Metrics-TinyStories_Case_Study

April 17, 2025

1 Case Study of the'TinyStories' Dataset

The 'TinyStories' dataset is a large dataset that was generated using AI story request prompts and responses. It helps facilitate quick fine-tuning of models that were trained with real-world data - such as LLaMA. As this is a large dataset, and this data was generated with the help of AI, the data may have unintentionally been skewed or biased during generation. It is essential to examine the contents of this dataset and whether it is suitable to our applications content-moderation objectives with regard to literacy, mental-health, and creativity.

[1]:
```
# importing libraries
from pyspark.sql import SparkSession
from datasets import load_dataset
```

[2]:
```
from pyspark.sql.functions import udf, col, countDistinct
from pyspark.sql.types import StringType, ArrayType, IntegerType
from pyspark.accumulators import AccumulatorParam
```

[38]:
```
import pandas as pd
from matplotlib import pyplot as plt
```

1.1 Setting up the Distributed Processing Environment

In this section, the spark distributed cluster is connected using AWS's elastic map reduce service via sagemaker studio.

[4]:
```
%load_ext sagemaker_studio_analytics_extension.magics
%sm_analytics emr-serverless connect --application-id 00fra2001bfrlm09␣
  ↪--language python --emr-execution-role-arn arn:aws:iam::597161074694:role/
  ↪service-role/AmazonEMR-ServiceRole-20250211T131858
```

```
Waiting for EMR Serverless application state to become STARTED
Waiting for EMR Serverless application state to become STARTED
Initiating EMR Serverless connection..
Starting Spark application
```

```
<IPython.core.display.HTML object>
```

```
FloatProgress(value=0.0, bar_style='info', description='Progress:',␣
 ↪layout=Layout(height='25px', width='50%'),…
```

```
SparkSession available as 'spark'.
```

[5]:
```
# connecting to the spark session
spark = SparkSession.builder \
    .master('local[*]') \
    .config("spark.driver.memory", "64g") \
    .appName('spark') \
    .getOrCreate()
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
25/04/01 22:33:25 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
```

2 Statistical Metrics

In this section, the basic statistical metrics are calculated to be used for further data analysis and visualization in the following sections.

2.1 Loading the Data

[6]:
```
data = load_dataset("skeskinen/TinyStories-GPT4")
```

[7]:
```
train = spark.createDataFrame(data['train'])
```

2.2 Defining the Spark Functions

[8]:
```
class WordCountAccumulator(AccumulatorParam):
    def zero(self, value):
        return {}

    def addInPlace(self, v1, v2):
        # v1 is the accumulator's current state
        # v2 is the new dictionary being added from a partition
        for word, count in v2.items():
            if word is not None and word.strip():
                v1[word] = v1.get(word, 0) + count
        return v1
```

[9]:
```
def normalize_and_count_unique(strings):
    if strings is None:
        return None
    normalized_strings = [s.strip().lower() for s in strings if s is not None␣
 ↪and s.strip()]
    return normalized_strings
```

```
# the sparkerized user defined function to normalize the words/features for␣
↪comparison
normalize_count_udf = udf(normalize_and_count_unique, ArrayType(StringType()))␣
↪# Assuming string type
```

[10]:
```
# applying the accumulator across partitions of the dataset based on col type
def process_features_partition(partition_iterator, col_type):
    # Initialize local counter for this partition
    partition_features = {}

    # Process each row in the partition
    for row in partition_iterator:
        features = row["normalized_" + col_type]
        if features is not None:
            # Handle both string and list cases
            words = features.split() if isinstance(features, str) else features

            # Count words in this row
            for word in words:
                if word and word.strip():
                    word = word.strip()
                    partition_features[word] = partition_features.get(word, 0)␣
↪+ 1

    # Add the partition counts to the accumulator
    if partition_features and col_type == 'features':
        features_accumulator.add(partition_features)
    elif partition_features and col_type == 'words':
        words_accumulator.add(partition_features)

    # Return the iterator for the partition
    return iter([1])  # Return dummy value to force evaluation
```

2.3 Data Prep

2.3.1 Adding the Normalized Columns For Words/Features

UDF / user defined functions applied on the spark dataset should be added as columns prior to partitioning / converting the dataset.

[11]:
```
train = train.withColumn("normalized_features",␣
↪normalize_count_udf(col("features")))
```

[12]:
```
train = train.withColumn("normalized_words", normalize_count_udf(col("words")))
```

2.3.2 Partitioning the Dataset for Mapping the Accumulator

[13]:
```
# Ensure proper partitioning
num_partitions = 200  # Adjust based on your cluster size
train = train.repartition(num_partitions)
```

[14]:
```
# Verify the number of partitions
print(f"Number of partitions: {train.rdd.getNumPartitions()}")
```

```
25/04/01 22:38:28 WARN TaskSetManager: Stage 0 contains a task of very large
size (223671 KiB). The maximum recommended task size is 1000 KiB.
[Stage 0:==============================================>          (13 + 3) / 16]
```

```
Number of partitions: 200
```

```
[Stage 0:=====================================================>   (15 + 1) / 16]
```

2.3.3 Narrative Features

[15]:
```
train.columns
```

[15]:
```
['features',
 'prompt',
 'source',
 'story',
 'summary',
 'words',
 'normalized_features',
 'normalized_words']
```

[16]:
```
features_accumulator = spark.sparkContext.accumulator(
    {}, WordCountAccumulator())
```

[17]:
```
# Force evaluation and verify processing
col_type = "features"  # or whatever column name you want to process
total_partitions = train.rdd.mapPartitions(
    lambda partition: process_features_partition(partition, col_type)
).count()
```

[18]:
```
# Get the final word counts
features_accumulator.value
```

[18]:
```
{'dialogue': 1470404,
 'moralvalue': 274152,
 'twist': 539383,
 'foreshadowing': 250789,
 'badending': 250300,
 'conflict': 250696}
```

[19]:
```
feature_occurences = pd.Series(features_accumulator.value)
feature_occurences
```

[19]:
```
dialogue         1470404
moralvalue        274152
twist             539383
foreshadowing     250789
badending         250300
conflict          250696
dtype: int64
```

2.3.4 Key Words

[20]:
```
words_accumulator = spark.sparkContext.accumulator(
    {}, WordCountAccumulator())
```

[21]:
```
# Force evaluation and verify processing
col_type = "words"  # or whatever column name you want to process
total_partitions = train.rdd.mapPartitions(
    lambda partition: process_features_partition(partition, col_type)
).count()
```

[22]:
```
word_occurrences = pd.Series(words_accumulator.value)
word_occurrences.head()
```

[22]:
```
admire      6971
cane        2546
bald       11245
collect     6956
aunt        2554
dtype: int64
```

[23]:
```
spark.stop()
```

2.3.5 Saving the Metrics

As the metrics calculated require a different computation environment for analysis, the metrics are saved to CSV for later use.

[78]:
```
feature_occurences.name = 'feature_occurrences'
```

[90]:
```
feature_occurences = feature_occurences.reset_index(drop=False)
```

[91]:
```
feature_occurences.columns = ['feature', 'feature_occurrences']
```

[92]:
```
feature_occurences.to_csv('feature_occurrences.csv', index=False)
```

```
[93]: word_occurrences.name = 'word_occurrences'
```

```
[94]: word_occurrences = word_occurrences.reset_index(drop=False)
```

```
[95]: word_occurrences.columns = ['word', 'word_occurrences']
```

```
[96]: word_occurrences.to_csv('word_occurrences.csv', index=False)
```