Alumni Profile Matching

Networking with alumni is highly valuable for graduate students as it enables them to leverage shared academic backgrounds, seek mentorship support, and potentially secure referrals in companies where alumni are currently or previously associated. In pursuit of this objective, our project focused on developing a sophisticated machine learning model that can effectively match current students with alumni profiles that exhibit significant similarities.

by Yuhsin Wang

Problem/Motivation Statement

Often, students only have limited opportunities to connect with alumni, typically during university meetups and fireside chats that offer brief interactions. Moreover, the career trajectory of alumni may differ significantly from that of current students, making their advice less relevant due to the disparity in backgrounds.

The objective of our project is to develop a sophisticated machine learning model that can effectively identify the top 5 alumni profiles most similar to each current student. This will enable students to reach out to alumni who share commonalities by leveraging similarity scores derived from a comprehensive analysis of student and alumni profiles. Additionally, we prioritize the creation of a robust and distributed data processing pipeline that seamlessly integrates with our machine learning model to achieve this objective.

Dataset & Analytics Goals

Data Sources:

We employed two primary data sources for our project:

- **List of alumni profiles for each cohort:**

We contacted the administrative head to obtain a comprehensive list of alumni from all previous cohorts of the graduate program. Subsequently, we stored this file in a dedicated Google Cloud Platform (GCP) storage bucket, allowing us to maintain a focused and accessible collection of profiles for further analysis.

- **Scraped LinkedIn profiles data (in JSON) per cohort:**

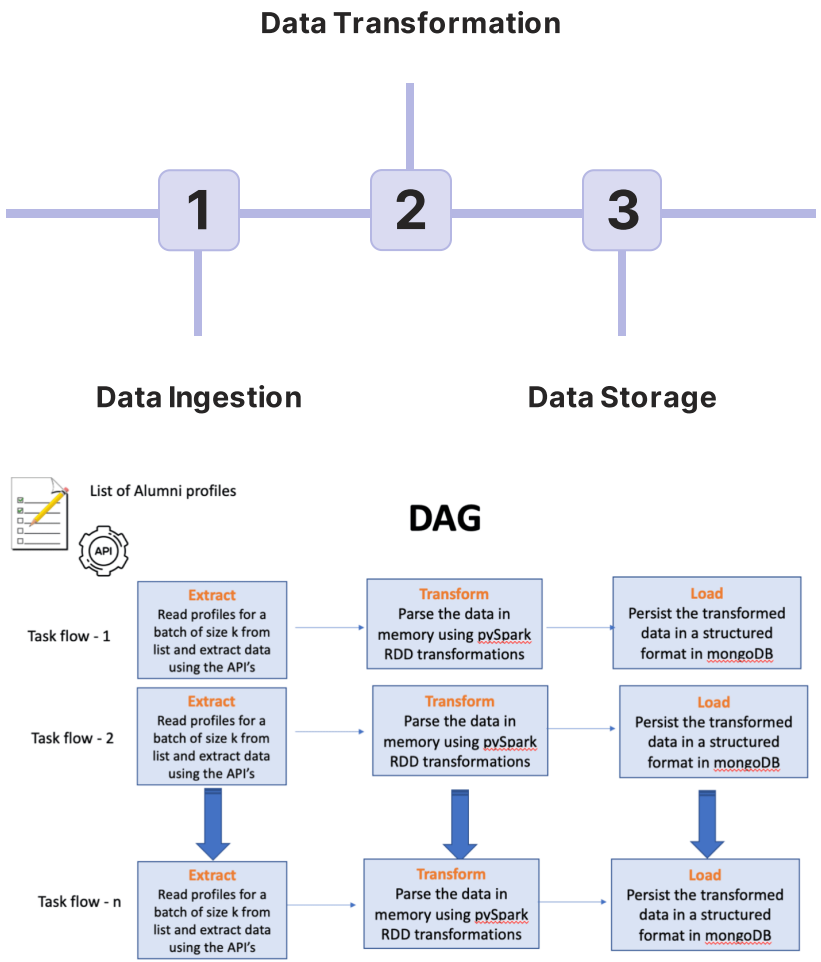
To augment our dataset, we leveraged the RapidAPI interface to seamlessly integrate with LinkedIn's developer APIs. This integration facilitated the retrieval of scraped LinkedIn profile data for each alumnus within our system, enhancing the richness and diversity of our data sources.

Analytics Goals:

Our analytics goals encompassed the following key objectives:

- **BERT Embedding:** Utilize BERT embedding techniques to vectorize the user profiles, thereby extracting high-quality numerical representations of the textual data.
- **GloVe Embedding:** Employ GloVe embedding to identify the top 5 similar profiles and compare the results with those obtained through the BERT model. This comparative analysis allows us to assess the effectiveness of different embedding approaches.
- **Pre-processing and Feature Engineering:** Apply pre-processing techniques to enhance data quality, followed by feature engineering to derive meaningful features from the dataset. This ensures optimal performance of our machine learning models.
- **TF-IDF:** Perform TF-IDF (Term Frequency-Inverse Document Frequency) on the transformed and simplified data, enabling us to calculate the significance of words within each profile and across the entire dataset. This approach aids in identifying key terms and patterns that contribute to profile similarity.

Overview of Data Engineering Pipeline



The tasks on the left collectively form an Extract-Transform-Load (ETL) pipeline. An ETL pipeline plays a crucial role in extracting data, transforming it into the desired format, and ultimately loading the organized and aggregated data into a data lake.

To automate this sequence of tasks, we implemented the **Apache Airflow DAG Scheduler**. A Directed Acyclic Graph (DAG) serves as the underlying representation, comprising nodes and edges. The nodes symbolize individual tasks, while the edges represent the dependencies and workflow direction between these tasks.

By leveraging the capabilities of the Apache Airflow DAG Scheduler, we effectively automated the execution and coordination of these tasks within the ETL pipeline. This automation ensures a streamlined and efficient workflow, allowing for seamless data extraction, transformation, and loading into the designated data lake.

DAG Task Flow — Task 1: Data Extraction

We defined two tasks that will run for each batch of cohort - one for **extracting data** and the other for **transforming and loading the data**.

The **first task** encompasses the **Extract block** of our **ETL pipeline**. This task is designed to execute a script responsible for crawling through the list of alumni profiles associated with a given cohort, which is obtained from the GCP storage bucket. Subsequently, the script retrieves the scraped LinkedIn profile data by making API requests. The extracted data is then stored in JSON formatted files within the GCP storage bucket, with separate files generated for each cohort. To ensure comprehensive handling of potential **exceptions**, such as API invocation failures or undesirable responses, we have implemented robust logging mechanisms. Moreover, a list of profiles that encountered issues during the data fetching process is maintained and tracked within the GCP bucket.

This script is encapsulated within a task and executed within the Airflow framework, utilizing the **PythonOperator** to facilitate seamless integration and execution.

```
with DAG(
    dag_id="msds697-task2",
    schedule=None,
    start_date=datetime(2023, 1, 1),
    catchup=False
) as dag:

    '''
    Below loop would dynamically generate task workflows for each cohort.
    We read the main file(source_file#1) in get_cohorts() function.
    Each flow consists of two tasks: one for extract and other for tranform&load
    '''
    for cohort, profiles in get_cohorts().items():

        create_insert_aggregate = SparkSubmitOperator(
            task_id=f"aggregates_to_mongo_cohort_{cohort}",
            packages="com.google.cloud.bigdataoss:gcs-connector:hadoop2-1.9.17,org.mongodb.spark:mongo-spark-connector_2.12:3.0.1",
            exclude_packages="javax.jms:jms,com.sun.jdmk:jmxtools,com.sun.jmx:jmxri",
            conf={"spark.driver.userClassPathFirst":True,
                "spark.executor.userClassPathFirst":True,
                #  "spark.hadoop.fs.gs.impl":"com.google.cloud.hadoop.fs.gcs.GoogleHadoopFileSystem",
                #  "spark.hadoop.fs.AbstractFileSystem.gs.impl":"com.google.cloud.hadoop.fs.gcs.GoogleHadoopFS",
                #  "spark.hadoop.fs.gs.auth.service.account.enable":True,
                #  "google.cloud.auth.service.account.json.keyfile":service_account_key_file,
                },
            verbose=True,
            application=spark_application,
            # Pass args to Spark job
            application_args=[f'{profiles_folder_path}Cohort_'+str(cohort)+'.json']
        )

        crawl_alumni_profiles = PythonOperator(task_id = f"crawl_alumni_profiles_cohort_{cohort}",
                                               provide_context=True,
                                               python_callable=_crawl_alumni_profiles,
                                               op_kwargs={'cohort_id': cohort , 'profile_urls': profiles},
                                               dag=dag)

        crawl_alumni_profiles >> create_insert_aggregate
```

DAG Task Flow — Task 2: Data Transformation and Loading

The **second task** within our DAG focuses on implementing the **Transform and Load blocks** of our **ETL pipeline**. To accomplish this, we have leveraged the Apache PySpark library to facilitate initial data cleaning and transformations. During this process, we encountered instances of missing data and inconsistency within our crawled profile data, which we addressed through the application of **PySpark transformations**. Given the deep nesting of the initial data structure, we carefully extracted and flattened the essential fields within each alumni profile, including education details, professional experience details, and profile summaries.

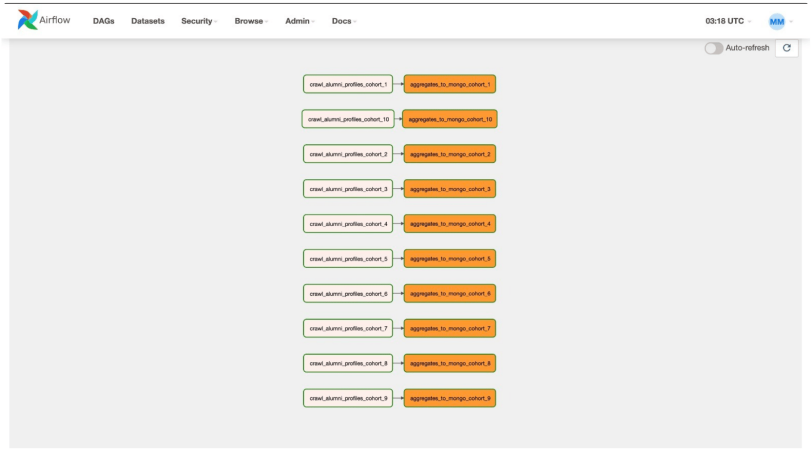
Subsequently, we aggregated each flattened profile document with the corresponding record from the alumni list, forming key-value pairs. These aggregates were then loaded into **MongoDB** as JSON documents using the **PyMongo** library's APIs.

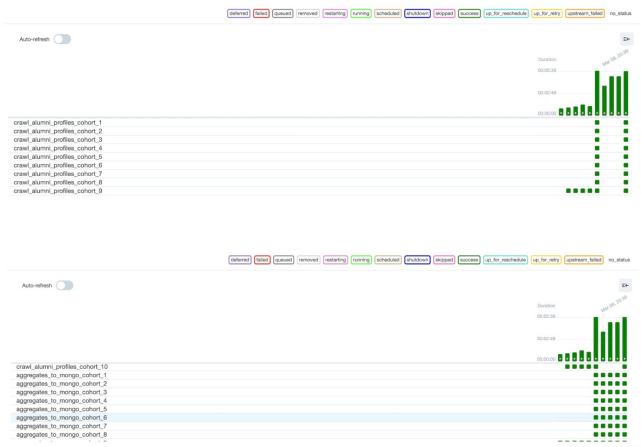
Given that Spark was predominantly used for our data cleaning and transformation processes, it was crucial to establish a Spark context to enable the execution of Spark jobs by the driver node, even within the automated Airflow tasks. To achieve this, we incorporated the **SparkSubmitOperator** DAG operator. This operator facilitates the submission of Spark jobs to a cluster as tasks within a DAG, ensuring seamless integration and execution within the Spark environment.


```
_id: ObjectId('64096b183f8dd3c19c129068')
public_identifier: "spencer-aiello-111a7a41"
full_name: "Spencer Aiello"
headline: "Machine Learning Engineer at Instagram"
summary: "https://spenai.org/"
country: "United States"
country_full_name: "United States"
city: "Brooklyn, New York"
state: "Brooklyn, New York"
languages: ""
numDegrees: "3"
recent_degree: "Master of Science (M.S.)"
recent_field_of_study: "Analytics"
recent_school: "University of San Francisco"
previous_degrees: "Bachelor of Arts (BA),Bachelor of Science (B.Sc.)"
previous_fields_of_study: "Mathematics,Physics"
previous_schools: "University of California, Santa Cruz,University of California, Santa C…"
volunteer_work: null
titles: ""
companies: "Instagram,Discord,Bloomberg LP,Neurensic (now part of Trading Technolo…"
total_years_of_experience: "8"
exp_descriptions: ""
certifications: null
cohort: "Cohort_1"
```


Aggregated JSON Document Structure

DAG Task Flow





Displayed above is a snippet of the DAG task flows represented as a flow chart, encompassing the aforementioned two tasks. These tasks are **dynamically** created for each cohort, allowing for iterative generation within a loop. During execution, parameters are passed to the Airflow operators to ensure seamless and efficient workflow orchestration.

The two snapshots showcase the landing page, providing a comprehensive view of the **DAG task flow progress**. Each successfully completed task can be marked as "Done" to proceed to the subsequent task. Conversely, any failed tasks necessitate inspection by referring to the logs and rectifying any encountered errors. This approach ensures the smooth execution and integrity of the workflow.

Cluster Specifications

To carry out data preprocessing and modeling, we utilized a Databricks platform cluster. Below are the specifications for both the Databricks compute and MongoDB database clusters employed in our project.

MongoDB Cluster Specification

- M30 (8 GB RAM, 40 GB Storage per Shard)
- 2,400 IOPS per Shard
- Encrypted storage with auto-expand capability
- MongoDB 5.0, Backup 2 shards

Databricks Cluster Specification

- Runtime version: 7.3 LTS (includes Apache Spark 3.0.1, Scala 2.12)
- Cluster specification: I3.xlarge
- Driver: 30.5 GB Memory, 4 cores
- Workers: 2-5 workers, with memory ranging from 61 GB to 152 GB, and 8-20 cores

To establish a connection between the Spark cluster and **MongoDB**, we employed a **Spark-mongo** connector, which was installed on the cluster. By providing the necessary credentials, we facilitated the reading of data from MongoDB into a Spark DataFrame, enabling seamless integration and data processing between the two systems.

Data Cleaning and Preprocessing

Handle missing data and inconsistency issues

```
degrees_mappings = {
    'B': 'Bachelor Degree',
    'BS': 'Bachelor of Science' ,
    'BA': 'Bachelor of Arts',
    'BTech': 'Bachelor of Technology',
    'BASc': 'Bachelor of Applied Science',
    'BBA': 'Bachelor of Business Administration',
    'BE: ': 'Bachelor of Engineering',
    'BMgmt': 'Bachelor of Management',
    'MBA': 'Master of Business Administration',
    'ME': 'Master of Engineering',
    'M': 'Master Degree',
    'MS': 'Master of Science',
    'MA': 'Master of Arts',
}
```

```
Command took 0.02 seconds -- by ppathi@dons.usfca.edu at 3/10/2023, 7:43:55 PM on SNACK_WEB_Cluster
```

Cmd 56

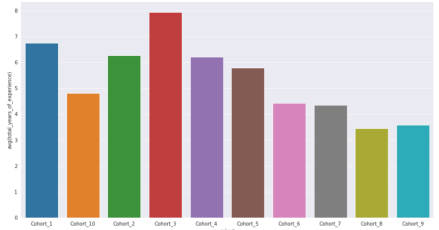
```
def _map_degree(x):
    if x is None:
        return ''
    _lstr = x.replace("'s", '').replace('.', '').replace("'s', '').replace('in', 'of').split(',')
    _lstr = [d.split('-')[0].split('(')[0].strip() for d in _lstr]
    r = list()
    for d in _lstr:
        flag = True
        # Only keep the main degrees
        if 'Certificate' in d or 'Course' in d:
            continue
        for dmk, dmv in degrees_mappings.items():
            if d.casefold() in dmk.casefold() or d.casefold() in dmv.casefold():
                r.append(dmk)
                flag = False
                break
        if flag:
            r.append('misc')
    return ' '.join(sorted(set(r)))
```

Perform feature engineering, such as obtaining geographical coordinates

```
from geopy.geocoders import Nominatim
geolocator = Nominatim(user_agent="my_user_agent")
misc_lat_long = {
    'New York City Metropolitan Area': (40.712776, -74.005974),
    'Greater Vancouver Metropolitan Area': (49.282730, -123.120735),
    'Austin, Texas Metropolitan Area': (30.267153, -97.743057),
    'Greater Phoenix Area': (33.448376, -112.074036),
    'Greater Seattle Area': (47.606209, -122.332069),
    'Greater Chicago Area': (41.8781, -87.6298),
    'Washington DC-Baltimore Area': (39.299236, -76.609383),
    'San Jose, California': (37.3387, -87.6298),
    'Greensboro--Winston-Salem--High Point Area': (36.044659, -79.766235),
    'Greater San Luis Obispo Area': (35.270378, -120.680656)
}
```

```
cities = df.rdd.map(lambda x: x[1]).distinct().collect()
lat, long = dict(), dict()
for city in cities:
    try:
        loc = geolocator.geocode(city)
        lat[city] = loc.latitude
        long[city] = loc.longitude
    except:
        loc = misc_lat_long[city]
        lat[city] = loc[0]
        long[city] = loc[1]
```

EDA (Exploratory Data Analysis)



Machine Learning Goal and Approaches

Our ultimate objective in machine learning is to identify the top 5 alumni profiles that closely align with a given input profile, based on significant academic and professional background similarities.

To accomplish this goal, we explored several prominent Natural Language Processing (NLP) approaches aimed at measuring similarity between profile sequences. These approaches were carefully selected and evaluated to determine the most effective method for capturing the desired similarity metrics among profiles.

BERT Embeddings

To process the text sequence data, we employed the **SparkNLP** library and constructed a pipeline for this purpose.

Within the pipeline, we utilized the **BertSentenceEmbeddings** model, which generates **sentence embeddings** for each sentence found in the documents created by the DocumentAssembler. This model is pretrained on a substantial English text corpus and employs a contextualized masked language modeling (CMLM) approach to produce embeddings of exceptional quality.

In order to recommend the most similar profiles to a given user from their alumni using **BERT Embeddings**, we employed the technique of computing **cosine similarity** between the embedding vectors. This approach allowed us to quantify the similarity between profiles based on the angular similarity between their respective embedding vectors.

```
document_assembler = DocumentAssembler() \
            .setInputCol("agged") \
            .setOutputCol("document")

bert_cmlm = BertSentenceEmbeddings.pretrained('sent_bert_use_cmlm_en_base', 'en')\
              .setInputCols(["document"])\
              .setOutputCol("sentence_embeddings")

pipe = Pipeline(stages=[
    document_assembler,
    bert_cmlm
])

nlp_model = pipe.fit(df)
processed = nlp_model.transform(df).persist()
```

```
4
5  vecs = pp['embeddings'].apply(pd.Series).values
6
7  def cosine_sim(a,b):
8      return np.dot(a,b)/(norm(a)*norm(b))
9
10 a = vecs[86]
11
12 sims = [(i,cosine_sim(a,vecs[i])) for i in range(0,len(vecs))]
13
14 sims = sorted(sims, key=lambda x:x[1], reverse=True)
15 top5= [pp.iloc[i[0]]['public_identifier'] for i in sims[1:6]]
```

Machine Learning Goal and Approaches

GloVe Embeddings

In this approach, we generated **GloVe** sequence embeddings that encapsulate word embeddings with contextual information. Subsequently, we computed **cosine similarity** between these embeddings. To accomplish this, we developed a machine learning pipeline that encompasses all the essential Natural Language Processing (NLP) operations required to obtain the embeddings, as demonstrated below.

```
document_assembler = DocumentAssembler() \
            .setInputCol("agged") \
            .setOutputCol("document")

tokenizer = Tokenizer() \
            .setInputCols(["document"]) \
            .setOutputCol("token")

normalizer = Normalizer() \
            .setInputCols(["token"]) \
            .setOutputCol("normalized")

stopwords_cleaner = StopWordsCleaner()\
            .setInputCols("normalized")\
            .setOutputCol("cleanTokens")\
            .setCaseSensitive(False)

lemma = LemmatizerModel.pretrained('lemma_antbnc') \
            .setInputCols(["cleanTokens"]) \
            .setOutputCol("lemma")

glove_embeddings = WordEmbeddingsModel().pretrained() \
                        .setInputCols(["document",'lemma'])\
                        .setOutputCol("embeddings")\
                        .setCaseSensitive(False)

embeddingsSentence = SentenceEmbeddings() \
                        .setInputCols(["document", "embeddings"]) \
                        .setOutputCol("g_sentence_embeddings") \
                        .setPoolingStrategy("AVERAGE")

glove_pipe = Pipeline(stages=[
    document_assembler,
    tokenizer,
    normalizer,
    stopwords_cleaner,
    lemma,
    glove_embeddings,
    embeddingsSentence
])


glove_model = glove_pipe.fit(df)
glove_processed = glove_model.transform(df).persist()
```

TF-IDF Vectorizer

In this approach, we employed the widely used **TF-IDF** method to generate a mapping of sequences to vectors. TF-IDF (Term Frequency-Inverse Document Frequency) assigns weights to words based on their frequency within a document and across all documents.

To implement this approach, we leveraged the functions provided by the **Spark MLlib** library. We constructed a pipeline to execute a series of operations, linking them together to obtain the TF-IDF vector representation of the sequences. This pipeline effectively processed the data, transforming the textual information into numerical vectors using the TF-IDF technique.

```
tokenizer = Tokenizer(inputCol="agged", outputCol="words")
remover = StopWordsRemover(inputCol="words", outputCol="filtered")
ngram = NGram(n=3, inputCol="filtered", outputCol="ngrams")
# cv = CountVectorizer(inputCol="ngrams", outputCol="features", minDF=2.0)
hashingTF = HashingTF(inputCol="ngrams", outputCol="tf")
idf = IDF(inputCol="tf", outputCol="feature")
normalizer = Normalizer(inputCol="feature", outputCol="norm")

cv_pipe = Pipeline(stages=[
    tokenizer,
    remover,
    ngram,
    hashingTF,
    idf,
    normalizer
])
cv_model = cv_pipe.fit(df)
cv_processed = cv_model.transform(df).persist()

cv_processed.count()
```

Execution Times

For our dataset consisting of approximately 500 alumni records, the execution times using an I3.xlarge Databricks cluster for the respective operations in the following approaches were as follows:

- **Bert Embedding:** Command took 2.09 minutes
- **Cosine Similarities:** Command took 0.39 seconds
- **TF-IDF:** Command took 3.43 seconds
- **GloVe Embeddings:** Command took 12.97 seconds
- **TF-IDF after Feature Engineering (Encodings & Geographical Distance):** Command took 5.07 seconds

These execution times provide insights into the efficiency and speed of the different approaches utilized for embedding generation, similarity computation, and feature engineering. Such measurements help assess the performance characteristics of the methods employed and inform decision-making regarding the optimal approach for achieving the desired results.

Machine Learning Outcome

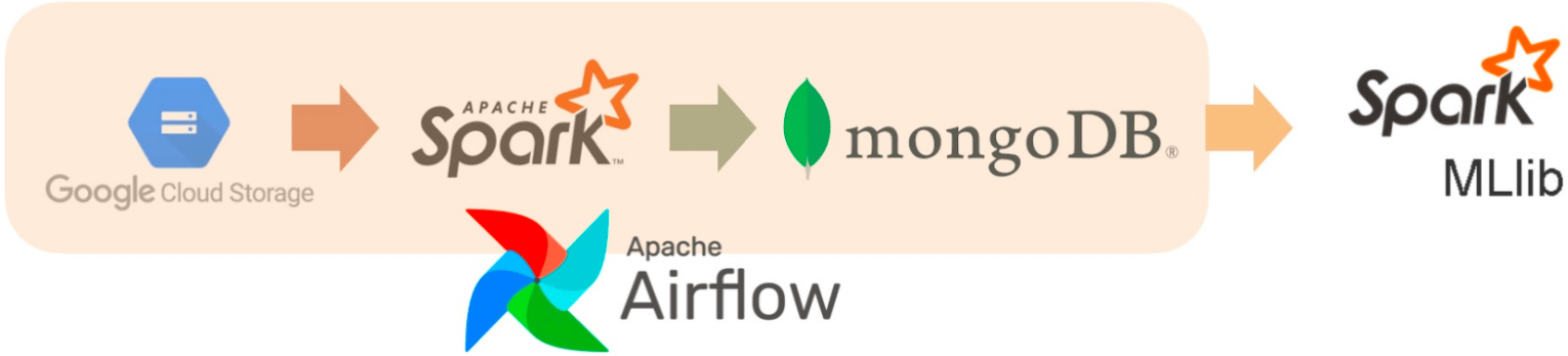
As our problem of suggesting the top 5 similar profiles involves an unsupervised learning approach, we conducted a manual comparison of the results obtained from each of the aforementioned approaches. After careful evaluation, we determined that the **TF-IDF approach**, coupled with the consideration of **geographical distance**, yielded the most favorable outcomes for our specific case.

The TF-IDF approach proved to be the most suitable due to its ability to generate feature vectors by weighting words based on their occurrence within individual profiles and across all profiles. This approach demonstrated superior performance compared to computing contextual sentence embeddings, which would have been excessive given the simplicity of the features we employed for modeling.

By leveraging the TF-IDF approach alongside the consideration of geographical distance, we were able to achieve optimal results in suggesting the top 5 similar profiles for a given input profile. This outcome reinforces the importance of carefully selecting the appropriate techniques that align with the characteristics and requirements of the dataset and problem domain.

Conclusion

We successfully developed a data processing pipeline that matches students with alumni who share similar backgrounds and career goals. Our pipeline recommends the top 5 alumni profiles based on academic and professional background similarities, using a combination of TF-IDF and geographical distance comparisons. This will help students network with alumni who can provide valuable career advice and guidance.



Source code available at: [Alumni Profile Matching]