

**Machine Learning -**

**Services Specialization**

Demo 3: Example of a machine learning model using pre-trained machine learning APIs and AutoML



Index

[Business goal and machine learning solution 3](#_Toc163718690)

[Data Exploration 4](#_Toc163718691)

[Feature engineering 8](#_Toc163718692)

[Preprocessing and data pipeline 13](#_Toc163718693)

[Machine learning model design(s) and selection 17](#_Toc163718694)

[Machine learning model training and development 18](#_Toc163718695)

[Machine learning model Evaluation 24](#_Toc163718696)

[Proof of Deployment 27](#_Toc163718697)

## Business goal and machine learning solution

In today's digital age, where countless news articles are published every day on various platforms, it becomes imperative for news aggregation platform to efficiently categorize news articles into specific topics. This not only enhances the user experience by providing tailored content, but also streamlines content management and optimizes content discovery processes.

**Our goal is to leverage machine learning to automate the categorization of news articles, ensuring fast and accurate dissemination of information to relevant audience segments.**

Our **machine learning solution** falls under supervised learning, specifically, **classification**. By training a model on a labelled dataset of news articles, where each article is tagged with a specific topic (e.g., politics, sports, technology), we aim to develop a system that can automatically predict the topic of new, unlabelled articles. To train our model, in addition to the title and body of the article and the most common metadata already present in our source dataset (e.g., author, newspaper, publication date), we extracted relevant features, such as sentiment, moderation, and entities, from the news text, by leveraging Google's Natural Language API.

Extracting such features not only provides additional value by offering interesting insights into the current news narrative, but also help us discover whether there are recurring trends or unique characteristics of particular topics.

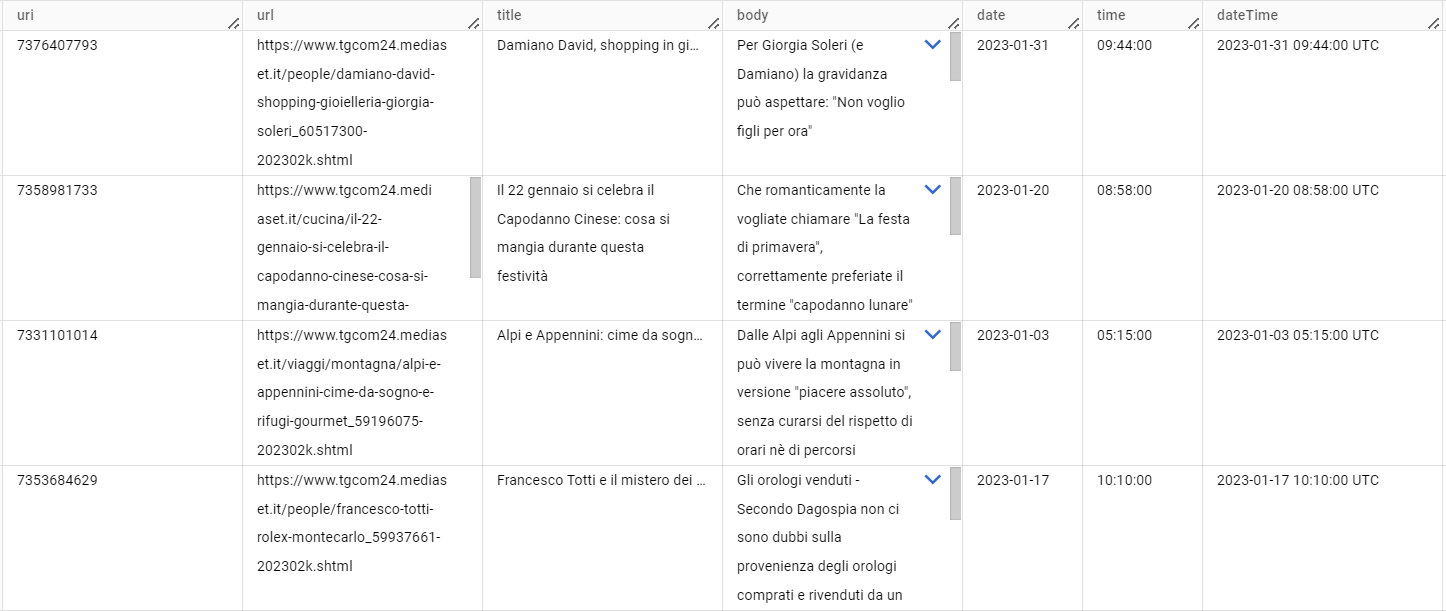
This machine learning solution directly addresses our business question by automating the categorization of news, saving considerable time and resources, and allowing resources to focus on content quality rather than manual categorization. In addition, extracting new features from the text of the article, as well as analysing whether they correlate with the specific topic at hand, provides a more comprehensive view of news trends and could be an important tip that can be leveraged independently of editorial teams.

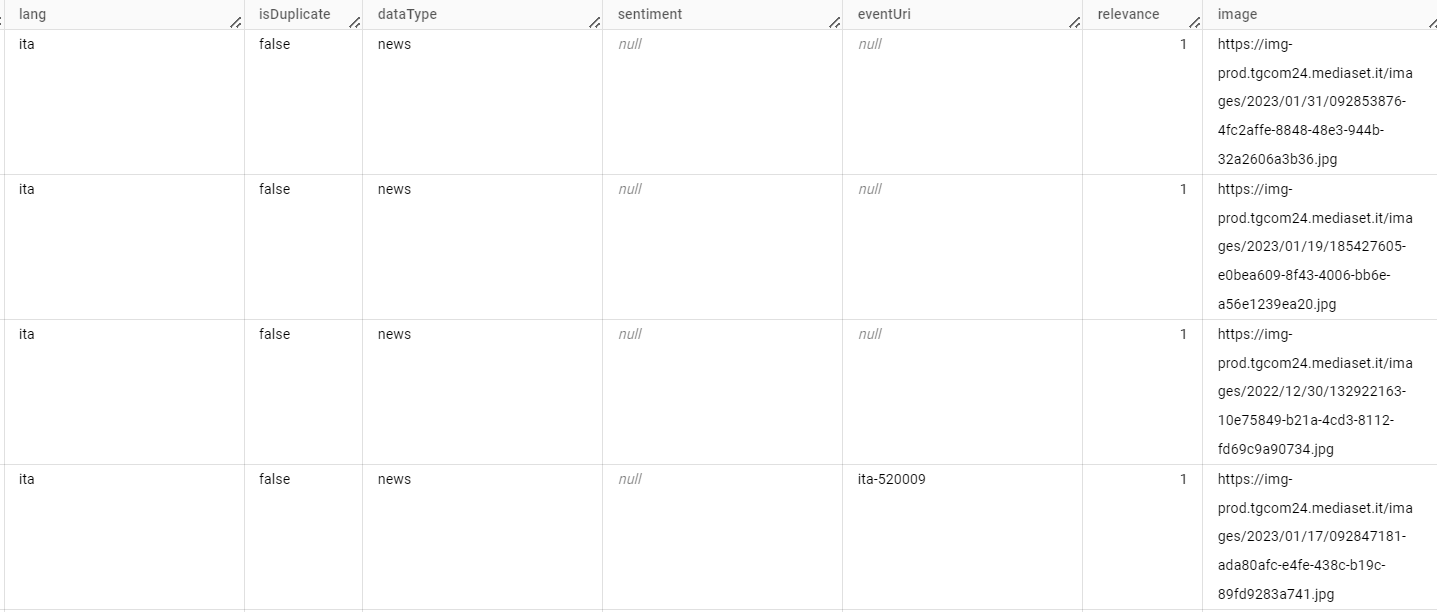
## Data Exploration

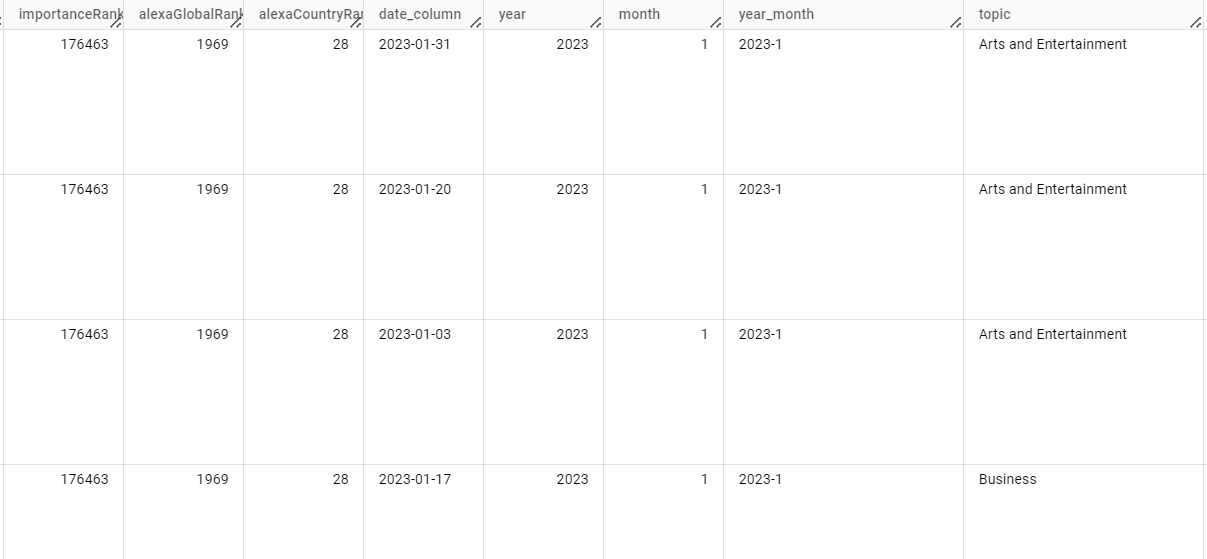
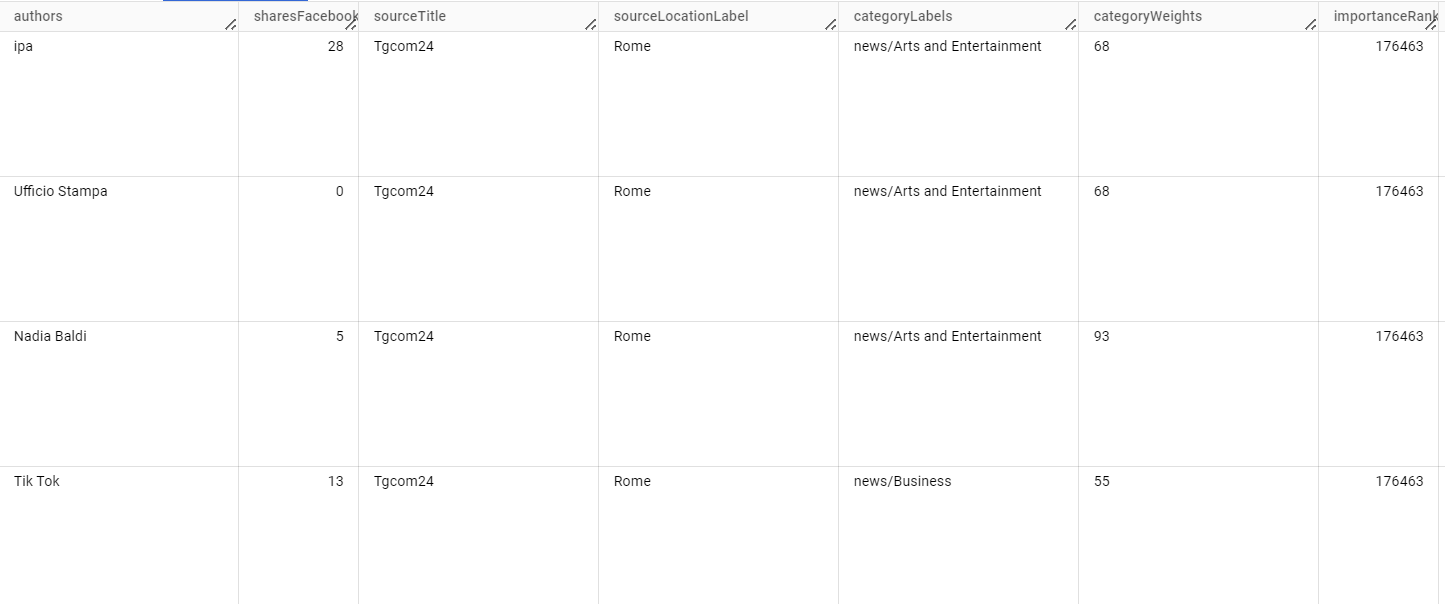
Our source dataset comprises 1,176,306 newspaper articles collected over a decade, from January 2014 to December 2023. Each article in the dataset is richly detailed, capturing not just the narrative through its **title** and **body** but also encompassing a broad array of metadata that enhances its informational value and utility for analysis. Key columns include:

* **Title**: This field represents the article title.
* **body**: Provides the article content.
* **URI** and **URL**: These fields uniquely identify an article and provide a link to its online source, respectively.
* **Date**, **time**, and **dateTime**: These columns record the publication date and time, offering precise temporal context for each article.
* **Lang**: Indicates the language of the article, crucial for linguistic analyses and ensuring appropriate processing techniques are applied.
* **IsDuplicate:** The column, whose possible values are true or false, represents whether the object is duplicated or not in the dataset.
* **dataType:** The field represents the type of data corresponding to that record, in our case it is always news as a value.
* **Sentiment**: Provides an automated sentiment analysis score, indicating the overall tone of the article.
* **EventUri:** Provides reference to the event discussed within the article.
* **Relevance:** The relevance of the document to the query used to extract the news items when constructing the dataset.
* **Image**: Links are provided to the images in the source from which the article is taken.
* **Authors**: Lists the authors of the article, offering insights into authorship patterns and contributions.
* **SharesFacebook**: Tracks the number of times the article was shared on Facebook, providing a proxy for its social media engagement.
* **SourceTitle** and **sourceLocationLabel**: Identify the source of the article and its geographical origin, respectively, highlighting the diversity and reach of the news sources included.
* **CategoryLabels**: Describe the topic associated with an article. This will be the label we will try to predict.
* **importanceRank**, **alexaGlobalRank**, and **alexaCountryRank**: The importance of the source from which the news is extracted according to Amazon Alexa globally and by country.

Dataset Overview:







Despite the wealth of information available initially, evaluating the computational costs and assessing the state of cleanliness and quality of the data directed us to make some decisions preliminary to the entire analysis.

1. nan\_count = df\_no\_dups.isna().sum()

2. print("Total NaN values:", nan\_count)

Total NaN values:

|  |  |
| --- | --- |
| uri | 0 |
| url | 0 |
| title | 1’540 |
| body | 0 |
| date | 0 |
| time | 0 |
| dateTime | 0 |
| dateTimePub | 536’954 |
| lang | 0 |
| isDuplicate | 0 |
| dataType | 0 |
| sentiment | 1175552 |
| eventUri | 897040 |
| relevance | 0 |
| image | 206248 |
| authors | 973795 |
| sharesFacebook | 0 |
| sourceTitle | 0 |
| sourceLocationLabel | 1830 |
| categoryLabels | 286009 |
| importanceRank | 0 |
| alexaGlobalRank | 0 |
| alexaCountryRank | 0 |
| date\_column | 0 |
| year | 0 |
| month | 0 |
| year\_month | 0 |

As we can see in the exploration, we have many rows with missing **CategoryLabels** and **Author** values, so we decided to remove them because we want to keep the author variable in the model.

1. # Remove rows where both "CategoryLabels" and "authors" are NaN

2. df\_no\_dups\_remov = df\_no\_dups.dropna(subset=['categoryLabels', 'authors'])

In addition to the obvious elimination of **duplicates** (identified through the URI), 148’456 with a percentage of 12.5%:

1. # Count the number of rows in each DataFrame

2. Num articles = articles\_df.shape[0]

3. print(f"Number of articles: {num\_articles}")

4.

5. # Find duplicates based on the 'uri' field in articles DataFrame

6. duplicate articles = articles\_df[articles\_df.duplicated(['uri'], keep=False)]

7. print(f"Number of duplicate articles based on 'uri': {duplicate\_articles.shape[0]}")

8.

Number of articles: 1176306

Number of duplicate articles based on 'uri': 148456

Considering that not all of the time span is equally covered by the news in the dataset and that indeed we are dealing with a huge time hole from the end of 2016 to the beginning of 2023, and considering the time variable an important piece of information for our use case (think trivially of the fact that it is very likely that an issue discussed in 2016 has no place at all among the news in 2023), we decided to focus our analysis entirely on the **news from the year 2023 alone**. This decision seemed to us not only reasonable for the above reasons, but also convenient in terms of lowering computational costs.

1. # We get the unique values of years and months

3. articles\_df['year\_month'] = articles\_df['year'].astype(str) + '-' + articles\_df['month'].astype(str)

4.

5.

6. unique\_year\_month = articles\_df['year\_month'].unique()

7.

8. print("Unique Year-Month Combinations:", unique\_year\_month)

9.

|  |
| --- |
| *Unique Year-Month Combinations:* |
| *['2016-12' '2016-11' '2016-10' '2016-9' '2016-8' '2016-7' '2016-6'*  *'2016-5' '2016-4' '2016-3' '2016-2' '2016-1' '2023-7' '2023-6' '2023-5'*  *'2023-4' '2023-3' '2023-2' '2023-1' '2015-12' '2015-11' '2015-10'*  *'2015-9' '2015-8' '2015-7' '2015-6' '2015-5' '2015-4' '2015-3' '2015-2'*  *'2015-1' '2023-11' '2023-10' '2014-12' '2014-11' '2014-10' '2014-9'*  *'2014-8' '2014-7' '2014-6' '2014-5' '2014-4' '2014-3' '2014-2' '2014-1'*  *'2023-12']* |

Since our goal is topic classification, we choose the column **categoryLabels** as a reference for our topics and exclude all records where it is empty. Since we find more than one topic associated (only for a few cases) in our source data for each item, we solve these borderline cases by choosing only the first of the proposed ones. By performing these operations, we created from the column categoryLabels our target column, which we renamed **Topic** for simplicity.

Our dataset will therefore possess for each article one of the following eight labels:

|  |  |
| --- | --- |
| **TOPIC** | Number of items |
| **Politics** | 11535 |
| **Arts and Entertainment** | 11511 |
| **Business** | 7392 |
| **Sports** | 3633 |
| **Health** | 2221 |
| **Technology** | 1202 |
| **Environment** | 391 |
| **Science** | 317 |

Table1. Topic sample of the input dataset.

The imbalance between the represented classes assessed during data exploration, along with the evaluation of computational costs, led us to consider a down-sampling strategy in creating the training set of our model, the reasons and methods of which will be specified in detail in the specific section on model training.

Due to the state of cleanliness of the source dataset, and mainly due to the presence of many **null values**, some of the originally present features are unusable as is. This will lead us to some choices at the features selection and features engineering steps.

* Consider the **sentiment** column, which, as we shall see, counting most of the values as null, will need to be replaced by a new one, created by us through Natural Language API.
* Equally mainly empty, the column for events (**EventUri**) we decided instead to exclude.
* Other columns did not seem usable to us, either those that have no informational value (think of the **Lang** column, which always has Italian as its value, or **dataType**, always filled in as News), or those that show an overlap of information with respect to the others (such as the three columns on the **date**, from which we will construct only one of the format that will be most convenient for us) or those that are of no interest to us for our specific case (such as the reference to the links to the images in the text).

The complete list of columns that we have omitted will be specified in the section Feature engineering.

* We also explored the **location** column, with most news items located in Rome, many news items labelled generically as "Italy", while only a few reported a location that was also Italian but different. Since the information in this column appears to be both unbalanced but also somewhat inconsistent for the above reason, we will exclude it.

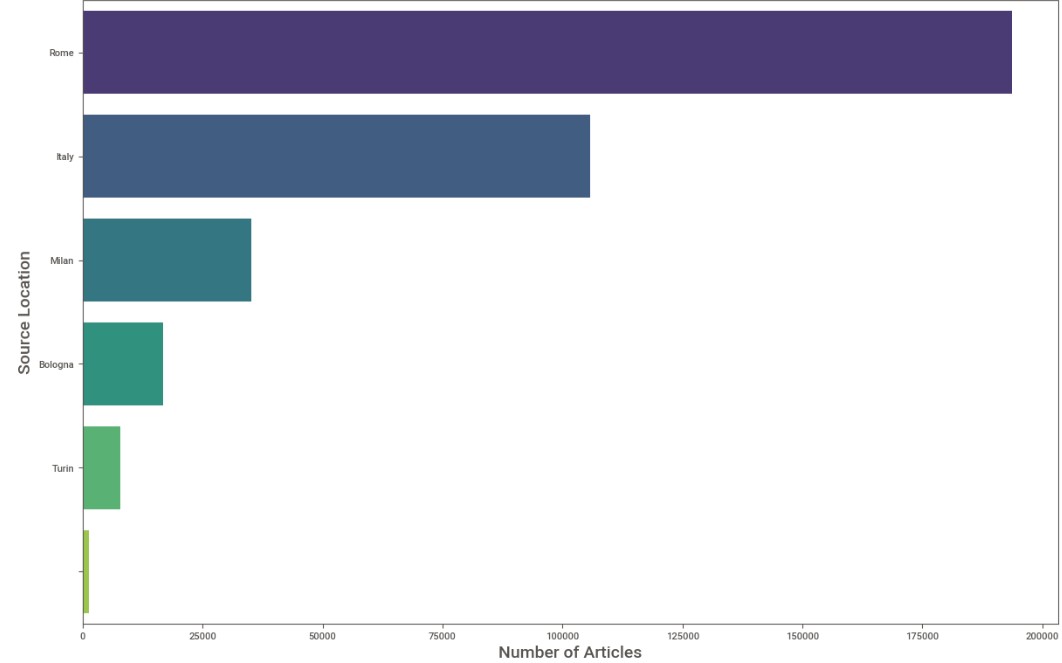


Figure 1 - Top Source Locations

Although always unbalanced, we still choose to consider the author and source columns that still contain some informational value with respect to our case, since there may be newspapers and/or authors more inclined to write about certain topics instead of others.

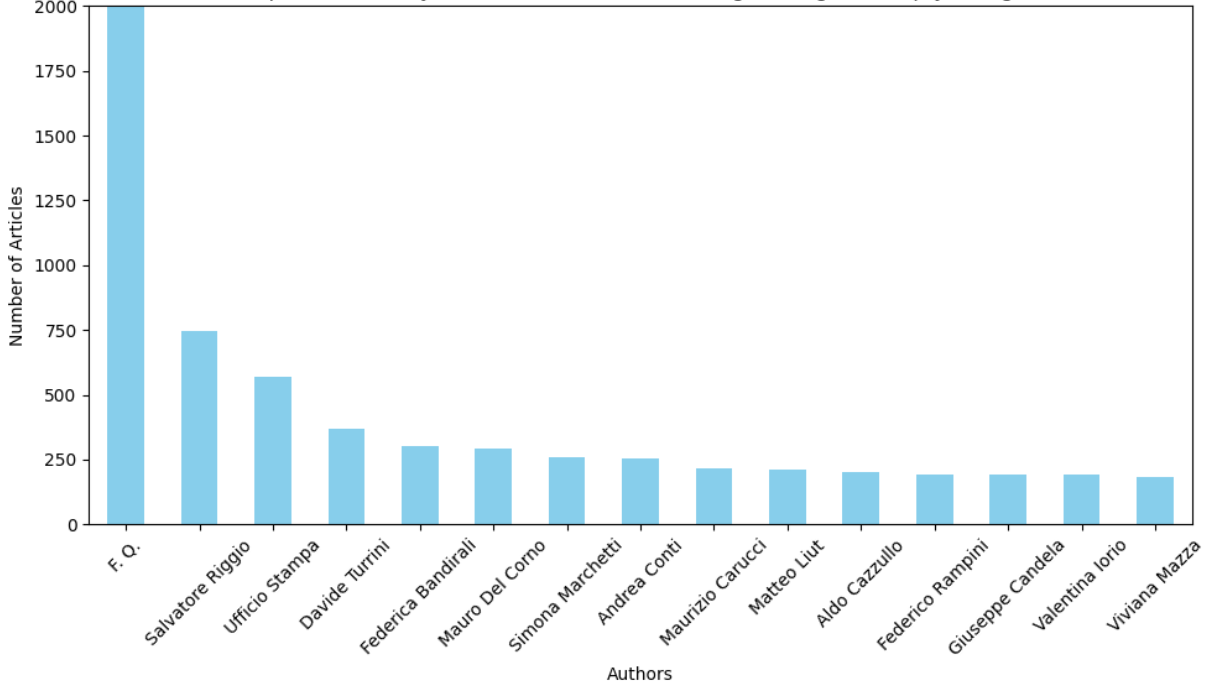


Figure 2 - Top 15 Authors by number of news

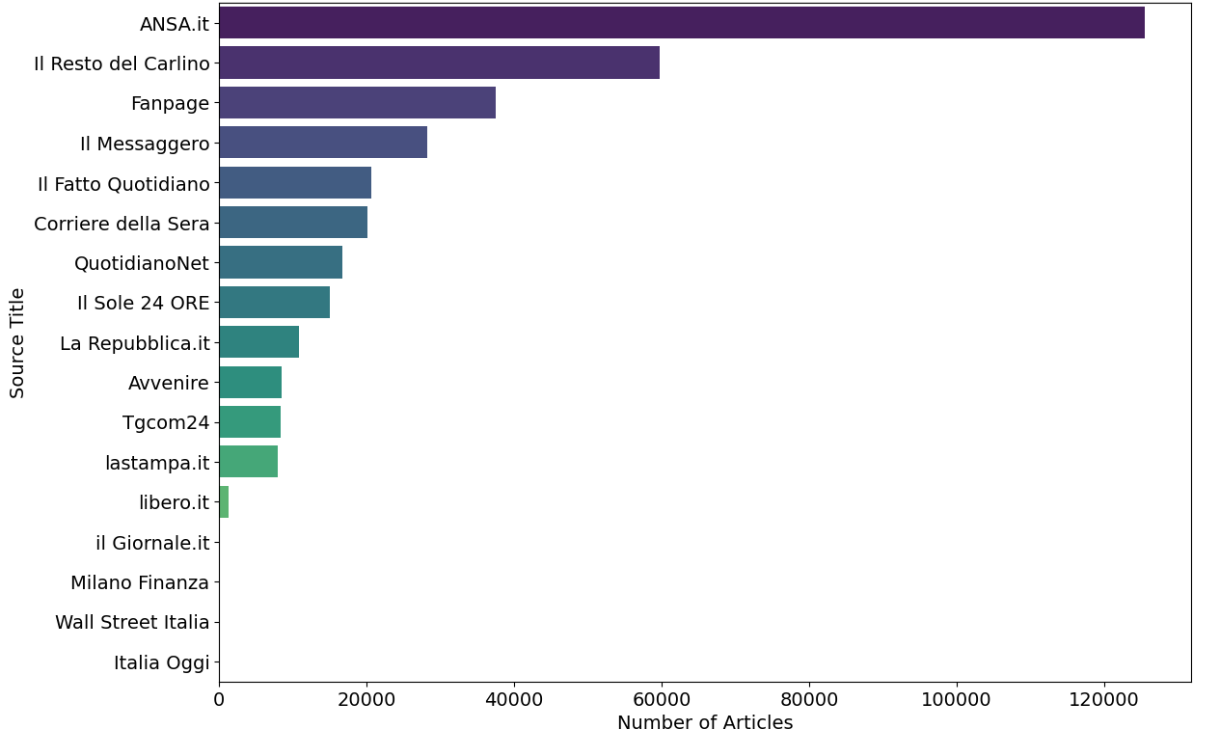


Figure 3-Top 15 Source by number of news

## Feature engineering

Utilizing Google's pre-trained API, we enriched our dataset with new features.

As we have seen in data exploration section, the sentiment values within our dataset were predominantly null, prompting us to replace these with the more comprehensive **Sentiment analysis** provided by the Natural Language API. We get, therefore, two new features, the **sentiment score** of a document i.e., the overall emotional leaning of the text, ranging from -1.0 (indicating a negative sentiment) to 1.0 (indicating a positive sentiment), and the **magnitude** of sentiment i.e., the strength or intensity of emotion present in the text, which can range from 0.0 to an upper bound that is theoretically infinite. This value aggregates the emotional intensity of each expression within the article, irrespective of the sentiment's polarity, thus providing an indication of the emotional richness or emotional charge of the content.

Since a neutral score (around 0.0) may indicate a document with low emotionality, or it may indicate mixed emotions, with high positive and negative values cancelling each other out. We decided to carry magnitude values to disambiguate these cases, since truly neutral documents will have a low magnitude value, while mixed documents will have higher magnitude values, and so the information contained in the magnitude column also seemed relevant.

|  |
| --- |
| 1. import pandas as pd  2. from google.cloud import language\_v1  3. import json  4.  5. def analyze\_text\_sentiment(row, text\_column: str = 'preprocessed\_text', num\_documents: int = 10) -> dict:  6. client = language\_v1.LanguageServiceClient()  7. text = row[text\_column]  8. document = language\_v1.Document(content=text, type\_=language\_v1.Document.Type.PLAIN\_TEXT)  9. response = client.analyze\_sentiment(request={'document': document})  10.  11. # Returns a dictionary with the results of sentiment analysis  12. return {  13. 'score': response.document\_sentiment.score,  14. 'magnitude': response.document\_sentiment.magnitude,  15. 'num\_documents': num\_documents  16. }  17.  18. num\_documents=10  19. # Applies the analyze\_text\_sentiment function to each row in the DataFrame  20. sentiment\_results = final\_df1.head(num\_documents).apply(lambda row: analyze\_text\_sentiment(row, text\_column='body\_pre', num\_documents=num\_documents), axis=1)  21.  22. # Create a DataFrame from the results of sentiment analysis  23. df\_sentiment = pd.DataFrame(sentiment\_results.tolist())  24.  25. # Merges the original DataFrame with the results of sentiment analysis  26. df\_sentiment = pd.concat([final\_df1.head(num\_documents), df\_sentiment], axis=1) |

This code snippet performs sentiment analysis on text data using Google Cloud Natural Language API. It defines a function analyze\_text\_sentiment to analyze sentiment for each row in a DataFrame, then applies this function to a subset of rows, creates a DataFrame from the sentiment analysis results, and merges it with the original DataFrame.

In addition, again using the Natural Language API, we performed **Text Moderation** analysis, which is the analysis of a document against a list of Security Attributes, which include "malicious categories" and topics that may be considered sensitive. Each Safety Attribute became a column in the final dataset, whose values for each document consisted of a **confidence score**, a value between 0.00 and 1.00, reflecting the probability that the input belongs to a given category.

|  |
| --- |
| 1. def moderate\_text(text: str) -> language.ModerateTextResponse:  2. client = language.LanguageServiceClient()  3. document = language.Document(  4. content=text,  5. type\_=language.Document.Type.PLAIN\_TEXT,  6. )  7. return client.moderate\_text(document=document)  8.  9. class MLStripper(HTMLParser):  10. def \_\_init\_\_(self):  11. super().\_\_init\_\_()  12. self.reset()  13. self.strict = False  14. self.convert\_charrefs = True  15. self.text = []  16.  17. def handle\_data(self, d):  18. self.text.append(d)  19.  20. def get\_data(self):  21. return ''.join(self.text)  22.  23. def strip\_html\_tags(html):  24. s = MLStripper()  25. s.feed(html)  26. return s.get\_data()  27.  28. def moderate\_text(text: str) -> language.ModerateTextResponse:  29. client = language.LanguageServiceClient()  30. document = language.Document(  31. content=text,  32. type\_=language.Document.Type.PLAIN\_TEXT,  33. )  34.  35. return client.moderate\_text(document=document)  36.  37. def analyze\_documents(df: pd.DataFrame, num\_documents: int = 2, column:str = 'text'):  38. documents = []  39. for index, row in df.iterrows():  40. linetext\_cleaned = row.get(column, '')  41. if linetext\_cleaned:  42. # Perform moderation analysis on the cleaned text  43. response = moderate\_text(linetext\_cleaned)  44. categories = []  45. confidences = []  46. for category in response.moderation\_categories:  47. categories.append(category.name)  48. confidences.append(category.confidence)  49. document = row.to\_dict()  50. for category, confidence in zip(categories, confidences):  51. # Create new columns for each category with confidence scores  52. document[category] = confidence  53. documents.append(document)  54. if len(documents) >= num\_documents:  55. break # Stop after processing the specified number of documents  56.  57. df\_result = pd.DataFrame(documents)  58. return df\_result  59.  60. df\_moderated = analyze\_documents(final\_df2, num\_documents=row\_count, column="body\_pre") |

The code performs moderate\_text function utilizes the Google Cloud Natural Language API to moderate text content. The MLStripper class is used to strip HTML tags from text. The analyze\_documents function iterates through rows in a DataFrame, performs moderation analysis on the text content, and creates new columns for each moderation category with confidence scores. Finally, it calls the analyze\_documents function with specified parameters and assigns the result to a DataFrame df\_moderated.

Furthermore, we performed **Entity analysis**. The Natural Language API returns in this case a set of detected entities for each article and parameters associated with those entities, such as the entity's type (for example if the entity is a person, location, consumer good, etc.), salience, i.e., relevance of the entity to the overall text, and locations in the text that refer to the same entity.

In the interests of maintaining a concise and focused dataset, we opted not to include all extracted entities. This decision was made to avoid the creation of an overly sparse dataset and to prevent the duplication of information already encapsulated within the text. Instead, we introduced a novel approach where we summarized the entities by their types within the dataset columns and captured the average salience of each type as the column values. So, our final dataset will contain as many new columns as there are entity types extracted and for each item, the value in the column is the average salience of the entities of that type for that specific item. This methodology allows us to preserve essential information while ensuring the dataset remains manageable and relevant.

|  |
| --- |
| 1. from google.cloud import language  2.  3. def analyze\_text\_entities(text: str) -> language.AnalyzeEntitiesResponse:  4. client = language.LanguageServiceClient()  5. document = language.Document(  6. content=text,  7. type\_=language.Document.Type.PLAIN\_TEXT,  8. )  9. return client.analyze\_entities(document=document)  10.  11. def analyze\_document\_entities(df: pd.DataFrame, num\_documents: int = None, column:str = 'text'):  12. if num\_documents is None:  13. num\_documents = len(df)  14.  15. documents = []  16.  17. for index, row in df.iterrows():  18. if len(documents) >= num\_documents:  19. break # Stop after processing the specified number of documents  20.  21. linetext\_cleaned = row.get(column, '')  22.  23. if linetext\_cleaned:  24. # Perform entity analysis on the cleaned text  25. response = analyze\_text\_entities(linetext\_cleaned)  26. entity\_type\_counts = {}  27. entities\_data = {}  28.  29. for entity in response.entities:  30. entity\_type = entity.type\_.name  31. salience = entity.salience  32.  33. # Update entity counts  34. entity\_type\_counts[entity\_type] = entity\_type\_counts.get(entity\_type, 0) + 1  35.  36. # Append saliences to the list corresponding to the entity type  37. if entity\_type not in entities\_data:  38. entities\_data[entity\_type] = []  39.  40. entities\_data[entity\_type].append(salience)  41.  42. # Calculate mean salience for each entity type  43. mean\_salience\_per\_entity\_type = {}  44. for entity\_type, salience\_list in entities\_data.items():  45. mean\_salience\_per\_entity\_type[entity\_type] = sum(salience\_list) / len(salience\_list)  46. #print(mean\_salience\_per\_entity\_type)  47.  48. # Create a copy of the current row as a dict and update it with entity counts  49. document = row.to\_dict()  50. document.update(entity\_type\_counts)  51.  52. # Add mean salience for each entity type to the document  53. for entity\_type, mean\_salience in mean\_salience\_per\_entity\_type.items():  54. document[f"{entity\_type}\_mean\_salience"] = mean\_salience  55.  56. documents.append(document)  57.  58. # Create a new DataFrame that includes the original data and the new entity count columns  59. df\_result = pd.DataFrame(documents)  60.  61. # Fill missing entity count values with 0  62. entity\_columns = [col for col in df\_result.columns if col not in df.columns]  63. df\_result[entity\_columns] = df\_result[entity\_columns].fillna(0).astype(float)  64.  65. return df\_result  66.  67. df\_entities = analyze\_document\_entities(final\_df3, column="body\_pre") |

The code performs analyze\_text\_entities function to analyze entities in text content using Google Cloud Natural Language API. Create analyze\_document\_entities function to iterate through DataFrame rows, perform entity analysis on text content, calculate entity counts and mean salience, and return a new DataFrame with entity count columns appended. Finally, call analyze\_document\_entities function with specified parameters and assign the result to DataFrame df\_entities.

Each of these steps, now shown as examples, will become a component of our data pipeline following common preprocessing tasks. An overall idea of the pipeline that includes each of the cases discussed here will be given in the following section.

The original dataset, specifically the **sharesFacebook** column, exhibited a significant presence of outliers and a non-normal distribution, common in social media metrics where a few articles might receive disproportionately high shares. Given these characteristics, we opted to apply the **RobustScaler**. This choice was driven by the scaler's ability to mitigate the influence of outliers by using the median and interquartile range for scaling, rather than the mean and standard deviation, which can be heavily affected by extreme values. This approach ensures that our model training process is not skewed by these anomalies, allowing for a more accurate and reliable analysis of how Facebook shares relate to other features in predicting outcomes.

1. final\_df["sharesFacebook"].describe()

2. count 4322.000000

3. mean 30.348681

4. std 109.517453

5. min 0.000000

6. 25% 1.000000

7. 50% 7.000000

8. 75% 26.000000

9. max 3091.000000

10. from sklearn.preprocessing import RobustScaler

11. # Create a RobustScaler object

12. scaler = RobustScaler()

13. # If you want to apply it to a DataFrame column:

14. final\_df['shares\_scaled'] = scaler.fit\_transform(final\_df['sharesFacebook'].values.reshape(-1, 1))

15. final\_df["shares\_scaled"].describe()

16. count 4322.000000

17. mean 0.933947

18. std 4.380698

19. min -0.280000

20. 25% -0.240000

21. 50% 0.000000

22. 75% 0.760000

23. max 123.360000

24.

The code shows a describe from the final dataset.

**Features Selection**

It was decided to exclude the following columns because they contained either irrelevant information or were directly correlated with the topic, such as category weights and category labels. Not all features extracted with the entities analysis were used; we only kept those recalculated with the mean salience within the document and excluded those related to dates and numbers, which we deemed less relevant.

And to reduce the complexity of the model, only those features that did not have correlated information were retained.

**Columns omitted:**

*uri,url,date,body,time,dateTime,dateTimePub,lang,isDuplicate,dataType,sentiment,eventUri,image,sharesFacebook,sourceLocationLabel,categoryLabels,categoryWeights,alexaCountryRank,date\_column,year,year\_month,num\_documents.*

*Columns keeped*

## Preprocessing and data pipeline

The **data-preprocessing** pipeline component begins with the retrieval and validation of data stored in a parquet file in Google Cloud Storage (GCS). Leveraging the Google Cloud Storage API, the pipeline ensures data integrity and completeness before proceeding with preprocessing.

1. @component(base\_image=f"{REGION}-docker.pkg.dev/{PROJECT\_ID}/{REPO\_NAME}/data\_preparation:latest")

2.

3. def data\_preprocessing(

4. bucket: str,

5. file\_path: str,

6. folder: str,

7. parquet\_file\_name: str,

8. processed\_dataset: Output[Artifact]

9. ):

10.

11. import

12.

13. from processing.data\_preparation import GCSParquetLoader

14.

15. logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s\n')

16.

17. processor = GCSParquetLoader(bucket, file\_path, folder, parquet\_file\_name)

18. processed\_dataset.uri = processor.process()

19.

The code shows the component from the preprocessing pipeline.

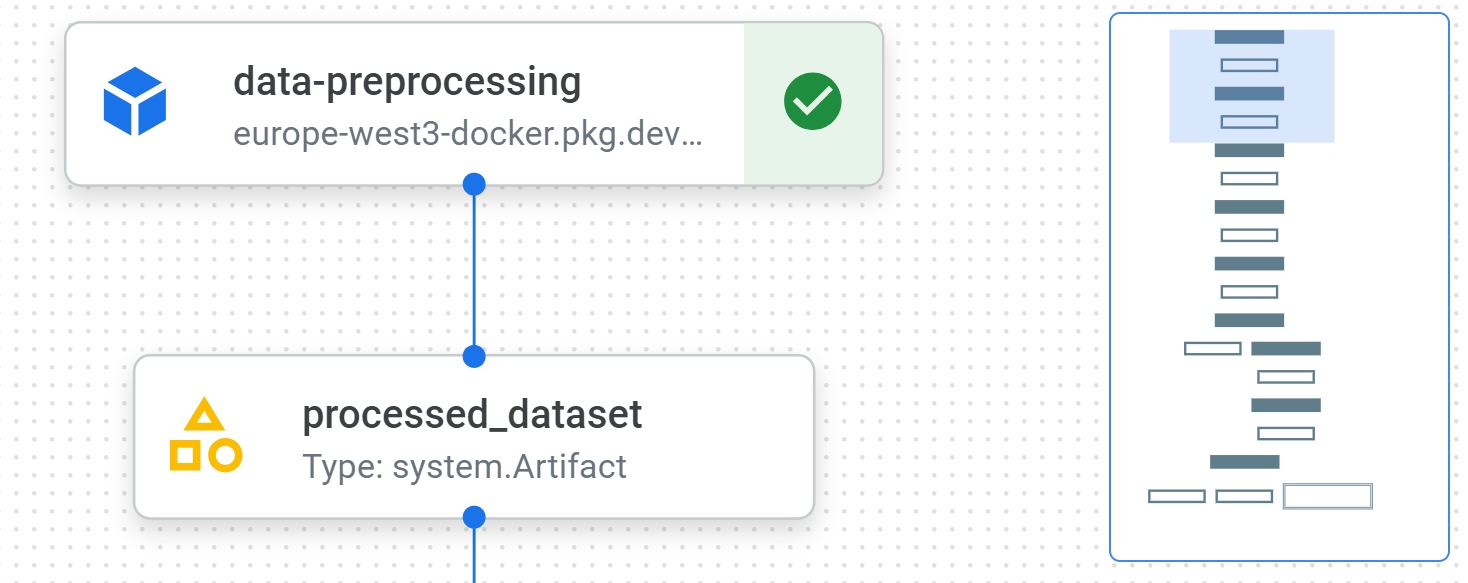


Figure 4. Pipeline first component data-preprocessing on VertexAI

The data preprocessing component encompasses several steps.

* **Data Loading and Transformation**: Upon validation, the data is loaded into memory and transformed into a structured format suitable for analysis. Utilizing Pandas, the pipeline handles data loading, manipulation, and transformation operations seamlessly.
* **Data Cleaning and Feature Engineering**: The heart of the preprocessing pipeline lies in data cleaning Techniques such as handling missing values, duplicate removal, and feature extraction are employed to ensure data quality.

and relevance.

* **Temporal Analysis and Filtering**: The pipeline incorporates temporal analysis to filter data based on specific time ranges. This enables the extraction of insights from data collected within defined temporal boundaries, enhancing the relevance of the dataset.
* **Normalization and Scaling**: To mitigate the effects of data skewness and heterogeneity, the preprocessing pipeline employs normalization and scaling techniques. RobustScaler from scikit-learn is utilized to scale numerical features, ensuring uniformity and stability across the dataset.
* **Splitting and Down-sampling**: Finally, the preprocessed dataset is split into training, validation, and test sets using stratified sampling. This facilitates model training, evaluation, and validation while ensuring representative distribution across classes.

The preprocessed dataset will be saved to Google Cloud Storage at each step and subsequently read at each step. The subsequent steps include:

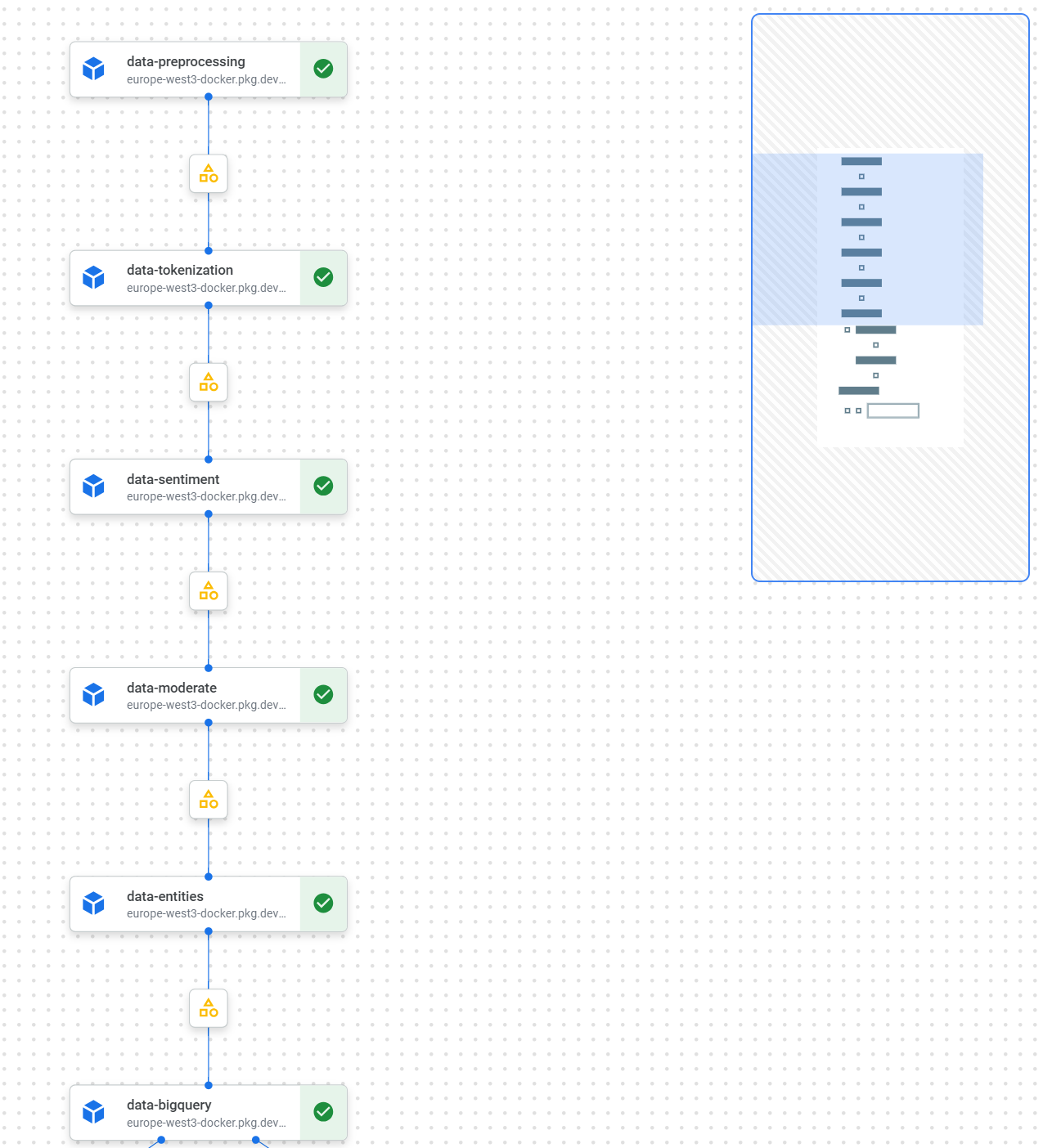


Figure 5 – Full preprocessing pipeline

1. **Data-tokenization** using method *preprocess\_text* tokenizes and preprocesses text data by removing stopwords and lemmatizing words.
2. **Data-sentiment** using method *analyze\_text\_sentiment* uses Google Cloud Natural Language API to analyze the sentiment of text data.
3. **Data-moderate** using method *moderate\_text* moderates text content using Google Cloud Natural Language API.
4. **Data-entities** using method *analyze\_text\_entities* analyzes entities in text data using Google Cloud Natural Language API.
5. **Data-bigquery** using method *upload\_dataframe\_to\_bigquery* uploads the DataFrame to BigQuery.

The described code defines a class named **GCS\_preprocessing**, which encapsulates various methods for preprocessing and analyzing data stored in Google Cloud Storage (GCS) and then uploading it to BigQuery.

1. @app.post("/preprocess/")

2. def preprocess(file\_path: str, num\_doc: int):

3. print("----- Running Preprocessing ---------")

4. try:

5. # Preprocess the data

6. instances, preprocessed\_df = preprocess\_data(file\_path, num\_doc)

7.

8. # Convert DataFrame to JSON

9. json\_data = preprocessed\_df.to\_json(orient="records")

10.

11. print("JSON data:", json\_data)

12. return json\_data

13.

14. except Exception as e:

15. raise HTTPException(status\_code=500, detail=str(e))

The code shows the preprocess callable API that returns a json with the dataset preprocessed.

## Machine learning model design(s) and selection

Since we have used pre-trained machine learning APIs with the main goal of building new features to train AutoML, we have detailed the main description of their use in the Feature Engineering section. Since our goal was primarily to pick up interesting hints from textual content, we confronted with the Natural Language API, which offers various methods for performing text analysis. For completeness, we give here the list of the used Natural Language API methods.

* Natural Language API for Sentiment Analysis
* Natural Language API for Text Moderation Analysis
* Natural Language API for Entity Analysis

Some of the general reasons justifying the choice of having used these methods to enrich our training dataset:

* **Enhanced Semantic Understanding**: The Natural Language API provides insights into the semantic structure of texts, such as entity recognition, sentiment analysis, and moderation analysis. By including these features, the model can better understand the context of news articles. This leads to more accurate classification, as the model can differentiate topics based on deeper textual understanding.
* **Data enrichment**: The addition of these features effectively enriches the dataset, providing the classification model with a wider range of signals from which to learn. Enriched datasets usually lead to better performing models because they capture a wider range of patterns and relationships within the data.
* **Addressing implicit topics**: Finally, some topics may not be explicitly stated in the text but implied through some patterns or associations. Security attributes and other functions of the natural language API can help uncover these implicit topics by highlighting underlying themes or concerns that may not be immediately apparent through traditional text analysis techniques.

Without going into the details of the individual analyses that were covered in the appropriate section, let us move on to describe the **AutoML** that we used for our final topic classification.

Because our source data, the original data and the data constructed later, goes beyond simple textual information, we immediately moved toward using an **AutoML on a tabular database**. In fact, because it can handle text along with other data types within the same dataset, it immediately seemed the most effective solution for our use case. In this way we could take advantage of all the information at our disposal, and evaluate, based on the model's response to such rich and varied input, which features were actually relevant for classification, without initially orienting ourselves to text alone. Since our labels are ultimately the nine possible topics with which a news can be associated and, therefore, ours is a multi-label classification problem, we opted to use **the AutoML for multi-class classification.**

1. AutoMLTabularTrainingJob(  
    display\_name: str,  
    optimization\_prediction\_type: str,  
    optimization\_objective: typing.Optional[str] = None,  
    column\_specs: typing.Optional[typing.Dict[str, str]] = None,  
    column\_transformations: typing.Optional[  
        typing.List[typing.Dict[str, typing.Dict[str, str]]]  
    ] = None,  
    optimization\_objective\_recall\_value: typing.Optional[float] = None,  
    optimization\_objective\_precision\_value: typing.Optional[float] = None,  
    project: typing.Optional[str] = None,  
    location: typing.Optional[str] = None,  
    credentials: typing.Optional[google.auth.credentials.Credentials] = None,  
    labels: typing.Optional[typing.Dict[str, str]] = None,  
    training\_encryption\_spec\_key\_name: typing.Optional[str] = None,  
    model\_encryption\_spec\_key\_name: typing.Optional[str] = None,  
)

The code shows the AutoML performed for the training.

## Machine learning model training and development

For the training step, we employed a strategy for dividing a dataset into training, validation, and test, specifically tailored for datasets that encompass both a diversity of topics and a temporal dimension. This approach aligns with the dynamic nature of news content, where the relevance and distribution of topics can evolve over time. Although it is possible to set up autoML training directly with a chronological order in the data, we decided to do this preliminarily to ensure that not only the temporal order was respected but also that all topics were equally represented in the training dataset. In fact, this strategy allowed us to handle the imbalance in label representation that was highlighted during data exploration. Let us see how.

In creating the training dataset, we followed the following steps in order:

* **Temporal Sorting**: Initially, the dataset is sorted based on a 'date\_column', ensuring that all entries are organized chronologically. This step is crucial for maintaining the temporal integrity of the dataset, allowing for realistic simulation of how the model will be applied in practice, where future data points follow those seen during training.
* **Topic Grouping**: The sorted dataset is then grouped by 'topic', which allows for a nuanced approach to handling diverse data. This step ensures that each topic is represented in the training, validation, and test sets.
* **Custom Split Allocation**: For each topic group, a custom function **assign\_labels** is used to allocate entries to the training, validation, and test sets. Specifically, 95% of the data is designated for training, with the remaining 5% equally split between validation and test sets. This allocation is performed sequentially within each group, reflecting a commitment to respecting the chronological order of data points, an approach particularly important for datasets where temporal trends and patterns may influence model performance.

Following this strategy, we obtained the following desiderata:

* **Balanced Representation Across Time**: The strategy ensures that each split not only contains a proportional representation of each topic but also spans a broad temporal range. This is achieved by calculating the date range for each topic within each split, allowing for a clear understanding of the temporal coverage of the training, validation, and test sets. This step guarantees that the model is exposed to, and evaluated on, a wide array of scenarios, enhancing its ability to handle new, unseen data.
* **Ensuring Model Robustness**: By organizing the dataset in this manner, the strategy enhances the robustness and generalizability of the model. It ensures that the model is trained on a comprehensive sample of the data, evaluated on recent, unseen examples, and tested on the latest data points, closely mimicking real-world conditions where the model needs to predict outcomes for future data.

|  |
| --- |
| \1. # Sort the DataFrame by date  2. df\_sorted = df\_2023\_cl.sort\_values(by='date\_column')  3.  4. # Group the DataFrame by topic  5. grouped = df\_sorted.groupby('topic')  6.  7. # Define a function to assign labels based on date ranges  8. def assign\_labels(group):  9. total\_count = len(group)  10. train\_count = int(0.95 \* total\_count) # 95% of observations for TRAIN  11. validate\_test\_count = (total\_count - train\_count) // 2 # Remaining observations for VALIDATE and TEST  12.  13. # Assign labels based on date ranges  14. group.loc[group.index[:train\_count], 'split'] = 'TRAIN'  15. group.loc[group.index[train\_count:train\_count + validate\_test\_count], 'split'] = 'VALIDATE'  16. group.loc[group.index[train\_count + validate\_test\_count:], 'split'] = 'TEST'  17.  18. return group['split'] # Return the 'split' column as a Series  19.  20. # Apply the function to each group and concatenate the results  21. split\_series = pd.concat([assign\_labels(group) for \_, group in grouped])  22.  23. # Assign the resulting Series to a new column in the original DataFrame  24. df\_2023\_cl['split'] = split\_series |

The code shows the method for splitting the dataset in training, validation and test.

After the initial split, the training dataset is further refined through down-sampling to address class imbalance:

* **Identifying the Minority Class Size**: The smallest class size within the training data is identified. This size becomes the target number of samples for each topic in the training dataset, ensuring equal representation across all classes. This step is crucial for mitigating the risk of a model that overfits to more frequently occurring topics at the expense of minority classes.
* **Equalizing Class Representation**: The training dataset is down sampled by randomly selecting a number of instances for each topic that matches the size of the smallest class. This process creates a balanced training dataset where each topic is equally represented.

1. # Filter the DataFrame to include only "TRAIN" observations

2. train\_df: pd.DataFrame = df\_2023\_cl[df\_2023\_cl['split'] == 'TRAIN']

3.

4. # Downsample the "TRAIN" section

5. minority\_class\_size = min(train\_df['topic'].value\_counts())

6. downsampled\_train\_df = train\_df.reset\_index(drop=True).groupby('topic').apply(lambda x: x.sample(minority\_class\_size)).reset\_index(drop=True)

7. # Concatenate downsampled DataFrames for "TEST" and "VALIDATION" splits

8. final\_df = pd.concat([downsampled\_train\_df, df\_2023\_cl[df\_2023\_cl['split'] == 'VALIDATE'], df\_2023\_cl[df\_2023\_cl['split'] == 'TEST']])

9. final\_df.reset\_index(drop=True)

10. print(final\_df['split'].value\_counts())

11.

12. split

13. TRAIN 2408

14. TEST 958

15. VALIDATE 956

16.

The code shows a print of our sample training set.

**Development**

Once the training dataset was prepared, we created the model with these characteristics:

1. training\_op = AutoMLTabularTrainingJobRunOp(

2. project = project,

3. display\_name = display\_name,

4. optimization\_prediction\_type = optimization\_prediction\_type,

5. optimization\_objective=optimization\_objective,

6. budget\_milli\_node\_hours = budget\_milli\_node\_hours,

7. disable\_early\_stopping=disable\_early\_stopping,

8. column\_specs = features,

9. #column\_transformations = features\_list,

10. dataset = dataset\_create\_op.outputs['dataset'],

11. target\_column = var\_target,

12. predefined\_split\_column\_name = 'split',

13. labels = labels,

14. location = location

15. )

16.

17. model\_eval\_task = classification\_model\_eval\_metrics(

18. project=project,

19. location=location,

20. staging\_bucket=staging\_bucket,

21. thresholds\_dict\_str=thresholds\_dict\_str,

22. model=training\_op.outputs["model"],

23. )

24.

25. endpoint = aip.Endpoint.list(filter=f'labels.notebook={NOTEBOOK}')[0].display\_name

26. with dsl.If(

27. model\_eval\_task.outputs["dep\_decision"] == "true",

28. name="deploy\_decision",

29. ):

30. print("Create a new Endpoint")

31. endpoint\_op = EndpointCreateOp(

32. project=project,

33. location=location,

34. labels=labels,

35. display\_name=endpoint\_name,

37. )

38.

39. print("Endpoint: Deployment of Model")

40. deployment = ModelDeployOp(

41. model=training\_op.outputs["model"],

42. endpoint=endpoint\_op.outputs["endpoint"], # Use the newly created endpoint

43. dedicated\_resources\_min\_replica\_count=1,

44. dedicated\_resources\_max\_replica\_count=1,

45. traffic\_split={"0": 100},

46. dedicated\_resources\_machine\_type=deploy\_machine

47. )

48.

The code shows the training pipeline where evaluation were performed.

Another important consideration is the choice of optimization objective in the training phase.

Choosing **log loss** for optimizing multi-label news article classification taps into the nuanced challenges of categorizing complex content. It's a choice that values precision in predicting probabilities and acknowledges the inherent ambiguity in news topics. Log loss uniquely penalizes overconfident incorrect predictions, encouraging models to be accurate yet humble in their certainty. This approach aligns with the fluid nature of news, where articles often overlap across multiple topics, requiring a careful balance between confidence and caution.

Furthermore, log loss's compatibility with gradient-based optimization means models can be efficiently trained to navigate the subtleties of news classification. It ensures models not only make accurate predictions but also provide insights into their confidence levels, offering calibrated probabilities that reflect the true likelihood of an article belonging to certain categories. This sophistication in prediction is critical for a domain as dynamic and overlapping as news, aiming for a system that's both precise and reflective of the complexities it seeks to categorize.

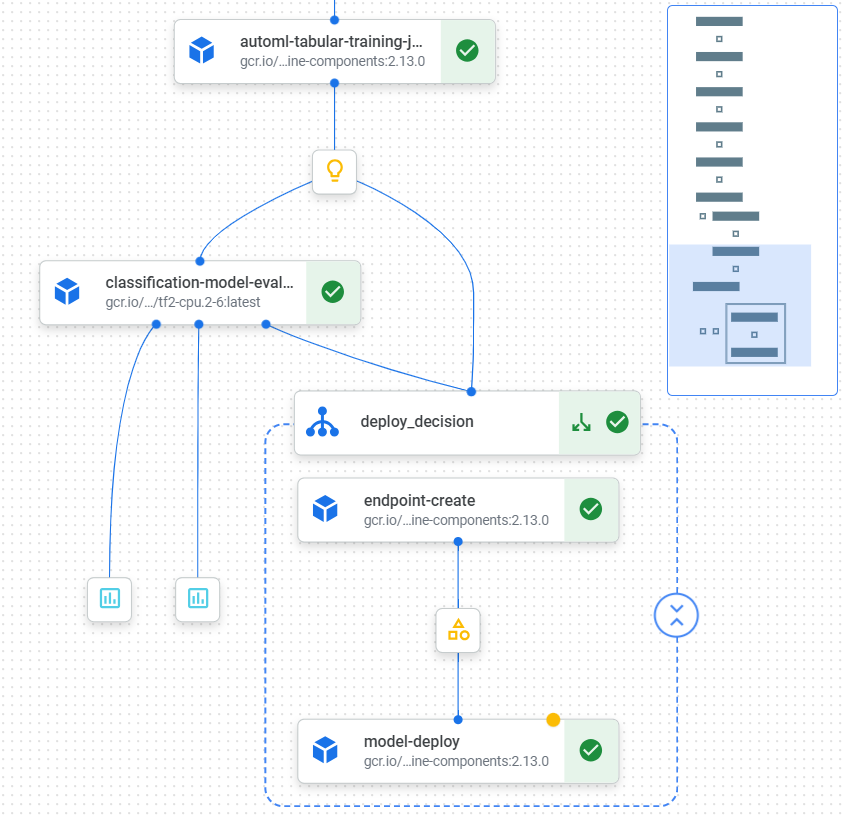


Figure 6. AutoML component with evaluation on VertexAI Pipeline.

After training the model, we evaluated the metrics for potential deployment. The model will only be deployed if the AUC ROC score is above 0.95 and has a higher value than the previous model.

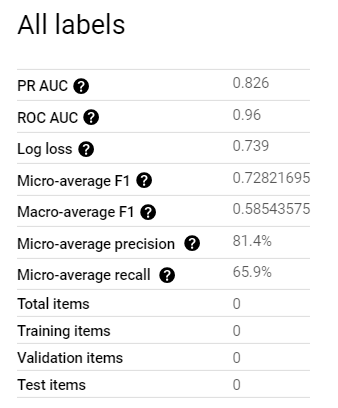
In this final section, we showcase the full pipeline.

Immagine che contiene testo, schermata, diagramma, design

Descrizione generata automaticamente

Figure 7. Full pipeline on VertexAI.

## Machine learning model Evaluation

****  
The evaluation output was directly extracted from Vertex AI. The displayed metrics are:

Confidence threshold: The threshold used for confidence in predictions is set at 0.5.

All labels PR AUC: The precision-recall area under the curve (PR AUC) for all categories combined is 0.826.

ROC AUC: The receiver operating characteristic area under the curve (ROC AUC) is 0.96, indicating strong overall performance.

Log loss: The log loss metric is calculated as 0.739.

Micro-average F1: The micro-average F1 score, which considers the overall precision and recall, is 0.72821695.

Table 2. Evaluation metrics on VertexAI.

Macro-average F1: The macro-average F1 score, averaging the F1 scores for each category, is 0.58543575.

Micro-average precision: The micro-average precision, calculated across all categories, is 81.4%.

Micro-average recall: The micro-average recall, calculated across all categories, is 65.9%.

Total items, Training items, Validation items, Test items: These indicate the number of items or samples used in the evaluation, split into training, validation, and test sets. However, these values are reported as 0, so the actual number of items used is not provided in this context.

In classification tasks, evaluating model performance typically involves metrics like accuracy, precision, recall, and F1-score. However, these measures might not be optimal for imbalanced datasets, where one class significantly outnumbers the others. In such cases, these metrics can be misleading or fail to adequately capture the performance of the minority class.

For example, accuracy may appear high if the model predominantly predicts the majority class, while precision and recall can fluctuate based on the chosen threshold or cutoff value. Although the F1-score provides a harmonic mean of precision and recall, it might not effectively represent the balance between them.

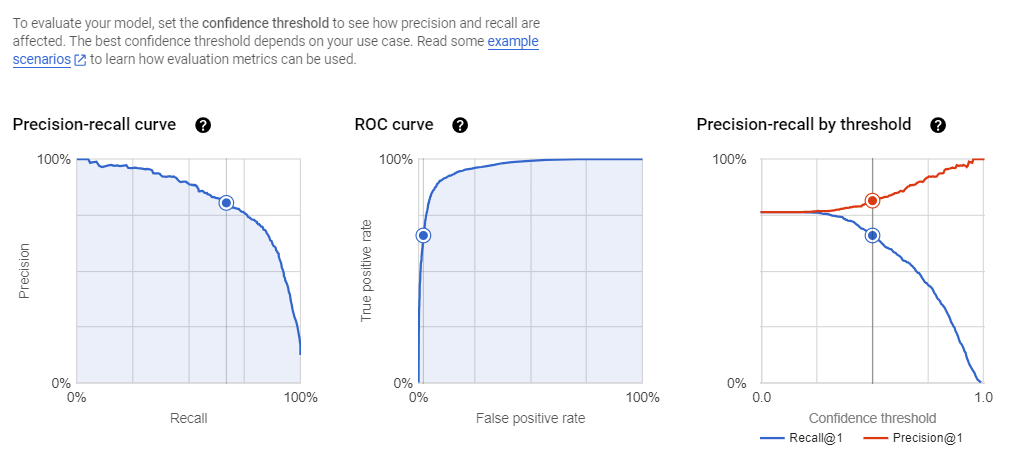
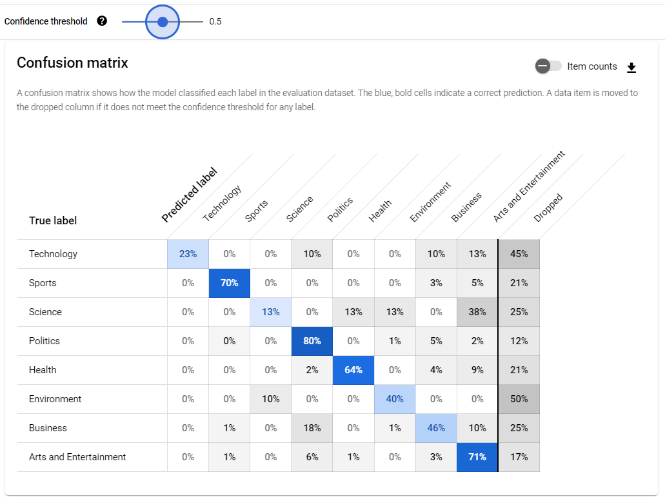
****To address these challenges, alternative evaluation metrics are often employed for imbalanced data. One widely used approach is the **Receiver Operating Characteristic** (ROC) curve and the corresponding Area Under the **Curve (AUC) metric**. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR) across various threshold values. The AUC quantifies the area under this curve, with a higher AUC indicating a better model capable of effectively distinguishing between different classes, even in imbalanced datasets.

Figure 8. Evaluation metrics on VertexAI.

The confusion matrix can vary depending on the threshold used. However, it's important to note that the model is currently in a testing phase. This is because the training dataset comprises only three thousand observations. To enhance performance, a larger sample size would be necessary. The decision to use a relatively small training set was made to mitigate processing times and the costs associated with feature creation, which would have been considerably higher with a larger dataset.

****

*a. b.*

Figure (9.a) Confusion matix with threshold 0.5. (9.b) Confusion matrix with threshold 0.25.

The classes that perform the best are Sports, Arts and Entertainment, Politics, and Business. The misclassified categories are those that are more challenging to train because the extracted articles are not explicitly related to topics such as Science, Technology, and Business, while categories like Politics and Sports are more easily classifiable. Certainly, with a much larger training sample, we can improve the classification even for these categories.

It's important to note that the training dataset is of limited size, as follows: [specific dimensions of the training dataset].

1. topic

2. Arts and Entertainment 301

3. Business 301

4. Environment 301

5. Health 301

6. Politics 301

7. Science 301

8. Sports 301

9. Technology 301

The code shows the output of the print for the training set by topic category.

## Proof of Deployment

After deploying the model, we created a testing app using FastAPI and built a Docker image containing the inference pipeline, which includes preprocessing and prediction steps. Subsequently, we deployed the image on Cloud Run.

1. # File parquet name test set

2. FILE\_PATH = 'test\_file.parquet'

3.

4. # Number of articles to analyze

5. NUM\_DOC = 2

6.

7. # Get prediction API

8.

9. # Url App

10. URL = "https://fast-api-automl-serving-bpfl6lx4ta-ey.a.run.app/predict"

11.

12. request\_client(file\_path=FILE\_PATH, num\_doc=NUM\_DOC, url=URL)

13.

Output

14. '[{"uri": "2023-12-196637772", "title": "Tutto il gusto dell\'Abruzzo d\'inverno, tra castagne e ferratelle", "author": "Marco Ciaffone", "predictions": {"Arts and Entertainment": 0.9001039862632751, "Sports": 0.0239704716950655, "Business": 0.02023078873753548, "Environment": 0.01892109028995037, "Technology": 0.01822514459490776, "Science": 0.01558459643274546, "Health": 0.001601254916749895, "Politics": 0.00136266800109297}}, {"uri": "2023-12-196631748", "title": "GF, Varrese sotto attacco: pesanti critiche da due ex Vipponi", "author": "Debora Manzoli", "predictions": {"Arts and Entertainment": 0.9462506175041199, "Technology": 0.01517451461404562, "Politics": 0.01214464288204908, "Sports": 0.01031678635627031, "Business": 0.009728465229272842, "Science": 0.004114052746444941, "Health": 0.00126795272808522, "Environment": 0.001002991339191794}}]'

15.

The code shows the callable API built in Cloud Run and its output.