Contents

[1 SQL Queries 4](#_Toc183478051)

[1.1 Query no 1 4](#_Toc183478052)

[1.1.1 Basic query 4](#_Toc183478053)

[1.1.2 Optimization using indices 4](#_Toc183478054)

[1.1.3 Further optimization 5](#_Toc183478055)

[1.1.4 Full code 5](#_Toc183478056)

[1.2 Query no 2 6](#_Toc183478057)

[1.2.1 Identify what is ‘technology’ 6](#_Toc183478058)

[1.2.2 Case 1 6](#_Toc183478059)

[1.2.3 Case 2 10](#_Toc183478060)

[1.3 Query no 3 12](#_Toc183478061)

[1.3.1 Determine what is top 5 12](#_Toc183478062)

[1.3.2 Basic query 13](#_Toc183478063)

[1.3.3 Optimization using indices 14](#_Toc183478064)

[1.3.4 Full code 15](#_Toc183478065)

[2 Data integration and Database insertion 16](#_Toc183478066)

[2.1 Merge the two company datasets 16](#_Toc183478067)

[2.1.1 Define a key for merging 16](#_Toc183478068)

[2.1.2 Duplicate removal 17](#_Toc183478069)

[2.1.3 Choose the right merge type 20](#_Toc183478070)

[2.1.4 Other columns 20](#_Toc183478071)

[2.1.5 Merging the datasets 21](#_Toc183478072)

[2.1.6 Rename columns 21](#_Toc183478073)

[2.1.7 Load the dataset in SQlite3 22](#_Toc183478074)

[3 Exploratory Data Analysis (EDA) 23](#_Toc183478075)

[3.1 Basic understanding of the data 23](#_Toc183478076)

[3.1.1 DataFrame.shape 23](#_Toc183478077)

[3.1.2 DataFrame.columns 23](#_Toc183478078)

[3.1.3 DataFrame.dtypes 23](#_Toc183478079)

[3.1.4 DataFrame.describe 24](#_Toc183478080)

[3.1.5 Missing values 24](#_Toc183478081)

[3.2 Data Preparation 25](#_Toc183478082)

[3.2.1 Drop irrelevant columns 25](#_Toc183478083)

[3.2.2 Ensure dtypes are correct 25](#_Toc183478084)

[3.2.3 Ensure there are no duplicates 26](#_Toc183478085)

[3.3 Numeric features 26](#_Toc183478086)

[3.3.1 Current/Total employee estimate 26](#_Toc183478087)

[3.3.2 Year founded 33](#_Toc183478088)

[3.4 Categorical features 35](#_Toc183478089)

[3.4.1 Category 35](#_Toc183478090)

[3.4.2 Industry 36](#_Toc183478091)

[3.4.3 Country 37](#_Toc183478092)

[3.4.4 Locality 38](#_Toc183478093)

[3.5 Text columns 39](#_Toc183478094)

[3.5.1 Text data merging 39](#_Toc183478095)

[3.5.2 Text data insights 40](#_Toc183478096)

[3.6 Questions about the data 48](#_Toc183478097)

[3.6.1 What categories are the most employed companies? 48](#_Toc183478098)

[3.6.2 What is the distribution of companies founded over the years? 49](#_Toc183478099)

[3.6.3 How have the distributions of different company categories changed over time? 50](#_Toc183478100)

[3.6.4 What’s the top word in Information Technology category? 51](#_Toc183478101)

[4 Model development 53](#_Toc183478102)

[4.1 Categorical Feature Encoding 53](#_Toc183478103)

[4.2 Numerical Feature Encoding 53](#_Toc183478104)

[4.3 Text Feature Encoding 53](#_Toc183478105)

[4.4 Classification Model 54](#_Toc183478106)

[5 Model evaluation 55](#_Toc183478107)

[5.1 Evaluation results 55](#_Toc183478108)

[5.2 Potential Improvements 56](#_Toc183478109)

[5.2.1 Grid Search 56](#_Toc183478110)

[5.2.2 Cross Validation 57](#_Toc183478111)

[5.2.3 Text Pre-processing 58](#_Toc183478112)

[5.2.4 Text Representation / Text Embeddings 58](#_Toc183478113)

[5.2.5 Feature Engineering of Categorical Data 58](#_Toc183478114)

[5.2.6 Number of features 58](#_Toc183478115)

[5.2.7 Non-Considered Parameters 59](#_Toc183478116)

[5.2.8 Evaluate different models 59](#_Toc183478117)

# SQL Queries

## Query no 1

Find the top 10 industries with the highest average number of employees, only considering companies founded after 2000 that have more than 10 employees:

### Basic query

SELECT

    industry

FROM

    CompanyDataset

WHERE

    "year founded" > 2000 AND "current employee estimate" > 10

    industry

ORDER BY

    AVG("current employee estimate") DESC

LIMIT 10;

**Average running time**: 1s  
  
A screenshot of a computer

Description automatically generated

### Optimization using indices

To optimize the query, we will have to create an index based on the columns of interest:

CREATE INDEX

    idx\_industry\_year\_employee

ON

    CompanyDataset(industry, "year founded", "current employee estimate");

**Average running time**: 0.35s

A number of numbers on a white background

Description automatically generated

### Further optimization

If we want to optimize further this specific query, we can create an index based on the columns of interested **AND** the data of interest:

CREATE INDEX

    idx\_industry\_year\_employee

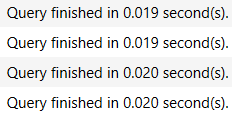
ON

    CompanyDataset(industry, "year founded", "current employee estimate")

WHERE

    "year founded" > 2000 AND "current employee estimate" > 10;

**Average running time**: 0.02s!



It should be noted that the order of indexing is very important. For example, if we did

CompanyDataset("year founded", "current employee estimate", industry)

The query would need about 0.09 seconds to run instead of 0.02 seconds. This is because the query's primary goal is to retrieve industries, making it more critical to quickly identify.

### Full code

-- Create the index

CREATE INDEX

    idx\_industry\_year\_employee

ON

    CompanyDataset(industry, "year founded", "current employee estimate")

WHERE

    "year founded" > 2000 AND "current employee estimate" > 10;

-- Create the query

SELECT

    industry -- Return the industry

FROM

    CompanyDataset -- Dataset of interest

WHERE

    -- Companies founded after 2000 that have more than 10 employees

    "year founded" > 2000 AND "current employee estimate" > 10

GROUP BY

    industry

ORDER BY

    -- Sort by average number of employyes in descending order

    AVG("current employee estimate") DESC

LIMIT 10; -- Take the top 10 industries

## Query no 2

Identify companies in the 'Technology'-like industry that do not have effective 'homepage\_text' and have fewer than 100 employees based on data merged from both datasets.

### Identify what is ‘technology’

Since the definition of a "Technology"-like industry is not explicitly provided, we must establish one ourselves.

* **Case 1:** A "Technology"-like industry could be defined as any industry that we determine to fall within the technology sector. For instance, industries such as medical devices and nanotechnology are considered technology industries, even though they do not fall under the "Information Technology" category.
* **Case 2:** Alternatively, a "Technology"-like industry could be defined as those industries that fall within the broader "Information Technology" category.

### Case 1

#### Basic query

WITH technology\_industries AS (

    SELECT 'accounting' AS industry UNION ALL

    SELECT 'animation' UNION ALL

    SELECT 'automotive' UNION ALL

    SELECT 'aviation & aerospace' UNION ALL

    SELECT 'banking' UNION ALL

    SELECT 'biotechnology' UNION ALL

    SELECT 'computer & network security' UNION ALL

    SELECT 'computer games' UNION ALL

    SELECT 'computer hardware' UNION ALL

    SELECT 'computer networking' UNION ALL

    SELECT 'computer software' UNION ALL

    SELECT 'consumer electronics' UNION ALL

    SELECT 'defense & space' UNION ALL

    SELECT 'e-learning' UNION ALL

    SELECT 'electrical/electronic manufacturing' UNION ALL

    SELECT 'financial services' UNION ALL

    SELECT 'human resources' UNION ALL

    SELECT 'industrial automation' UNION ALL

    SELECT 'information technology and services' UNION ALL

    SELECT 'internet' UNION ALL

    SELECT 'logistics and supply chain' UNION ALL

    SELECT 'machinery' UNION ALL

    SELECT 'management consulting' UNION ALL

    SELECT 'marketing and advertising' UNION ALL

    SELECT 'mechanical or industrial engineering' UNION ALL

    SELECT 'medical devices' UNION ALL

    SELECT 'nanotechnology' UNION ALL

    SELECT 'online media' UNION ALL

    SELECT 'program development' UNION ALL

    SELECT 'public safety' UNION ALL

    SELECT 'security and investigations' UNION ALL

    SELECT 'semiconductors' UNION ALL

    SELECT 'telecommunications' UNION ALL

    SELECT 'wireless' UNION ALL

    SELECT 'writing and editing'

)

SELECT

    cc.CompanyName

FROM

    CompanyClassification cc

JOIN

    CompanyDataset cd

ON

    cc.CompanyName = cd.CompanyName

WHERE

    cc.homepage\_text IS NULL

    AND cd."current employee estimate" < 100

    AND cd.industry IN (SELECT industry FROM technology\_industries);

**Average running time:** 1.4s

A screenshot of a computer

Description automatically generated

#### Optimization using indices

We cannot create an index for the joined table, but we can create an index for each table. Like Query no 1 case, we will make the indices based on the columns of interested **AND** the data of interest for optimal performance.

CREATE INDEX

    idx\_company\_employee\_industry

ON

    CompanyDataset(CompanyName, "current employee estimate", industry)

WHERE

    "current employee estimate" < 100;

CREATE INDEX

    idx\_company\_homepage

ON

    CompanyClassification(CompanyName, homepage\_text)

WHERE

    homepage\_text IS NULL;

**Average running time:** 0.001s

A screenshot of a computer

Description automatically generated

#### Full code

-- Create index for CompanyDataset

CREATE INDEX

    idx\_company\_employee\_industry

ON

    CompanyDataset(CompanyName, "current employee estimate", industry)

WHERE

    "current employee estimate" < 100;

-- Create index for CompanyClassification

CREATE INDEX

    idx\_company\_homepage

ON

    CompanyClassification(CompanyName, homepage\_text)

WHERE

    homepage\_text IS NULL;

-- Identify the technology industries

WITH technology\_industries AS (

    SELECT 'accounting' AS industry UNION ALL

    SELECT 'animation' UNION ALL

    SELECT 'automotive' UNION ALL

    SELECT 'aviation & aerospace' UNION ALL

    SELECT 'banking' UNION ALL

    SELECT 'biotechnology' UNION ALL

    SELECT 'computer & network security' UNION ALL

    SELECT 'computer games' UNION ALL

    SELECT 'computer hardware' UNION ALL

    SELECT 'computer networking' UNION ALL

    SELECT 'computer software' UNION ALL

    SELECT 'consumer electronics' UNION ALL

    SELECT 'defense & space' UNION ALL

    SELECT 'e-learning' UNION ALL

    SELECT 'electrical/electronic manufacturing' UNION ALL

    SELECT 'financial services' UNION ALL

    SELECT 'human resources' UNION ALL

    SELECT 'industrial automation' UNION ALL

    SELECT 'information technology and services' UNION ALL

    SELECT 'internet' UNION ALL

    SELECT 'logistics and supply chain' UNION ALL

    SELECT 'machinery' UNION ALL

    SELECT 'management consulting' UNION ALL

    SELECT 'marketing and advertising' UNION ALL

    SELECT 'mechanical or industrial engineering' UNION ALL

    SELECT 'medical devices' UNION ALL

    SELECT 'nanotechnology' UNION ALL

    SELECT 'online media' UNION ALL

    SELECT 'program development' UNION ALL

    SELECT 'public safety' UNION ALL

    SELECT 'security and investigations' UNION ALL

    SELECT 'semiconductors' UNION ALL

    SELECT 'telecommunications' UNION ALL

    SELECT 'wireless' UNION ALL

    SELECT 'writing and editing'

)

-- Create the query

SELECT

    cc.CompanyName -- Return the company name

FROM

    CompanyClassification cc -- Inner join of CompanyClassification

JOIN

    CompanyDataset cd -- and CompanyDataset

ON

    cc.CompanyName = cd.CompanyName -- CompanyName is the common column between the two tables

WHERE

    cc.homepage\_text IS NULL -- Filter to have an effective homepage

    AND cd."current employee estimate" < 100 -- Filter for companies that have fewer than 100 employees

    AND cd.industry IN (SELECT industry FROM technology\_industries); -- Filter for 'Technology' like industry

### Case 2

#### Basic query

SELECT

    cc.CompanyName

FROM

    CompanyClassification cc

JOIN

    CompanyDataset cd

ON

    cc.CompanyName = cd.CompanyName

WHERE

    cc.homepage\_text IS NULL

    AND cc.Category = 'Information Technology'

    AND cd."current employee estimate" < 100;

**Average running time:** 109s



#### Optimization using indices

Like the previous case we will make the indices based on the columns of interested **AND** the data of interest for optimal performance.

CREATE INDEX

    idx\_company\_employee

ON

    CompanyDataset(CompanyName, "current employee estimate")

WHERE

    "current employee estimate" < 100;

CREATE INDEX

    idx\_company\_homepage\_category

ON

    CompanyClassification(CompanyName, homepage\_text, Category)

WHERE

    homepage\_text IS NULL

    AND Category = 'Information Technology';

**Average running time:** 0.001s

A number of numbers on a white background

Description automatically generated

#### Full code

-- Create index for CompoanyDataset

CREATE INDEX

    idx\_company\_employee

ON

    CompanyDataset(CompanyName, "current employee estimate")

WHERE

    "current employee estimate" < 100;

-- Create index for CompanyClassification

CREATE INDEX

    idx\_company\_homepage\_category

ON

    CompanyClassification(CompanyName, homepage\_text, Category)

WHERE

    homepage\_text IS NULL

    AND Category = 'Information Technology';

-- Create the query

SELECT

    cc.CompanyName -- Return the company name

FROM

    CompanyClassification cc -- Inner join of CompanyClassification

JOIN

    CompanyDataset cd -- and CompanyDataset

ON

    cc.CompanyName = cd.CompanyName -- CompanyName is the common column between the two tables

WHERE

    cc.homepage\_text IS NULL -- Filter to have an effective homepage

    AND cc.Category = 'Information Technology' -- Filter for 'Technology' like industry

    AND cd."current employee estimate" < 100; -- Filter for companies that have fewer than 100 employees

## Query no 3

Rank companies within each country by their total employee estimate in descending order, showing only companies that rank in the top 5 within their country.

### Determine what is top 5

We rank the companies based on their total employee estimate. Since larger companies typically have higher employee counts, it is unlikely that we will encounter identical values. However, in the event that this occurs, we have three ranking options: ROW\_NUMBER(), DENSE\_RANK(), and RANK(). The disadvantages of each method will be discussed using hypothetical tables to illustrate the differences.

* Disadvantage of **ROW\_NUMBER**:

|  |  |  |
| --- | --- | --- |
| **Company** | **Total employees** | **ROW\_NUMBER** |
| Company\_1 | 500 | 1 |
| Company\_2 | 400 | 2 |
| Company\_3 | 300 | 3 |
| Company\_4 | 200 | 4 |
| Company\_5 | 100 | 5 |
| Company\_6 | 100 | 6 |

If we choose ROW\_NUMBER(), we will always have a fixed number of 5 companies. However, if the 5th and 6th companies have the same number of employees, the 6th company will be excluded from the results. Additionally, if there are a significant number of companies with the same employee count (e.g., 100 employees), many of these companies may not be included in the rankings, as ROW\_NUMBER() assigns a unique rank to each row, even when values are identical.

* Disadvantage of **DENSE\_RANK**:

|  |  |  |
| --- | --- | --- |
| **Company** | **Total employees** | **DENSE\_RANK** |
| Company\_1 | 500 | 1 |
| Company\_2 | 400 | 2 |
| Company\_3 | 400 | 2 |
| Company\_4 | 300 | 3 |
| Company\_5 | 200 | 4 |
| Company\_6 | 100 | 5 |
| Company\_7 | 100 | 5 |
| Company\_8 | 100 | 5 |
| Company\_9 | 100 | 5 |

If we choose DENSE\_RANK(), the ranking will be fair to all companies, as it does not skip ranks. However, if many companies share the same number of employees, we could end up with more than 5 companies in the results. For instance, it is possible to end up with 9 or more companies ranked equally. In a more extreme case, we could potentially retrieve 50 companies, even though we are only interested in the top 5. While this scenario is unlikely, it is still a possibility that needs to be considered.

* Disadvantage of **RANK**:

Let’s consider the same case as above. The ranks would be as shown:

|  |  |  |
| --- | --- | --- |
| **Company** | **Total employees** | **RANK** |
| Company\_1 | 500 | 1 |
| Company\_2 | 400 | 2 |
| Company\_3 | 400 | 2 |
| Company\_4 | 300 | 4 |
| Company\_5 | 200 | 5 |
| Company\_6 | 200 | 6 |
| Company\_7 | 100 | 6 |
| Company\_8 | 100 | 6 |
| Company\_9 | 100 | 6 |

The RANK() method shares a similar disadvantage to DENSE\_RANK(), in that it may result in more than 5 companies being included in the ranking. However, the number of additional companies is generally smaller, unless there is a significant number of companies sharing the same "Total employees" value. This scenario is less common than with DENSE\_RANK(), making it a less frequent issue.

In our case, it is rare for companies to have the same number of "Total employees." However, to ensure fairness and avoid excluding a 6th company, I consider RANK() to be the most appropriate option. This method ensures that no company is unfairly left out of the ranking

### Basic query

WITH RankedCompaniesByCountry AS (

    SELECT

        CompanyName,

        country,

        RANK() OVER (PARTITION BY country ORDER BY "total employee estimate" DESC) AS RankInCountry

    FROM

        CompanyDataset

    WHERE

        country IS NOT NULL

)

SELECT

    CompanyName,

    country,

    RankInCountry

FROM

    RankedCompaniesByCountry

WHERE

    RankInCountry <= 5;

**Average running time**: 3.3s  
A screenshot of a number

Description automatically generated

### Optimization using indices

Like the previous cases we will make the index based on the columns of interested **AND** the data of interest for optimal performance.

CREATE INDEX

    idx\_company\_country\_total

ON

    CompanyDataset(country, "total employee estimate", CompanyName)

WHERE

    Country IS NOT NULL;

**Average running time**: 0.001s

A number of numbers on a white background

Description automatically generated

### Full code

--Create the index. We choose country as first parameter since we partition by country.

-- We choose "total employee estimate" as the second parameter since we compute the ranking based on this.

-- We choose CompanyName as third parameter since it's the least important parameter

CREATE INDEX

    idx\_company\_country\_total

ON

    CompanyDataset(country, "total employee estimate", CompanyName)

WHERE

    Country IS NOT NULL;

-- Create a Common Table Expression (CTE) to rank companies between each country

WITH RankedCompaniesByCountry AS (

    SELECT

        CompanyName, -- We need the CompanyName

        country, -- We also need the country in the result

        -- Rank companies within each country based on the total employee estimate in descending order

        RANK() OVER (PARTITION BY country ORDER BY "total employee estimate" DESC) AS RankInCountry

    FROM

        CompanyDataset

    WHERE

        -- Only include records where the country field is not null

        country IS NOT NULL

)

-- Create the query

SELECT

    CompanyName, -- Return the company name

    country,  -- Return the country

    RankInCountry -- Return the rank within the country

FROM

    RankedCompaniesByCountry

WHERE

    RankInCountry <= 5;  -- Filter to only include the top 5 companies per country

# Data integration and Database insertion

## Merge the two company datasets

### Define a key for merging

The two datasets have 2 common columns. CompanyName and Website. Intuitively CompanyName seems better option since there can be a company without a website, but it can’t be a website without a CompanyName.

Despite that we can prove that choosing CompanyName is a better option. Using CompanyClassification dataset (we care about Companies that are also in this dataset, we will explain later the reason) we can run the following queries:

SELECT CompanyName, COUNT(DISTINCT Website)

FROM CompanyClassification

GROUP BY CompanyName

HAVING COUNT(DISTINCT Website) > 1;

SELECT Website, COUNT(DISTINCT CompanyName)

FROM CompanyClassification

GROUP BY Website

HAVING COUNT(DISTINCT CompanyName) > 1;

Thus, we can see that there’s not a company with more than one websites but there’s some websites associated with more than one companies:

A screenshot of a computer

Description automatically generated  
  
Having said that we should choose CompanyName as the key for merging.

### Duplicate removal

Before merging the data, we’ll have to handle the possible duplicates.

#### Duplicates in CompanyClassification

Let’s check the duplicates on our merging key:

SELECT

    \*

FROM

    CompanyClassification

WHERE

    CompanyName IN (

        SELECT

            CompanyName

        FROM

            CompanyClassification

        GROUP BY

            CompanyName

        HAVING

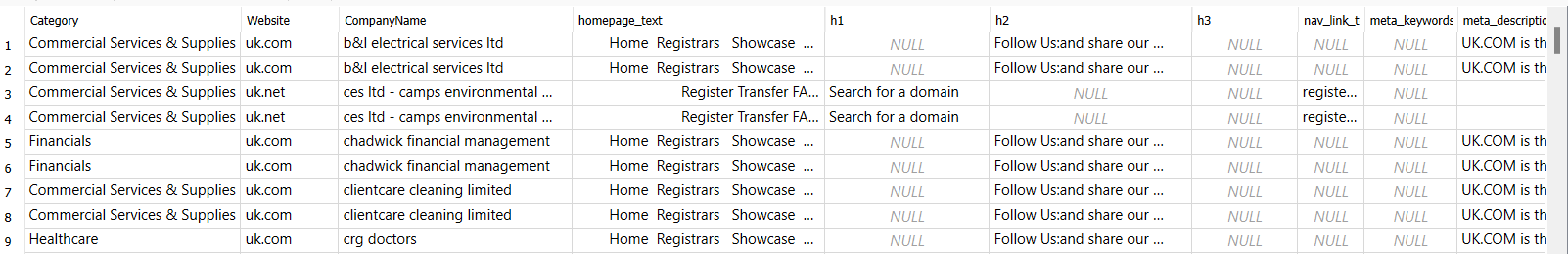
            COUNT(\*) > 1

    )

ORDER BY

    CompanyName;

We have several duplicates, where all columns are identical:



Since the rows are identical, we can delete the duplicates and retain only the first occurrence (first row):

DELETE FROM

    CompanyClassification

WHERE

    ROWID NOT IN (

        SELECT

            MIN(ROWID)

        FROM

            CompanyClassification

        GROUP BY

            CompanyName

)

For consistency, we will also include this in our Python code. Here, cc represents the CompanyClassification dataset as a Pandas DataFrame, as will be demonstrated later in the code:

cc = cc.drop\_duplicates(subset='CompanyName', keep='first')

#### Duplicates in CompanyDataset

To handle duplicates in the CompanyDataset, we will use a Pandas DataFrame, as the process is more complex.

##### Find the duplicates

We are only interested in the CompanyNames that are also present in the CompanyClassification dataset. The duplicates we are looking for appear as follows:

A screenshot of a computer

Description automatically generated

##### Identify how to choose

As shown in the results, the key difference in the duplicates is the website. Since the CompanyClassification dataset also includes a Website column, we have chosen to retain the record that matches the website in the CompanyClassification dataset. This approach handles most of the duplicates. However, there are still some duplicates remaining:

A screenshot of a computer

Description automatically generated

Upon inspecting the websites in the CompanyClassification dataset, we noticed entries like "blogstop.in," "uk.com," etc. Therefore, instead of selecting the duplicate where CompanyDataset.Website equals CompanyClassification.Website, we decided to retain the record where the CompanyDataset.Website contains the CompanyClassification.Website. This approach allows us to handle all the duplicates.

##### Code

import sqlite3

import pandas as pd

# Path to your SQLite database file

db\_path = "db/combined\_data.db"

# Connect to the SQLite database

connection = sqlite3.connect(db\_path)

# Load the datasets into pandas DataFrames

cd = pd.read\_sql\_query("SELECT \* FROM CompanyDataset", connection)

cc = pd.read\_sql\_query("SELECT \* FROM CompanyClassification", connection)

# Close the connection

connection.close()

# Step 1: Filter cd to only include rows with CompanyName in cc

cd = cd[cd['CompanyName'].isin(cc['CompanyName'])]

# Step 2: Identify duplicates in cd

duplicates\_cd = cd[cd.duplicated(subset=['CompanyName'], keep=False)] \

                    .sort\_values(by='CompanyName')

# Step 3: Process duplicates

result = []

other = []

for company, group in duplicates\_cd.groupby('CompanyName'):

    # Get the CompanyClassification website for this CompanyName

    cc\_website = cc.loc[cc['CompanyName'] == company, 'Website'].iloc[0] \

                    if company in cc['CompanyName'].values else None

    if cc\_website:

        # Check if cc\_website matches or is a substring in any of group['Website']

        matches = group['Website'].str.contains(cc\_website, na=False)

        if matches.sum() == 1:

            # Only one duplicate match

            result.append(group[matches])

        else:

            # All duplicates match or none of them match

            other.append(group)

    else:

        # No corresponding Website in cc, keep all duplicates

        other.append(group)

# Create a dataframe containing only the values that we want from the duplicates

filtered\_duplicates = pd.concat(result).sort\_index()

# Step 4: Final deduplicated DataFrame

cd = pd.concat([cd[~cd.index.isin(duplicates\_cd.index)],

                 filtered\_duplicates]).sort\_index()

### Choose the right merge type

The CompanyDataset contains approximately 7,000,000 rows, while the CompanyClassification dataset has only 70,000 rows. Since our goal is to predict the "Category" column from CompanyClassification, we need data from this dataset to train and test our model. Later, if we wish to predict the category for the remaining 99% of the data, we can easily scrape the CompanyClassification data based on the website, as the extra columns in CompanyClassification are associated with the company website.

Furthermore, every company in CompanyClassification is also present in CompanyDataset, ensuring that we will have 70,000 rows of complete data. To merge the datasets, we will use an inner join.

### Other columns

The only other common column for the two datasets is the Website column. It’s right to assume that the correct website is on CompanyClassification dataset since it’s the dataset with all the website info (homepage text, description etc). But as seen above there are some cases that the full website is in CompanyDataset e.g., brightman.uk.com while in CompanyClassification is uk.com

Therefore, if the CompanyDataset website includes the CompanyClassification website, we will retain the website from CompanyDataset. Otherwise, we will keep the website from CompanyClassification.

### Merging the datasets

Based on the points discussed above, we will perform an inner join between the two datasets on CompanyName. The Website column will be selected as described in section 2.1.4

# Step 1: Inner join on CompanyName

merged = pd.merge(cd, cc, on='CompanyName', how='inner', suffixes=('\_cd', '\_cc'))

# Step 2: Resolve Website column

def resolve\_website(row):

    # if CompanyDataset.Website exists and includes CompanyClassification.Website

    if pd.notna(row['Website\_cd']) and row['Website\_cc'] in row['Website\_cd']:

        return row['Website\_cd']

    return row['Website\_cc']

merged['Resolved\_Website'] = merged.apply(resolve\_website, axis=1)

# Drop the original Website columns if not needed

merged = merged.drop(columns=['Website\_cd', 'Website\_cc'])

### Rename columns

We aim to maintain a consistent format across all columns. It is not ideal to have blank spaces (e.g., "current employee estimate") or use multiple formats. Therefore, for column names, we will adopt snake\_case, where each word is separated by an underscore (\_) character.

rename\_dictionary = {

    'CompanyName': 'company\_name',

‘Category’: 'category’,

    'Resolved\_Website': 'website',

    'year founded': 'year\_founded',

    'size range': 'size\_range',

    'linkedin url': 'linkedin\_url',

    'current employee estimate': 'current\_employee\_estimate',

    'total employee estimate': 'total\_employee\_estimate',

    }

# Rename columns using the rename dictionary

merged = merged.rename(columns=rename\_dictionary)

### Load the dataset in SQlite3

# Create a SQLite3 connection

conn = sqlite3.connect(db\_path)  # Creates a file-based database

# Save DataFrame to a table named 'company\_table'

merged.to\_sql('CompanyData', conn, if\_exists='replace', index=False)

# Close the connection

conn.close()

# Exploratory Data Analysis (EDA)

## Basic understanding of the data

### DataFrame.shape

Our data consists of 74856 rows and 19 columns.

### DataFrame.columns

The columns of our dataset are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| company\_name | category | website | Unnamed: 0 | year\_founded |
| industry | size\_range | locality | country | linkedin\_url |
| current\_employee\_estimate | total\_employee\_estimate | homepage\_text | h1 | h2 |
| h3 | nav\_link\_text | meta\_keywords | meta\_description |  |

### DataFrame.dtypes

|  |  |
| --- | --- |
| **Column Name** | **Data Type** |
| company\_name | object |
| category | object |
| website | object |
| Unnamed: 0 | int64 |
| year\_founded | float64 |
| industry | object |
| size\_range | object |
| locality | object |
| country | object |
| linkedin\_url | object |
| current\_employee\_estimate | int64 |
| total\_employee\_estimate | int64 |
| homepage\_text | object |
| h1 | object |
| h2 | object |
| h3 | object |
| nav\_link\_text | object |
| meta\_keywords | object |
| meta\_description | object |

### DataFrame.describe

A screenshot of a data

Description automatically generated

This provides some basic insights into our numerical data. We can observe that most companies (75%) have a relatively small number of employees, as reflected by the mean of just 27.4. Additionally, we notice a significant number of NaN values in the year\_founded column, with a count of only 48,164. It is also evident that most companies were founded after 1990.

### Missing values

Running the command

df.isna().sum()

We can identify all the missing values for each column. It’s good to have a good idea of this before we start our analysis. We will see how we’ll handle each case later.

|  |  |
| --- | --- |
| **Feature** | **Number of missing values** |
| company\_name | 0 |
| category | 0 |
| year\_founded | 25878 |
| industry | 0 |
| size\_range | 0 |
| locality | 1730 |
| country | 0 |
| current\_employee\_estimate | 0 |
| total\_employee\_estimate | 0 |
| homepage\_text | 669 |
| h1 | 27177 |
| h2 | 20664 |
| h3 | 29156 |
| nav\_link\_text | 25787 |
| meta\_keywords | 50005 |
| meta\_description | 7046 |

## Data Preparation

### Drop irrelevant columns

* “Unnamed: 0” seems like an id column. We won't need for our data exploration or category prediction.
* “website” typically only contains the company name, so it will not be necessary for our analysis.
* “linkedin\_url” serves a similar purpose to the "website" column and will also be excluded.

columns\_to\_drop = ['Unnamed: 0', 'website', 'linkedin\_url']

df.drop(columns=columns\_to\_drop, inplace=True)

### Ensure dtypes are correct

From our initial data exploration, all data types appeared correct, except for the year\_founded, which was incorrectly set as a float. We have converted it to an integer type.

df['year\_founded'] = pd.to\_numeric(df['year\_founded'],  
 errors='coerce').astype('Int64')

### Ensure there are no duplicates

We have already addressed the duplicates prior to merging the datasets. However, to be thorough, we check for duplicates again.

duplicate\_companies = df[df.duplicated(subset='company\_name', keep=False)]

As expected, no duplicates are found.

## Numeric features

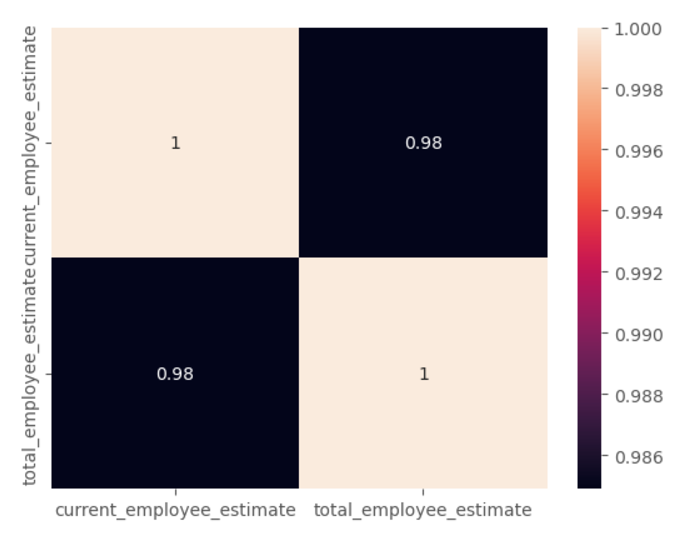
The numeric features of our dataset are the year\_founded, the current\_employee\_estimate and the total\_employee\_estimate.

### Current/Total employee estimate

These features represent the estimated number of employees. Typically, to handle this type of data, we categorize it into bins. The size\_range column already serves this purpose. As a result, we may not need these two features. However, there is still the possibility that analyzing them could provide valuable insights and a deeper understanding of the dataset. Additionally, we may identify that a different binning approach would be more appropriate.

#### Correlation

First let’s investigate the corelation between these two features:



The two features are very highly correlated, so we will proceed with our analysis using only one of them.

#### Value Frequency

If we count the occurrences of each value in total\_employee\_estimate, the most frequent values are shown here:

A graph of a number of employees

Description automatically generated

We can observe that most companies have a very small number of employees. To visualize this, let's plot a histogram:  
A graph with a red bar

Description automatically generated  
  
The histogram doesn't provide much insight, suggesting the presence of outliers.

#### Peak Values

A graph with numbers and a red line

Description automatically generated  
By printing the peak values, we can see that there are some outliers, and it's clear that most of our data have a small total\_employee\_estimate.

#### Company Distribution by Employee Estimate

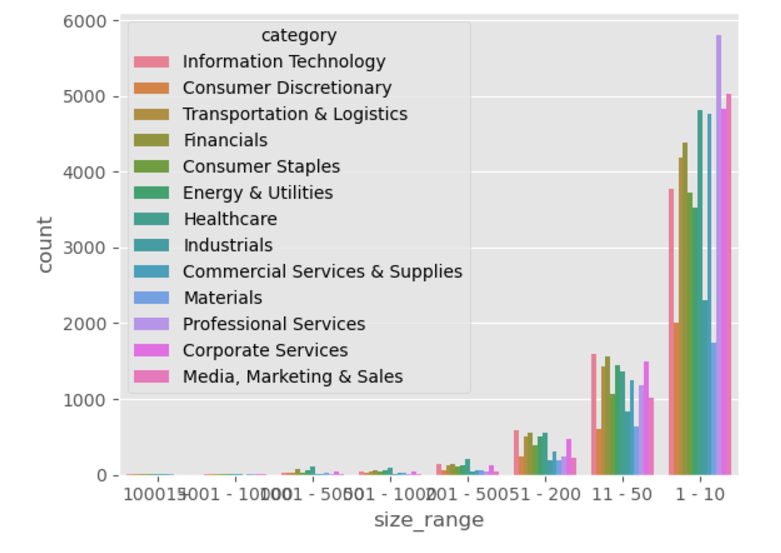
Below is the distribution of companies based on the employee estimate, showing the number of companies that have at least the specified number of employees (threshold).

|  |  |  |
| --- | --- | --- |
| **Threshold** | **Count** | **Percentage** |
| 25000 | 14 | 0.02 |
| 10000 | 43 | 0.06 |
| 5000 | 117 | 0.16 |
| 1000 | 575 | 0.78 |
| 500 | 1098 | 1.49 |
| 300 | 1710 | 2.32 |
| 200 | 2518 | 3.42 |
| 100 | 4493 | 6.11 |
| 50 | 7675 | 10.43 |
| 40 | 9092 | 12.35 |
| 30 | 11212 | 15.24 |
| 20 | 14825 | 20.15 |
| 10 | 22771 | 30.94 |
| 9 | 24246 | 32.95 |
| 8 | 25794 | 35.05 |
| 7 | 27592 | 37.49 |
| 6 | 29712 | 40.37 |
| 5 | 32419 | 44.05 |
| 4 | 35663 | 48.46 |
| 3 | 40053 | 54.43 |
| 2 | 46338 | 62.97 |
| 1 | 56180 | 76.34 |

Indeed, only 10% of the companies have more than 50 employees, while 70% have fewer than 10 employees. This suggests that we may need to adjust the size\_range, which currently only has a single bin for the 1-10 employee range.

#### Size\_range distribution by Category

Let's examine the distribution of size\_range across different categories:



And by excluding the top 3 categories, we can better observe the distribution of the other categories:

A graph with different colored bars

Description automatically generated

Our goal is to create a better separation based on the table above. Low values play a significant role. For example, 24% of the companies have only one employee (total\_employee\_estimate = 1), which means they should be treated differently from companies with 10 employees. Similarly, companies with 3-4 employees should be differentiated from those with 10 employees, as indicated by the large number of companies with only that many employees.

However, what about the outliers mentioned earlier? Are 0.02% or 0.06% too small to consider? Let's examine the outliers based on the categories (target column).

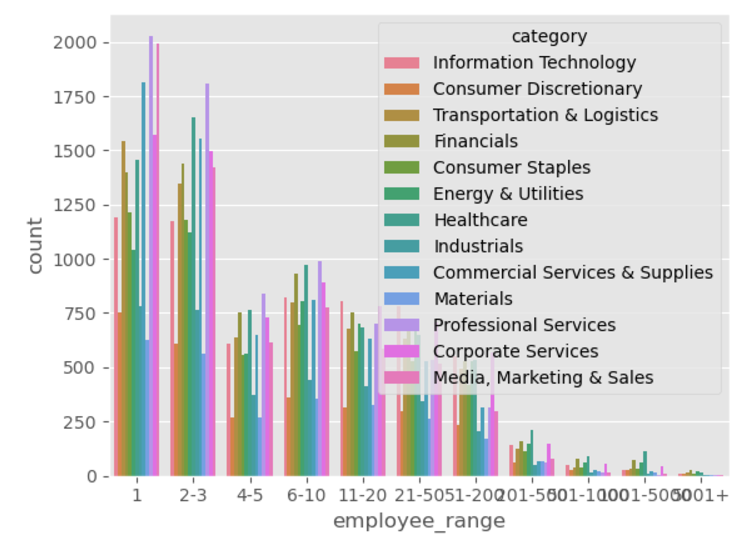
#### Outliers

A graph of a number of employees

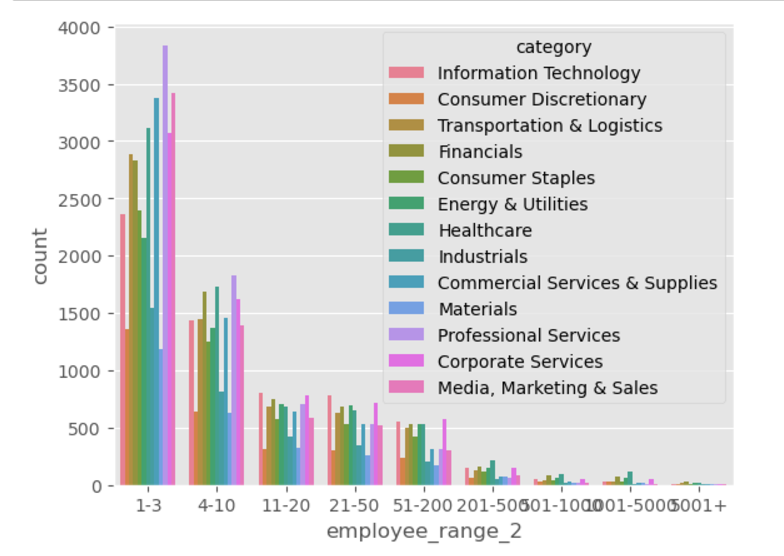
Description automatically generated with medium confidence

We can observe that most of the outliers are concentrated in specific categories. Although their numbers are low, they are likely to provide valuable insights for our prediction model. However, since the number of these companies is minimal, we will merge the 5001-10000 employee range with the 10001+ category.

#### Best Bin Separation

Based on the previous analysis, let's create a new separation and examine the resulting graph:

The category distribution appears very similar between the 1 and 2-3 employee cases, as well as between the 4-5 and 6-10 employee cases. Therefore, it has been decided to merge these categories.



Here, we can observe clear differences between the 1-3 and 4-10 employee ranges. For example, the 1-3 range has more companies in Media, Marketing & Sales compared to Corporate Services, and more in Consumer Staples than in Energy & Utilities. Based on these observations, we create a new feature called employee\_range based on these bins.

### Year founded

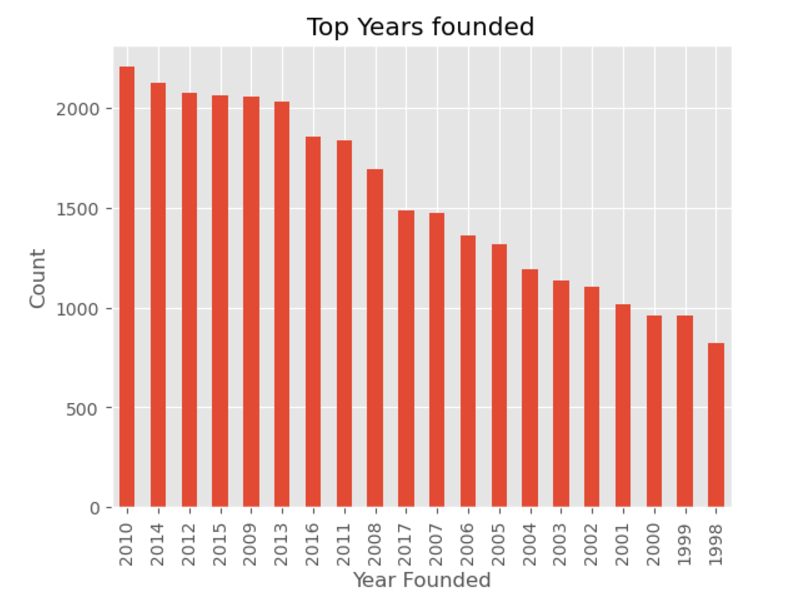
#### Value Frequency

It is clear that as time progresses, more companies are being founded, with almost all companies established after 1850:

A graph with numbers and a red line

Description automatically generated

However, this doesn't necessarily mean that more companies were founded in 2015 compared to 2010, for example:



#### Missing Values

To handle the missing values, let’s examine their distribution across categories:

A graph of a number of values

Description automatically generated with medium confidence

Since the distribution is not uniform, we choose a distribution-based imputation method, where missing values in the year\_founded column are filled based on the distribution of values in the category feature. Let's examine the year\_founded distribution after applying the changes:

A graph with a red line

Description automatically generated

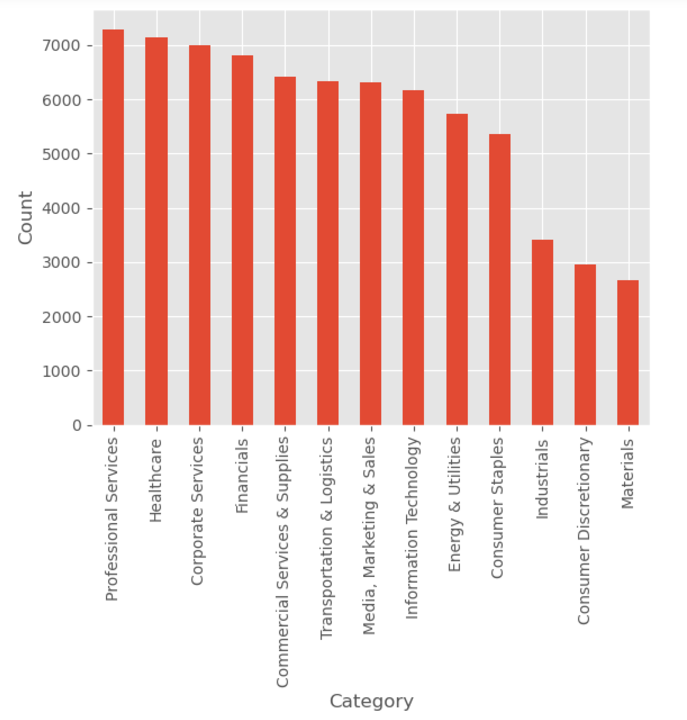
The distribution remains the same, as expected.

## Categorical features

Our categorical features are category, industry, country, and locality. Size\_range can also be considered as a categorical feature but we handled with that in the previous section.

### Category

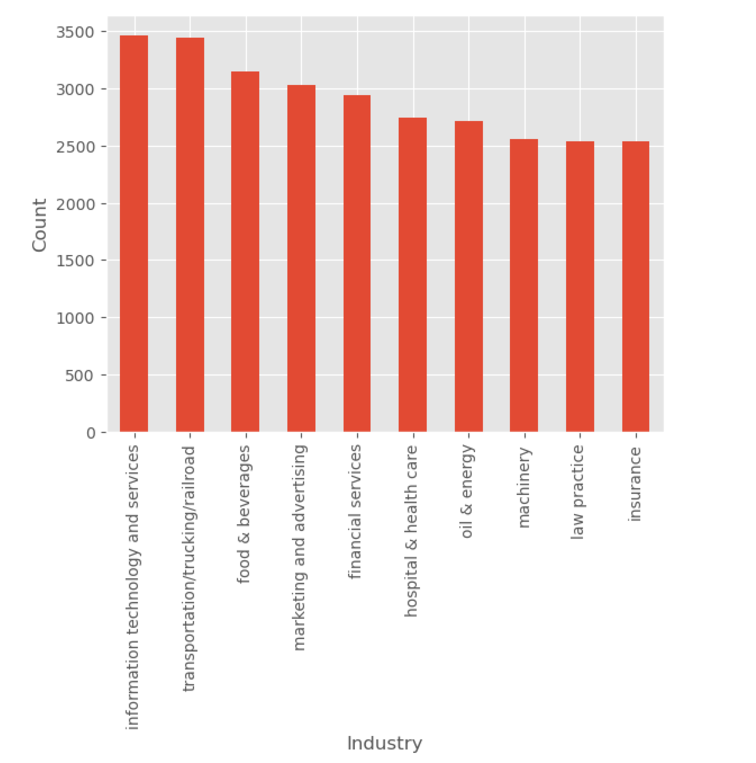
Let’s have a basic understanding of the distribution of our target feature.



Here, we can observe the most common categories as well as the least frequent ones.

### Industry

The next step is to examine the top industries, as our intuition suggests that category and industry should be correlated.



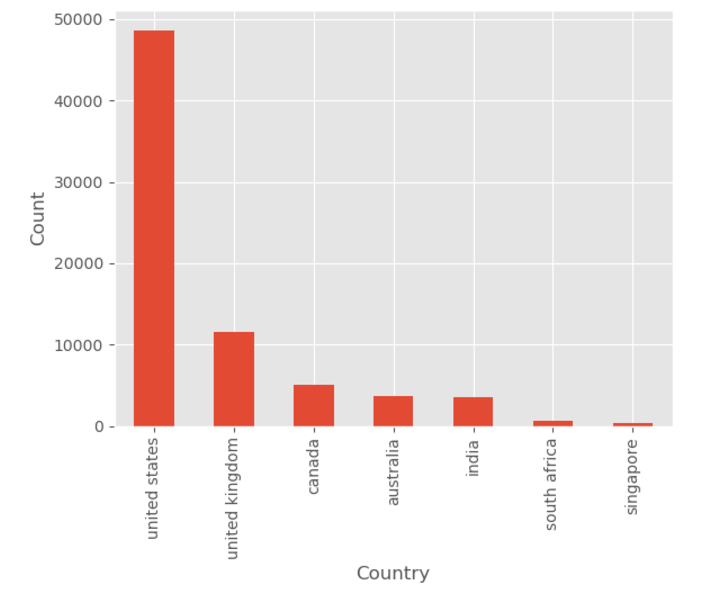
It is evident that there is a strong correlation between the two columns. Each category is associated with specific industries, which indicates that the industry column will be a crucial feature for our prediction model.

A graph with blue dots and white text

Description automatically generated

### Country

By examining the frequency of companies from each country, we can see that the majority are from the United States or the United Kingdom:



There also appears to be a strong correlation between each country and its respective category:

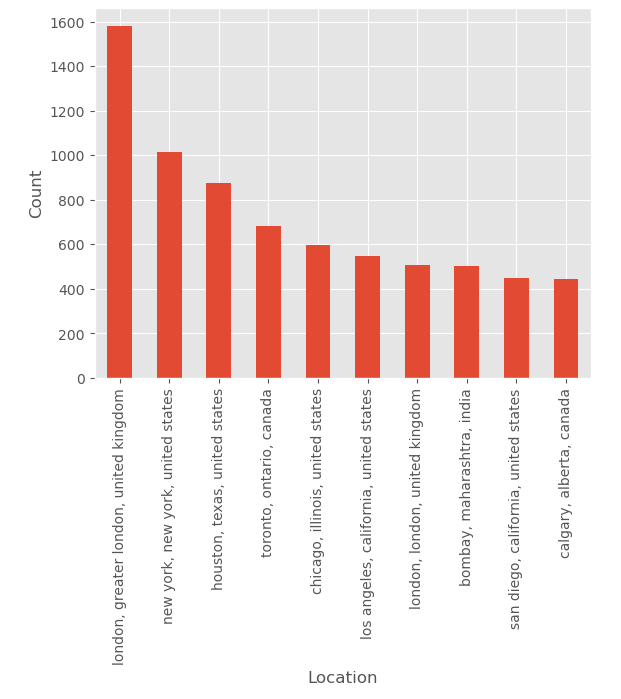
A group of colorful bars

Description automatically generated with medium confidence

### Locality

In the locality column, as mentioned in section 3.2.4, there are 1,730 missing values. Given that this number is relatively low compared to the entire dataset, we have three options. The first option is to fill the missing values with another category or by predicting them. The second option is to exclude locality from our prediction, as we already have the country column, which appears to have a strong correlation with our target column. The third option is to drop the 1,730 rows with missing values, given that the number is small .

Let’s first gain some insights about locality.



Although the United States has significantly more companies than the United Kingdom, London stands out as the location with the most companies.

There are a total of 11,024 unique locations. Given the large amount of data, it would be challenging to gain meaningful insights about the categories if we exclude the top locations. Grouping locations together could be an option, but this is already handled by the country column. Therefore, we have decided to drop the locality column for our prediction model.

## Text columns

### Text data merging

The h1, h2, and h3 columns each have about one-third of the data missing. Based on our intuition, these columns should be combined, as they all represent headings. After combining them, we are left with 7,434 rows missing, which justifies merging them into a new column named headings.

The columns homepage\_text, headings, nav\_link\_text, and meta\_keywords consist of keywords and have the following missing data counts: 669, 7,434, 25,787, and 50,005, respectively. Let’s also examine their average word counts:

|  |  |
| --- | --- |
| **Column** | **Average word count** |
| headings | 25.16 |
| homepage\_text | 411.55 |
| nav\_link\_text | 13.86 |
| meta\_keywords | 26.10 |

Columns like headings, nav\_link\_text, and meta\_keywords seem to contain basic page data. Given their relatively low average word counts and high number of missing values, we have decided to combine them into a new column named homepage\_keywords.

After this combination, we have three text columns. Of these, 350 rows have all columns as NaN, and 1,645 rows have at least two NaN values. Since text data plays a crucial role in our prediction model, we have decided to drop these 1,645 rows.

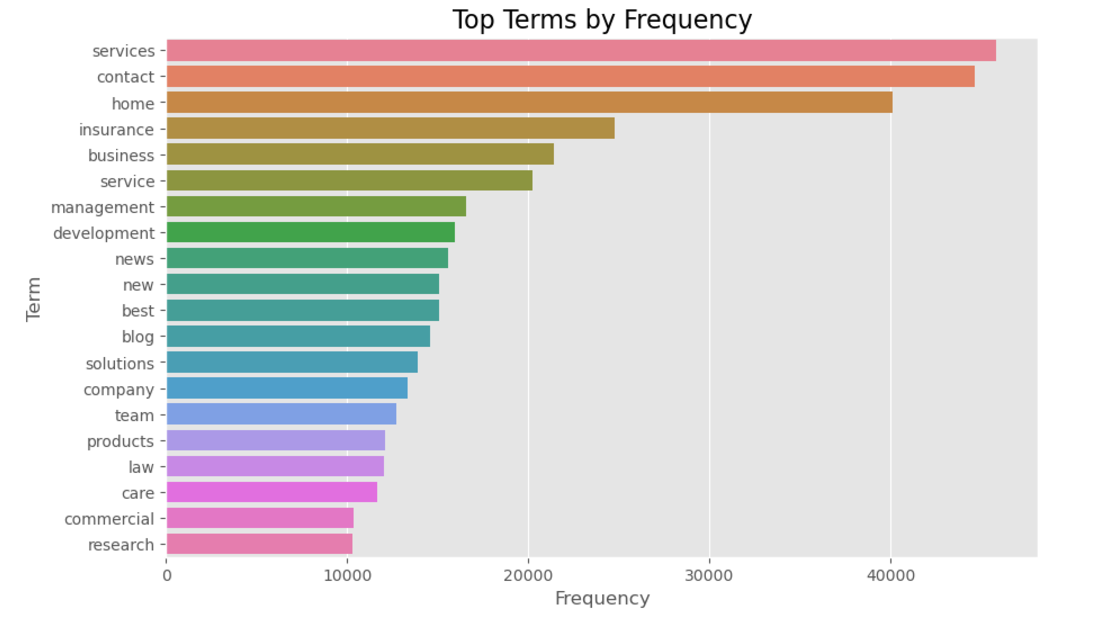
### Text data insights

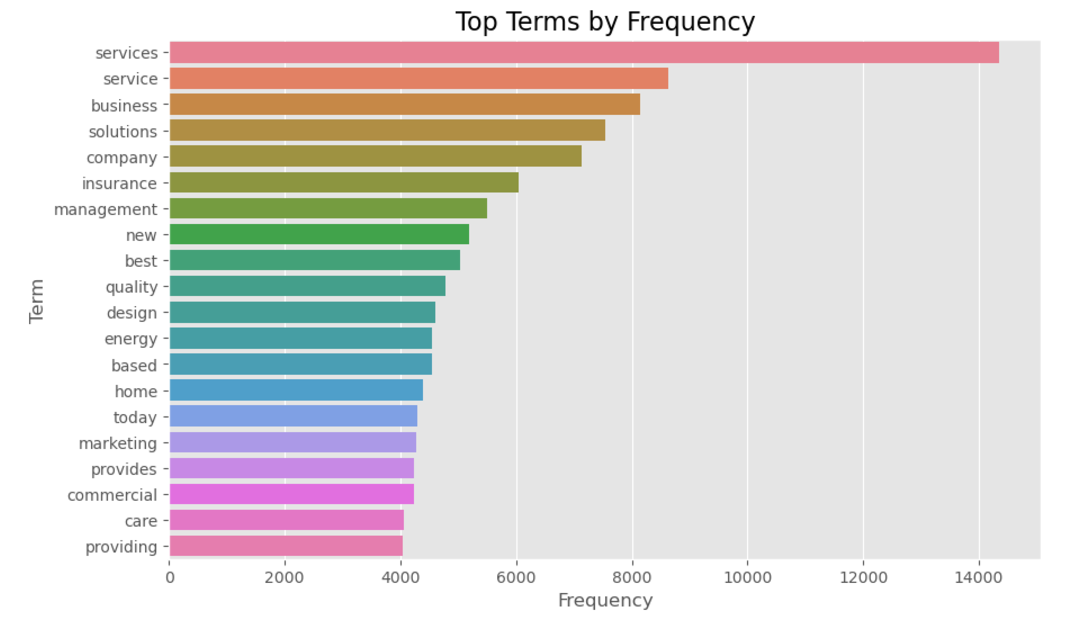
#### Top terms frequency

Below are the top terms by frequency for homepage\_text, homepage\_keywords, and meta\_description, respectively.

A graph of different colored bars

Description automatically generated





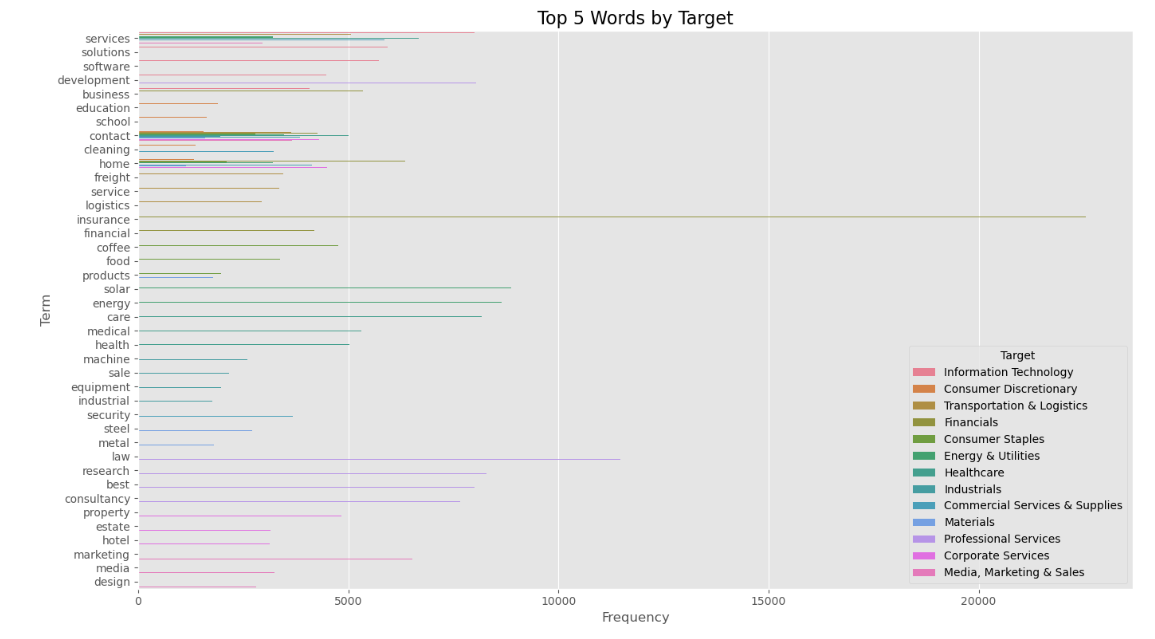
We can observe that while some top terms differ across the three features, there are also terms that appear frequently in all of them. Are these terms stop words, or do they occur often in certain categories and potentially contribute to the classification? To answer this, we will examine the top terms for each category.

#### Top terms by category

Below are the top terms by frequency for each category in homepage\_text, homepage\_keywords, and meta\_description, respectively.

A screen shot of a graph

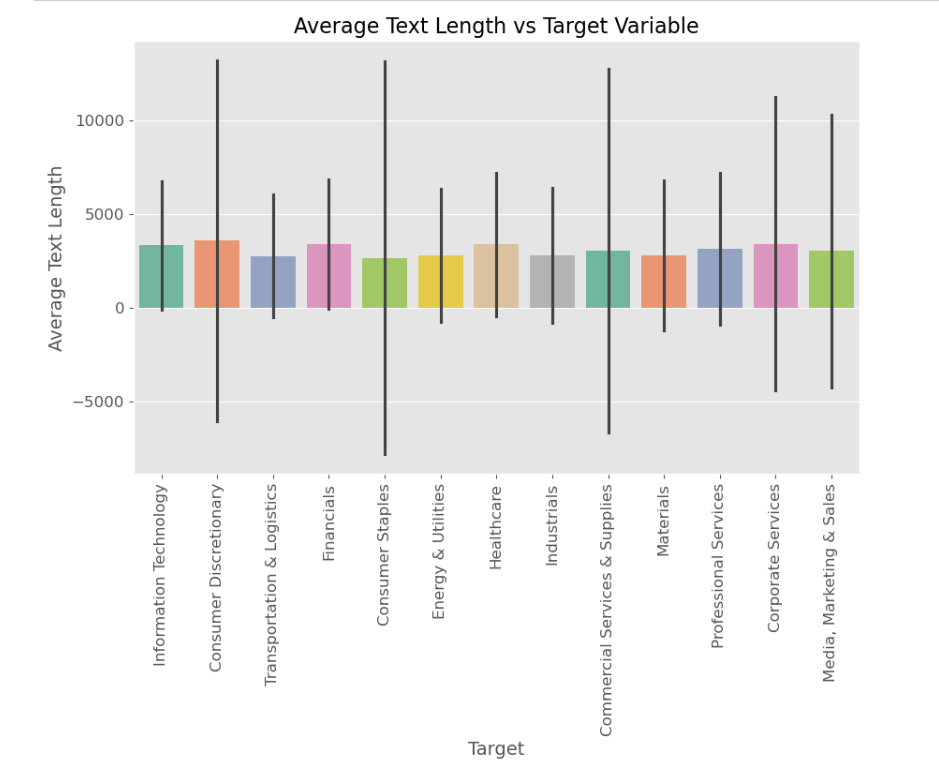
Description automatically generated





We can observe a clear correlation between the top terms used and the category. However, as mentioned in the previous section, words like "services," "home," and "contact" appear across all categories. Therefore, we will treat these as stop words and remove them from the data.

#### Text length by category



A graph with different colored lines

Description automatically generated

A graph with text length and target variable

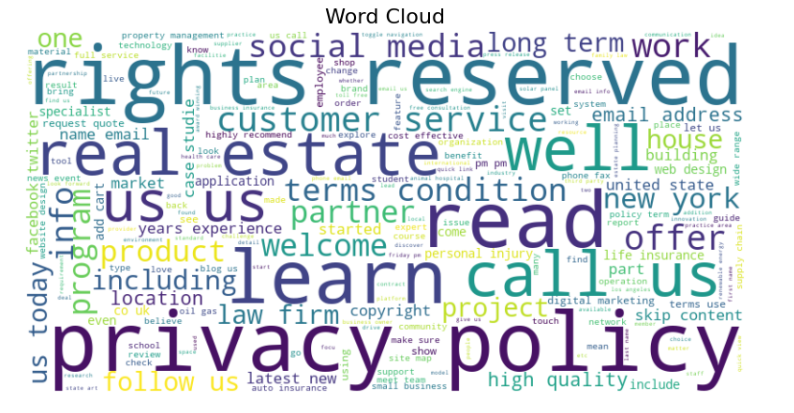
Description automatically generated

The text length isn’t correlated with the category, but as shown above, the term frequency is. Therefore, using just a bag of words for embeddings may not be ideal. Instead, TF-IDF would be a much better choice.

#### Word cloud

After removing the additional stop words, here are the resulting word clouds.

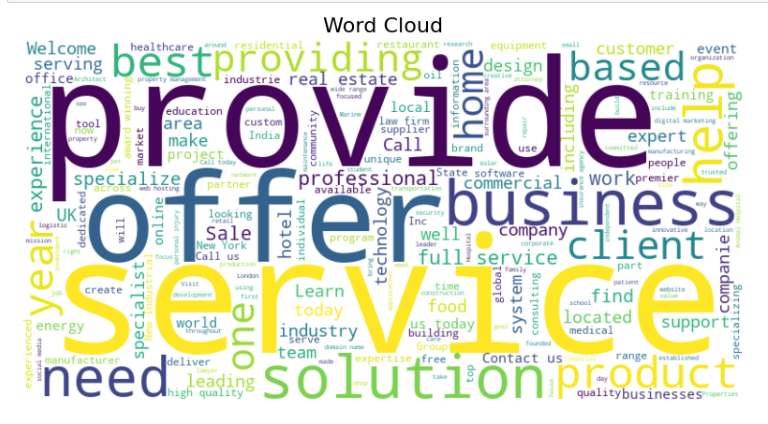
##### Homepage\_text



##### Homepage\_keywords



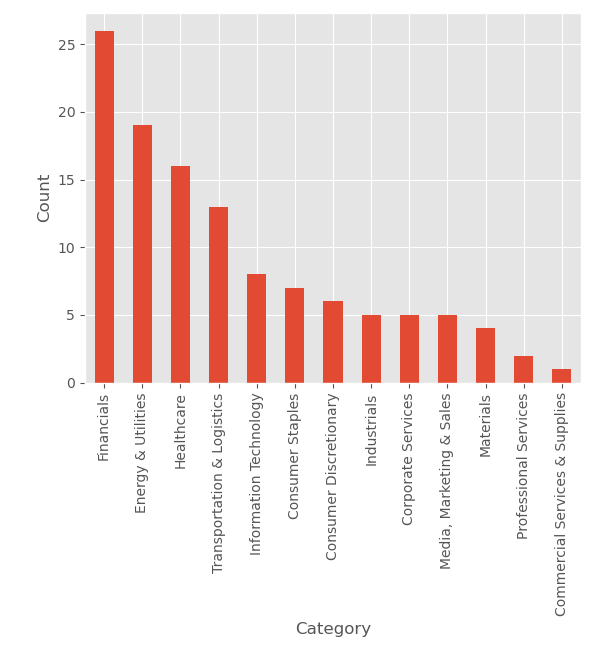
##### Meta\_description



## Questions about the data

In this section, we explore some interesting questions about the data that help us better understand the dataset.

### What categories are the most employed companies?



By considering companies with a total employee estimate greater than 5000, we can identify the categories of companies with the most employees. As shown in the figure, the Financials category has the largest companies based on employee count.

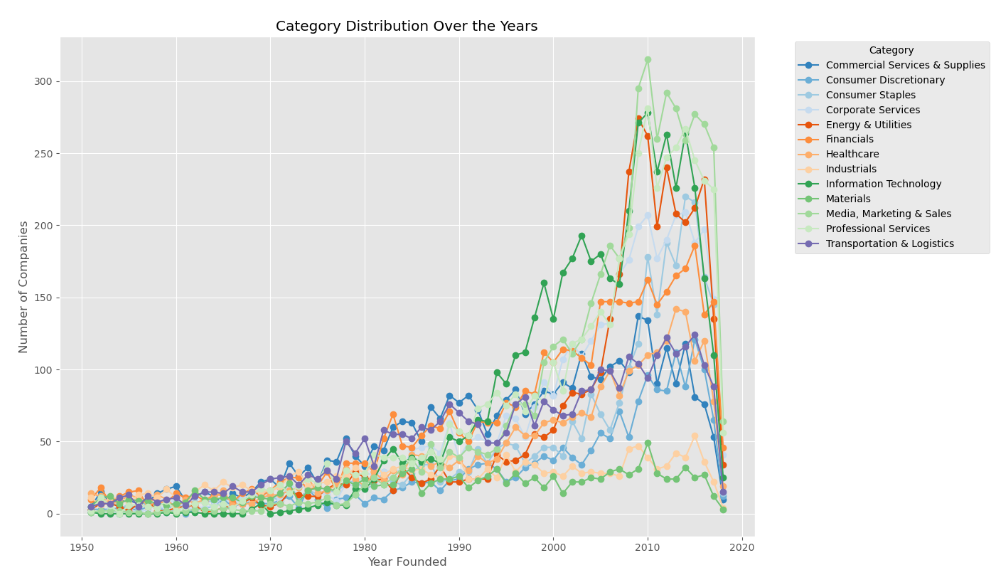
### What is the distribution of companies founded over the years?

A graph with numbers and lines

Description automatically generated

We can observe a steady increase in the number of companies founded over the years, with a peak around 2010. After that, the number of new companies tends to decline.

### How have the distributions of different company categories changed over time?

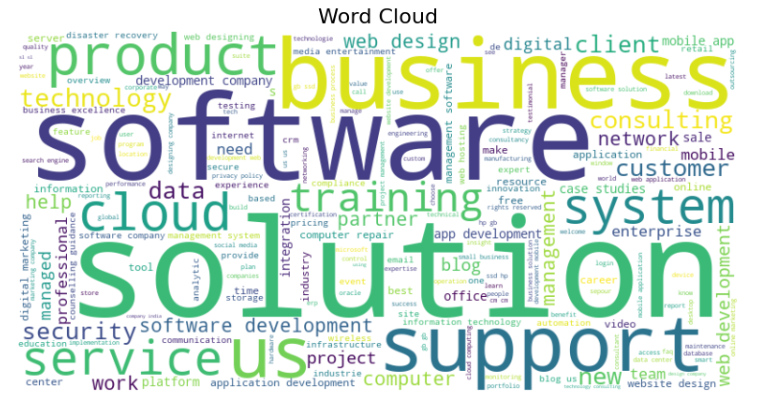


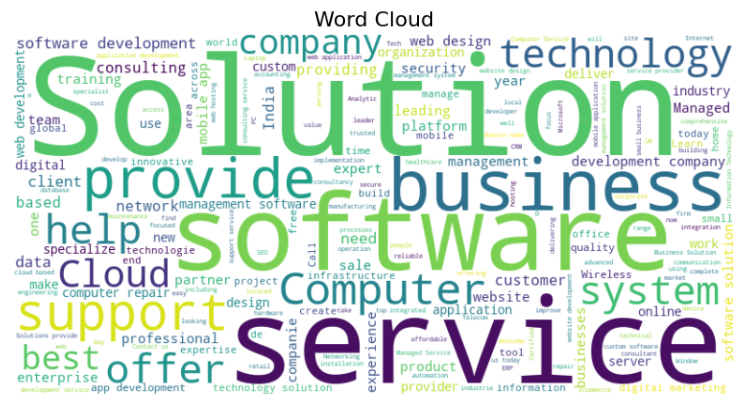
For clarity, we focus on companies founded after 1950. We can observe that categories previously at the bottom, such as Professional Services, Energy & Utilities, and Information Technology, have experienced significant growth in recent years compared to the past.

### What’s the top word in Information Technology category?

A close-up of words

Description automatically generated





As seen, all of these categories are related to Information Technology.

# Model development

## Feature Engineering

To optimize model performance, it is essential to consider diverse transformation methods for each type of feature. For this experiment, we will explore the following approaches:

### Categorical Feature Encoding

For our categorical features, we use both one-hot encoding and target encoding.

#### One hot encoding

This method creates binary columns for each unique category, ensuring that the model can process the categorical data as numerical input. The implementation was as follows:

one\_hot\_encoder = OneHotEncoder(sparse=False)

feature\_encoded = one\_hot\_encoder.fit\_transform(df[[feature]])

#### Target encoding

Target encoding replaces categorical feature values with the mean of the target variable for each category. This approach reduces dimensionality compared to one-hot encoding while capturing the relationship between the feature and the target variable. The following steps outline the process:

# Step 1: Encode the target variable as numeric values

target\_mapping = {label: idx for idx, label in enumerate(df[target].unique())}

df[f'{target}\_encoded'] = df[target].map(target\_mapping)

# Step 2: Calculate the mean target value for each target value in the feature

mean\_target\_per\_feature = df.groupby(feature)[f'{target}\_encoded'].mean()

# Step 3: Replace 'industry' with the calculated target encoding

df[f'{feature}\_encoded'] = df[feature].map(mean\_target\_per\_feature)

# Step 4: Apply smoothing to prevent overfitting

global\_mean = df[f'{target}\_encoded'].mean()

counts = df.groupby(feature).size()

smooth\_factor = 10

smoothed\_target\_encoded = (mean\_target\_per\_feature \* counts + global\_mean \* smooth\_factor) / (counts + smooth\_factor)

X\_encoded = df[feature].map(smoothed\_target\_encoded).to\_numpy().reshape(-1, 1)

### Numerical Feature Encoding

For the year\_founded feature, we applied the MinMaxScaler to scale the values within a range of 0 to 1. This transformation standardizes the feature, ensuring it is on a comparable scale with other numerical features. The implementation is as follows:

scaler = MinMaxScaler()

year\_scaled = scaler.fit\_transform(df[['year\_founded']])

### Text Representation

For the text features, we utilized **TF-IDF** and **Word2Vec** for keyword-based features, while **BERT** embeddings were used for sentence-based features. To optimize text preprocessing, we applied lemmatization to the TF-IDF and Word2Vec inputs, as these models perform better with normalized text. In contrast, BERT works more effectively with raw text, so lemmatization was not applied to its inputs. We explored three different approaches for processing the text:

* **Case 1**: Apply **TF-IDF** to all keyword-based features (homepage\_text, homepage\_keywords) and use **BERT** embeddings for the meta\_description.
* **Case 2**: For the long keyword feature (homepage\_text, average of 411 words as described in section 3.5.1), we used **Word2Vec**, as we have enough data to train the model. For the shorter keyword feature (homepage\_keywords, average of 65 words), we applied **TF-IDF**. **BERT** **embeddings** were used for the meta\_description.
* **Case 3**: Combine all text features and use **BERT embeddings** for the entire merged set of features.

The following sections provide the implementation details for **TF-IDF**, **Word2Vec**, and **BERT embeddings**.

#### TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a method that identifies important terms by weighing their frequency within a document against their occurrence across the entire dataset. This allows it to highlight words that are distinctive and significant to the document while reducing the influence of common words that appear across many documents.

The implementation is as follows:

tfidf\_vectorizer = TfidfVectorizer(

                max\_df=0.7,

                max\_features=500,

                min\_df=2,

                use\_idf=True

        )

tfidf\_feature = tfidf\_vectorizer \

                        .fit\_transform(df[feature]) \

                        .toarray()

* **max\_df=0.7**: Ignores terms that appear in more than 70% of the documents.
* **max\_features=500**: Limits the feature set to the top 500 most frequent terms.
* **min\_df=2**: Ignores terms that appear in fewer than 2 documents.
* **use\_idf=True**: Uses inverse document frequency to adjust for the frequency of words across the entire dataset.

This implementation creates a TF-IDF matrix, which transforms the text feature into numerical vectors.

#### Word2Vec

The Word2Vec model converts text into continuous vector representations by capturing semantic relationships between words. The training process builds word embeddings, which are then used to represent documents.

To train the Word2Vec model:

def train\_word2vec(documents, sg\_value, min\_count, max\_epochs=30):

    # Get number of cores from computer

    cores = multiprocessing.cpu\_count()

    # Create word2vec model

    model = Word2Vec(min\_count=min\_count,

                        window=5,

                        sample=6e-5,

                        alpha=0.03,

                        min\_alpha=0.0007,

                        negative=20,

                        workers=cores-1,

                        sg=sg\_value)

    # Build vocabulary

    model.build\_vocab(documents)

    # Train the model

    model.train(documents, total\_examples=model.corpus\_count,

                epochs=max\_epochs, report\_delay=1)

    return model

* **sg (Skip-gram model)**: When set to 1, the model uses the skip-gram approach to predict context words given a target word. If set to 0, it uses the Continuous Bag of Words (CBOW) model.
* **min\_count**: Ignores words with a frequency lower than the specified value.
* **max\_epochs**: Defines the number of training epochs.

Next, we extract embeddings from the trained model:

def vectorize\_word2vec(documents, model):

    document\_vectors = []

    # For each document (a document is a list of tokens)

    for document in documents:

        # Zero vector with size equal to word2vec vector size

        zero\_vector = np.zeros(model.vector\_size)

        # List to append word2vec vectors

        vectors = []

        # For each token in the document

        for token in document:

            # Get word2vec vector representation of that token

            # and append that vector to the vector list

            if token in model.wv:

                try:

                    vectors.append(model.wv[token])

                except KeyError:

                    # if there's not a word2vec vector representation

                    # for that token, skip that token

                    continue

        # If there's at least one non zero word2vec vector

        if vectors:

            # Get a vector that's the mean of all word2vec vectors

            vectors = np.asarray(vectors)

            avg\_vec = vectors.mean(axis=0)

            # That way the document is represented as the mean of the

            # word vectors that consist the document

            document\_vectors.append(avg\_vec)

        # If there's non any word2vec vector

        else:

            # This document is represented as the zero vector

            document\_vectors.append(zero\_vector)

    return document\_vectors

This function converts each document into a vector by averaging the Word2Vec embeddings of its tokens. If no embeddings are available for a document, a zero vector is used.

#### BERT

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based model designed to understand the context of words in a sentence. It captures semantic information and is particularly effective for sentence-level embeddings.

The implementation for generating BERT embeddings is as follows

model\_name = 'all-MiniLM-L6-v2'

model = SentenceTransformer(model\_name)

# Define a batch size

batch\_size = 64

documents = df[feature].tolist()

dataloader = DataLoader(documents, batch\_size=batch\_size)

embeddings = []

for batch in dataloader:

    embeddings.append(model.encode(batch))

X = np.vstack(embeddings)

In this implementation:

* **SentenceTransformer**: Used to load the pre-trained BERT model for sentence-level embeddings.
* **batch\_size**: Defines the number of documents processed at once. Adjusting batch size can optimize memory usage and computation time.
* **model.encode()**: Converts a batch of documents into their corresponding BERT embeddings.
* **np.vstack(embeddings)**: Stacks the embeddings from each batch into a final matrix.

## Feature Selection

We evaluate two configurations for feature selection. In the first, we use all the features extracted during the feature engineering phase. In the second configuration, we focus solely on the text features to assess their impact on model performance. This allows us to understand how much value the text features contribute compared to the entire set.

## Dataset Splitting

The dataset is divided into training and test sets. Ideally, we would employ **cross-validation** to prevent overfitting and ensure that the model generalizes well to unseen data. Cross-validation would also provide more robust estimates of model performance. However, due to **limited resources and time constraints**, we proceed with a simple train-test split.

## Classification Models

We experiment with three classification models that are well-suited for CPU processing: **RandomForestClassifier**, **LogisticRegression**, and **KNeighborsClassifier**. For simplicity, we use the default hyperparameters for each model. While tuning hyperparameters through **Grid Search** would optimize model performance, it was not performed in this experiment due to time and resource constraints.

## Identify the Best Configuration

Using **accuracy** as our evaluation metric, we identify the best configuration. This includes the choice of categorical feature encoding, text representation method, feature selection approach, and classification model. This process allows us to understand which combination of techniques yields the best performance for our dataset.

## Cross-Validation

After determining the best configuration, we apply **cross-validation** to validate the robustness of our model. Cross-validation ensures that our results are reliable and not overly dependent on a particular train-test split. This step provides a more accurate estimate of model performance on unseen data.

## Grid Search

In addition to cross-validation, we conduct **Grid Search** to fine-tune the hyperparameters of our selected model configuration. Ideally, **Grid Search** and **cross-validation** would be performed simultaneously during the model selection process. However, due to constraints, we first identified the best configuration using a single set of model parameters and now use Grid Search to optimize them.

# Model evaluation

## Evaluation Metrics

For our evaluation, we use a combination of performance metrics to gain a comprehensive understanding of the model's effectiveness. These metrics include **accuracy**, **precision**, **recall**, **F1 score**, and **Cohen’s Kappa**. Additionally, we provide a **classification report** and a **confusion matrix** to visualize and further analyze the model's performance.

* **Accuracy**: This metric measures the overall correctness of the model, calculating the proportion of correctly classified instances out of the total instances. While easy to understand, accuracy may not be ideal for imbalanced datasets.
* **Precision**: Precision indicates how many of the positive predictions made by the model are actually correct. It is particularly useful when the cost of false positives is high.
* **Recall**: Recall measures how many of the actual positive instances were correctly identified by the model. It is important in situations where the cost of false negatives is high.
* **F1 Score**: The F1 score is the harmonic mean of precision and recall, providing a balance between the two. It is especially useful when dealing with imbalanced classes, where both false positives and false negatives need to be minimized.
* **Cohen’s Kappa**: This statistic measures the agreement between the predicted and actual classifications, considering the possibility of the agreement occurring by chance. A higher Kappa score indicates better agreement, with values closer to 1 signifying strong agreement.

We also present a **classification report**, which includes a summary of key metrics such as precision, recall, and F1 score for each class. Finally, the **confusion matrix** visually displays the number of true positives, true negatives, false positives, and false negatives, providing a clearer view of the model’s performance in distinguishing between classes.

## Identify the Best Configuration

This section involves selecting the best combination of categorical feature encoding, text representation method, feature selection strategy, and classification model to achieve the highest model performance

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Text Representation | Encoding | Features | Classifier | Accuracy | Precision | Recall | F1 Score | Cohen Kappa |
| TWB | OH | all | RF | 0.959 | 0.959 | 0.959 | 0.959 | 0.955 |
| TWB | OH | all | LR | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| TWB | OH | all | kNN | 0.990 | 0.990 | 0.990 | 0.990 | 0.989 |
| TWB | OH | text | RF | 0.806 | 0.809 | 0.806 | 0.804 | 0.788 |
| TWB | OH | text | LR | 0.863 | 0.863 | 0.863 | 0.863 | 0.851 |
| TWB | OH | text | kNN | 0.827 | 0.831 | 0.827 | 0.827 | 0.811 |
| TWB | T | all | RF | 0.918 | 0.919 | 0.918 | 0.917 | 0.910 |
| TWB | T | all | LR | 0.982 | 0.982 | 0.982 | 0.982 | 0.980 |
| TWB | T | all | kNN | 0.995 | 0.995 | 0.995 | 0.995 | 0.994 |
| TWB | T | text | RF | 0.808 | 0.812 | 0.808 | 0.806 | 0.791 |
| TWB | T | text | LR | 0.863 | 0.863 | 0.863 | 0.863 | 0.851 |
| TWB | T | text | kNN | 0.827 | 0.831 | 0.827 | 0.827 | 0.811 |
| TTB | OH | all | RF | 0.957 | 0.958 | 0.957 | 0.957 | 0.953 |
| TTB | OH | all | LR | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| TTB | OH | all | kNN | 0.983 | 0.983 | 0.983 | 0.983 | 0.981 |
| TTB | OH | text | RF | 0.808 | 0.810 | 0.808 | 0.806 | 0.790 |
| TTB | OH | text | LR | 0.860 | 0.860 | 0.860 | 0.860 | 0.847 |
| TTB | OH | text | kNN | 0.773 | 0.782 | 0.773 | 0.775 | 0.752 |
| TTB | T | all | RF | 0.906 | 0.907 | 0.906 | 0.906 | 0.898 |
| TTB | T | all | LR | 0.981 | 0.981 | 0.981 | 0.981 | 0.979 |
| TTB | T | all | kNN | 0.991 | 0.991 | 0.991 | 0.991 | 0.990 |
| TTB | T | text | RF | 0.807 | 0.809 | 0.807 | 0.806 | 0.790 |
| TTB | T | text | LR | 0.860 | 0.860 | 0.860 | 0.860 | 0.847 |
| TTB | T | text | kNN | 0.773 | 0.782 | 0.773 | 0.775 | 0.752 |
| B | OH | all | RF | 0.958 | 0.958 | 0.958 | 0.957 | 0.954 |
| B | OH | all | LR | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| B | OH | all | kNN | 0.929 | 0.930 | 0.929 | 0.929 | 0.923 |
| B | OH | text | RF | 0.716 | 0.721 | 0.716 | 0.711 | 0.690 |
| B | OH | text | LR | 0.794 | 0.794 | 0.794 | 0.794 | 0.776 |
| B | OH | text | kNN | 0.771 | 0.776 | 0.771 | 0.771 | 0.750 |
| B | T | all | RF | 0.904 | 0.906 | 0.904 | 0.902 | 0.895 |
| B | T | all | LR | 0.969 | 0.969 | 0.969 | 0.969 | 0.967 |
| B | T | all | kNN | 0.976 | 0.976 | 0.976 | 0.975 | 0.973 |
| B | T | text | RF | 0.720 | 0.724 | 0.720 | 0.714 | 0.694 |
| B | T | text | LR | 0.794 | 0.794 | 0.794 | 0.794 | 0.776 |
| B | T | text | kNN | 0.771 | 0.776 | 0.771 | 0.771 | 0.750 |

**Key-findings**:

* **Feature Selection:** Using all features rather than just the text-based features significantly impact the model's performance, suggesting that the additional features contribute valuable information.
* **Text Representation**: The combination of **tf-idf, word2vec, and BERT** generally outperforms **tf-idf, tf-idf, BERT**, which in turn performs slightly better than an **all-BERT** approach. This is expected, as **homepage\_text** (with an average length of 411 words) benefits from word2vec's ability to capture semantic relationships between words. Additionally***,* homepage\_text**and **homepage\_keywords**, being keyword-based rather than sentence-based, are more effectively represented by a keyword-based vectorizer like tf-idf, compared to a sentence-based model like BERT. Another factor is that splitting the features, rather than merging them into one, allows each feature to provide more distinct insights, preventing important information from being lost. This is particularly relevant for **homepage\_keywords,** which has a shorter length compared to the other features and may lose relevance when combined with the longer text features.
* **Encoding Method**: **One-hot encoding** works best with **Logistic Regression** and **RandomForestClassifier**, while **target encoding** shows better performance with **k-NN**.
* **Classifier**: The best performing classifier is **Logistic Regression**, particularly when paired with **One-hot encoding**, closely followed by **k-NN** when paired with **target encoding.**

In conclusion, **Logistic Regression** with **One-hot encoding** and **all** **features** appears to perform consistently well across all text representation methods. Taking into account the overall performance of the text representation methods, we opt for the **tf-idf**, **word2vec**, and **BERT** combination for the final model configuration.

## Results with Cross Validation

To validate our results, we employed **K-Fold cross-validation**. This technique helps to assess the model’s ability to generalize to unseen data by partitioning the dataset into multiple folds and iterating through each one as the validation set while using the remaining folds for training. The cross-validation results show an impressive accuracy of **0.9999**.

from sklearn.model\_selection import KFold

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, cohen\_kappa\_score

import numpy as np

# Parameters

n\_splits = 5  # Number of folds

kf = KFold(n\_splits=n\_splits, shuffle=True, random\_state=42)

# Initialize logistic regression

model = LogisticRegression()

# Store metrics for each fold

results = []

# K-Fold Cross Validation

for train\_index, test\_index in kf.split(X):

    # Split data

    X\_train, X\_test = X[train\_index], X[test\_index]

    y\_train, y\_test = y[train\_index], y[test\_index]

    # Train model

    model.fit(X\_train, y\_train)

    # Predict

    y\_pred = model.predict(X\_test)

    # Compute metrics

    metrics = {

        "accuracy": accuracy\_score(y\_test, y\_pred),

        "precision": precision\_score(y\_test, y\_pred, average='weighted'),

        "recall": recall\_score(y\_test, y\_pred, average='weighted'),

        "f1\_score": f1\_score(y\_test, y\_pred, average='weighted'),

        "cohen\_kappa": cohen\_kappa\_score(y\_test, y\_pred),

    }

    results.append(metrics)

# Average metrics over all folds

avg\_metrics = {

    metric: np.mean([fold[metric] for fold in results]) for metric in results[0]

}

# Print results

print("Average metrics over", n\_splits, "folds:")

for metric, value in avg\_metrics.items():

    print(f"{metric}: {value:.4f}")

## Results with Grid Search

To further improve model performance, we conducted a **Grid Search** over the following hyperparameters:

param\_grid = {

    'C': [0.01, 0.1, 1, 10, 100],

    'solver': ['liblinear', 'lbfgs', 'saga'],

    'class\_weight': [None, 'balanced']

}

After performing the grid search, we found that the model performed best with **C: 1**, **solver: liblinear**, and **class\_weight: None**. This change in solver from **'lbfgs'** to **'liblinear'** was the only modification compared to our previous model configuration. Upon retraining the model with these optimal parameters and conducting cross-validation again, we achieved an outstanding accuracy of **1.0000**.

## Evaluation results of the best model

The performance of the **Logistic Regression** classifier was evaluated using several key metrics. Below are the primary evaluation results:

**Overall Accuracy:**

* **Accuracy**: 1.00

**Classification Metrics:**

* **Precision:** 1.00
* **Recall:** 1.00
* **F1 Score:** 1.00

**Cohen's Kappa:** 1.00

The detailed evaluation through the classification report indicates that the model performs excellently across all classes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Precision** | **Recall** | **F1-Score** | **Support** |
| Commercial Services & Supplies | 1.00 | 1.00 | 1.00 | 1272 |
| Consumer Discretionary | 1.00 | 1.00 | 1.00 | 533 |
| Consumer Staples | 1.00 | 1.00 | 1.00 | 1029 |
| Corporate Services | 1.00 | 1.00 | 1.00 | 1333 |
| Energy & Utilities | 1.00 | 1.00 | 1.00 | 1085 |
| Financials | 1.00 | 1.00 | 1.00 | 1371 |
| Healthcare | 1.00 | 1.00 | 1.00 | 1516 |
| Industrials | 1.00 | 1.00 | 1.00 | 662 |
| Information Technology | 1.00 | 1.00 | 1.00 | 1189 |
| Materials | 1.00 | 1.00 | 1.00 | 517 |
| Media, Marketing & Sales | 1.00 | 1.00 | 1.00 | 1231 |
| Professional Services | 1.00 | 1.00 | 1.00 | 1391 |
| Transportation & Logistics | 1.00 | 1.00 | 1.00 | 1261 |
| **Macro Average** | 1.00 | 1.00 | 1.00 | 14390 |
| **Weighted Average** | 1.00 | 1.00 | 1.00 | 14390 |

Finally, the **Confusion Matrix** confirms the model's strong classification performance:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Actual \ Predicted** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| **0** | 1272 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 533 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 1029 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 1033 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 1085 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 | 1371 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 1516 | 0 | 0 | 0 | 0 | 0 | 0 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 662 | 0 | 0 | 0 | 0 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1189 | 0 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 517 | 0 | 0 | 0 |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1231 | 0 | 0 |
| **11** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1391 | 0 |
| **12** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1261 |

# Next Steps

## Evaluate Alternative Models

While we have utilized models optimized for CPU usage, it would be beneficial to explore models better suited for GPU acceleration, such as **Gradient Boosting**, **Support Vector Machines (SVM)**, or **Multi-Layer Perceptrons (MLP)**. These models may offer performance improvements, particularly when dealing with large datasets, and could enhance our results.

## Investigate Feature Selection

It would be insightful to evaluate how the model performs when excluding text-based features. By testing with just the structured features, we can assess their individual impact on the model's performance and determine whether text data is crucial for achieving high accuracy.

## Grid Search and Cross-Validation

Although our current model achieves an accuracy of 1.000, it would be valuable to explore other models that may also perform perfectly. For instance, we observed an accuracy of 0.995 with target encoding and k-NN. Conducting grid search on these models could potentially yield configurations that also achieve perfect performance. These alternative models may generalize better to unseen data, in case the current model has overfitted.

## Experiment with Large Language Models (LLMs)

A promising next step would be to evaluate the performance of the model using a Large Language Model (LLM) such as DistilBERT. Given the complexity and potential of LLMs, their integration could significantly improve the model’s ability to understand and process text data. Below is an example of how we can integrate and fine-tune a DistilBERT model for classification:

# Combine columns into text

def row\_to\_text(row):

    return f"Year Founded: {row['year\_founded']}; Country: {row['country']}; Employee Range: {row['employee\_range']}; " \

           f"Industry: {row['industry']}; Homepage Text: {row['homepage\_text']}; Homepage Keywords: {row['homepage\_keywords']}; " \

           f"Meta Description: {row['meta\_description']}"

df['text'] = df.apply(row\_to\_text, axis=1)

# Encode target labels

label\_encoder = LabelEncoder()

df['label'] = label\_encoder.fit\_transform(df['category'])

# Split data into train and test sets

train\_texts, test\_texts, train\_labels, test\_labels = train\_test\_split(

    df['text'], df['label'], test\_size=0.2, random\_state=42

)

# Tokenization

tokenizer = DistilBertTokenizer.from\_pretrained('distilbert-base-uncased')

def tokenize\_function(examples):

    return tokenizer(examples['text'], padding="max\_length", truncation=True)

train\_dataset = Dataset.from\_dict({"text": train\_texts, "label": train\_labels})

test\_dataset = Dataset.from\_dict({"text": test\_texts, "label": test\_labels})

# Tokenize the datasets

train\_dataset = train\_dataset.map(tokenize\_function, batched=True)

test\_dataset = test\_dataset.map(tokenize\_function, batched=True)

# Set format for PyTorch

train\_dataset = train\_dataset.with\_format("torch")

test\_dataset = test\_dataset.with\_format("torch")

# Load pre-trained DistilBERT model

num\_labels = len(label\_encoder.classes\_)

model = DistilBertForSequenceClassification.from\_pretrained("distilbert-base-uncased", num\_labels=num\_labels)

# Define training arguments

training\_args = TrainingArguments(

    output\_dir="./results",

    evaluation\_strategy="epoch",

    save\_strategy="epoch",

    logging\_dir="./logs",

    num\_train\_epochs=3,

    per\_device\_train\_batch\_size=4,

    per\_device\_eval\_batch\_size=4,

    learning\_rate=5e-5,

    weight\_decay=0.01,

    save\_total\_limit=2,

    gradient\_accumulation\_steps=4,

    load\_best\_model\_at\_end=True

)

# Define Trainer

trainer = Trainer(

    model=model,

    args=training\_args,

    train\_dataset=train\_dataset,

    eval\_dataset=test\_dataset

)

# Train the model

trainer.train()

# Evaluate the model

results = trainer.evaluate()