Contents

1 SQL Queries

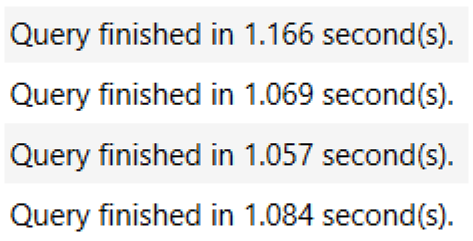
1.1 Query no 1

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

1.1.1 Basic query

```sql
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



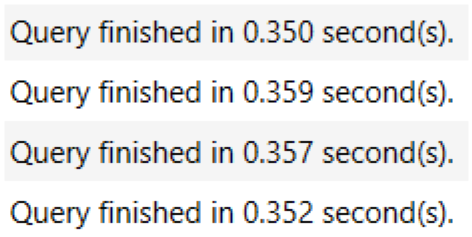
```
Query finished in 1.166 second(s).
Query finished in 1.069 second(s).
Query finished in 1.057 second(s).
Query finished in 1.084 second(s).
```

1.1.2 Optimization using indices

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

```sql
CREATE INDEX
    idx_industry_year_employee
ON
    CompanyDataset(industry, "year founded", "current employee estimate");
```

Average running time: 0.35s



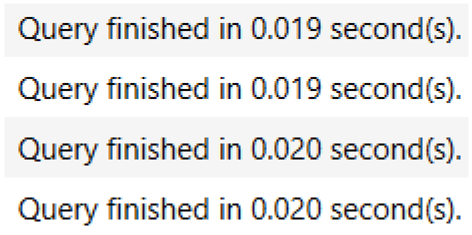
```
Query finished in 0.350 second(s).
Query finished in 0.359 second(s).
Query finished in 0.357 second(s).
Query finished in 0.352 second(s).
```

1.1.3 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!



```
Query finished in 0.019 second(s).
Query finished in 0.019 second(s).
Query finished in 0.020 second(s).
Query finished in 0.020 second(s).
```

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.

1.1.4 Full code

```sql
-- 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
```

1.2 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.

1.2.1 Identify what is ‘technology’

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

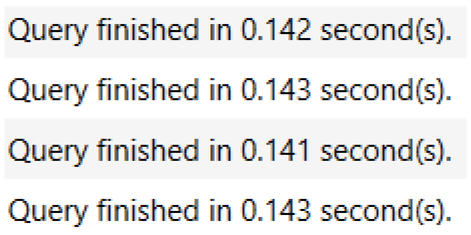
- **Case 1:** A "Technology"-like industry could be defined as any industry that we determine to fall within the technology sector. For instance, industries such as medical devices and nanotechnology are considered technology industries, even though they do not fall under the "Information Technology"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

1.2.2 Case 1

1.2.2.1 Basic query

```sql
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



```
Query finished in 0.142 second(s).
Query finished in 0.143 second(s).
Query finished in 0.141 second(s).
Query finished in 0.143 second(s).
```

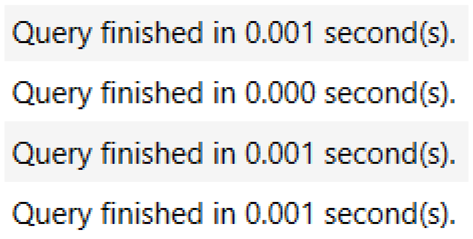
1.2.2.2 Optimization using indices

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



```
Query finished in 0.001 second(s).
Query finished in 0.000 second(s).
Query finished in 0.001 second(s).
Query finished in 0.001 second(s).
```

1.2.2.3 Full code

```sql
-- 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
```

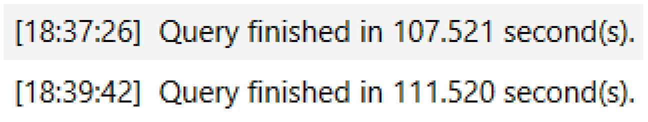
```
    CompanyClassification cc -- Inner join of CompanyClassification
JOIN
    CompanyDataset cd -- and CompanyDataset
ON
    cc.CompanyName = cd.CompanyName -- CompanyName is the common column between the
two tables
WHERE
    cc.homepage_text IS NULL -- Filter to have an effective homepage
    AND 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
```

1.2.3 Case 2

1.2.3.1 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



```
[18:37:26]  Query finished in 107.521 second(s).
[18:39:42]  Query finished in 111.520 second(s).
```

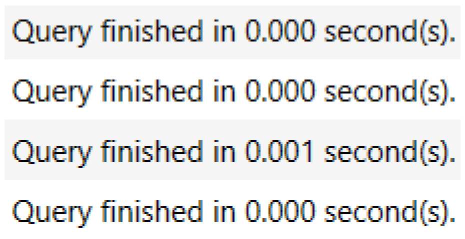
1.2.3.2 Optimization using indices

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;
```

```sql
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


```
Query finished in 0.000 second(s).
Query finished in 0.000 second(s).
Query finished in 0.001 second(s).
Query finished in 0.000 second(s).
```

1.2.3.3 Full code

```sql
-- 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
```

1.3 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.

1.3.1 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

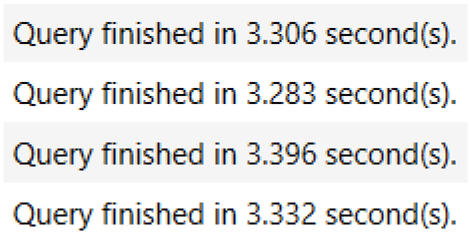
The RANK() method shares a similar disadvantage to DENSE_RANK(), in that it may result in more than 5 companies being included in the ranking. However, the number of additional companies is generally smaller, unless there is a significant number of companies sharing the same "Total employees" value. 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

1.3.2 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



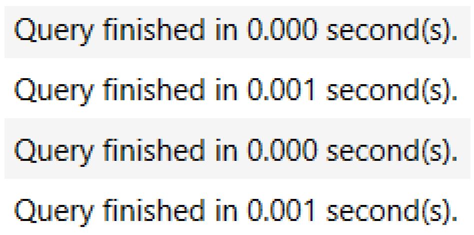
```
Query finished in 3.306 second(s).
Query finished in 3.283 second(s).
Query finished in 3.396 second(s).
Query finished in 3.332 second(s).
```

1.3.3 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



```
Query finished in 0.000 second(s).
Query finished in 0.001 second(s).
Query finished in 0.000 second(s).
Query finished in 0.001 second(s).
```

1.3.4 Full code

```sql
--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
```

2 Data integration and Database insertion

2.1 Merge the two company datasets

2.1.1 Define a key for merging

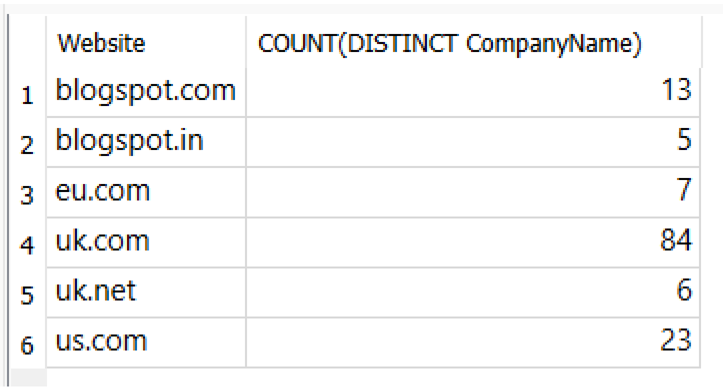
The two datasets have 2 common columns. CompanyName and Website. Intuitively CompanyName seems better option since there can be a company without a website, but it can’t be a website without a CompanyName.

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:

```sql
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:



	Website	COUNT(DISTINCT CompanyName)
1	blogspot.com	13
2	blogspot.in	5
3	eu.com	7
4	uk.com	84
5	uk.net	6
6	us.com	23

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

2.1.2 Duplicate removal

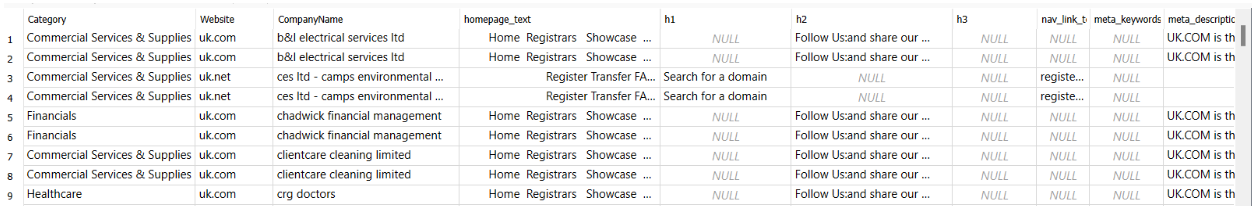
Before merging the data, we’ll have to handle the possible duplicates.

2.1.2.1 Duplicates in CompanyClassification

Let’s check the duplicates on our merging key:

```sql
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:



	Category	Website	CompanyName	homepage_text	h1	h2	h3	nav_link_t	meta_keywords	meta_descriptic
1	Commercial Services & Supplies	uk.com	b&l electrical services ltd	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
2	Commercial Services & Supplies	uk.com	b&l electrical services ltd	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
3	Commercial Services & Supplies	uk.net	ces ltd - camps environmental ...	Register Transfer FA...	Search for a domain	NULL	NULL	registe...	NULL	
4	Commercial Services & Supplies	uk.net	ces ltd - camps environmental ...	Register Transfer FA...	Search for a domain	NULL	NULL	registe...	NULL	
5	Financials	uk.com	chadwick financial management	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
6	Financials	uk.com	chadwick financial management	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
7	Commercial Services & Supplies	uk.com	clientcare cleaning limited	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
8	Commercial Services & Supplies	uk.com	clientcare cleaning limited	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th
9	Healthcare	uk.com	crg doctors	Home Registrars Showcase ...	NULL	Follow Us:and share our ...	NULL	NULL	NULL	UK.COM is th

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

```sql
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:

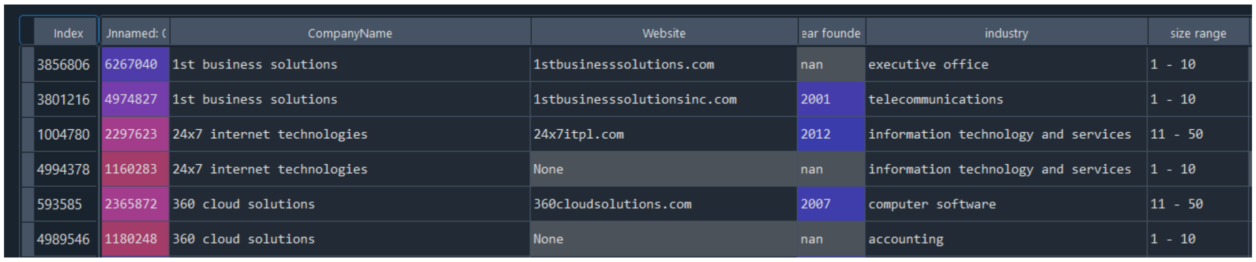
```
cc = cc.drop_duplicates(subset='CompanyName', keep='first')
```

2.1.2.2 Duplicates in CompanyDataset

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

2.1.2.2.1 Find the duplicates

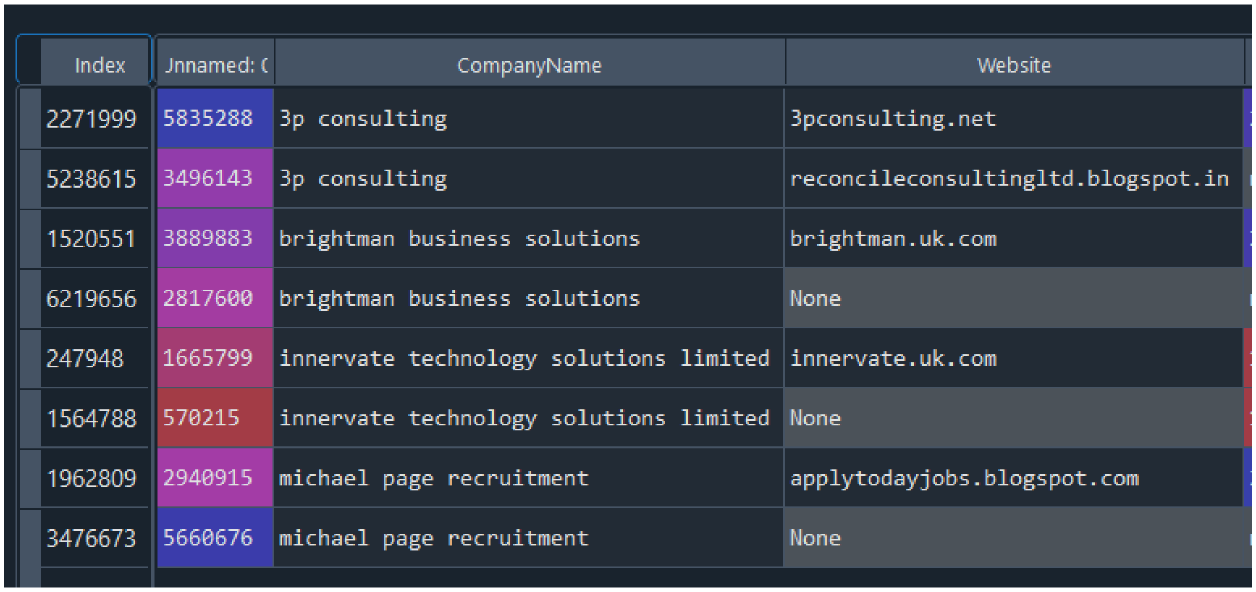
We are only interested in the CompanyNames that are also present in the CompanyClassification dataset. The duplicates we are looking for appear as follows:



Index	Jnnamed: (	CompanyName	Website	ear founde	industry	size range
3856806	6267040	1st business solutions	1stbusinesssolutions.com	nan	executive office	1 - 10
3801216	4974827	1st business solutions	1stbusinesssolutionsinc.com	2001	telecommunications	1 - 10
1004780	2297623	24x7 internet technologies	24x7itpl.com	2012	information technology and services	11 - 50
4994378	1160283	24x7 internet technologies	None	nan	information technology and services	1 - 10
593585	2365872	360 cloud solutions	360cloudsolutions.com	2007	computer software	11 - 50
4989546	1180248	360 cloud solutions	None	nan	accounting	1 - 10

2.1.2.2.2 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:



Index	Jnnamed: (	CompanyName	Website
2271999	5835288	3p consulting	3pconsulting.net
5238615	3496143	3p consulting	reconcileconsultingltd.blogspot.in
1520551	3889883	brightman business solutions	brightman.uk.com
6219656	2817600	brightman business solutions	None
247948	1665799	innervate technology solutions limited	innervate.uk.com
1564788	570215	innervate technology solutions limited	None
1962809	2940915	michael page recruitment	applytodayjobs.blogspot.com
3476673	5660676	michael page recruitment	None

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.

2.1.2.2.3 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()
```

2.1.3 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.

2.1.4 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.

2.1.5 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'])
```

2.1.6 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)
```

2.1.7 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()
```

3 Exploratory Data Analysis (EDA)

3.1 Basic understanding of the data

3.1.1 DataFrame.shape

Our data consists of 74856 rows and 19 columns.

3.1.2 DataFrame.columns

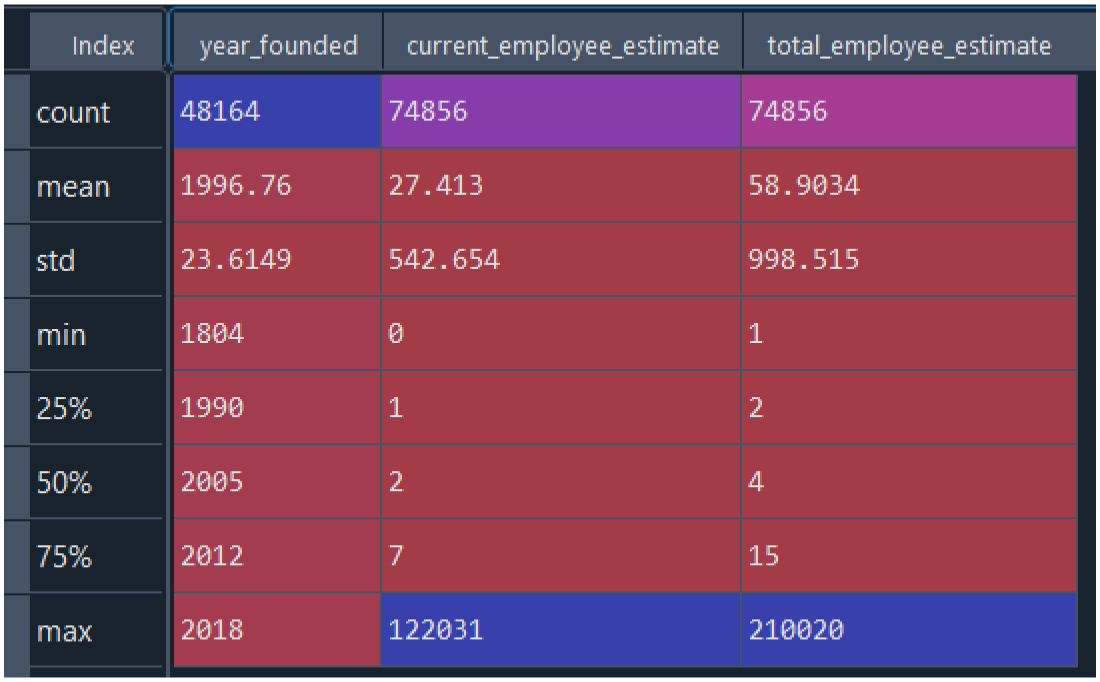
The columns of our dataset are:

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	

3.1.3 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

3.1.4 DataFrame.describe



Index	year_founded	current_employee_estimate	total_employee_estimate
count	48164	74856	74856
mean	1996.76	27.413	58.9034
std	23.6149	542.654	998.515
min	1804	0	1
25%	1990	1	2
50%	2005	2	4
75%	2012	7	15
max	2018	122031	210020

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.

3.1.5 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

3.2 Data Preparation

3.2.1 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)
```

3.2.2 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')
```

3.2.3 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.

3.3 Numeric features

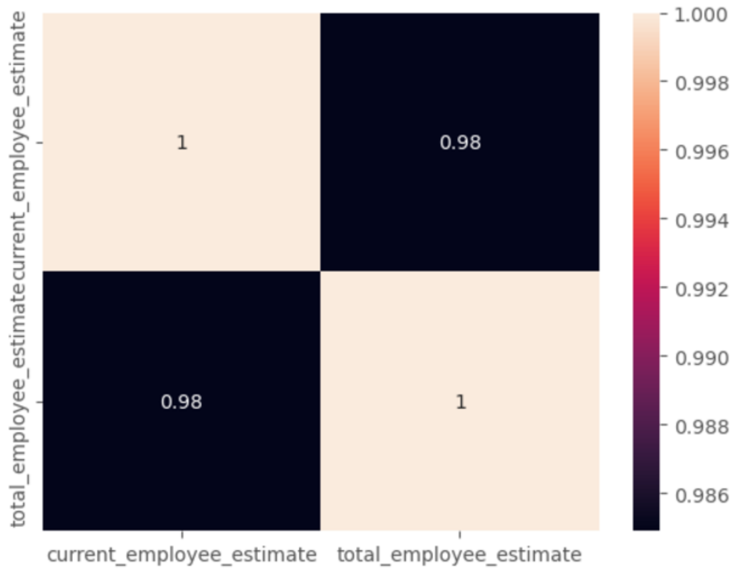
The numeric features of our dataset are the year_founded, the current_employee_estimate and the total_employee_estimate.

3.3.1 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.

3.3.1.1 Correlation

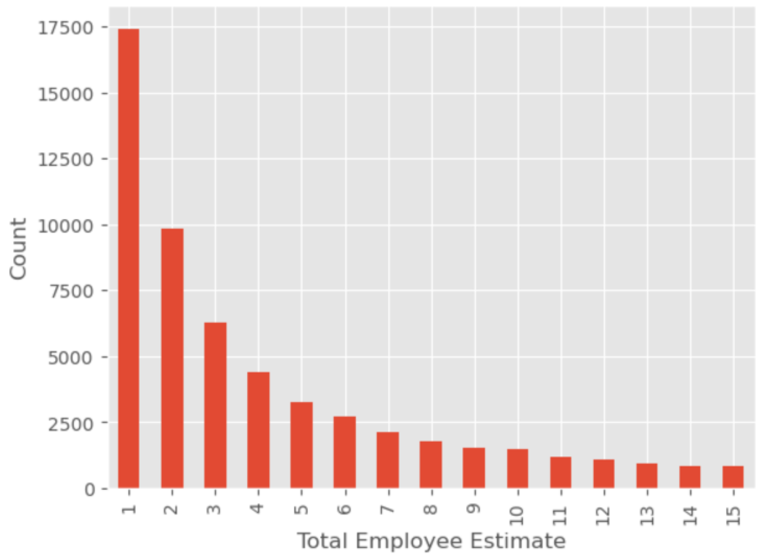
First let's investigate the corelation between these two features:



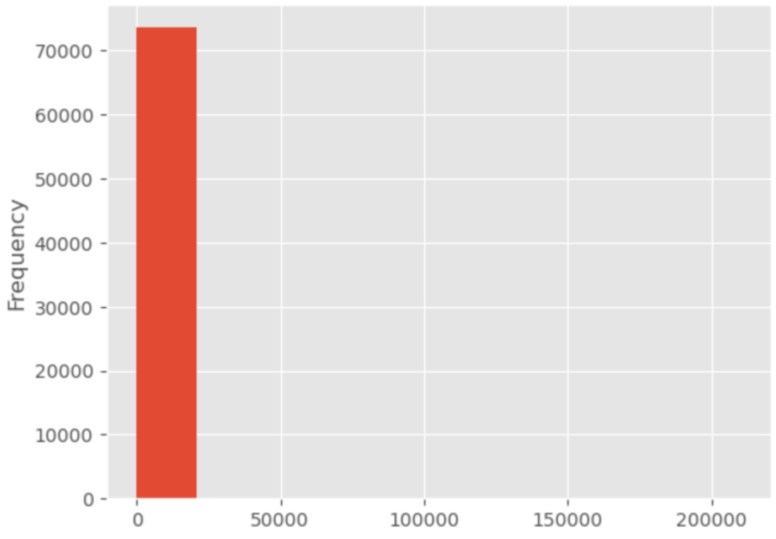
The two features are very highly correlated, so we will proceed with our analysis using only one of them.

3.3.1.2 Value Frequency

If we count the occurrences of each value in total_employee_estimate, the most frequent values are shown here:

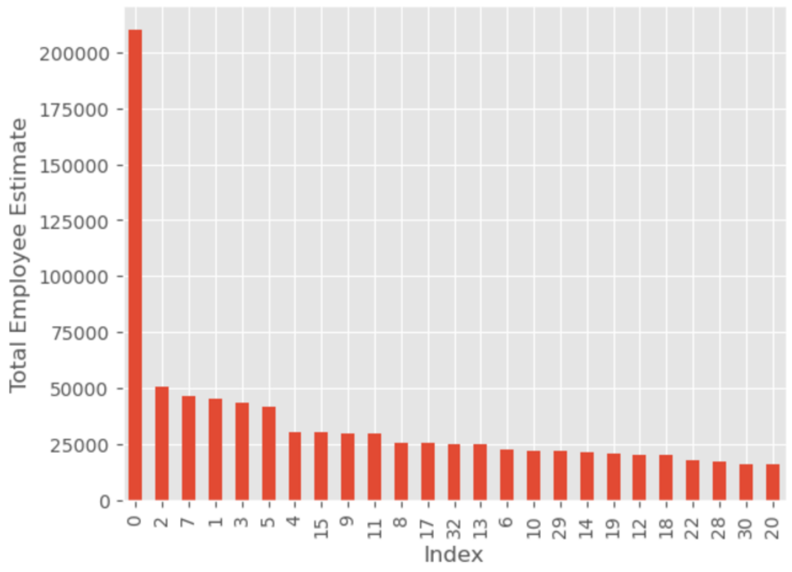


We can observe that most companies have a very small number of employees. To visualize this, let's plot a histogram:



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

3.3.1.3 Peak Values



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.

3.3.1.4 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).

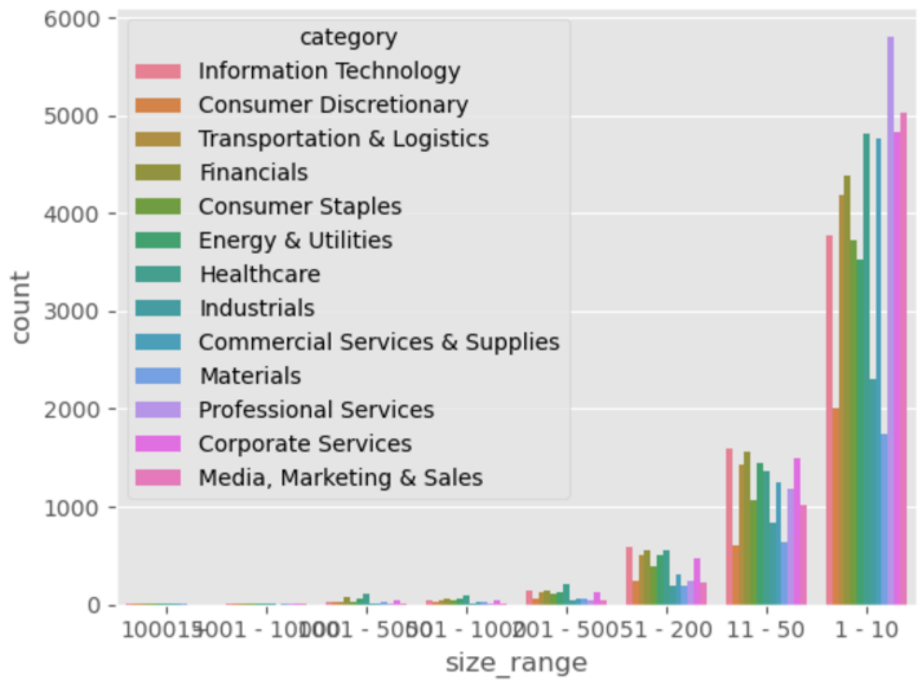
Threshold	Count	Percentage
25000	14	0.02
10000	43	0.06
5000	117	0.16
1000	575	0.78
500	1098	1.49
300	1710	2.32
200	2518	3.42
100	4493	6.11
50	7675	10.43
40	9092	12.35
30	11212	15.24
20	14825	20.15
10	22771	30.94
9	24246	32.95
8	25794	35.05
7	27592	37.49

6	29712	40.37
5	32419	44.05
4	35663	48.46
3	40053	54.43
2	46338	62.97
1	56180	76.34

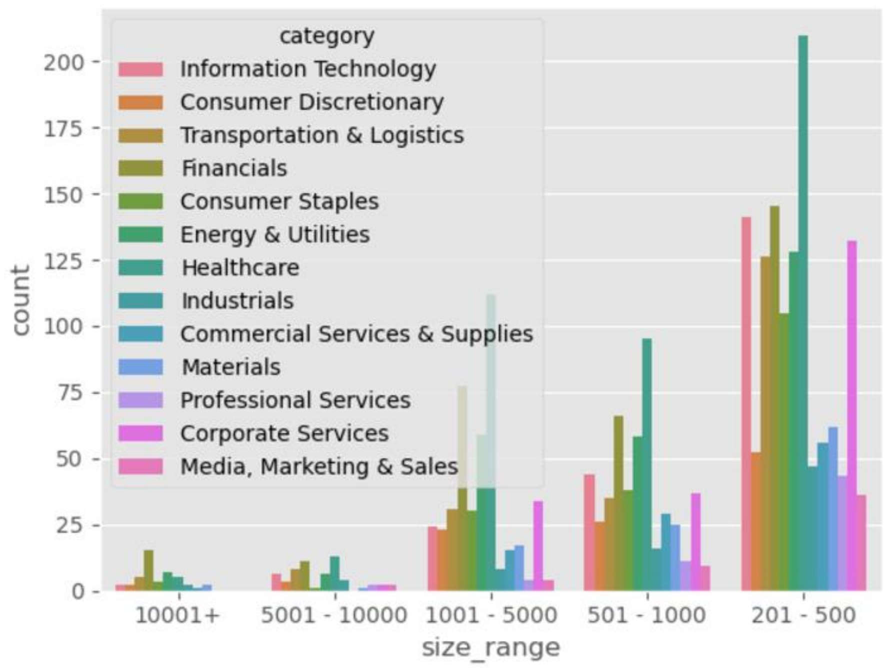
Indeed, only 10% of the companies have more than 50 employees, while 70% have fewer than 10 employees. This suggests that we may need to adjust the size_range, which currently only has a single bin for the 1-10 employee range.

3.3.1.5 Size_range distribution by Category

Let's examine the distribution of size_range across different categories:



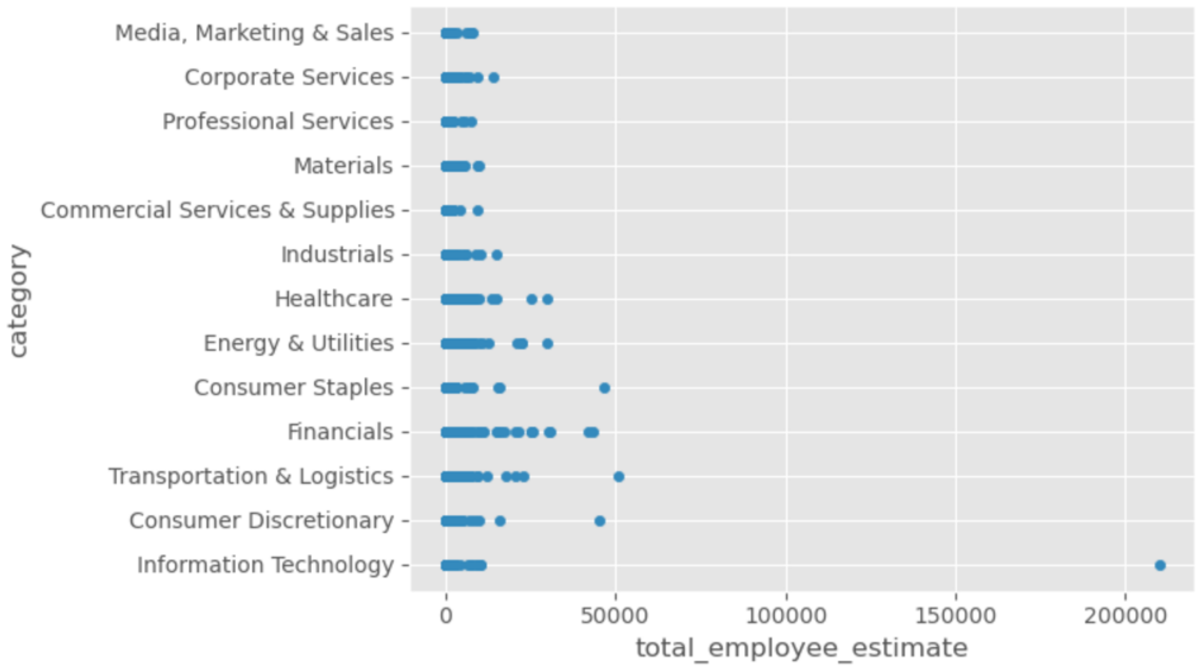
And by excluding the top 3 categories, we can better observe the distribution of the other categories:



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).

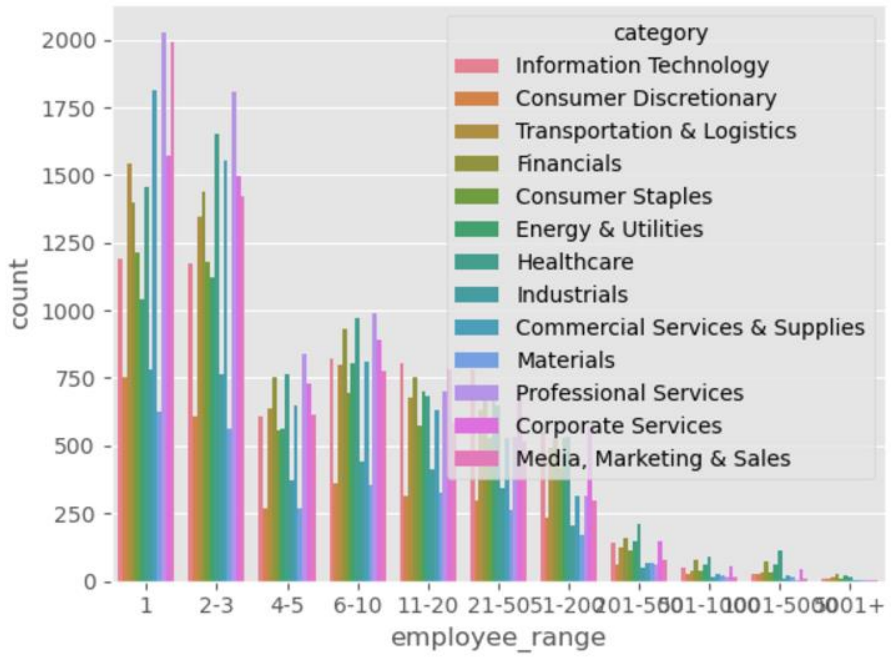
3.3.1.6 Outliers



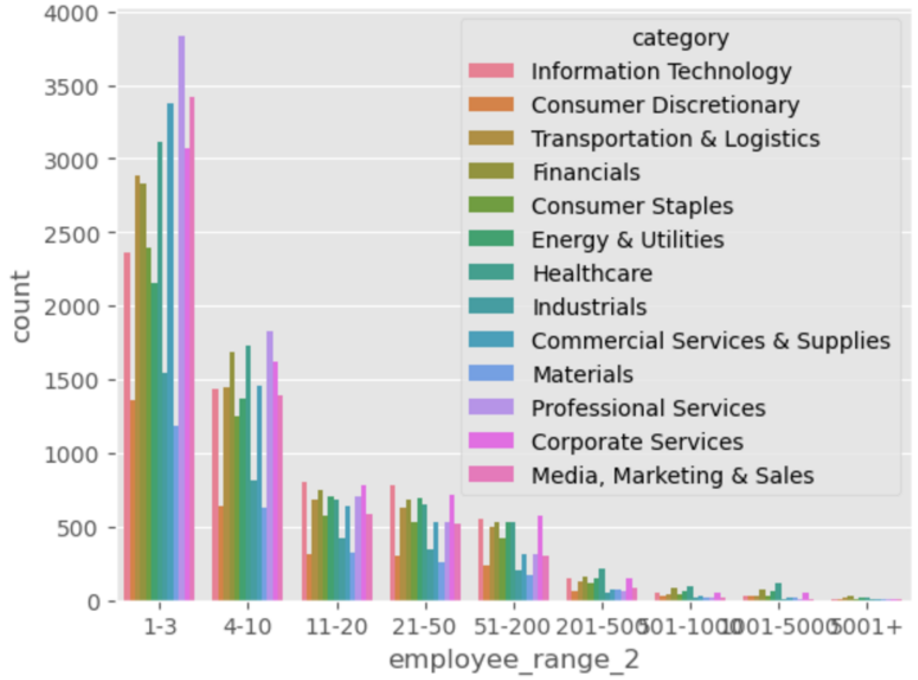
We can observe that most of the outliers are concentrated in specific categories. Although their numbers are low, they are likely to provide valuable insights for our prediction model. However, since the number of these companies is minimal, we will merge the 5001-10000 employee range with the 10001+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

3.3.1.7 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.

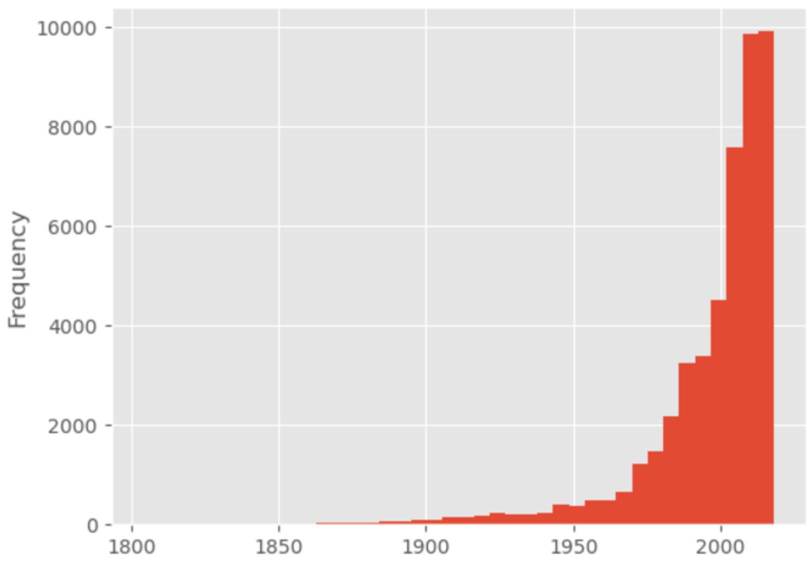


Here, we can observe clear differences between the 1-3 and 4-10 employee ranges. For example, the 1-3 range has more companies in Media, Marketing & Sales compared to Corporate Services, and more in Consumer Staples than in Energy & Utilities. Based on these observations, we create a new feature called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

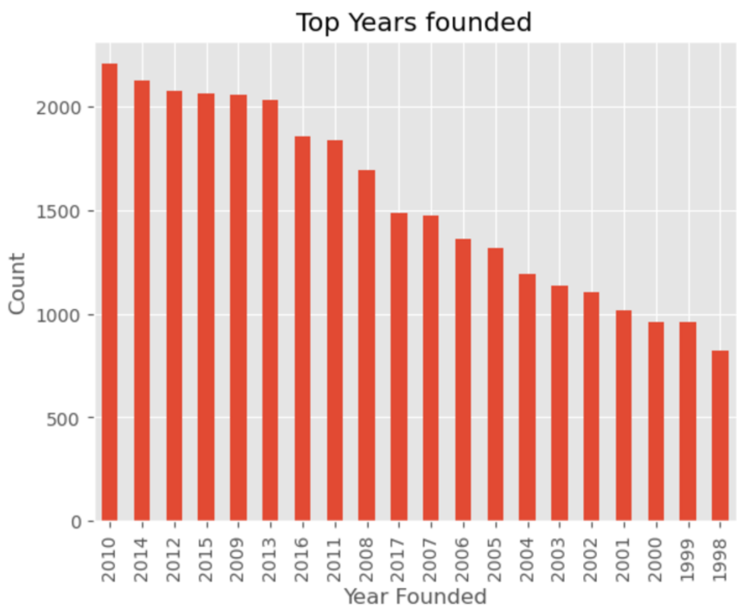
3.3.2 Year founded

3.3.2.1 Value Frequency

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

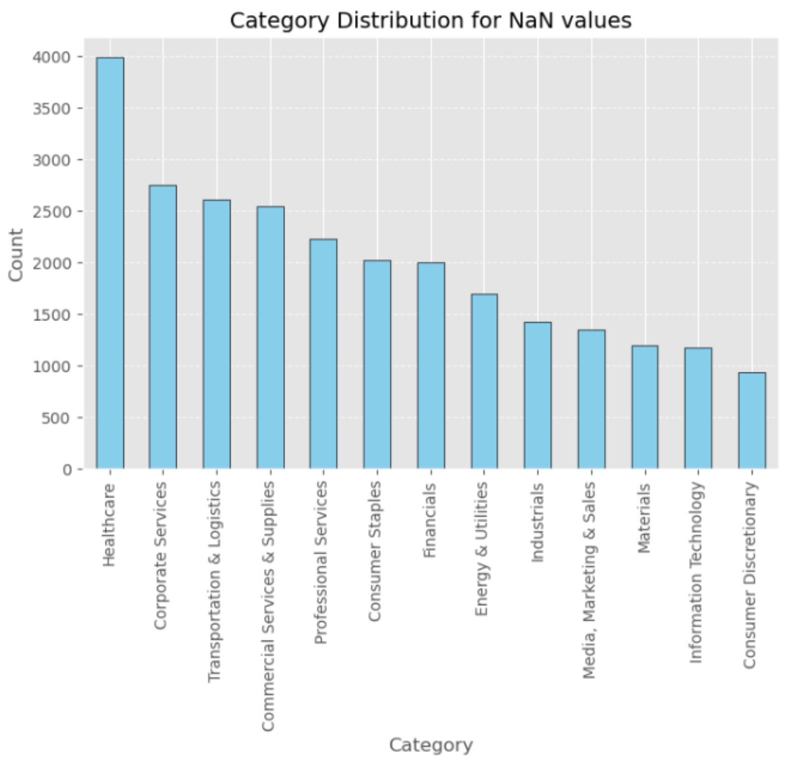


However, this doesn't necessarily mean that more companies were founded in 2015 compared to 2010, for example:

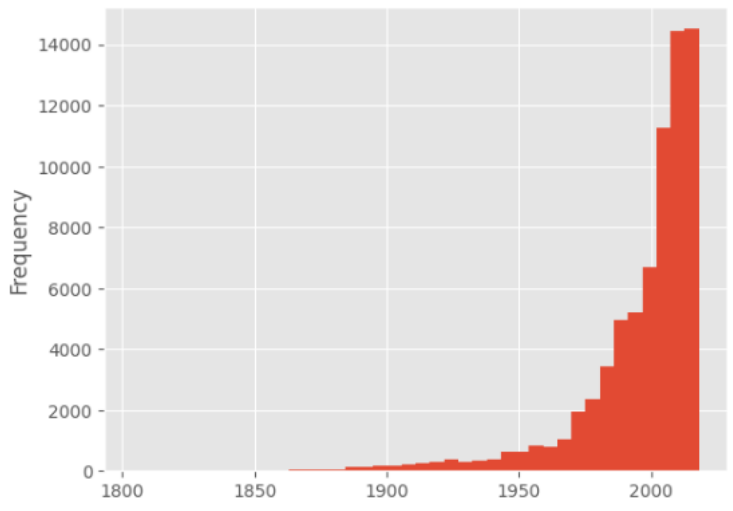


3.3.2.2 Missing Values

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



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:



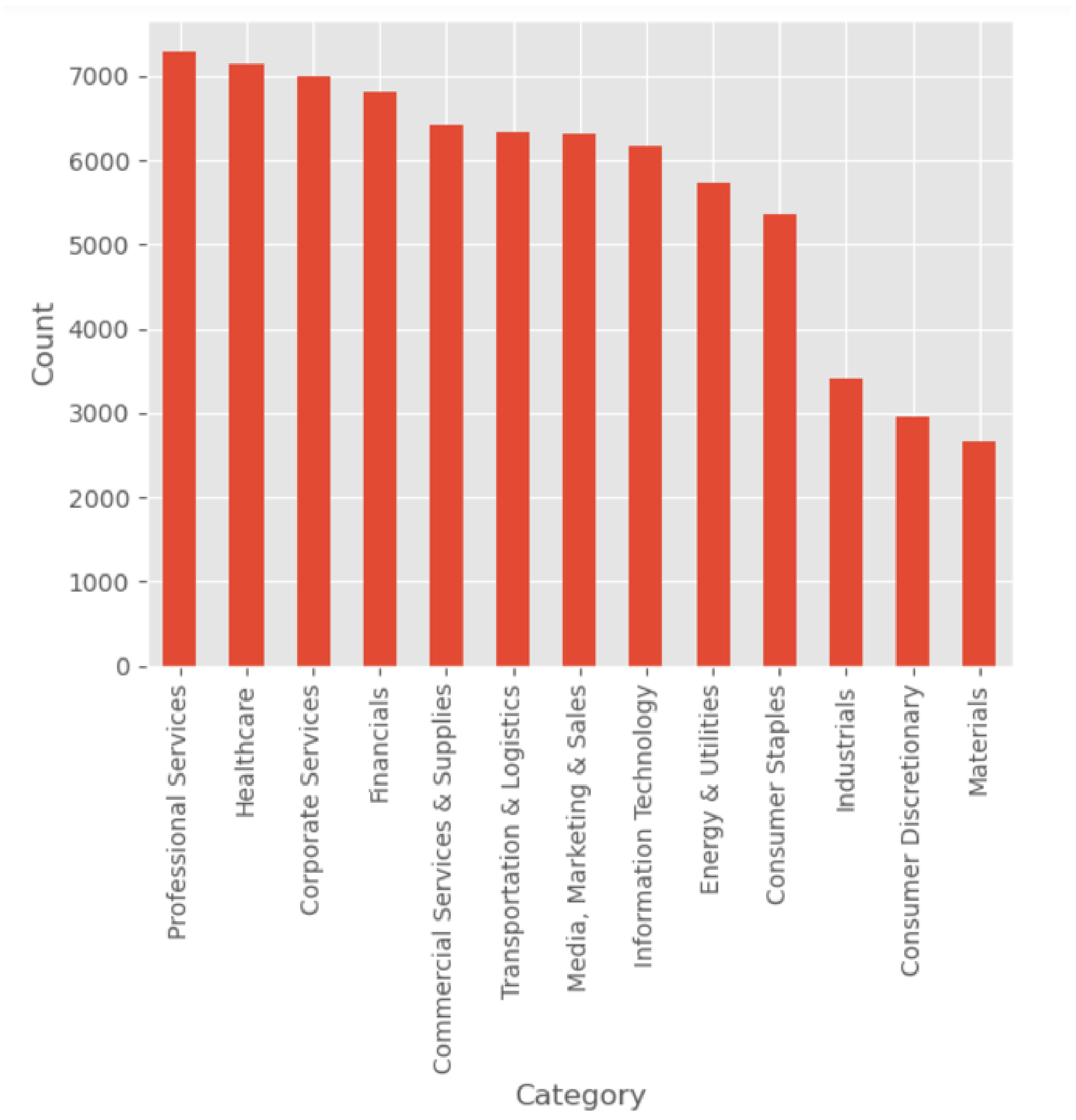
The distribution remains the same, as expected.

3.4 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.

3.4.1 Category

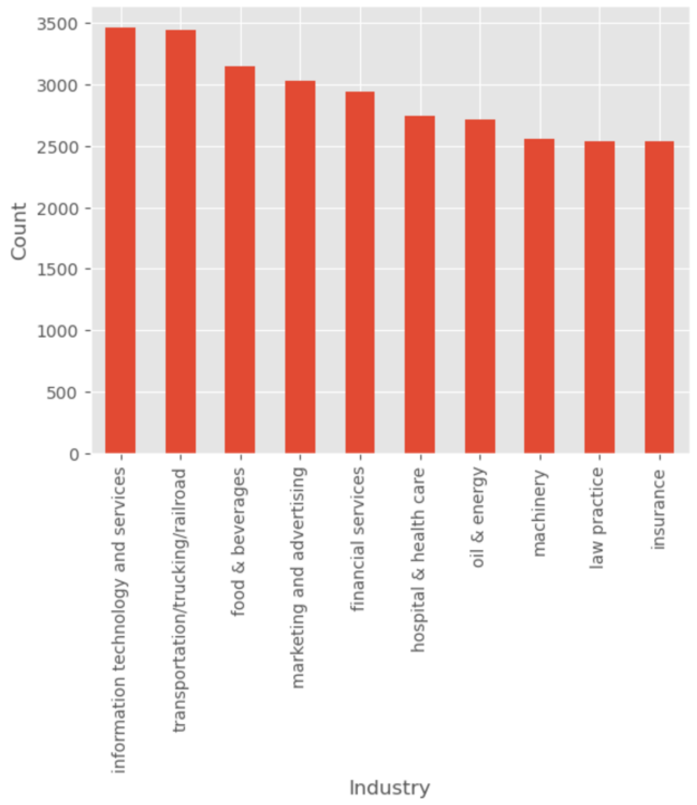
Let's have a basic understanding of the distribution of our target feature.



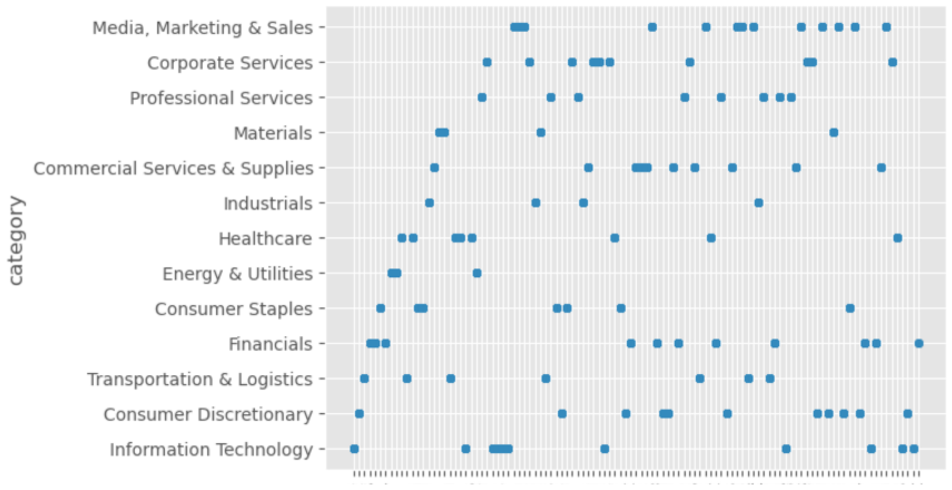
Here, we can observe the most common categories as well as the least frequent ones.

3.4.2 Industry

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

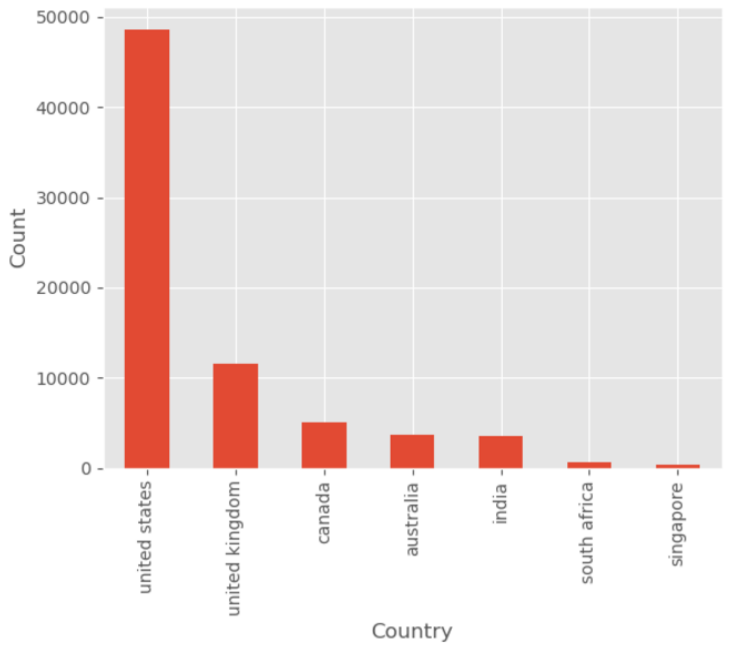


It is evident that there is a strong correlation between the two columns. Each category is associated with specific industries, which indicates that the industry column will be a crucial feature for our prediction model.

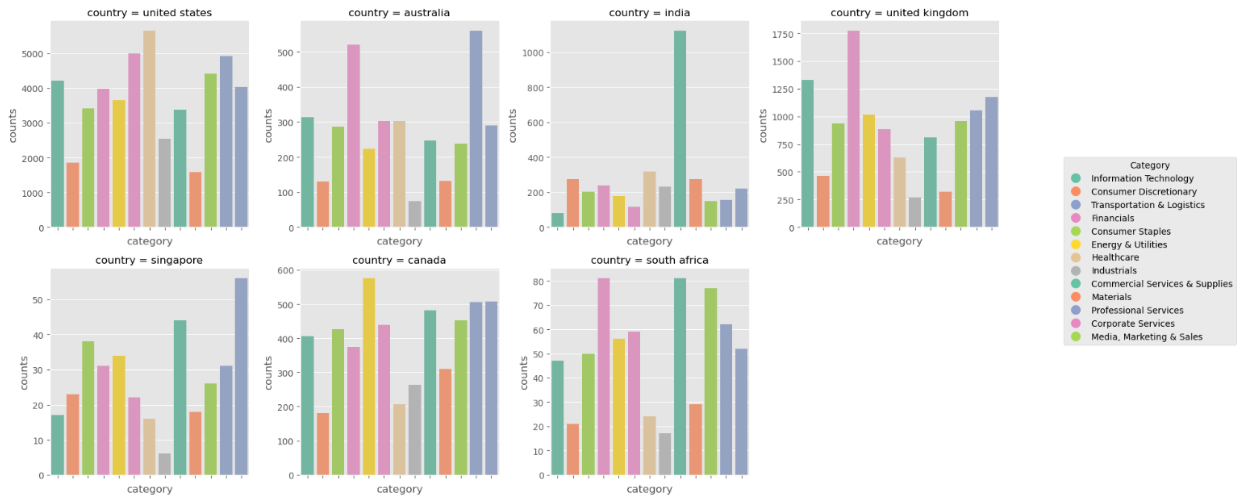


3.4.3 Country

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



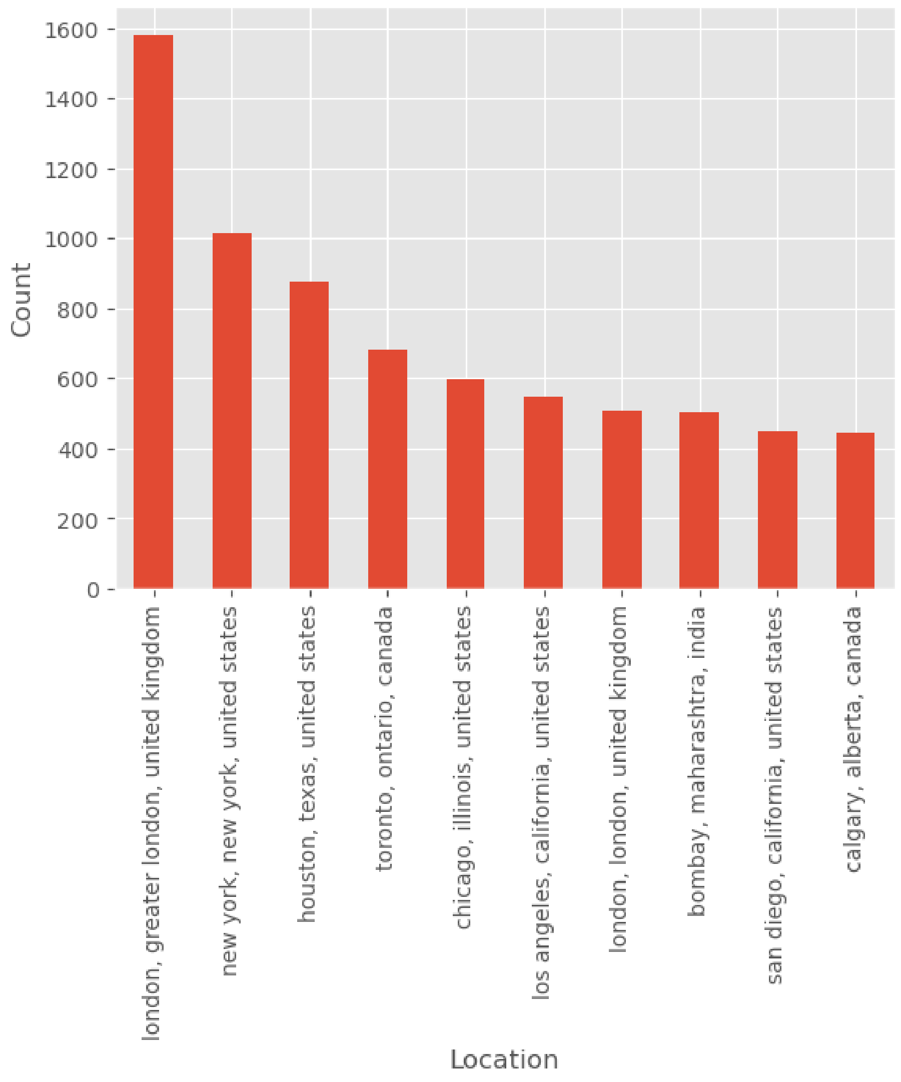
There also appears to be a strong correlation between each country and its respective category:



3.4.4 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.

3.5 Text columns

3.5.1 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

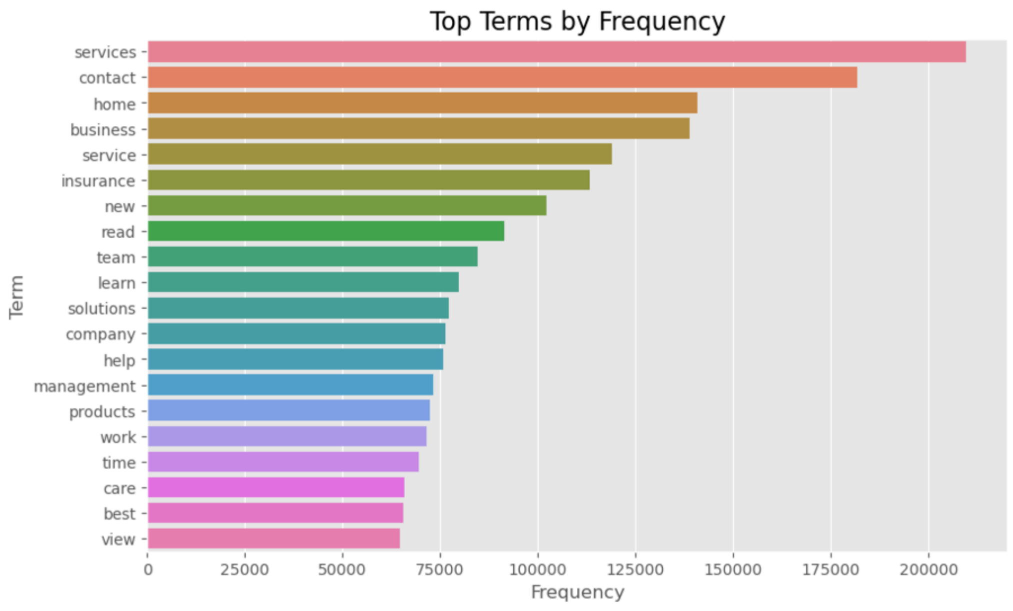
Columns like headings, nav_link_text, and meta_keywords seem to contain basic page data. Given their relatively low average word counts and high number of missing values, we have decided to combine them into a new column named homepage_keywords.

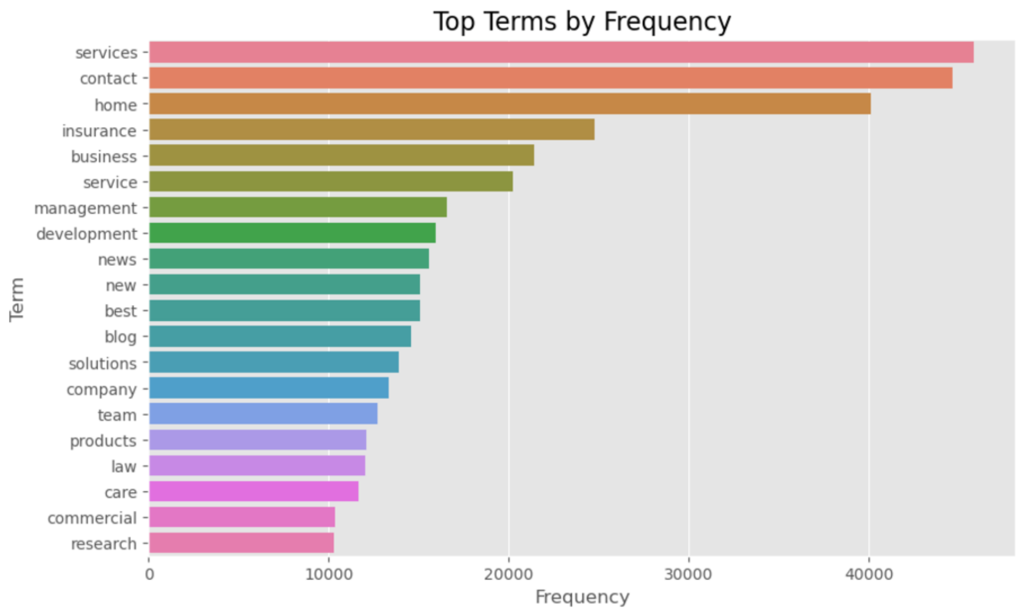
After this combination, we have three text columns. Of these, 350 rows have all columns as NaN, and 1,645 rows have at least two NaN values. Since text data plays a crucial role in our prediction model, we have decided to drop these 1,645 rows.

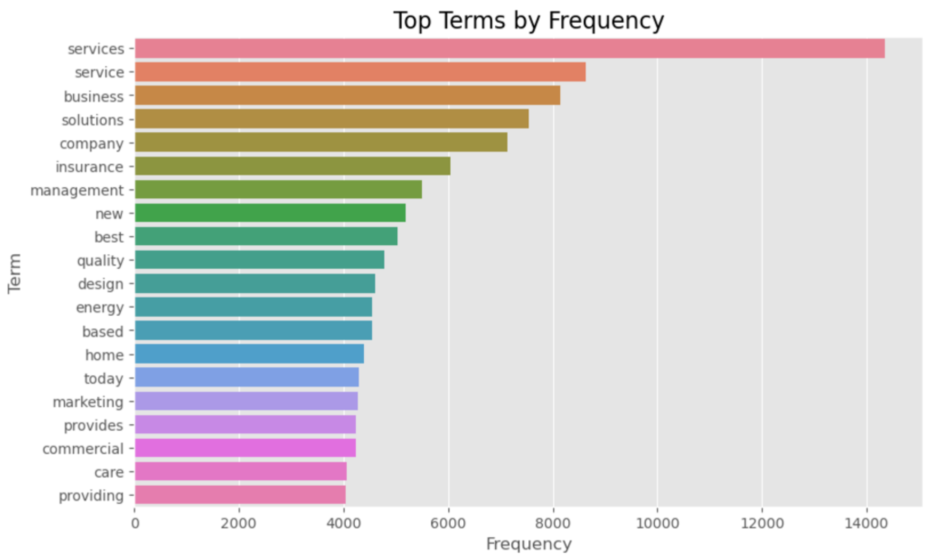
3.5.2 Text data insights

3.5.2.1 Top terms frequency

Below are the top terms by frequency for homepage_text, homepage_keywords, and meta_description, respectively.



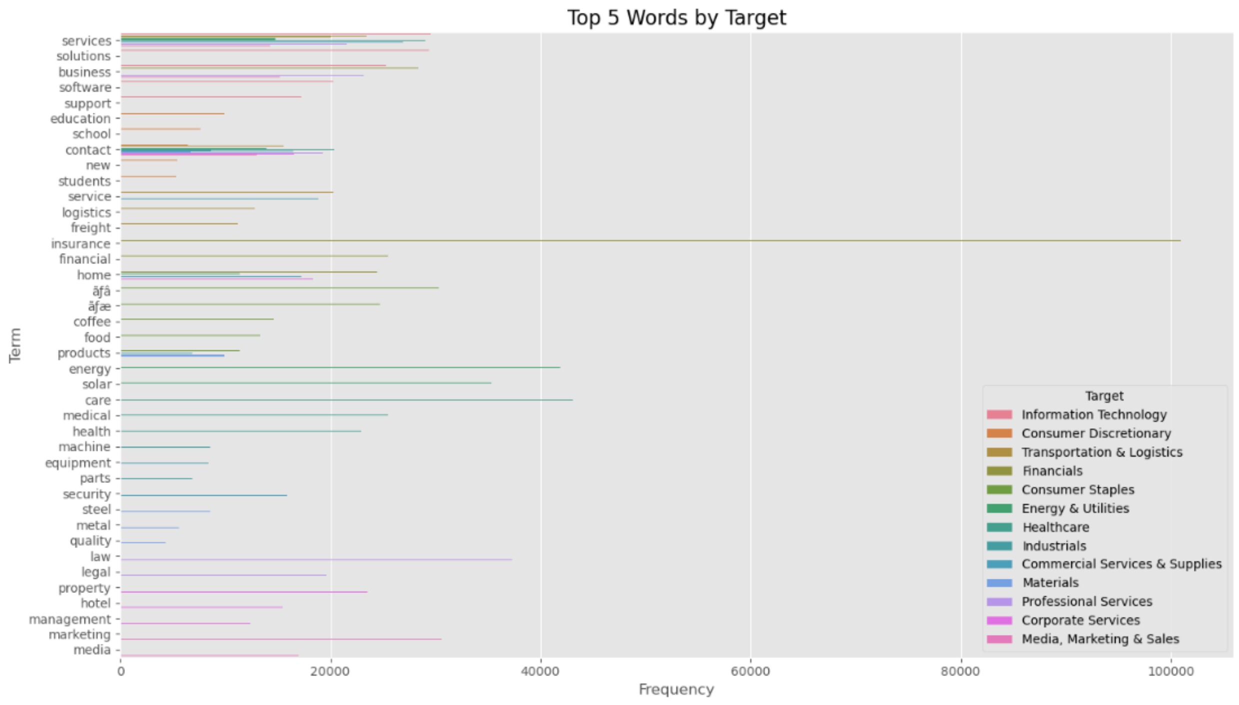


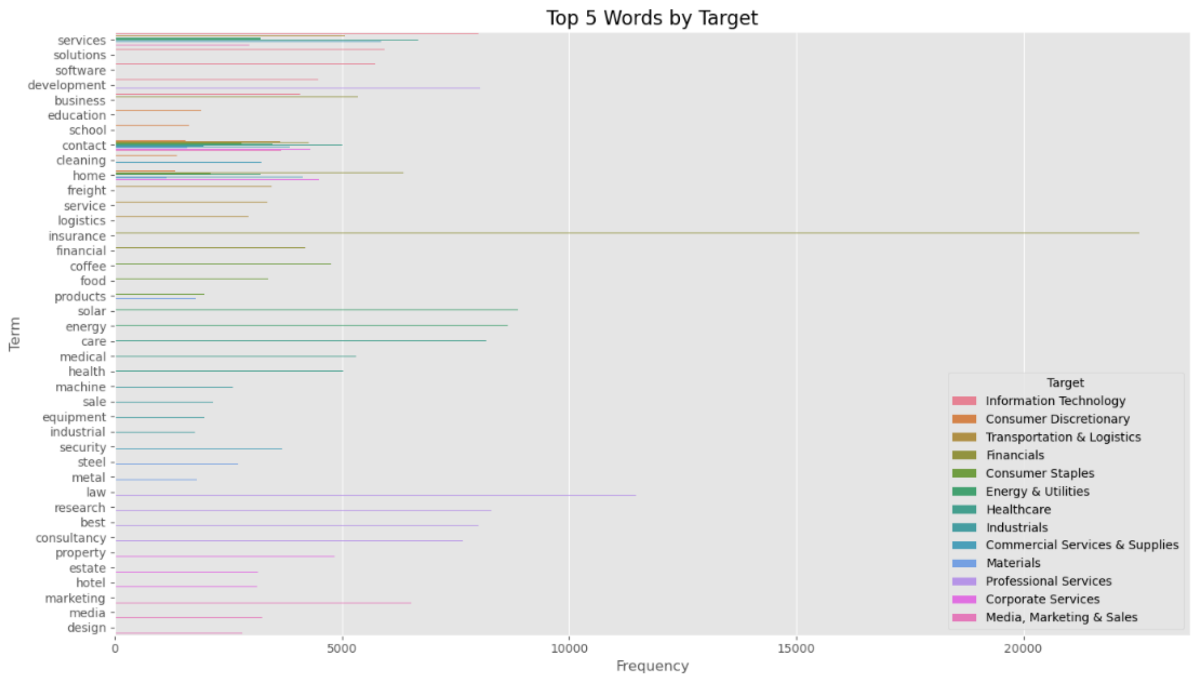


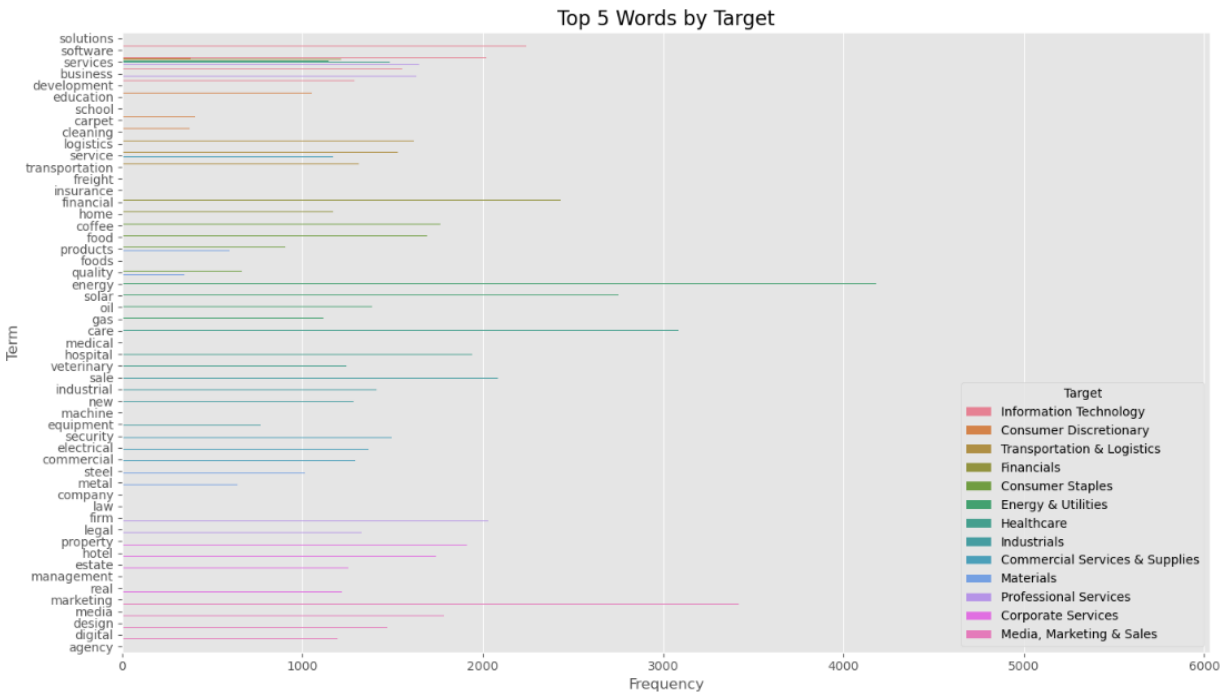
We can observe that while some top terms differ across the three features, there are also terms that appear frequently in all of them. Are these terms stop words, or do they occur often in certain categories and potentially contribute to the classification? To answer this, we will examine the top terms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

3.5.2.2 Top terms by category

Below are the top terms by frequency for each category in homepage_text, homepage_keywords, and meta_description, respectively.

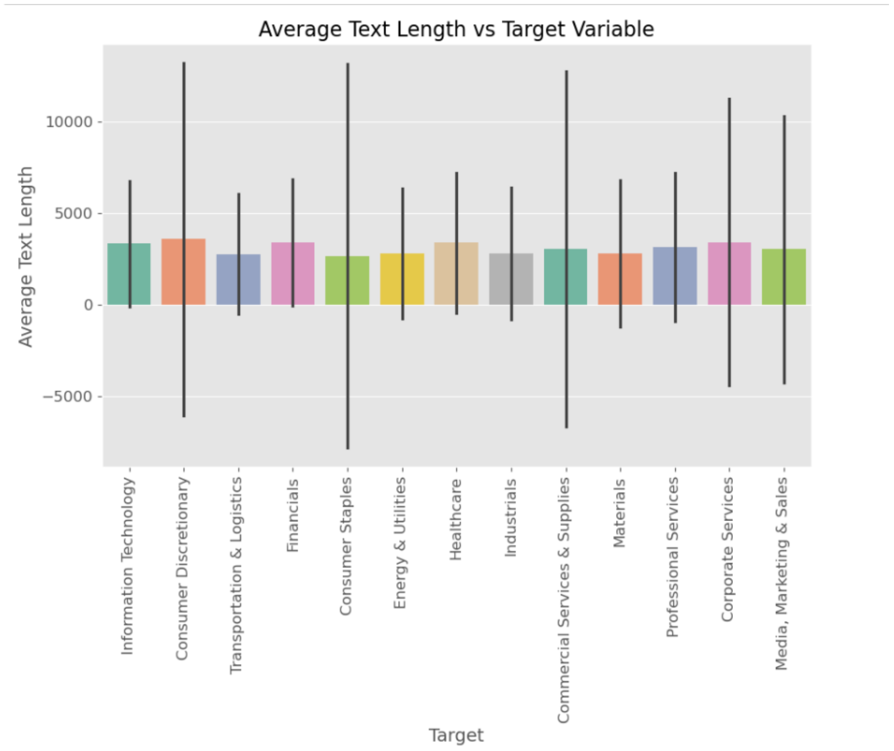


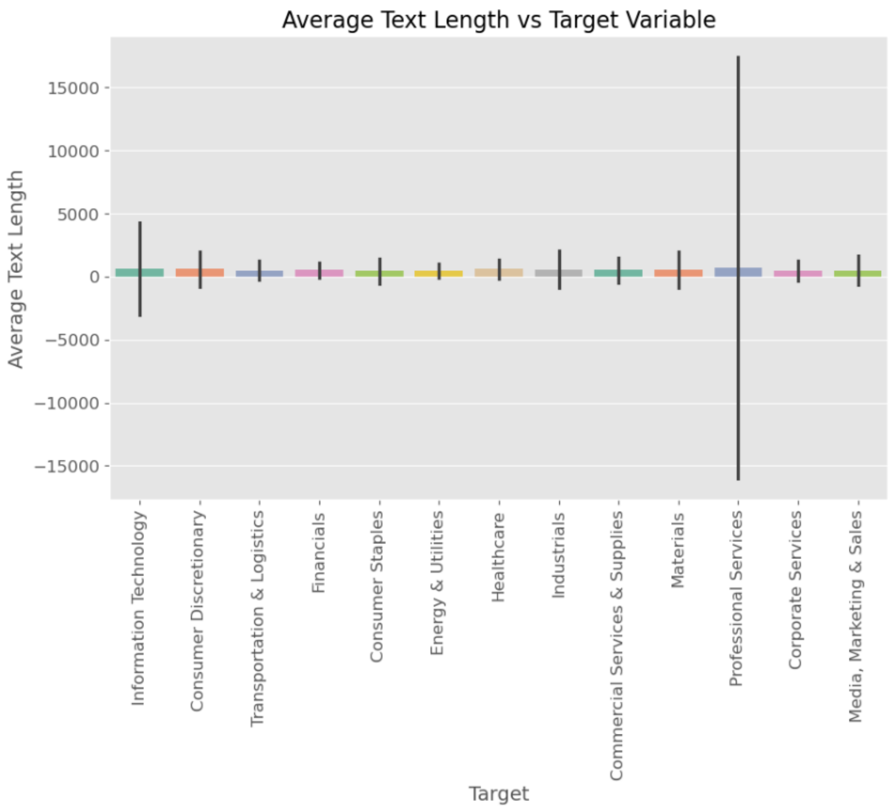


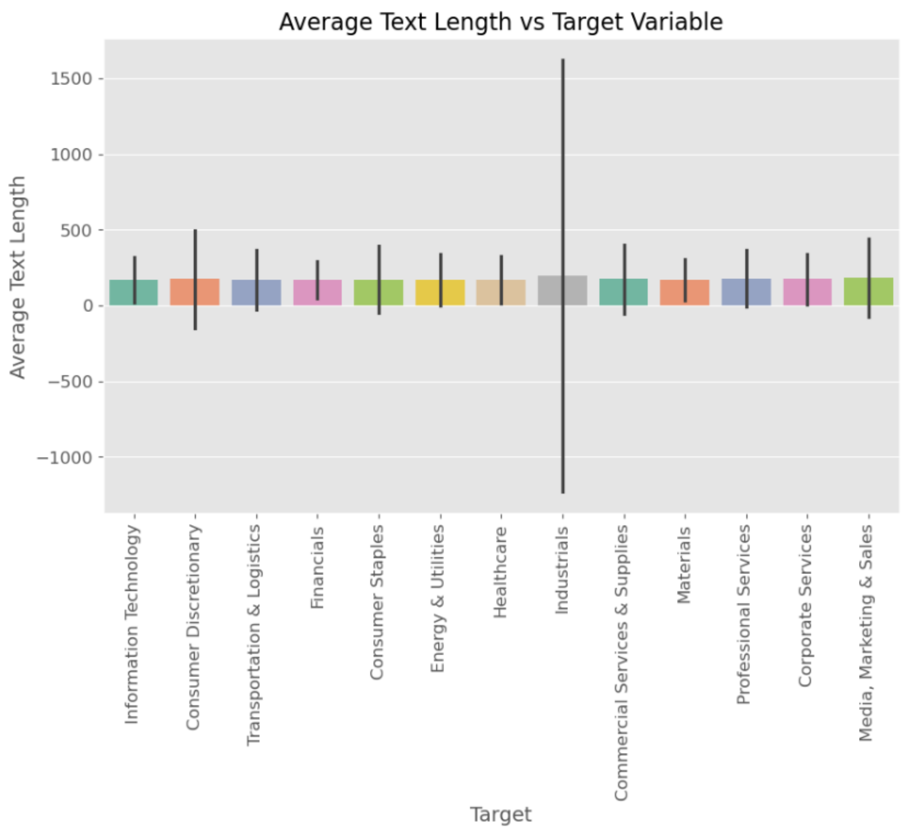


We can observe a clear correlation between the top terms used and the category. However, as mentioned in the previous section, words like "services," "home," and "contact" appear across all categories. Therefore, we will treat these as stop words and remove them from the data.

3.5.2.3 Text length by category





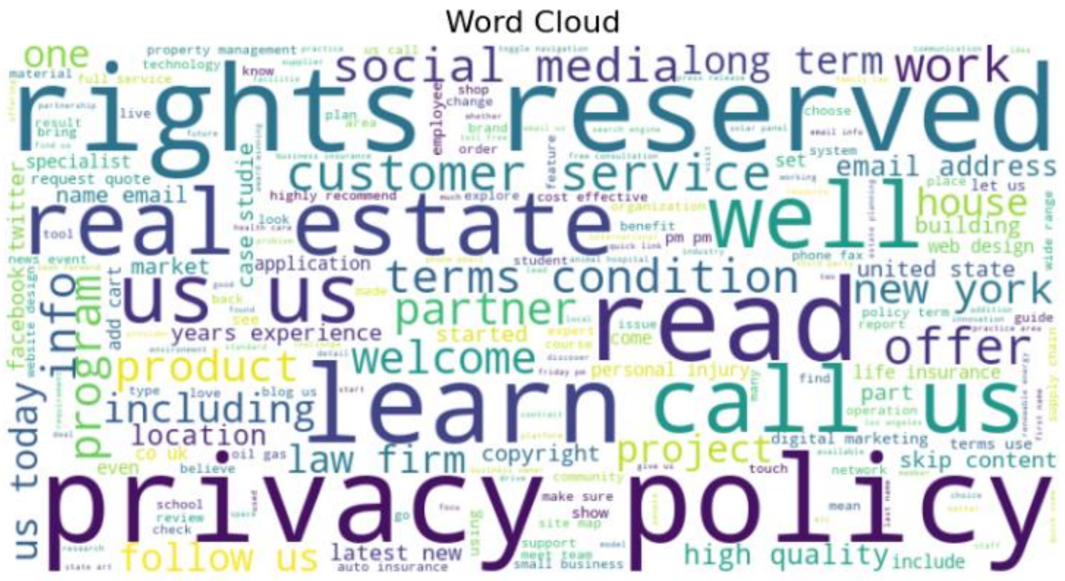


The text length isn't correlated with the category, but as shown above, the term frequency is. Therefore, using just a bag of words for embeddings may not be ideal. Instead, TF-IDF would be a much better choice.

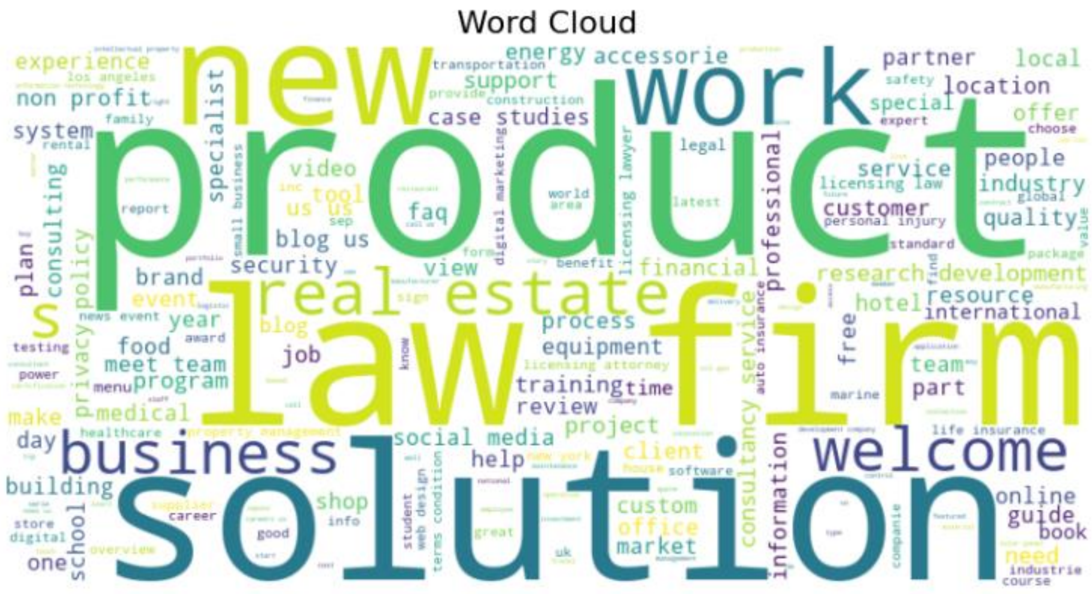
3.5.2.4 Word cloud

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

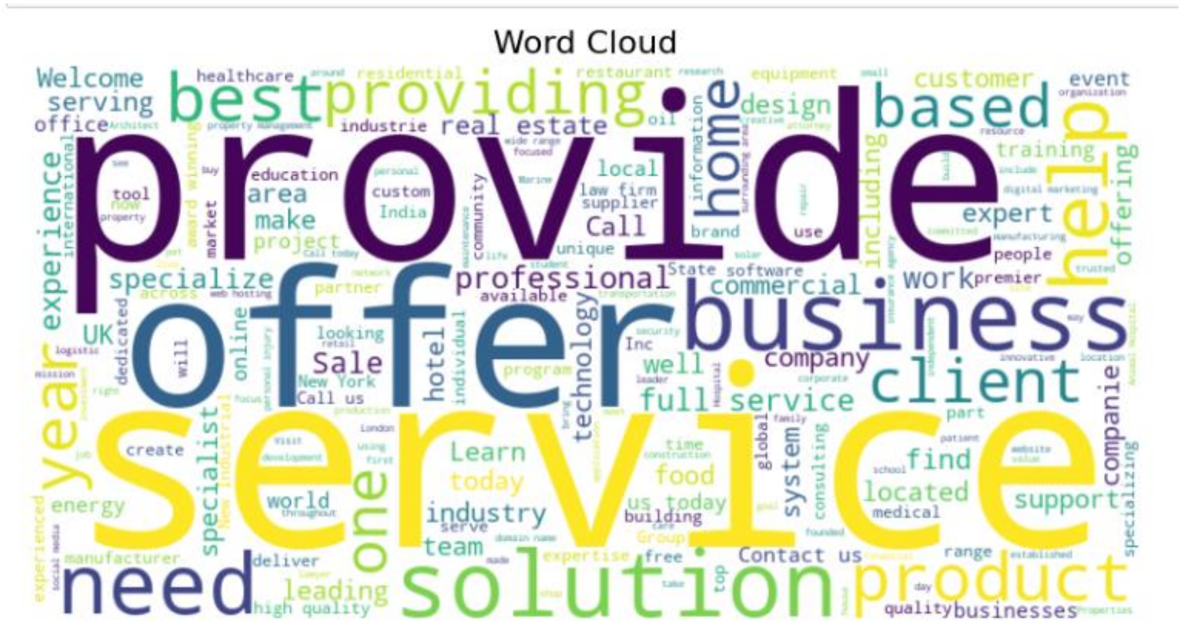
3.5.2.4.1 Homepage_text



3.5.2.4.2 Homepage_keywords



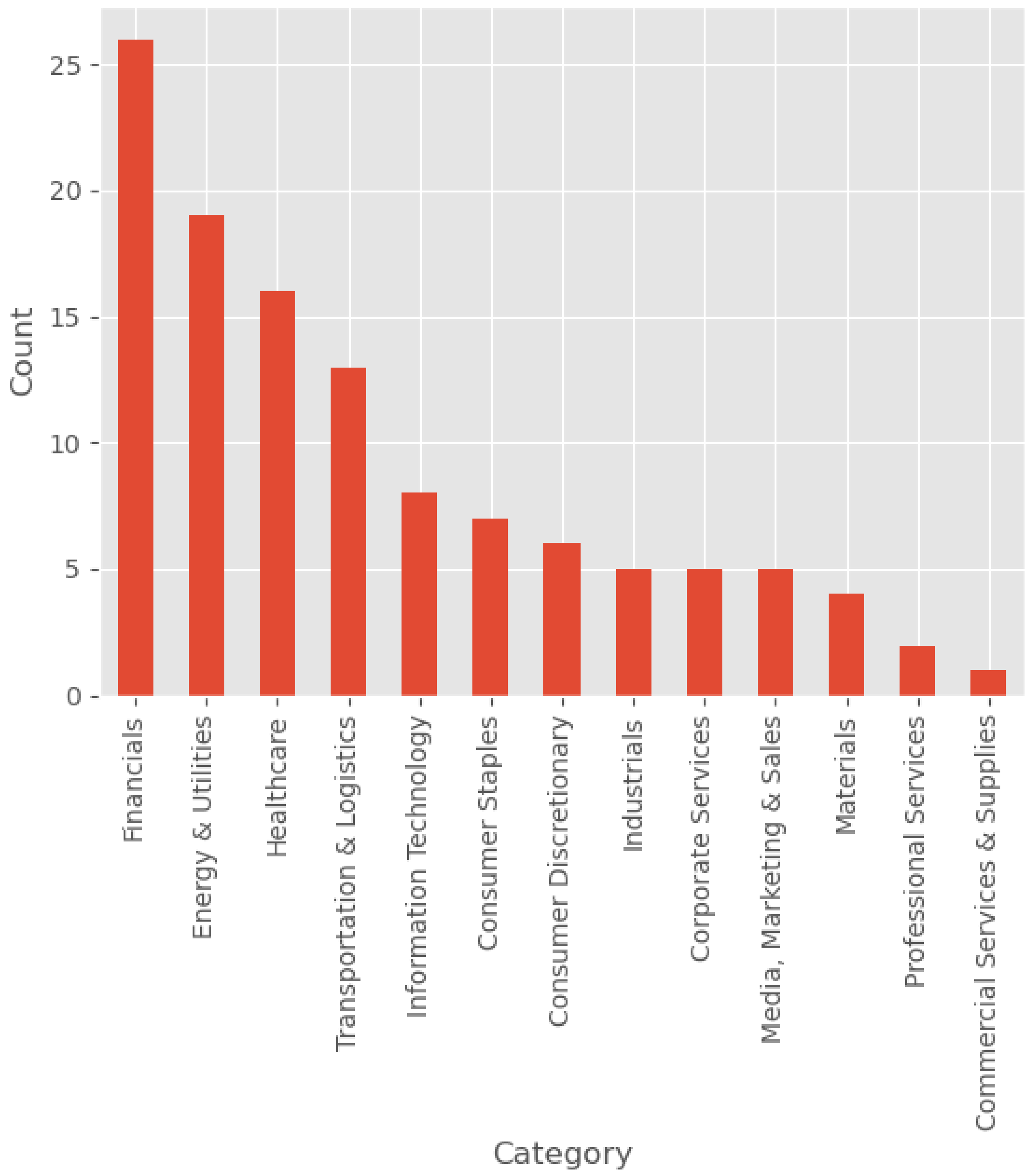
3.5.2.4.3 Meta_description



3.6 Questions about the data

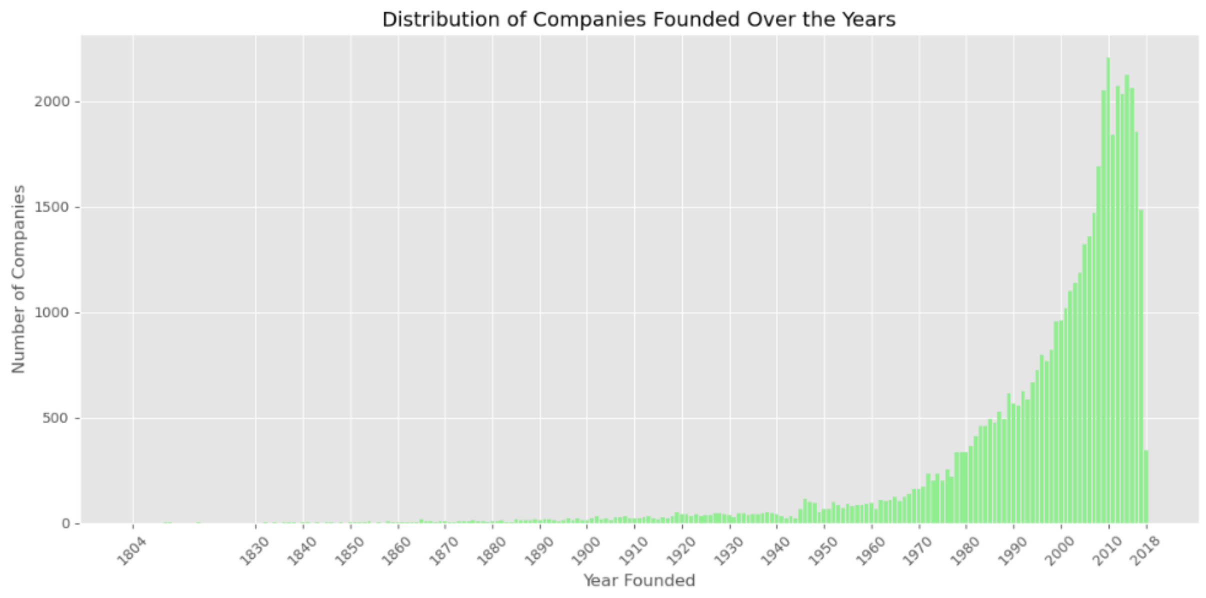
In this section, we explore some interesting questions about the data that help us better understand the dataset.

3.6.1 What categories are the most employed companies?



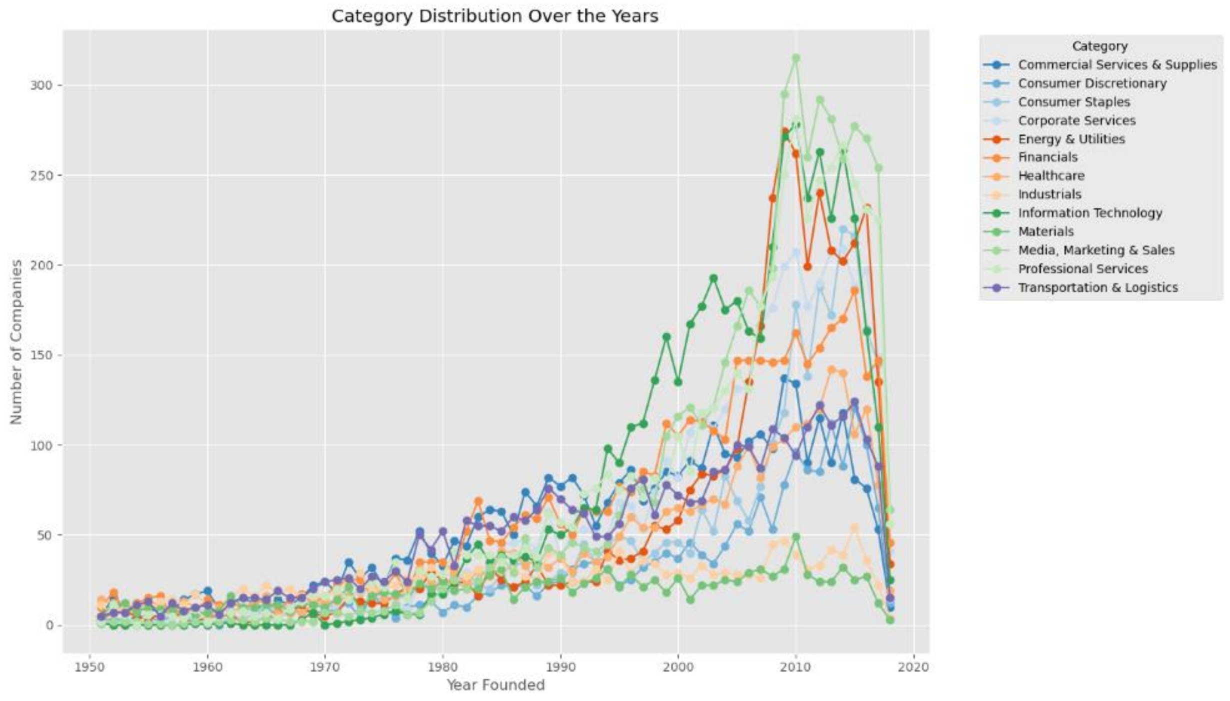
By considering companies with a total employee estimate greater than 5000, we can identify the categories of companies with the most employees. As shown in the figure, the Financials category has the largest companies based on employee count.

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



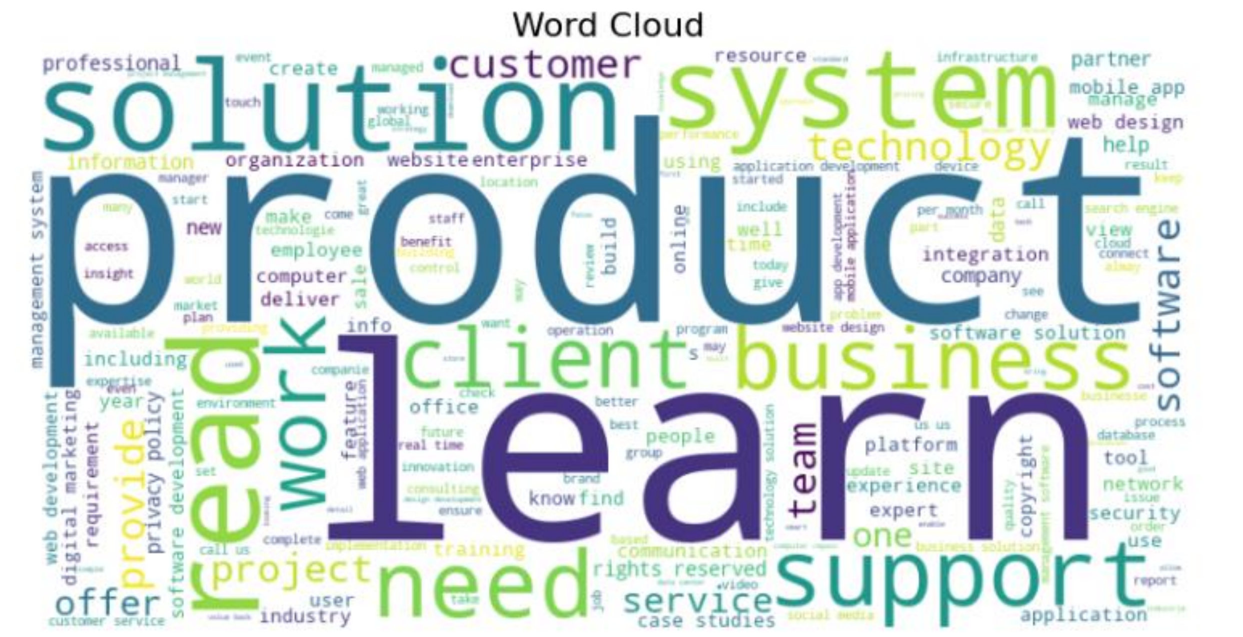
We can observe a steady increase in the number of companies founded over the years, with a peak around 2010. After that, the number of new companies tends to decline.

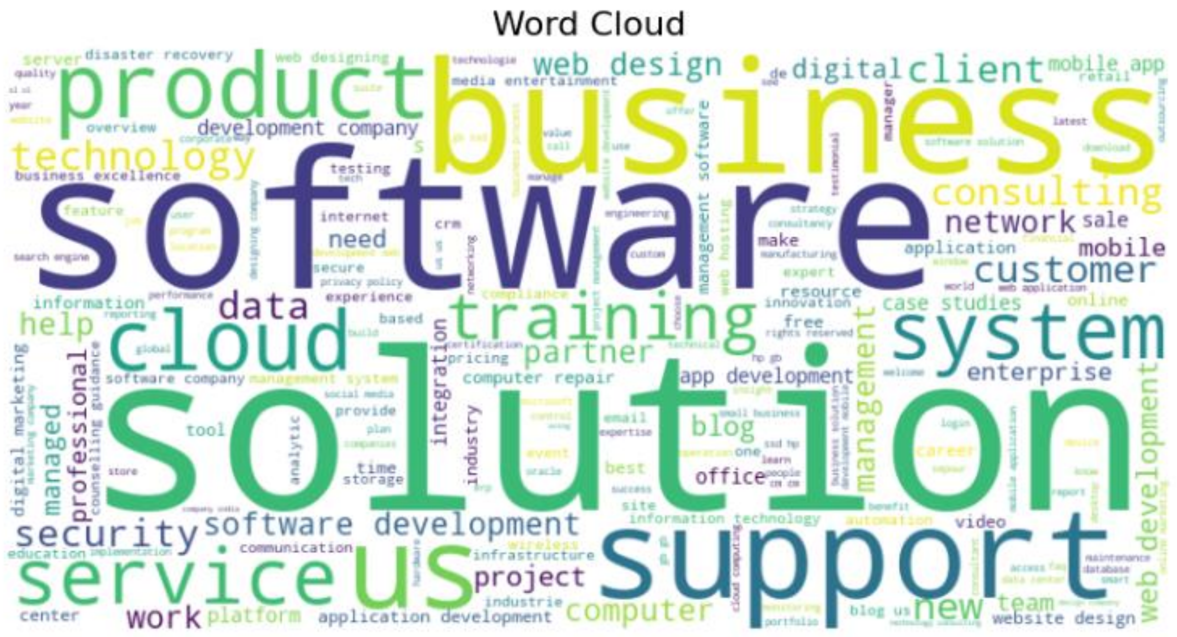
3.6.3 How have the distributions of different company categories changed over time?

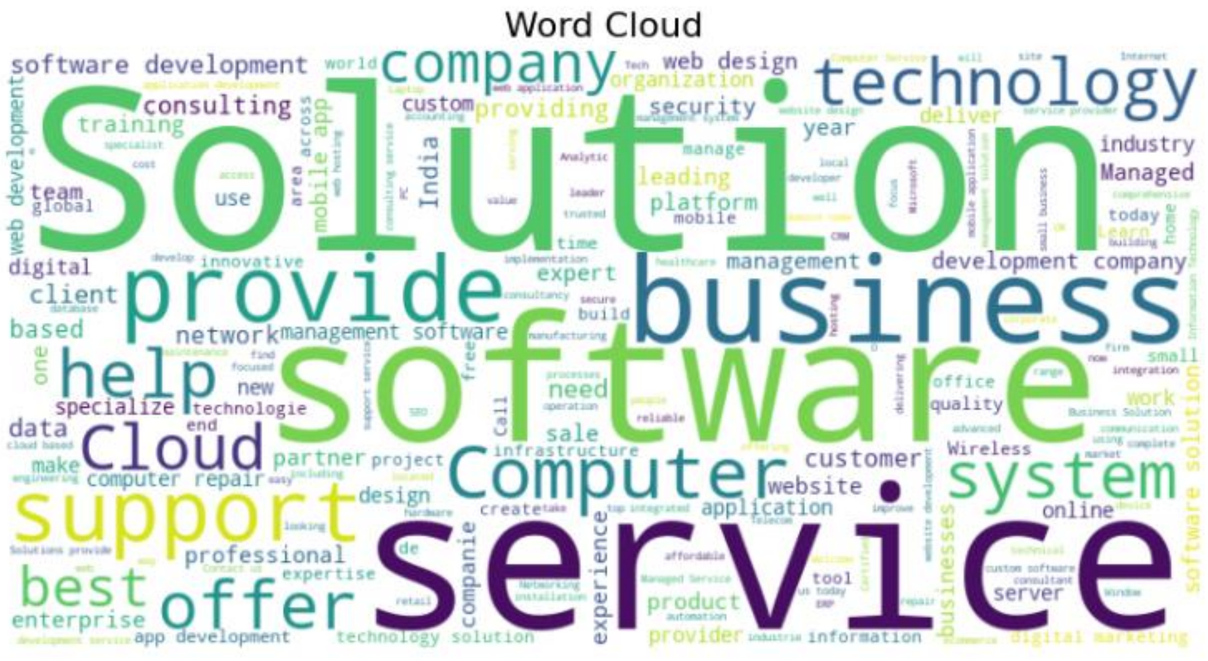


For clarity, we focus on companies founded after 1950. We can observe that categories previously at the bottom, such as Professional Services, Energy & Utilities, and Information Technology, have experienced significant growth in recent years compared to the past.

3.6.4 What's the top word in Information Technology category?







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

4 Model development

4.1 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:

4.1.1 Categorical Feature Encoding

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

4.1.1.1 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]])
```

4.1.1.2 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)
```

4.1.2 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']])
```

4.1.3 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**.

4.1.3.1 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.

4.1.3.2 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:

```python
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.

4.1.3.3 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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.

4.6 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.

4.7 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.

5 Model evaluation

5.1 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.

5.2 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.

5.3 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**.

```python
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}")
```

5.4 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**.

5.5 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

6 Next Steps

6.1 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.

6.2 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.

6.3 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.

6.4 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)
```

```python
# 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()
```