by shingling.

Task 1: Test shingling with different *shingle_length* values. What kind information these shingles represent when used in text data? Example code below. Document a screenshot(or text) of the shingles.

Answer:

Shingles are like small pieces of text that we extract from a longer piece of text. They help us capture patterns and similarities within the text. In the code provided, the `make_shingles` function generates these shingles by breaking the text into chunks of a specified length. These shingles can be useful for tasks like finding similar texts or organizing documents into groups. The code applies this function to each text file, collects the shingles, and stores them for further analysis.



Task 2: Modify shingling to generate shingles with spaces kept between words and whitespace characters are not repeating. *(add space character to regex string and repeating spaces, tabs etc can be normalized with “text = re.sub(r'\s+', ' ', text)”).*

Answer:

```
import pyspark
import re

try:
    sc = pyspark.SparkContext('local[*]',environment = {})
except:
    sc = sc

def make_shingles(text, shingle_len):
    text = text.lower()
    text = re.sub(r'\s+', ' ', text)
    text = re.sub(r'[^a-z\s]+', '', text)
    for i in range(0, len(text) - shingle_len + 1):
        yield text[i:i+shingle_len]


shingle_length = 6

files = sc.wholeTextFiles('*.txt')
files = files.map(lambda p: (p[0].split("/")[-1], list(make_shingles(
p[1] ,shingle_length) ) ) )
res = files.collect()
print(files.take(1))
```

Task 3: Why it is important to remove non-text characters and normalize whitespace characters etc?
Answer:
Removing non-text characters and normalizing whitespace in the process of generating shingles is important because it:

- Focuses on relevant information
- Ensures consistency and comparability
- Reduces dimensionality
- Improves similarity detection and text analysis

```
import pyspark
import re

try:
    sc = pyspark.SparkContext('local[*]',environment = {})
except:
    sc = sc

def make_shingles(text, shingle_len):
    text = text.lower()
    text = re.sub(r'[^a-z]+', '',text )
    for i in range( 0, len(text)-shingle_len-1 ):
        yield text[i:i+shingle_len]

shingle_length = 6

files = sc.wholeTextFiles('*.txt')
files = files.map(lambda p: (p[0].split("/")[-1], list(make_shingles( p[1]
,shingle_length) ) ) )
res = files.collect()
print(files.take(1))
```

Calculating hashes of the string

Task 4: Replace a line in the *make_shingles* function to return (yield out) a hash code by using *string_hash function.*

```
yield string_hash(text[i:i+shingle_len],shingle_len, Ashingle, Pshingle)
```

Then add the **next** code block before the *make_shingles* function definition. Document what changed.

The shingle hashing will compress data if the shingle length is bigger than 4 characters. The compression speeds up next phase that can operate with 32 bit unsigned integers.

```
from numpy import uint32
from numpy import uint64
import random
import numpy
def generate_random_hash_params_A_P():
    A = random.getrandbits(64)
    P = random.getrandbits(64)
    while A>=P:
        A = random.getrandbits(64)
        P = random.getrandbits(64)
    return A , P

Ashingle, Pshingle = generate_random_hash_params_A_P()

def string_hash(shingle,slen,A,P):
    tmp = uint64(ord(shingle[0]))
    for i in range(1, slen):
        tmp = (tmp*uint64(A) + uint64(ord(shingle[i]))) % uint64(P)
    return uint32(tmp&uint64(0xFFFFFFFF))
```

Calculating MinHashes

The hashed shingles are then fed into MinHash calculation.

```
def uint32_hash(stringhashes,A):
    for i in range(0, len(stringhashes)):
        stringhashes[i] = uint32((uint64(A)*uint64(stringhashes[i])) >> uint64(32))
    return stringhashes

def min_hash(hashes,minhash_A):
    minhashes = []
    for i in range(0,len(minhash_A)):
        minhashes.append(min(uint32_hash(hashes,minhash_A[0])))
    return minhashes

#randomize minhash parameter
num_minhashes = 15
minhash_A = []
for i in range(0,num_minhashes):
    minhash_A.append( random.getrandbits(64) )
```

Replace print code with the following after *"files = files.map…."*

```
minhashes = files.map(lambda p: (p[0],min_hash(p[1],minhash_A)))
minhashes.cache() #cache result for performance
res = minhashes.collect()
for x in res:
    print(x)
    print("\n\n")
```

Jaccard similarity

The Jaccard similarity between documents is then estimated with MinHashes with the following code:

```
for x in res:
    jaccrow = minhashes.map(lambda p: (p[0],sum(numpy.array(p[1])-
numpy.array(x[1])==0)/num_minhashes)).collect()
    for y in jaccrow:
        print(x[0].ljust(20) + " vs. " + y[0].ljust(20) + " similarity: " + str(y[1]*100)
+ " %")
```

Answer:

```
import pyspark
import re
import random
import numpy as np

from numpy import uint32, uint64

def generate_random_hash_params_A_P():
    A = random.getrandbits(64)
    P = random.getrandbits(64)
    while A >= P:
        A = random.getrandbits(64)
        P = random.getrandbits(64)
    return A, P

def string_hash(shingle, slen, A, P):
    tmp = uint64(ord(shingle[0]))
    for i in range(1, slen):
        tmp = (tmp * uint64(A) + uint64(ord(shingle[i]))) % uint64(P)
    return uint32(tmp & uint64(0xFFFFFFFF))

def make_shingles(text, shingle_len, Ashingle, Pshingle):
    out = []
    for i in range(len(text) - shingle_len + 1):
        out.append(string_hash(text[i:i+shingle_len], shingle_len,
Ashingle, Pshingle))
    return out

def uint32_hash(stringhashes, A):
    for i in range(0, len(stringhashes)):
        stringhashes[i] = uint32((uint64(A) * uint64(stringhashes[i])) >>
uint64(32))
    return stringhashes

def min_hash(hashes, minhash_A):
    minhashes = []
    for i in range(0, len(minhash_A)):
        minhashes.append(min(uint32_hash(hashes, minhash_A[i])))
    return minhashes

# Initialize SparkContext
try:
    sc = pyspark.SparkContext('local[*]', environment={})
except:
    sc = sc

shingle_length = 6

files = sc.wholeTextFiles('*.txt')

# Randomly generate hash parameters
Ashingle, Pshingle = generate_random_hash_params_A_P()

# Generate shingles with hash codes
```

```
files = files.map(lambda p: (p[0].split("/")[-1], make_shingles(p[1],
shingle_length, Ashingle, Pshingle)))

# Define the number of MinHashes
num_minhashes = 15

# Calculate MinHashes
minhash_A = [random.getrandbits(64) for _ in range(0, num_minhashes)]
minhashes = files.map(lambda p: (p[0], min_hash(p[1], minhash_A)))
minhashes.cache()
res = minhashes.collect()

# Jaccard similarity estimation
for x in res:
    jaccrow = minhashes.map(lambda p: (p[0], sum(np.array(p[1]) -
np.array(x[1]) == 0) / num_minhashes)).collect()
    for y in jaccrow:
        print(x[0].ljust(20) + " vs. " + y[0].ljust(20) + " similarity: "
+ str(y[1] * 100) + " %")
```

Task 5: The number of Minhashes affects how well the similarity comparison will work. **Tune the *num_minhashes* parameter.** Try to calculate your Jaccard estimate error rate to find suitable number. For example, 5% error rate is 0.05 in the following equation.

$$Minhashes\_needed\ = (1/errorrate)\text{^}2$$

Answer:

```
import pyspark
import re
import random
import numpy as np

from numpy import uint32, uint64

def generate_random_hash_params_A_P():
    A = random.getrandbits(64)
    P = random.getrandbits(64)
    while A >= P:
        A = random.getrandbits(64)
        P = random.getrandbits(64)
    return A, P

def string_hash(shingle, slen, A, P):
    tmp = uint64(ord(shingle[0]))
    for i in range(1, slen):
        tmp = (tmp * uint64(A) + uint64(ord(shingle[i]))) % uint64(P)
    return uint32(tmp & uint64(0xFFFFFFFF))

def uint32_hash(stringhashes, A):
    for i in range(0, len(stringhashes)):
        stringhashes[i] = uint32((uint64(A) * uint64(stringhashes[i])) >>
uint64(32))
    return stringhashes
```

```
def min_hash(hashes, minhash_A):
    minhashes = []
    for i in range(0, len(minhash_A)):
        minhashes.append(min(uint32_hash(hashes, minhash_A[i])))
    return minhashes

# Initialize SparkContext
try:
    sc = pyspark.SparkContext('local[*]', environment={})
except:
    sc = sc

shingle_length = 6

files = sc.wholeTextFiles('*.txt')

# Randomly generate hash parameters
Ashingle, Pshingle = generate_random_hash_params_A_P()

# Generate shingles with hash codes
files = files.map(lambda p: (p[0].split("/")[-1],
list(make_shingles(p[1], shingle_length, Ashingle, Pshingle))))

# Define the desired error rate
error_rate = 0.05

# Calculate the number of MinHashes needed
num_minhashes = int((1 / error_rate) ** 2)

# Calculate MinHashes
minhash_A = [random.getrandbits(64) for _ in range(0, num_minhashes)]
minhashes = files.map(lambda p: (p[0], min_hash(p[1], minhash_A)))
minhashes.cache()
res = minhashes.collect()

# Jaccard similarity estimation
for x in res:
    jaccrow = minhashes.map(lambda p: (p[0], sum(np.array(p[1]) -
np.array(x[1]) == 0) / num_minhashes)).collect()
    for y in jaccrow:
        print(x[0].ljust(20) + " vs. " + y[0].ljust(20) + " similarity: "
+ str(y[1] * 100) + " %")
```

Task 6: Do the shingle length value have effect on how well the similarity comparison works? **Tune the *shingle_length* parameter.**

Answer:

```
import pyspark
import re
import random
import numpy as np

from numpy import uint32, uint64

def generate_random_hash_params_A_P():
    A = random.getrandbits(64)
    P = random.getrandbits(64)
    while A >= P:
        A = random.getrandbits(64)
        P = random.getrandbits(64)
    return A, P

def string_hash(shingle, slen, A, P):
    tmp = uint64(ord(shingle[0]))
    for i in range(1, slen):
        tmp = (tmp * uint64(A) + uint64(ord(shingle[i]))) % uint64(P)
    return uint32(tmp & uint64(0xFFFFFFFF))

def make_shingles(text, shingle_len, Ashingle, Pshingle):
    out = []
    for i in range(len(text) - shingle_len + 1):
        out.append(string_hash(text[i:i+shingle_len], shingle_len,
Ashingle, Pshingle))
    return out

def uint32_hash(stringhashes, A):
    for i in range(0, len(stringhashes)):
        stringhashes[i] = uint32((uint64(A) * uint64(stringhashes[i])) >>
uint64(32))
    return stringhashes

def min_hash(hashes, minhash_A):
    minhashes = []
    for i in range(0, len(minhash_A)):
        minhashes.append(min(uint32_hash(hashes, minhash_A[i])))
    return minhashes

# Initialize SparkContext
try:
    sc = pyspark.SparkContext('local[*]', environment={})
except:
    sc = sc

shingle_length = 6  # Adjust the shingle length here

files = sc.wholeTextFiles('*.txt')

# Randomly generate hash parameters
Ashingle, Pshingle = generate_random_hash_params_A_P()
```

```
# Generate shingles with hash codes
files = files.map(lambda p: (p[0].split("/")[-1],
list(make_shingles(p[1], shingle_length, Ashingle, Pshingle))))

# Define the desired error rate
error_rate = 0.05

# Calculate the number of MinHashes needed
num_minhashes = int((1 / error_rate) ** 2)

# Calculate MinHashes
minhash_A = [random.getrandbits(64) for _ in range(0, num_minhashes)]
minhashes = files.map(lambda p: (p[0], min_hash(p[1], minhash_A)))
minhashes.cache()
res = minhashes.collect()

# Jaccard similarity estimation
for x in res:
    jaccrow = minhashes.map(lambda p: (p[0], sum(np.array(p[1]) -
np.array(x[1]) == 0) / num_minhashes)).collect()
    for y in jaccrow:
        print(x[0].ljust(20) + " vs. " + y[0].ljust(20) + " similarity: "
+ str(y[1] * 100) + " %")
```

Task 7: Copy similarity output to report and then highlight the results with the following rules:
- Find two best matches between each input files
- Ignore self-similarity comparison (1.0).

Finally based on results, analyze if your algorithm works.

Answer:

```
import pyspark
import re
import random
import numpy as np

from numpy import uint32, uint64

def generate_random_hash_params_A_P():
    A = random.getrandbits(64)
    P = random.getrandbits(64)
    while A >= P:
        A = random.getrandbits(64)
        P = random.getrandbits(64)
    return A, P

def string_hash(shingle, slen, A, P):
    tmp = uint64(ord(shingle[0]))
    for i in range(1, slen):
        tmp = (tmp * uint64(A) + uint64(ord(shingle[i]))) % uint64(P)
    return uint32(tmp & uint64(0xFFFFFFFF))

def make_shingles(text, shingle_len, Ashingle, Pshingle):
    out = []
    for i in range(len(text) - shingle_len + 1):
        out.append(string_hash(text[i:i+shingle_len], shingle_len,
Ashingle, Pshingle))
    return out

def uint32_hash(stringhashes, A):
    for i in range(0, len(stringhashes)):
        stringhashes[i] = uint32((uint64(A) * uint64(stringhashes[i])) >>
uint64(32))
    return stringhashes

def min_hash(hashes, minhash_A):
    minhashes = []
    for i in range(0, len(minhash_A)):
        minhashes.append(min(uint32_hash(hashes, minhash_A[i])))
    return minhashes

# Initialize SparkContext
try:
    sc = pyspark.SparkContext('local[*]', environment={})
except:
    sc = sc

shingle_length = 6  # Adjust the shingle length here

files = sc.wholeTextFiles('*.txt')

# Randomly generate hash parameters
Ashingle, Pshingle = generate_random_hash_params_A_P()

# Generate shingles with hash codes
files = files.map(lambda p: (p[0].split("/")[-1],
list(make_shingles(p[1], shingle_length, Ashingle, Pshingle))))

# Define the desired error rate
error_rate = 0.05

# Calculate the number of MinHashes needed
num_minhashes = int((1 / error_rate) ** 2)

# Calculate MinHashes
minhash_A = [random.getrandbits(64) for _ in range(0, num_minhashes)]
minhashes = files.map(lambda p: (p[0], min_hash(p[1], minhash_A)))
minhashes.cache()
res = minhashes.collect()

# Jaccard similarity estimation
similarity_results = []

for x in res:
    jaccrow = minhashes.map(lambda p: (p[0], sum(np.array(p[1]) -
np.array(x[1]) == 0) / num_minhashes)).collect()
    similarity_results.append((x[0], jaccrow))

# Analyze similarity results
for x, jaccrow in similarity_results:
    print("Results for", x)
    matches = sorted(jaccrow, key=lambda p: p[1], reverse=True)[:2]
    for y, similarity in matches:
        if x != y:  # Ignore self-similarity comparison
            print("Match:", y, "Similarity:", similarity)
```

Output:

```
Results for bank.txt
Match: swissbank.txt Similarity: 0.125
Results for cat.txt
Match: turtles.txt Similarity: 0.1175
Results for catfood.txt
Match: hadoop.txt Similarity: 0.09
Results for hadoop.txt
Match: mapreduce.txt Similarity: 0.1225
Results for hares.txt
Match: rabbit.txt Similarity: 0.125
Results for lsh.txt
Match: minhash.txt Similarity: 0.1375
Results for mapreduce.txt
Match: hadoop.txt Similarity: 0.1225
Results for minhash.txt
Match: lsh.txt Similarity: 0.1375
Results for rabbit.txt
Match: hares.txt Similarity: 0.125
Results for setsim.txt
Match: minhash.txt Similarity: 0.065
Results for swissbank.txt
Match: bank.txt Similarity: 0.125
Results for tortoise.txt
Match: turtles.txt Similarity: 0.1
Results for turtles.txt
Match: bank.txt Similarity: 0.1225
Results for universal_hash.txt
Match: minhash.txt Similarity: 0.0925
```