



**Cracow University of Technology**

**Department of Computer Sciences**



**Natural Language Processing – Erasmus**

**Ac. Year 2023/2024**

**Project title**

**Sarcasm Detector**

***Team's group***

Kenza Bennani

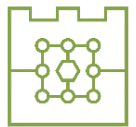
Deborah Renard



**Cracow University of Technology**

**Department of Computer Sciences**





# Contents

|  |    |
|--|----|
| Contents .....                                   | 3  |
| 1. Abstract .....                                | 4  |
| 2. Introduction.....                             | 5  |
| a. Aim .....                                     | 5  |
| b. Scope .....                                   | 5  |
| c. Methodology.....                              | 5  |
| 3. Theoretical part.....                         | 7  |
| a. Supervised learning algorithms.....           | 7  |
| Logistic regression .....                        | 7  |
| Gradient Boosting .....                          | 7  |
| b. Neural network models.....                    | 9  |
| LSTM (Long Short-Term Memory).....               | 9  |
| GRU (Gated Recurrent Unit) .....                 | 9  |
| c. Transformer-based models.....                 | 10 |
| RoBERTa (Robustly optimized BERT approach) ..... | 10 |
| 4. Practical Part .....                          | 11 |
| a. Data collection .....                         | 11 |
| b. Data Preprocessing.....                       | 12 |
| c. Model Implementation .....                    | 22 |
| d. Model Evaluation .....                        | 28 |
| 5. Summary .....                                 | 32 |
| 6. Bibliography.....                             | 33 |

## 1. Abstract

Detecting sarcasm in text is a complex and nuanced challenge in natural language processing, crucial for improving sentiment analysis and enhancing human-computer interaction. In this project we have used several machine learning and deep-learning models for sarcasm detection, including supervised learning algorithms (logistic regression and gradient boosting), neural network models (Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)), and transformer-based models (RoBERTa from Hugging Face).

First, we introduce the aim, scope, and methodology of our project. Next, we provide a theoretical overview of the principles and advantages of each model type, explaining their pertinency to sarcasm detection.

In the practical part, we document the data sourcing from Kaggle, followed by extensive preprocessing and exploratory data analysis (EDA). We describe how the data was prepared for modeling. Right after we detail the implementation phase, showcasing code snippets and graphical analysis used in our approach.

Finally, we present a comparative analysis of the model's performance, and we discuss the most effective model for sarcasm detection.

Our project concludes with a summary discussing whether the project goals were achieved, reflecting on the results, and offering subjective thoughts on the process and potential future directions. Through this study, we demonstrate the potential of advanced NLP techniques in improving the detection and understanding of sarcastic content in text.

## 2. Introduction

### a. Aim

The primary aim of this project is to develop and evaluate various machine learning and deep-learning models for detecting sarcasm in text. Achieving the goal to identify sarcastic expression, we can improve the performance of systems involved in sentiment analysis, social media monitoring, and human-computer interaction.

### b. Scope

In this project we have covered several key areas:

- Data collection: Utilizing a well-known dataset for sarcasm detection sourced from Kaggle. [1]
- Data Preprocessing: Loading data and performing exploratory data analysis (EDA). Implementing steps to clean and prepare the data for analysis.
- Model Implementation: Developing and training 3 different types of models for sarcasm detection.
  - Supervised learning algorithms: logistic regression and gradient boosting.
  - Neural network models: Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).
  - Transformer-based models: RoBERTa from Hugging Face.
- Model Evaluation: Providing a detailed analysis of the results, discussing the strengths and limitations of each model. Comparing the performance of these models to determine which is most effective in detecting sarcasm.

### c. Methodology

The methodology for this project involves several steps and tools, which were implemented using Python in Jupyter Notebook and Google colab environments. Let's detail the process:

- Data sourcing and Preparation
  - Dataset: We used a well-suited dataset from Kaggle.
  - Libraries and tools: For manipulating data we used Pandas library and for visualizing data we used Seaborn and WordCloud libraries
  - We used NLTK for NLP tasks, including stopwords removal and tokenization



- We used TfidfVectorizer from Scikit-learn for feature extraction.

→ Model development and Training

- Supervised Learning: We implemented logistic Regression and gradient boosting model using Scikit-learn, with a pipeline for TF-IDF vectorization and model training.
- Neural Network: We have build LSTM and GRU Models using TensorFlow and Keras. These models were structured with an embedding layer, LSTM or GRU layers, dropout for regularization, and dense layers for output.
- Transformer-based Models: Leveraging Hugging Face's Transformers library, we used the AutoTokenizer and AutoModelForSequenceClassification for implementing the RoBERTa model.
- Metrics such as accuracy, precision, recall, F1-score, and confusion matrix were used to evaluate model performance. To ensure model robustness and prevent overfitting we employed cross\_val\_score from Scikit-learn.

This thorough approach, using various Python libraries and tools, enabled us to effectively test and compare different models for sarcasm detection.

## 3. Theoretical part

In our project, we employed three different types of machine learning models to achieve our objectives.

### a. Supervised learning algorithms

#### *Logistic regression*

**Logistic regression** is a supervised machine learning algorithm utilized for binary classification tasks, predicting the probability of an outcome, event, or observation.[3]

##### **Advantages for Sarcasm Detection:**

- **Simplicity and Interpretability:** Logistic regression is straightforward to implement and interpret, making it easy to understand the relationship between features (e.g., text features like word counts or sentiment scores) and the prediction of sarcasm.
- **Low Computational Cost:** This method requires less computational power compared to more complex models, allowing for faster training and evaluation.
- **Baseline Performance:** While simple, logistic regression can serve as a strong baseline to compare more advanced models against, providing insights into the effectiveness of feature engineering.

#### *Gradient Boosting*

**Gradient boosting** machines are a versatile and powerful family of machine learning techniques that excel in a wide range of practical applications. Their ability to be tailored to specific loss functions and incorporate regularization mechanisms ensures high predictive accuracy and robustness. [2]

##### **Advantages for Sarcasm Detection:**

- **High Predictive Accuracy:** Gradient Boosting often achieves high accuracy by combining the strengths of multiple weak learners (decision trees) into a strong predictive model. This is particularly beneficial for capturing the complex patterns associated with sarcasm.



- **Handling Non-Linearity:** Sarcasm often involves non-linear relationships between words and phrases. Gradient Boosting can model these complex interactions more effectively than linear models like logistic regression.
- **Feature Importance:** It provides insights into the importance of different features used in the model. This can help in understanding which words or phrases are most indicative of sarcasm.
- **Robustness to Overfitting:** Gradient Boosting includes regularization techniques (such as limiting tree depth and learning rate) to prevent overfitting, which is crucial for handling the variability in sarcastic expressions.
- **Flexibility with Different Types of Data:** Gradient Boosting can handle various types of input features, including numerical, categorical, and text features transformed through techniques like TF-IDF or word embeddings, making it versatile for sarcasm detection tasks.



## b. Neural network models

### *LSTM (Long Short-Term Memory)*

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) designed to model temporal sequences and their long-range dependencies more accurately than conventional RNNs.[4]

#### **Advantages for Sarcasm Detection:**

- **Handling Long-Term Dependencies:** LSTMs can capture long-term dependencies and context in text, which is crucial for understanding the nuanced patterns in sarcastic remarks.
- **Sequential Data Processing:** They are well-suited for processing sequences of words or sentences, maintaining context over longer text sequences.
- **Flexibility with Various Sequence Lengths:** LSTMs can handle varying lengths of text input, making them versatile for different sarcastic expressions.

### *GRU (Gated Recurrent Unit)*

Gated Recurrent Unit (GRU) is another type of recurrent neural network that is similar to LSTM but with a simpler architecture.[5]

#### **Advantages for Sarcasm Detection:**

- **Efficiency:** GRUs are computationally more efficient and faster to train than LSTMs, while still effectively capturing dependencies in sequential data.
- **Effective Sequence Modeling:** Despite their simpler structure, GRUs are capable of handling sequence data well, making them suitable for detecting sarcasm in context where temporal dependencies matter.
- **Reduced Complexity:** The simpler architecture of GRUs makes them less prone to overfitting and easier to tune.

## c. Transformer-based models

### *RoBERTa (Robustly optimized BERT approach)*

RoBERTa is a transformer-based model developed by Facebook AI and available through Hugging Face. It is an optimized version of BERT (Bidirectional Encoder Representations from Transformers) that enhances performance through extensive pre-training and fine-tuning.[6]

#### **Advantages for Sarcasm Detection:**

- **Contextual Understanding:** RoBERTa excels at understanding context and nuances in text, which is critical for accurately detecting sarcasm, often conveyed through subtle linguistic cues.
- **Pre-trained on Extensive Data:** Being pre-trained on a large corpus of text data, RoBERTa has a strong grasp of language patterns and can be fine-tuned for specific sarcasm detection tasks, improving its performance.
- **State-of-the-Art Performance:** RoBERTa often achieves state-of-the-art results in various NLP tasks, including sarcasm detection, by leveraging its deep and robust language understanding capabilities.
- **Scalability and Robustness:** The model scales well with more data and larger architectures, making it highly effective for complex sarcasm detection tasks that require nuanced interpretation.

## 4. Practical Part

### a. Data collection

For our sarcasm detection project, we chose to use a well-known and widely utilized dataset from Kaggle. Kaggle is a popular platform for data science competitions and hosting datasets. [1]

The dataset, published by Dan Ofer in 2018, contains 1.3 million sarcastic statements and is balanced.

The sarcasm detection dataset we used contains diverse and context-rich examples of sarcasm, which is crucial for training models capable of understanding and detecting sarcasm in text.

We chose to use a CSV (Comma-Separated Values) file to store and manipulate the data. The CSV format has several advantages: simplicity, readability, compatibility and ease of Manipulation.

## b. Data Preprocessing

### → Importing librairies

To begin our sarcasm detection project, we first loaded and analyzed the dataset to understand its structure and content. We started by importing the necessary libraries for data manipulation, visualization, and machine learning.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from wordcloud import WordCloud
from collections import Counter
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

from os import path
sns.set()
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score, precision_score, f1_score
import datetime as dt
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
import calendar
from sklearn.metrics.pairwise import cosine_similarity
%matplotlib inline
import time
```

### → Loading data and dropping the null comments

We loaded the dataset and then we removed any rows where the 'comment' field is null to ensure that we only work with complete data.

```
#dropping the null comments
data.dropna(subset=['comment'],inplace=True)
```

### → Clean text

We defined a function to clean the text by removing digits, punctuation, and converting the text to lowercase. Removing digits and punctuation helps in reducing noise in the data, which can improve the performance of natural language processing (NLP) tasks.

Then we converted the timestamp into a DateTime object for easier manipulation. The purpose of this code is to preprocess and clean the text data in the DataFrame, making it more suitable for further analysis or modeling.

```
# Clean text
def clean_text(comment):
    #removing digits
    comment = re.sub(r'\d+', '', comment)
    #removing punctuations
    comment = re.sub(r'[^\w\s]', '', comment)
    #converting to lowercase
    comment = comment.lower()
    return comment

data['cleaned_text'] = data['comment'].apply(clean_text)
```

### → Converting the timestamp into DateTime object

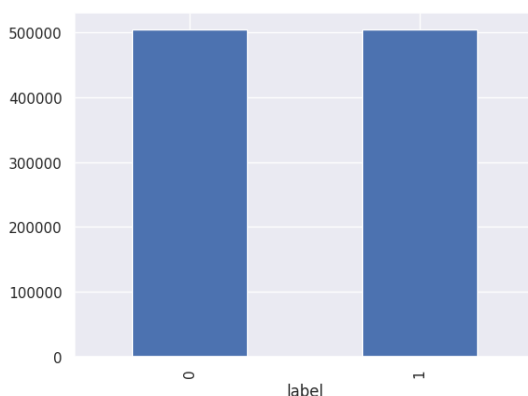
Converting the created\_utc column to DateTime objects ensures that date and time data is in a format suitable for complex temporal operations, which is crucial for accurate and efficient analysis involving time-based data.

```
# Converting the timestamp into DateTime object
data.created_utc = pd.to_datetime(data.created_utc)
```

.info() and .describe() methods are used to get a summary of the dataset.

### → Dataset Class Distribution

Then we explored the data. The figure below demonstrates that the dataset is balanced, with an equal proportion of sarcastic and non-sarcastic comments. Specifically, both categories represent 50% of the total dataset. This balance is crucial for ensuring unbiased analysis and accurate model training, as it prevents any one class from disproportionately influencing the results.

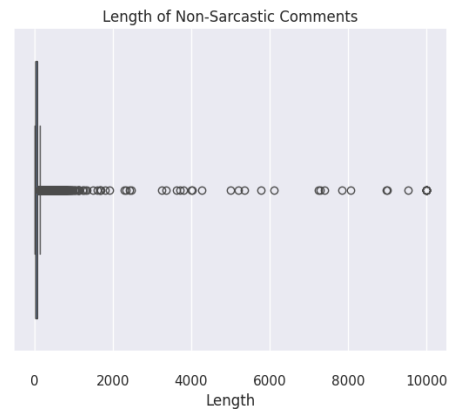
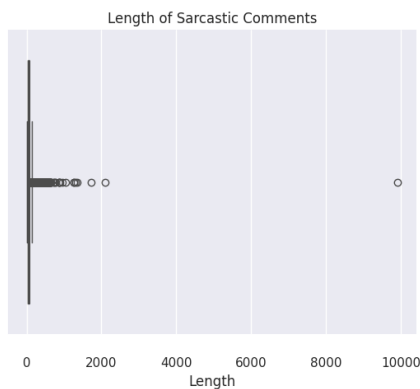


Then we remove stop words from the cleaned text to focus on the more meaningful words.

```
nlTK.download('stopwords')
# Remove stop words
stop_words = set(stopwords.words('english'))
data['cleaned_text'] = data['cleaned_text'].apply(lambda x: ' '.join([word for word in x.split() if word not in stop_words]))
```

→ Relation between the length of the comment and the comment being sarcastic

The figures below show that non-sarcastic comments tend to be shorter on average compared to sarcastic comments. The differing distributions of comment lengths between sarcastic and non-sarcastic comments may reflect distinct patterns in language usage and communication styles within the dataset.



→ The Word Cloud

To further understand the differences between sarcastic and non-sarcastic comments, we can visualize the most common words used in each category by generating word clouds





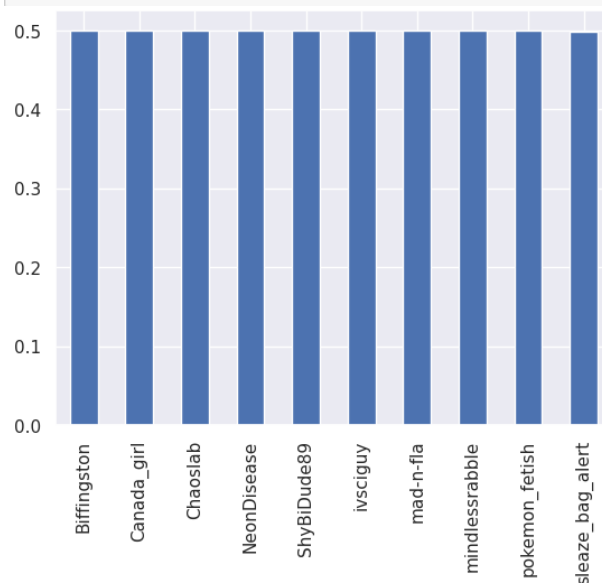
### Non-Sarcastic Comments



### → Analyzing Top Authors and Their Sarcasm Levels

Then we wanted to identify the top 10 authors who have contributed the most comments in the dataset. We accomplished this by counting the occurrences of each author's name in the 'author' column and selecting the top 10 based on frequency. Subsequently, we computed the average sarcasm level for each of these top authors by filtering the dataset to include only comments from these authors and then calculating the mean label value for each author group.

```
top_authors = data['author'].value_counts().head(10)
top_authors_sarcasm = data[data['author'].isin(top_authors.index)].groupby('author')['label'].mean()
top_authors_sarcasm.plot(kind='bar')
```



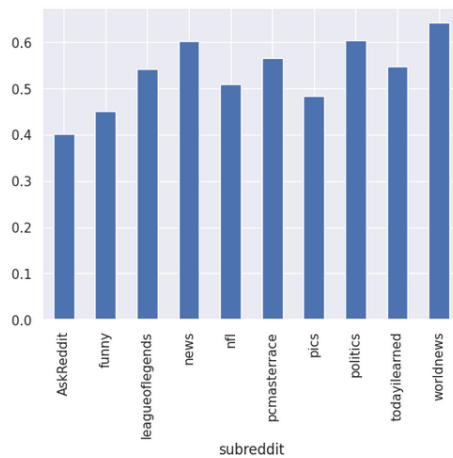
### → Examining Top Subreddits and Sarcasm Levels

Finally, we can visualize the sarcasm levels of these top authors using a bar plot. This visualization provides valuable insights into which authors tend to exhibit higher or lower levels of sarcasm in their comments, thus offering a glimpse into the sarcasm distribution among the most active contributors in the dataset.



```
top_subreddits = data['subreddit'].value_counts().head(10)
top_subreddits_sarcasm = data[data['subreddit'].isin(top_subreddits.index)].groupby('subreddit')['label'].mean()
top_subreddits_sarcasm.plot(kind='bar')
```

<Axes: xlabel='subreddit'>

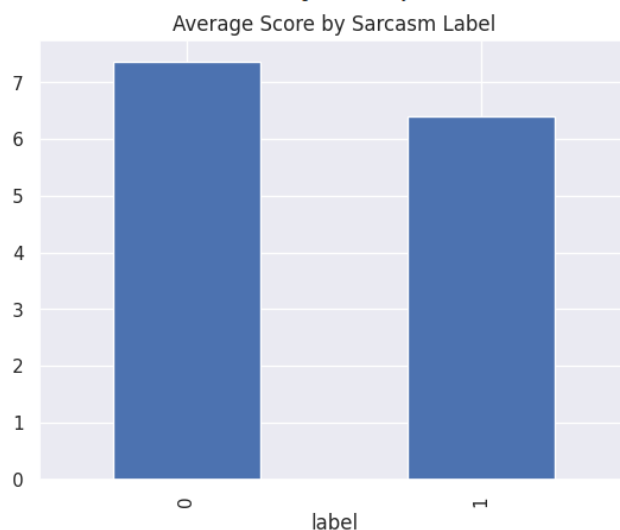


### → Analyzing Average Score by Sarcasm Label

According to the dataset, sarcastic comments tend to be less popular due to having lower overall scores. We have grouped the dataset by the sarcasm label ('label' column), and the mean score for each group is calculated. The 'score' column likely represents a numerical value associated with the engagement or popularity of each comment, such as the number of upvotes or likes it received.

```
data.groupby('label')['score'].mean().plot(kind='bar', title='Average Score by Sarcasm Label')
```

By computing the average score for both sarcastic and non-sarcastic comments separately, this analysis aims to compare the engagement levels between these two categories. The resulting bar plot visualizes the average score for sarcastic and non-sarcastic comments, providing insights into whether sarcastic comments tend to receive more, or fewer votes compared to non-sarcastic ones. This analysis helps in understanding the relationship between sarcasm and audience engagement within the dataset.

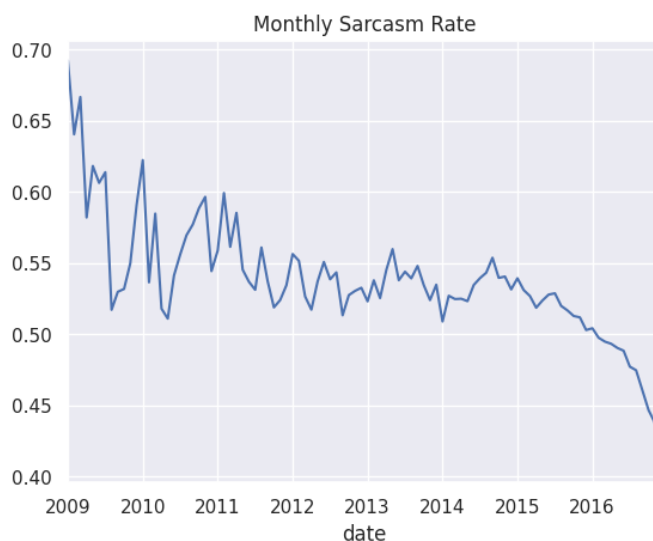


### → Analyzing Monthly Sarcasm Rate

After that, the 'date' column in the dataset is converted to a datetime format using the `pd.to_datetime()` function, enabling temporal analysis. Subsequently, the DataFrame is indexed by the 'date' column, and then resampled at a monthly frequency using `resample('M')`.

```
data['date'] = pd.to_datetime(data['date'])  
data.set_index('date').resample('ME')['label'].mean().plot(title='Monthly Sarcasm Rate')
```

This resampling aggregates the data into monthly intervals and calculates the mean sarcasm rate for each month. The resulting line plot visualizes the monthly sarcasm rate over time, allowing for the observation of trends or patterns in sarcasm usage within the dataset. This analysis provides insights into how sarcasm varies over different periods, potentially revealing seasonal or temporal trends in sarcastic communication.



Consequently, the frequency of sarcasm is on the decline.

### → Analyse the subreddits

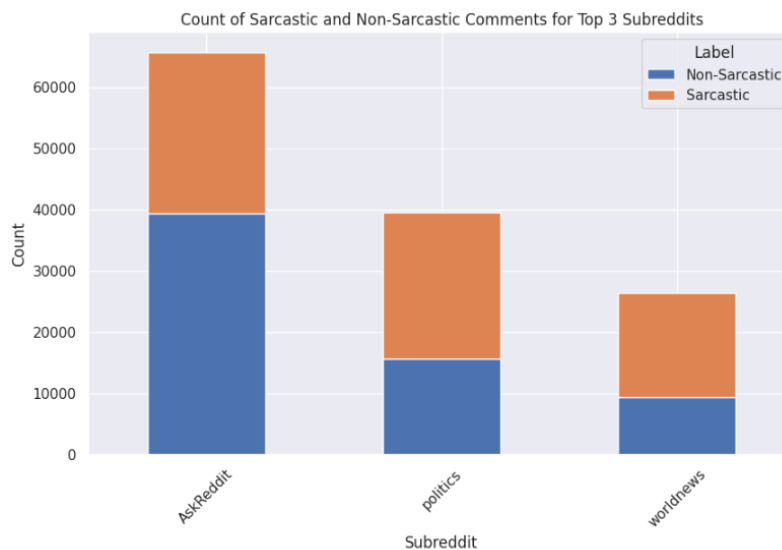
The analysis of subreddits focuses on understanding the distribution and prevalence of sarcastic comments across different subreddits. This helps us identify which subreddits have the highest number of sarcastic comments, and which subreddits have the highest proportion of sarcastic comments relative to the total number of comments in that subreddit.

```
# Identify the top 3 popular subreddits
top_subreddits = data['subreddit'].value_counts().head(3).index.tolist()

# Filter the dataset to include only comments from the top subreddits
top_subreddits_data = data[data['subreddit'].isin(top_subreddits)]

# Calculate the count of sarcastic and non-sarcastic comments for each subreddit
subreddit_counts = top_subreddits_data.groupby(['subreddit', 'label']).size().unstack(fill_value=0)
```

This bar plot shows the count of sarcastic and non-sarcastic comments for the top 3 subreddits with the highest total number of comments. The stacked bars make it easy to compare the relative number of sarcastic and non-sarcastic comments within each subreddit.



For example we can see that the subject of politics is mostly use with sarcasm.

### → Distribution of Sarcastic Comment Lengths vs. Parent Comment Lengths

The boxplot visualization below provides insights into the distribution of comment lengths for sarcastic comments compared to their parent comments.

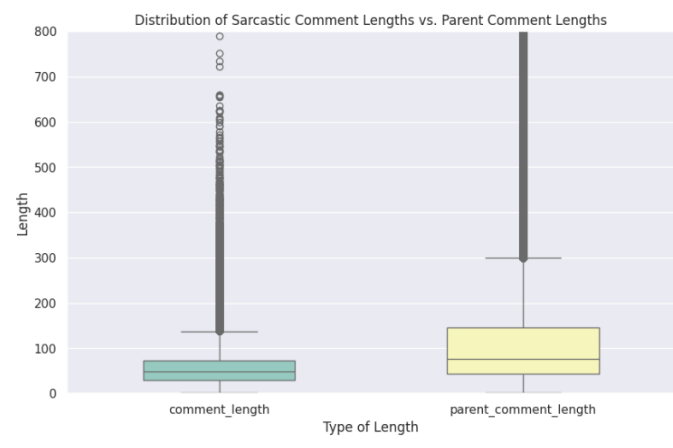
The mean length of sarcastic comments is approximately 56.45 characters and for the parent comments it is approximately 133.17 characters.

```
# Filter sarcastic comments and their corresponding parent comments
sarcastic_data = data[data['label'] == 1].copy()

# Calculate length of sarcastic comments and parent comments
sarcastic_data.loc[:, 'comment_length'] = sarcastic_data['comment'].apply(len)
print(sarcastic_data.loc[:, 'comment_length'].mean())
sarcastic_data.loc[:, 'parent_comment_length'] = sarcastic_data['parent_comment'].apply(len)
print(sarcastic_data.loc[:, 'parent_comment_length'].mean())

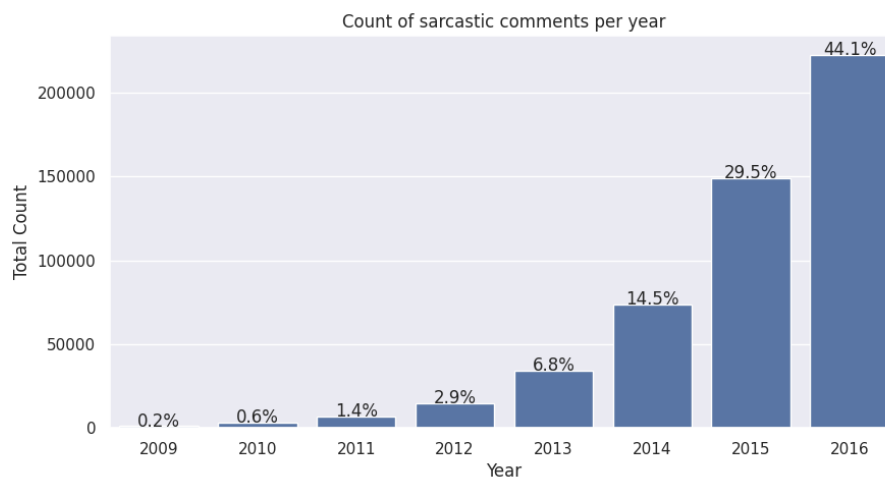
# Set up the figure and axis
plt.figure(figsize=(10, 6))

# Create a boxplot for comment lengths and parent comment lengths
sns.boxplot(data=[sarcastic_data['comment_length'], sarcastic_data['parent_comment_length']],
            width=0.5, palette="Set3")
```



→ Sarcasm threw the years

Our last graphic shows that the more the time goes the more we are sarcastic.



→ Modelling

Finally, we ensure that the model focuses on relevant features, potentially improving the performance and interpretability of the sarcasm detection model by removing unnecessary columns.



```
data_new = data.drop(['author', 'ups', 'downs', 'date', 'created_utc', 'Year'], axis=1)
data_new.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1010771 entries, 0 to 1010825
Data columns (total 6 columns):
#   Column          Non-Null Count  Dtype
---  ---
0   label           1010771 non-null  int64
1   comment         1010771 non-null  object
2   subreddit       1010771 non-null  object
3   score           1010771 non-null  int64
4   parent_comment  1010771 non-null  object
5   cleaned_text    1010771 non-null  object
dtypes: int64(2), object(4)
memory usage: 54.0+ MB
```

## c. Model Implementation

To implement and evaluate our machine learning models for sarcasm detection, we employed a combination of diverse techniques and libraries.

### → Logistic Regression

To implement this model, we have used TF-IDF vectorization. It is instantiated with specified parameters such as 'ngram\_range', 'max\_features', and 'min\_df', which define the range of n-grams to consider, the max number of features to extract, and the min document frequency respectively. In the other hand, the model is instantiated with parameters such as 'c', 'n\_jobs', 'solver', 'random\_state', 'verbose', and 'max\_iter', which control regularization strength, parallel processing, optimization solver, random seed, verbosity level, and maximum number of iterations respectively.

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

# Define the TF-IDF vectorizer and logistic regression model
tf_idf = TfidfVectorizer(ngram_range=(1, 2), max_features=50000, min_df=2)

logit = LogisticRegression(C=1, n_jobs=4, solver='lbfgs', random_state=17, verbose=0, max_iter=1000)

# Create a pipeline that combines the vectorizer and the model
#tfidf_logit_pipeline = Pipeline([('tf_idf', tf_idf), ('logit', logit)])
#This pipeline first transforms the text data using TF-IDF and then applies the logistic regression
from sklearn.pipeline import make_pipeline
tfidf_logit_pipeline = make_pipeline(tf_idf, StandardScaler(with_mean=False), logit)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data_new['comment'], data_new['label'],
                                                    #25% of the data is used for testing, 75% for training

# Train the pipeline on the training data
tfidf_logit_pipeline.fit(X_train, y_train)
#The TF-IDF vectorizer transforms the text data, and the logistic regression model is trained

# Make predictions on the test set
valid_pred = tfidf_logit_pipeline.predict(X_test)
#Prints a report with various classification metrics (precision, recall, F1-score)

# Evaluate the model
print(classification_report(y_test, valid_pred))
```

The 'classification\_report' function is used to print a report with various classification metrics such as precision, recall, and F1-score, comparing the predicted labels against the actual labels in the test set.

Next, we evaluate the model using Cross-Validation. We took the pipeline (tfidf\_logit\_pipeline), input features (data\_new['comment']), target variable (data\_new['label']), number of folds (cv=5), and scoring metric (scoring='accuracy') as parameters.

```
from sklearn.model_selection import cross_val_score

# Perform cross-validation on the pipeline
cv_scores = cross_val_score(tfidf_logit_pipeline, data_new['comment'], data_new['label'], cv=5, scoring='accuracy')

# Print the average accuracy and standard deviation
print(f"Cross-Validation Accuracy: {cv_scores.mean():.2f} ± {cv_scores.std():.2f}")
```

Cross-Validation Accuracy: 0.71 ± 0.00

## → Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
# Define the TF-IDF vectorizer and SVM model
tf_idf = TfidfVectorizer(ngram_range=(1, 2), max_features=50000, min_df=2)
gb_model = GradientBoostingClassifier(random_state=42)
tfidf_gb_pipeline = make_pipeline(tf_idf, StandardScaler(with_mean=False), logit)

# Make predictions on the test set
valid_pred = tfidf_logit_pipeline.predict(X_test)

# Evaluate the model
print(classification_report(y_test, valid_pred))

# Perform cross-validation on the pipeline
cv_scores = cross_val_score(tfidf_gb_pipeline, data_new['comment'], data_new['label'], cv=5, scoring='accuracy')

# Print the average accuracy and standard deviation
print(f"Cross-Validation Accuracy: {cv_scores.mean():.2f} ± {cv_scores.std():.2f}")
```

As we can see we used the same technique for implementing the Gradient Boosting model.

## → SVM

We tried to implement the linear SVM code, but the data was too heavy even with sample, so it didn't work.

```
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

# Define the TF-IDF vectorizer and SVM model
tf_idf = TfidfVectorizer(ngram_range=(1, 2), max_features=10000, min_df=2)
svm = SVC(C=1, kernel='linear', probability=True, random_state=17)

# Create a pipeline that combines the vectorizer and the model
tfidf_svm_pipeline = make_pipeline(tf_idf, StandardScaler(with_mean=False), svm)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data_new['comment'], data_new['label'], te

# Train the pipeline on the training data
tfidf_svm_pipeline.fit(X_train, y_train)

# Make predictions on the test set
valid_pred = tfidf_svm_pipeline.predict(X_test)

# Evaluate the model
print(classification_report(y_test, valid_pred))

# Perform cross-validation on the pipeline
cv_scores = cross_val_score(tfidf_svm_pipeline, data_new['comment'], data_new['label'], cv=5,

# Print the average accuracy and standard deviation
print(f"Cross-Validation Accuracy: {cv_scores.mean():.2f} ± {cv_scores.std():.2f}")
```

## → GRU and LSTM

Because of RAM issues we chosed to sample the data for this models.

To leverage the computational power of GPUs, we configure TensorFlow to recognize and utilize GPU resources. By calling `tf.config.list_physical_devices('GPU')`, we ensure that TensorFlow is aware of available GPU devices, which can significantly accelerate training and processing times for our models.

```
import tensorflow as tf
tf.config.list_physical_devices('GPU')
```

The text is split into individual words or tokens. This helps in analyzing the text on a word-by-word basis. Then we converted words to their base forms. This reduces the variability in the text and helps in focusing on the core meaning of the words.

```
import string
import nltk
nltk.download('punkt')
nltk.download('wordnet')
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
def preprocess_text(text):
    tokens = nltk.word_tokenize(text) # Tokenization

    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(t) for t in tokens] # Lemmatization
    return " ".join(tokens)

data_sample['comment'] = data_sample['comment'].apply(preprocess_text)
```

We tokenize the text into sequences of integers and pad them to ensure all sequences have the same length. The padding ensures all sequences are of the same length by adding zeros to shorter sequences. This is crucial for batch processing in neural networks.



```
# Tokenization

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
tokenizer = Tokenizer(num_words=20000)
tokenizer.fit_on_texts(data_sample['comment'].values)

tokens = tokenizer.texts_to_sequences(data_sample['comment'].values)
X = pad_sequences(tokens, maxlen=100) # Pad sequences to ensure uniform length

numTokens = [len(token) for token in tokens]
numTokens = np.array(numTokens)
print("Tokens'mean", np.mean(numTokens))
print("Max", np.max(numTokens))
print("Argmax", np.argmax(numTokens))
```

```
Tokens'mean 11.3104
Max 1665
Argmax 1332
```

We

define a Sequential model using TensorFlow Keras, which includes an embedding layer, GRU layer or LSTM layer, and a dense output layer.

- **Embedding Layer:** Converts words into dense vectors of fixed size, capturing the semantic meaning of words.
- **GRU Layer or LSTM Layer:** Captures temporal dependencies and patterns in the data. The dropout parameters help prevent overfitting.
- **Dense Layer:** A single neuron with sigmoid activation function to output a probability for binary classification.

```
-----Summary of the built modelLSTM-----
Model: "sequential_1"

Layer (type)                 Output Shape              Param #
-----
embedding_1 (Embedding)      (None, 200, 128)         2560000
spatial_dropout1d_1 (Spatia (None, 200, 128)         0
alDropout1D)
lstm (LSTM)                  (None, 100)              91600
dense_1 (Dense)              (None, 1)                101
-----
Total params: 2651701 (10.12 MB)
Trainable params: 2651701 (10.12 MB)
Non-trainable params: 0 (0.00 Byte)
None
```

```
-----Summary of the built modelGRU-----
Model: "sequential"

Layer (type)                 Output Shape              Param #
-----
embedding (Embedding)        (None, 200, 128)         2560000
spatial_dropout1d (Spatial (None, 200, 128)         0
Dropout1D)
gru (GRU)                    (None, 100)              69000
dense (Dense)                (None, 1)                101
-----
Total params: 2629101 (10.03 MB)
Trainable params: 2629101 (10.03 MB)
Non-trainable params: 0 (0.00 Byte)
None
```

We train the model on the training set and validate it using a small portion of the training data (validation split of 0.1).



```
# Train the model
history = model_GRU.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.1, verbose=1)

Epoch 1/5
57/57 [=====] - 55s 873ms/step - loss: 0.6875 - accuracy: 0.5250 - val_loss: 0.6767 - val_accuracy: 0.5875
Epoch 2/5
57/57 [=====] - 35s 613ms/step - loss: 0.6080 - accuracy: 0.6922 - val_loss: 0.6712 - val_accuracy: 0.5725
Epoch 3/5
57/57 [=====] - 32s 562ms/step - loss: 0.4025 - accuracy: 0.8311 - val_loss: 0.7546 - val_accuracy: 0.5750
Epoch 4/5
57/57 [=====] - 34s 608ms/step - loss: 0.2161 - accuracy: 0.9181 - val_loss: 0.9824 - val_accuracy: 0.5525
Epoch 5/5
57/57 [=====] - 32s 560ms/step - loss: 0.1205 - accuracy: 0.9556 - val_loss: 1.1858 - val_accuracy: 0.5500
# Train the model
history = model_LSTM.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.1, verbose=1)

Epoch 1/5
57/57 [=====] - 69s 1s/step - loss: 0.6878 - accuracy: 0.5336 - val_loss: 0.6778 - val_accuracy: 0.5725
Epoch 2/5
57/57 [=====] - 36s 618ms/step - loss: 0.6173 - accuracy: 0.6894 - val_loss: 0.6407 - val_accuracy: 0.6250
Epoch 3/5
57/57 [=====] - 35s 621ms/step - loss: 0.4374 - accuracy: 0.8047 - val_loss: 0.6815 - val_accuracy: 0.6325
Epoch 4/5
57/57 [=====] - 35s 617ms/step - loss: 0.2647 - accuracy: 0.8917 - val_loss: 0.7851 - val_accuracy: 0.6325
Epoch 5/5
57/57 [=====] - 36s 632ms/step - loss: 0.1549 - accuracy: 0.9458 - val_loss: 0.9571 - val_accuracy: 0.6000
```

## → RoBERTa Model

The RoBERTa tokenizer and pre-trained model for sarcasm detection are loaded using the `AutoTokenizer.from_pretrained` and `AutoModelForSequenceClassification.from_pretrained` functions respectively.

The `predict_sarcasm` function takes input text, preprocesses it, tokenizes it using the tokenizer, and passes it through the pre-trained RoBERTa model. It then calculates the sarcasm score based on the model's output probabilities.

A Tkinter GUI application named `SarcasmDetectorApp` is defined, which consists of a text entry field for input, a button to trigger the sarcasm detection, and a label to display the sarcasm score.

The `detect_sarcasm` method of the `SarcasmDetectorApp` class retrieves the input text from the text entry field, calls the `predict_sarcasm` function to get the sarcasm score, and updates the result label with the obtained score.

A Tkinter window is created, and the `SarcasmDetectorApp` instance is initialized, followed by the event loop (`mainloop()`) to start the GUI application.

```
# Tkinter Application
class SarcasmDetectorApp:
    def __init__(self, master):
        self.master = master
        master.title("Sarcasm Detector")

        self.label = tk.Label(master, text="Enter text:")
        self.label.pack()

        self.text_entry = tk.Text(master, height=5, width=50)
        self.text_entry.pack()

        self.detect_button = tk.Button(master, text="Detect Sarcasm", command=self.detect_sar
        self.detect_button.pack()

        self.result_label = tk.Label(master, text="")
        self.result_label.pack()

    def detect_sarcasm(self):
        input_text = self.text_entry.get("1.0", tk.END)
        sarcasm_score = predict_sarcasm(input_text)
        self.result_label.config(text=f"Sarcasm score: {sarcasm_score}")

# Create Tkinter window
root = tk.Tk()
app = SarcasmDetectorApp(root)
root.mainloop()
```

## d. Model Evaluation

### → Logistic Regression and Gradient Boosting

The classification report for the two models reveals an overall accuracy of 71% on the test set, with precision scores of 70% for non-sarcastic (class 0) and 72% for sarcastic (class 1) comments.

The recall scores indicate that the model correctly identifies 74% of non-sarcastic comments and 68% of sarcastic comments. The F1-score, which balances precision and recall, is 0.72 for non-sarcastic comments and 0.70 for sarcastic comments.

These results suggest that the two models model performs reasonably well in distinguishing between sarcastic and non-sarcastic comments. However, the exploration of additional techniques may enhance its performance in sarcasm detection. Even if the model is consistent and robust with no variability across the 5 folds as we have seen the results by doing Cross-Validation, the model is still not the best model for our project.

|              | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| 0            | 0.70      | 0.74   | 0.72     | 126226  |
| 1            | 0.72      | 0.68   | 0.70     | 126467  |
| accuracy     |           |        | 0.71     | 252693  |
| macro avg    | 0.71      | 0.71   | 0.71     | 252693  |
| weighted avg | 0.71      | 0.71   | 0.71     | 252693  |

## → GRU and LSTM

From the training history of the LSTM and GRU model, we observe that the training loss and accuracy generally improve with each epoch, indicating that those models are learning from the data. However, the validation loss and accuracy fluctuate and show signs of degradation in later epochs, suggesting potential overfitting. This means those models perform well on the training data but struggle to generalize to unseen data, which is a common challenge in machine learning and deep learning.

Upon evaluation on the test set, the models achieved an accuracy of around 62/61%, indicating their ability to correctly classify sarcastic and non-sarcastic comments. However, further analysis revealed a balanced but modest precision, recall, and F1-score for both classes, suggesting the need for improvements in distinguishing between the two classes

```
-----Classification report model LSTM-----
              precision    recall  f1-score   support

     0       0.62         0.58         0.60         503
     1       0.60         0.64         0.62         497

   accuracy          0.61         1000
  macro avg       0.61         0.61         0.61         1000
 weighted avg       0.61         0.61         0.61         1000

[[293 210]
 [178 319]]
```

```
-----Classification report model GRU-----
              precision    recall  f1-score   support

     0       0.61         0.62         0.62         503
     1       0.61         0.60         0.60         497

   accuracy          0.61         1000
  macro avg       0.61         0.61         0.61         1000
 weighted avg       0.61         0.61         0.61         1000

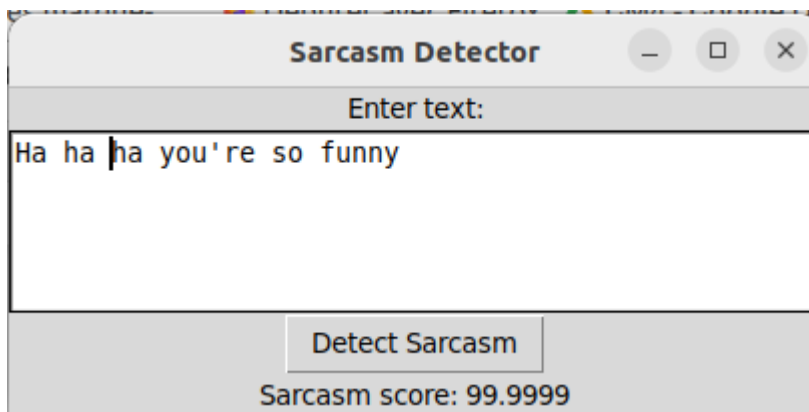
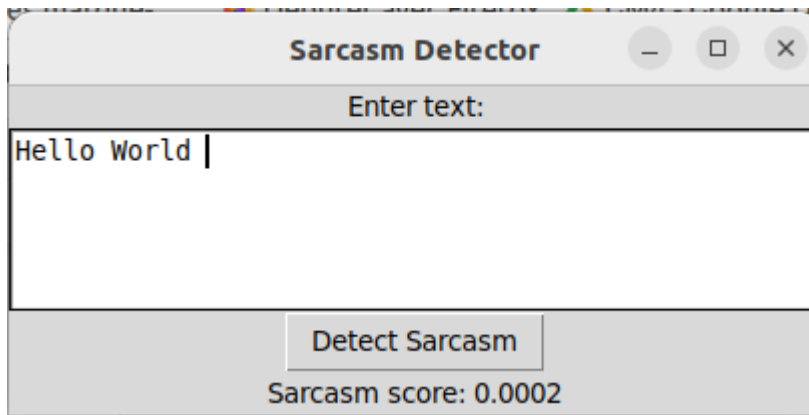
[[314 189]
 [200 297]]
```

The confusion matrix highlighted the model's struggle in correctly classifying instances, with a considerable number of false positives and false negatives. This indicates areas for improvement, possibly through feature engineering or model architecture adjustments.

In conclusion, while those models show promise in detecting sarcasm, their performance could be further enhanced to improve accuracy and robustness in real-world scenarios.

## → RoBERTa Model

In this model we have set up a GUI application using Tkinter to allow users to input text and detect sarcasm. The interface is presented below. We can write our message and then we have as result the sarcasm score.



## 5. Summary

In conclusion, this project enabled us to apply the knowledge we had acquired throughout the semester by developing a project to detect sarcasm in text.

Through the exploration and implementation of our models, it becomes evident that the best model to detect sarcasm in text is the RoBERTa Model.

While Logistic Regression and neural network models like LSTM and GRU were considered, they posed significant challenges in terms of implementation complexity and computational resource requirements. The execution time for these models was notably long and, in some cases, even impractical, hindering their feasibility for large-scale sarcasm detection tasks. Indeed, our code was taking too long to run due to the size of our dataset so one solution was to randomly sample a smaller subset of our data to work with.

On the other hand, the RoBERTa model showed really good results when it came to detecting sarcasm. It's built on a fancy type of technology called transformers and was trained specifically to understand sarcasm better. This model did a great job at picking up on the subtle meanings and tricky language tricks that often come with sarcasm. That's why it did better than the other models we tried.

## 6. Bibliography

- [1] <https://www.kaggle.com/danofer/sarcasm/data?select=train-balanced-sarcasm.csv>
- [2] Natekin A., Knoll A. Gradient boosting machines, a tutorial //Frontiers in neurorobotics. – 2013. – T. 7. – C. 21.
- [3] [logistic regression](#)
- [4] [https://en.wikipedia.org/wiki/Long\\_short-term\\_memory](https://en.wikipedia.org/wiki/Long_short-term_memory)
- [5] [https://en.wikipedia.org/wiki/Gated\\_recurrent\\_unit](https://en.wikipedia.org/wiki/Gated_recurrent_unit)
- [6] <https://towardsdatascience.com/roberta-1ef07226c8d8>