National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Web systems and applications - Final project report
Summer semester 2023-2024

Student 1	Triantafyllou Thanasis
	7115132300010
Student 2	Tsiompikas Dimitris
	7115112300036
Student 3	Syrios Konstantinos-Zois
	7115112300035

Movie Reviews Sentiment Mining and Analysis

Web Crawler implementation

For this project we wanted to crawl movies from IMDb and scrape the reviews(or a subset) of each movie. So we started by trying to find a way to crawl the review page from each movie . I found that the IMDb url stracture "https://www.imdb.com/title/tt0100000/reviews/?ref_=tt_ov_rt" had a unique number in it for each movie. For example the movie we start crawling is this one "**Lehrer leben länger** " with this unique number : "0100000". So by incrementing this number in the url by one the crawler could search as many movies as we would like . Some of the problems we encountered was that not every number/movie was valid . Some of the movies were giving 404 error . Also in the url structure if you start from number 1 there must be "00000" in front of the number/movie you want to crawl (example : "0000001" instead of "1") so we had to predefine how many movies we would like to crawl. For that reason we chose to target almost 1 million movies from 0100000 to 01999999 and scrape 100 reviews from each movie If it had more than 100. If not we would take all the reviews. The code analysis follows.

Code Analysis

```
# Get the HTML content
def get_html(url):
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3"
    }
    response = httpx.get(url, headers=headers)
    html = HTMLParser(response.text)

    return html

# Extractor function to bypass NoneType error
def extractor(html, selector):
    try:
        return html.css_first(selector).text()
    except AttributeError:
        return "None"
```

Function Definitions

get_html(url) Function

This function fetches the HTML content of a given URL. It uses the httpx library to make the HTTP request and a custom HTMLParser to parse the response content.

- **Headers:** A dictionary specifying the User-Agent to mimic a request from a web browser. We hoped that with this we would avoid being blocked by IMDb but it didn't work out as expected.
- **HTTP GET Request:** The httpx.get function sends a GET request to the URL with the given headers.
- **HTML Parsing:** The response text is passed to HTMLParser to parse the HTML content.
- **Return Value:** The parsed HTML content is returned.

extractor(html, selector) Function

This function extracts specific data from the HTML content using a CSS selector. We also included error handling to manage cases where the selector might not be found (ex.:404 error).

- **CSS Selection:** The html.css_first(selector) method is used to select the first element that matches the given CSS selector.
- **Error Handling:** A try-except block is used to handle AttributeError exceptions, which occurs if the selector does not match any elements. In such cases, the function returns "None".
- **Return Value:** The text content of the selected element is returned or "None" if an exception occurs.

```python
# Extract the data
def parse_page(html):
    movies = html.css("div.lister-item-content")
    #print(movies)

    for movie in movies:
        item = {
            "review": extractor(movie, "div.content"),
            "rating": extractor(movie, "span.rating-other-user-rating")
        }
        item["review"] = item["review"].replace("\n", " ")
        item["review"] = ' '.join(item["review"].split())
        item["rating"] = item["rating"].replace("\n", " ")
        item["rating"] = ' '.join(item["rating"].split())
        remove_string = "found this helpful. Was this review helpful? Sign in to vote. Permalink"
        item["review"] = item["review"].replace(remove_string, "")
        item["review"] = re.sub(r"\s*\d+\s*out\s*of\s*\d+\s*$", "", item["review"])
        item["review"] = item["review"].replace("\"", "")
        #print(item)
        with open('data.json', 'a') as f:
            json.dump(item, f)
            f.write('\n')
            f.close()
```

parse_page(html) Function

This function processes the HTML content so that we can extract movie reviews and ratings, cleans the extracted data, and appends it to a JSON file.

- **CSS Selection:** The html.css('div.lister-item-content') method is used to select all elements matching the CSS selector for movie content.
- **Loop Through Movies:** Here the code iterates over each movie item in the selected elements.
- **Data Extraction:** The extractor function is used here to extract the review and rating data from each movie using specific CSS selectors (div.content for reviews and span.rating-other-user-rating for ratings).

Data Cleaning:

- **Newline Removal:** Replace the Newline characters in the review and rating strings with spaces.
- **Whitespace Normalization:** Reduce whitespace characters to a single space.

- **Specific String Removal:** Remove a predefined string (remove_string) from the review text that was inside every review container.
- **Rating Pattern Removal:** Use a regex pattern to remove "number out of number *" string from the review text that was also in every rating container.
- **Quote Removal:** Remove double quotes from the review text.

Data Storage: We append the cleaned data to a JSON file (data.json), with each item written as a new line in the file.

```
def main():
    html = get_html(baseurl)
    parse_page(html)


if __name__ == "__main__":

    # Crawling Loop
    key = 100000
    while key < 999999 :

        baseurl = f"https://www.imdb.com/title/tt0{str(key)}/reviews/?ref_=tt_ov_rt"
        main()
        key += 1
        print("Done: " , key)
```

main() Function

Here we have our entry point for the web crawling/scraping process. It calls the get_html function to retrieve HTML content and then processes this content using the parse_page function.

- **HTML Retrieval:** get_html(baseurl) is called here to fetch the HTML content from the specified URL.
- **Data Parsing:** Passes the fetched HTML content to parse_page(html) for data extraction and processing.

Crawling Loop and Execution Control

This is the loop to iterate through the movies of IMDb review pages, and update the baseurl for each iteration. It ensures the main function is called with the updated URL for each page. We Initialize the key variable to 100000.

Crawling Loop: We run a while loop as long as key is less than 999999. In each iteration:

- **URL Formation:** Constructs the baseurl using the current key value, targeting IMDb review pages.
- **Main Function Call:** Execute the main function to fetch and parse data from the constructed URL.
- **Key Increment:** Increments the key by 1.
- **Progress Logging:** Print the current value of the key to track the progress so far.

Problems

As mentioned in the mid term report we had 2 main problems that made us use Selenium instances that work synergetic with the crawler/scraper. Firstly we wanted to be able to gather 100 reviews from each movie if there were as much. Here we had the problem with how the IMDb site controls the pagination and the mechanism behind it that loads more reviews, when someone presses the “Load More” button to load more reviews (starting with 25 reviews and every time someone clicks the load more button 25 more reviews get loaded). The IMDb site loads more reviews asynchronously (AJAX). This AJAX-based pagination means the reviews are loaded in the background and added to the current page content. So we weren’t able to find a way to crawl and scrape more than 25 reviews. The second problem was that because IMDb provides an API to gather data from the site, It disallows scraping and other automations in some cases stated in the robot.txt, so after a couple of hours of crawling we got our IP blocked. With that in mind we decided to use selenium to counter these problems.

NOTE: Some of the provided code above was discarded when we weren’t able to find solutions to our problems, so we continued with the use of Selenium.

Selenium Instances/Bots

We decided to use selenium instances to counter the said problem with pagination to load more reviews and make it copy the user behavior so that we won't have problems with IP blocks and bans.

```
# Random proxy function
"""
def rand_proxy():
    #proxy = random.choice(validProxies.ips)
    #return proxy"
"""
```

rand_proxy() Function

This function is the logic to randomly select a proxy IP from a predefined list of valid proxies and return it for use in web requests. We implemented that logic so that we could use a rotation of IPs while gathering data. We tried to do it with free IPs that we found online but they were very slow so this was not viable. Still I left the logic commented out because in a real research wise scenario we would have paid fast proxy IPs that would work. Instead of that we used a static free VPN.

```python
def loadBot():

    # List of proxy implementation
    """
    #proxy = rand_proxy()
    #chrome_options = webdriver.ChromeOptions()
    #chrome_options.add_argument(f'--proxy-server={proxy}')
    """



    #Bot implementation
    driver = webdriver.Chrome()

    key = 100000
    while key < 999999:
        driver.get(f"https://www.imdb.com/title/tt0{str(key)}/reviews/?ref_=tt_ov_rt")

        print("Bot 1: ")
        print("~" * 50)
        try:
            loadMore = driver.find_element(By.ID, "load-more-trigger")
            i = 0
            while i < 4:
                driver.execute_script("arguments[0].click();", loadMore);
                i += 1
                print(f"Done {i} loads")
                time.sleep(3)
        except:
                print("No more reviews to load")


        time.sleep(8)
        crawler.crawlerStart(driver.page_source)
        key += 1
        print("Done: " , driver.title)
        print("\n")

    driver.quit()
```

This is the logic for the web scraping bot using Selenium WebDriver to extract reviews from IMDb. The bot simulates the user, clicking the "Load More Reviews" button to load additional reviews dynamically. The commented out part is the logic for using proxy IPs.

loadBot() Function

This function serves as the main driver for the web scraping bot. It initializes a Selenium WebDriver instance, iterates through a range of IMDb review pages, handles the pagination to load more reviews and scraps the reviews and ratings.

- **Proxy Setup:** The commented-out section is the logic to set up a proxy using Selenium's ChromeOptions.
- **WebDriver Initialization:** Initialize a Chrome WebDriver instance to control the browser.
- **Loop Through IMDb Pages:** Iterate through a range of IMDb review pages using a while loop, it's the same logic as the previous code mentioned above.
- **Loading More Reviews:** Try to find and click the "Load More" button up to 4 times to load additional reviews dynamically (100 reviews in total if there are as many).
- **Page Source Handling:** After attempting to load more reviews, the bot passes the page source to the crawler.crawlerStart() function .
- **Driver Cleanup:** Quits the WebDriver instance after completing the loop.

```python
from selectolax.parser import HTMLParser
import re
import json
from filelock import FileLock

# Get the HTML content
def get_html(baseurl):
    html = HTMLParser(baseurl)
    return html

# Extractor function to bypass NoneType error
def extractor(html, selector):
    try:
        return html.css_first(selector).text()
    except AttributeError:
        return "None"

# Extract the data
def parse_page(html):
    movies = html.css("div.lister-item-content")
    #print(movies)

    for movie in movies:
        item = {
            "review": extractor(movie, "div.content"),
            "rating": extractor(movie, "span.rating-other-user-rating")
        }
        item["review"] = item["review"].replace("\n", " ")
        item["review"] = ' '.join(item["review"].split())
        item["rating"] = item["rating"].replace("\n", " ")
        item["rating"] = ' '.join(item["rating"].split())
        remove_string = "found this helpful. Was this review helpful? Sign in to vote. Permalink"
        item["review"] = item["review"].replace(remove_string, "")
        item["review"] = re.sub(r"\s*\d+\s*out\s*of\s*\d+\s*$", "", item["review"])
        item["review"] = item["review"].replace("\"", "")
        #print(item)
        with FileLock("lock"):
            with open('imdb_data.json', 'a') as f:
                json.dump(item, f)
                f.write('\n')
                f.close()

# Crawler function
def crawlerStart(baseurl):
    html = get_html(baseurl)
    parse_page(html)




if __name__ == "__main__":
    crawlerStart()
```

crawler.crawlerStart()Function

This function has almost the same logic as the one we discarded with the difference that the selenium bot handles the while loop that iterates through the movie review pages and the load more review pagination, here we parse the already loaded html to gather our data and save them to a json file. Also there is the use of FileLock to manage concurrent file writes, ensuring safe access to imdb_data.json.(The explanation why we used FileLock is below)

MoviesDB review scraping

We started here by trying to gather the reviews with the same approach as before without selenium instances but as before we again had IP blocks . Another problem was that the css containers of the reviews and ratings didn't return us the actual review and rating if the page didn't load. For those reasons we used selenium again to make another instance to handle the loading of the pages. (This time all the reviews of each movie were loading so we didn't have problems with pagination). As before crawler2.crawler2Start() was handling the parsing and scraping of the html to gather the reviews and ratings. Here we targeted 500.000 movies to gather data from. The code that we didn't use here is in the crawler_not_working2.py file and the code that we used is in the PaginationBot3.py, PaginationBot4.py and crawler2.py files.

Parallel crawling/scraping

```python
import multiprocessing
import importlib

# Making each bot a new module to run in diffrent processes
def load_bot(bot_name):
    bot_module = importlib.import_module(bot_name)
    bot_module.loadBot()


if __name__ == "__main__":

    # Running the modules in different processes
    bot_names = ['PaginationBot1', 'PaginationBot2', 'PaginationBot3', 'PaginationBot4']

    for bot_name in bot_names:
        p = multiprocessing.Process(target=load_bot, args=(bot_name,))
        p.start()
```

`load_bot(bot_name)` Function

Dynamically import the specified bot module and execute the `loadBot` function.

- Use `importlib.import_module(bot_name)` to dynamically import the module.
- Call the `loadBot` function from the imported module to start the bot.

Main Execution Block

- Defines a list of bot module names to be executed concurrently.
- Creates a new process for each bot using `multiprocessing.Process`.
- Starts the process, which runs the `load_bot` function with the specified bot module.

Here we used Python's `multiprocessing` and `importlib` modules to run multiple instances of the web scraping bot concurrently. Each bot is running as a separate process to take advantage of the multiple CPU cores and perform parallel scraping. We did that so that we can have 2 bots for each site scraping data. Bot 1 target was IMDb from movie 100000 up to 999999. Bot 2 had targets from movie 1999999 down to 1000000. That means we targeted almost 2 million movies . The reason we did this and not make both bots run in the same range was that bot2 that was going downwards was crawling/scraping in an area that IMDb had porn movies (from 999999 and downwards) so there weren't any reviews to crawl. Also we were going to let the bots run for 12 hours so we didn't want the second bot to not scrape any data because we didn't know how big the volume of the pornographic content was. The bot 3 scraped data from movies 1 up to 500000 and bot 4 from 1000000 down to 500001 so we have 1 more million movies from moviesDB. Also, the FileLock is used here to lock which bot writes on its corresponding JSON (Bot 1 and 2 write to imdb_data.json and Bot 3 and 4 to tmdb_data.json)so that we don't allow both bots write at the same time in the same JSON and have corrupted data.

Web scraping/crawling conclusions

In conclusion we managed to gather around 60k of reviews and after we cleaned the dataset from reviews that had no ratings we left with a solid 50k to train and evaluate our NLP models. Also with this approach we can scale up our project to use more CPU and RAM resources and run more concurrent processes of selenium instances to scrape a lot more data, from more websites than 2. For that purpose we will need more IP addresses though so that we won't have IP blocks.

Dataset Management

After gathering the data from the websites (at the time of writing from IMDb and TMDB) and storing them as separate json files, we process them in order to create two datasets to use for training the machine learning models, the “basic” one and the “keywords” one.

For the basic dataset we labeled the text reviews using [negative, neutral, positive] labels based on the review’s corresponding numerical rating. IMDb uses a 0-10 rating system whereas TMDB uses a 0-100 rating system. We first clean up both datasets from invalid values (for example reviews without ratings), then we transform the TMDB ratings to follow the more standard 0-10 system, and finally, we assign labels as follows:

- [0, 4] as Negative
- (4, 6] as Neutral
- (6,10] as Positive

The text of the reviews, their ratings, and their labels are saved as a separate json file to be used independently from the rest of the program. This dataset will also be uploaded to kaggle.

In addition, we experimented with the idea of detecting positive/negative keywords in the review text and then using their counts as a secondary feature for the models.

We start by splitting the negative and positive keywords from this kaggle dataset. Then we iterate through all the reviews detecting which keywords are used in each one. For each “hit” we associated the keyword with the review’s ID (a unique ID that we assigned to each review of the basic dataset), to create a kind of index with the keywords as the keys. After that we determine the 100 most common negative and positive keywords in the reviews, as these we estimate will have the most impact for movie reviews, and use their counts as the secondary features.

The intermediate dictionaries (e.g. positive keywords-reviewIDs) as well as the new “keywords” dataset are saved as separate json files to be used independently.

Machine Learning Models

Introduction

For this part we created three machine learning models that use classic ML algorithms, specifically Logistic Regression, Support Vector Machine (SVM) and Naive Bayes for multi label classification. The datasets obtained from the previous parts of the assignment will be used here for sentiment analysis and we will compare the models' performance using Learning curves along with some visualizations for the datasets. Amount of data was 47947 movie reviews from both websites.

Pre-processing

Data preprocessing had to occur in order for the data to be passed in the models. We used the following techniques that are commonly employed in NLP tasks:

- **Put all text from the reviews to lower case**
- **Remove all special characters (question marks, commas etc)**
- **Remove stopwords (commonly used words that don't offer anything to training)**
- **Lemmatization**
- **Remove numbers**

Methodology

We made two experiments with each model, one for the basic data and one with the extra features (positive, negative keywords) to see which one would have the best performance. After preprocessing we passed the data through a Hashing Vectorizer to convert the text data to numerical and then feed them to the models for training. For the extra features experiment we used a count vectorizer to also contain the numerical inputs from positive and negative keywords.

Hyperparameter tuning

We also used GridSearch for hyperparameter tuning on all models for the following hyperparameters:

Logistic Regression - C, solver, penalty
Naive Bayes - alpha
SVM - C, penalty

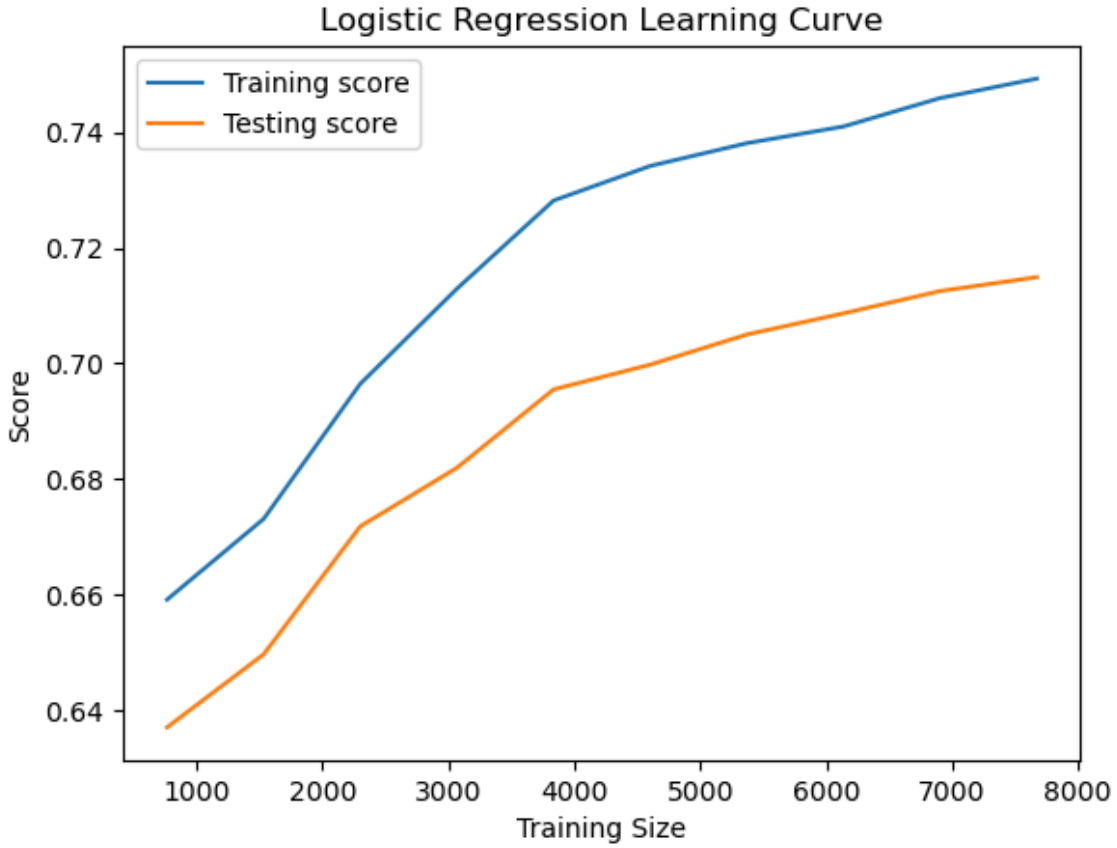
Results

Model	Precision	Recall	F1-Score	Accuracy
Logistic Regression (basic data)	**72%**	**74%**	**74%**	**74%**
Logistic Regression (extra features)	**45%**	**61%**	**61%**	**61%**
SVM (basic data)	**72%**	**75%**	**75%**	**75%**
SVM (extra features)	**46%**	**63%**	**63%**	**63%**
Naive Bayes (basic data)	**70%**	**72%**	**72%**	**72%**
Naive Bayes (extra features)	**46%**	**55%**	**55%**	**55%**

From the table we can see that SVM had the best performance of all models with a very satisfying 75% accuracy. Naive bayes had the worst performance of all with 72% accuracy on the basic data. Unfortunately, the extra features did not assist in achieving more performance scores and instead caused problems to the models lowering their performance by a large margin. SVM still had the best performance on this experiment as well and naive bayes had the worst. Below we will showcase the learning curves, bear in mind that due to the low amount of data some overfitting can be observed in some models, which can be fixed by obtaining more data from the crawlers.

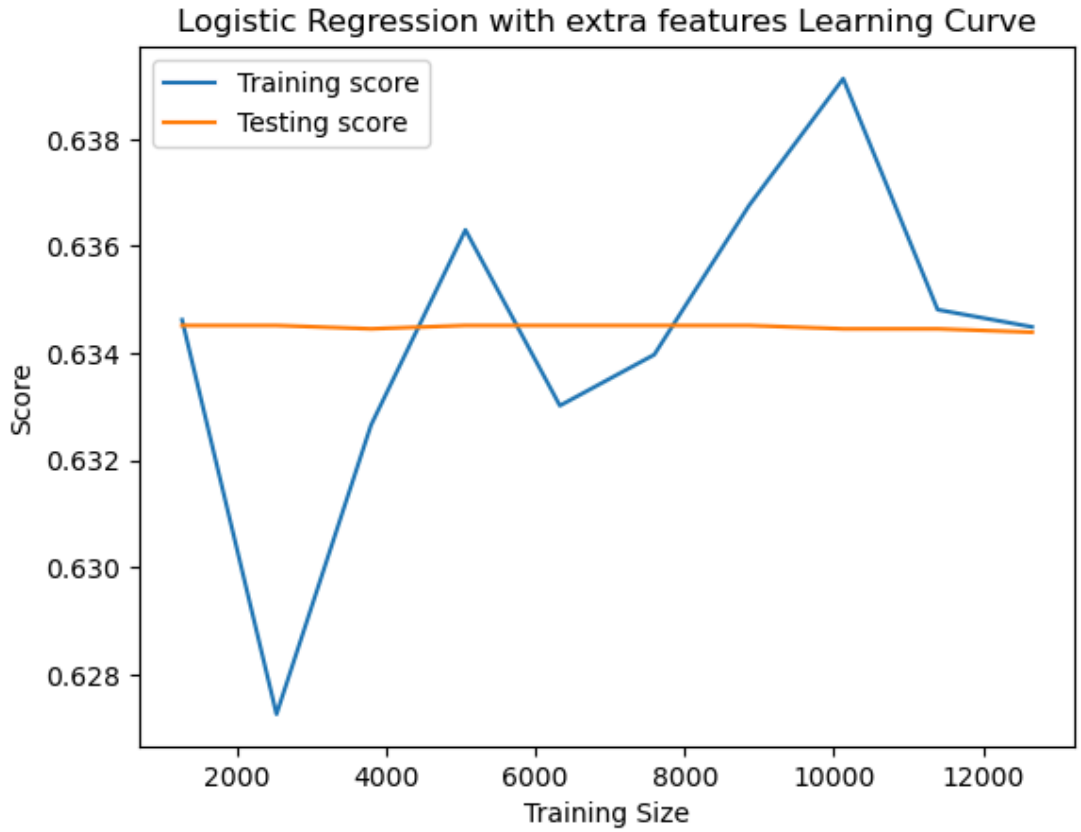
Learning curves

Logistic regression (basic data)



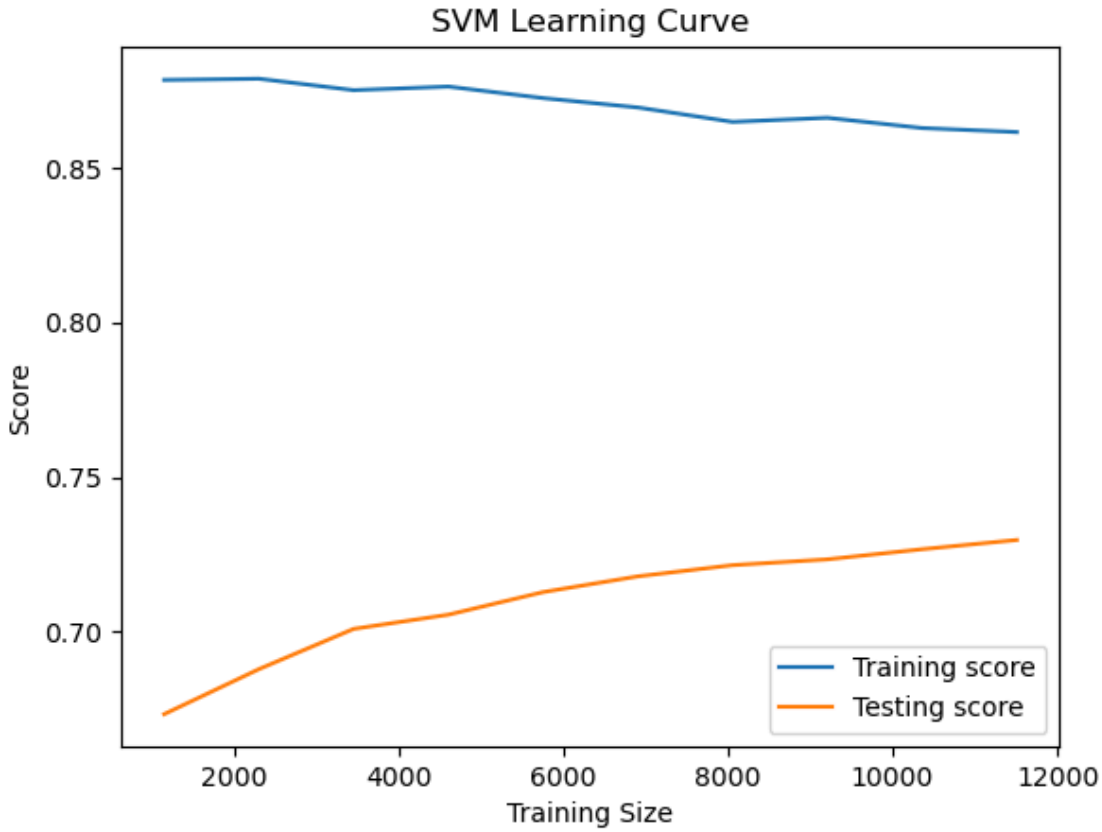
The curves are smooth here, without overfitting. Logistic regression had a normal learning rate despite the low amount of data and was the most stable model.

Logistic regression (extra features)



Every model with extra features had this abnormal learning rate , logistic regression didn't provide any meaningful results with this dataset.

SVM (basic data)

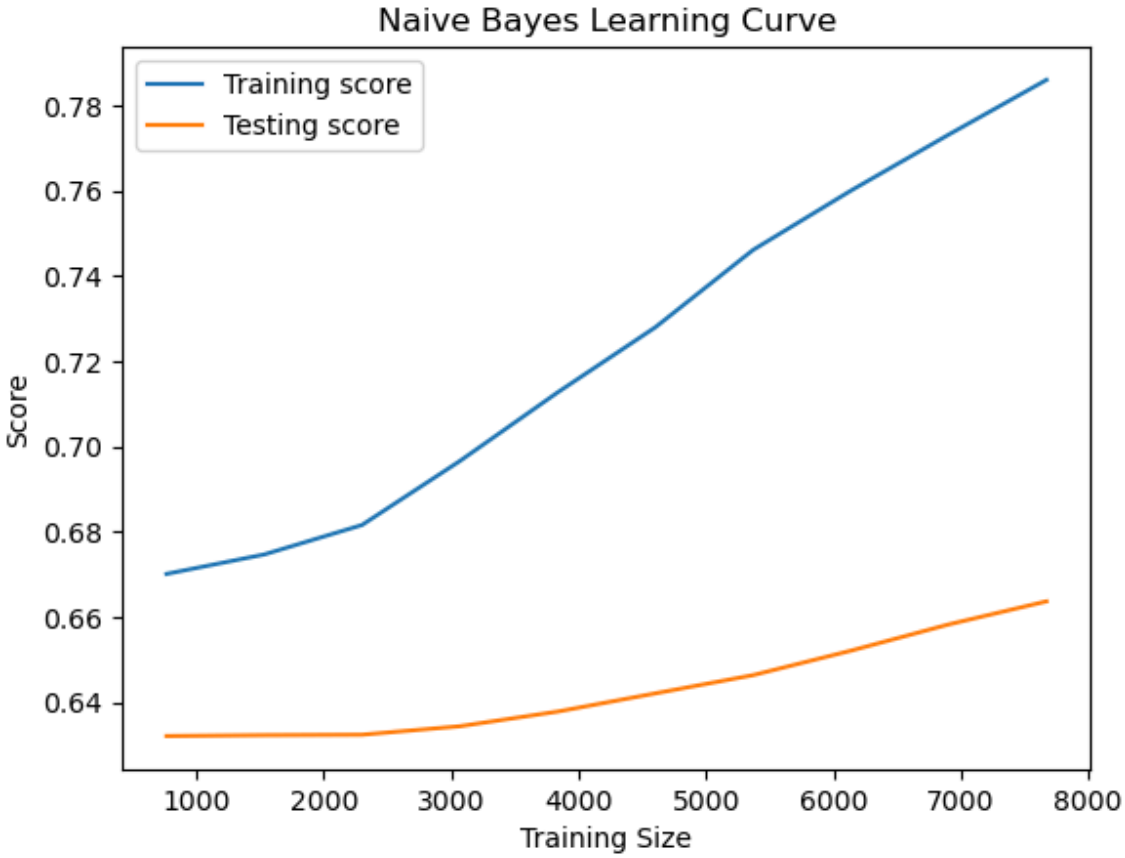


SVM had some overfitting as well, a larger amount of data would help it, still its performance for testing was admirable.

SVM (extra features)

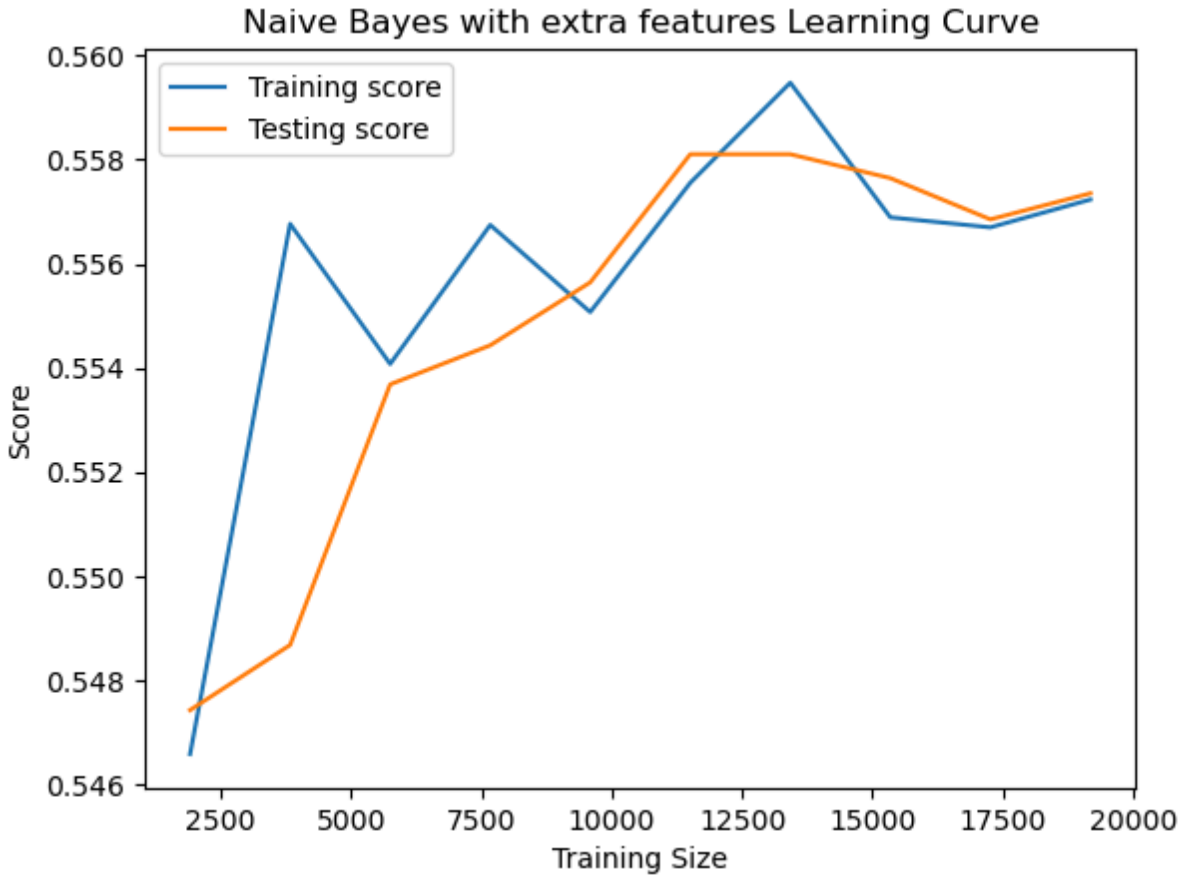
SVM had the same problem as the LR model, no meaningful results from the extra features here either.

Naive Bayes (basic data)



Naive Bayes showed a small overfitting curve here, this could also be easily fixed with adding more data.

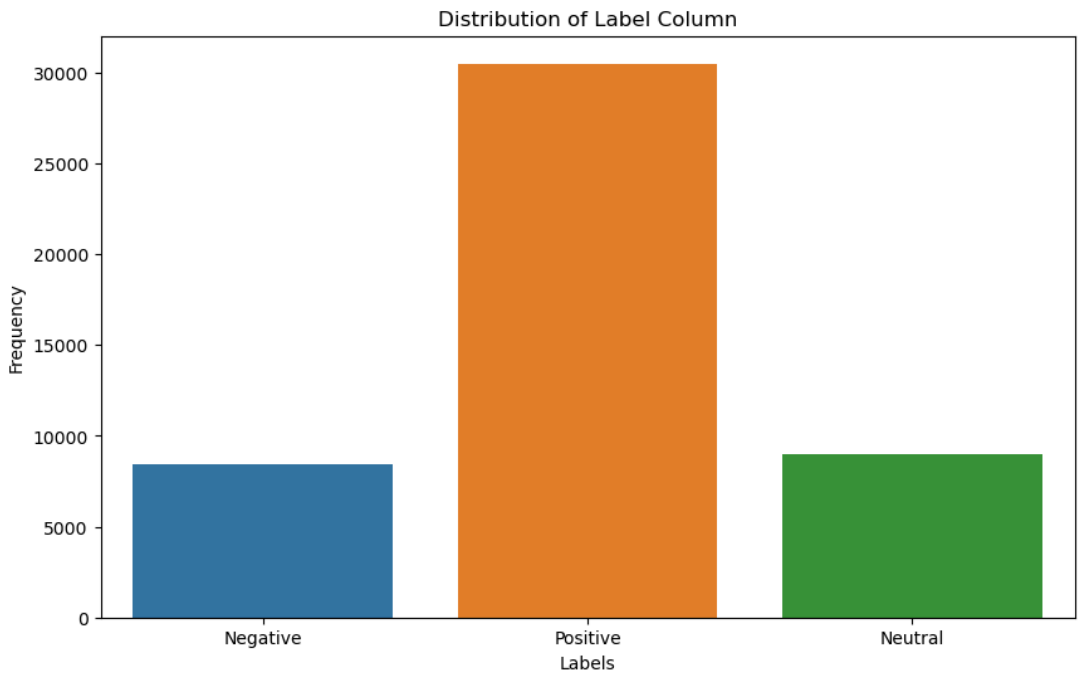
Naive Bayes (extra features)



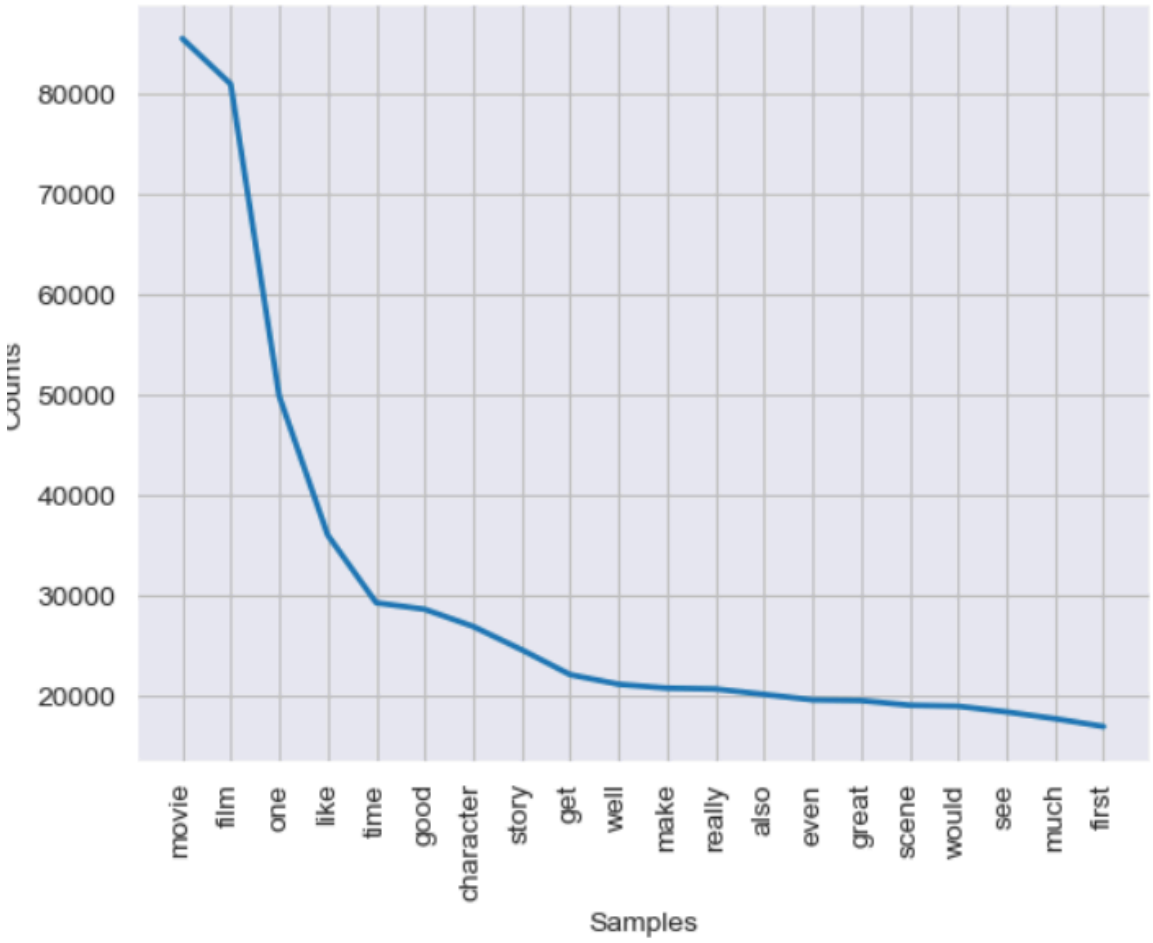
Naive bayes surprisingly had a pretty normal learning curve with the extra features, however the accuracy was really low to provide any meaningful results against the other models or the basic dataset.

Visualizations

Below we will show some visualizations of the dataset.

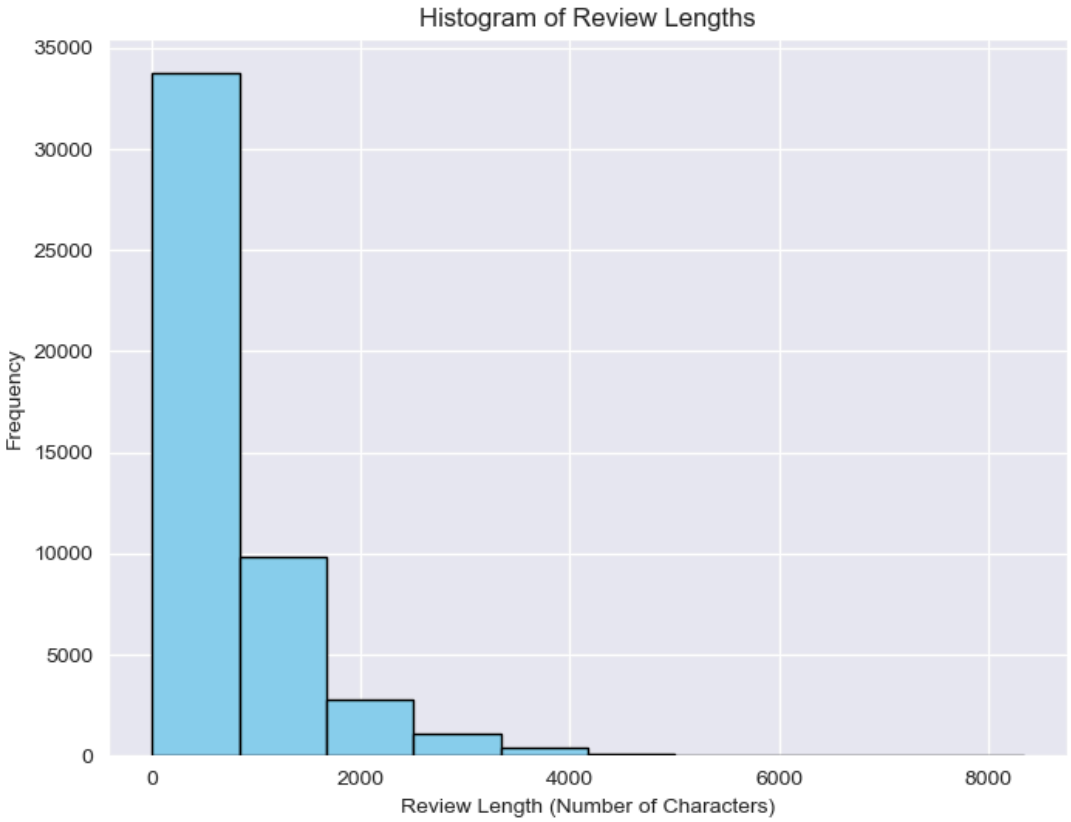


This label distribution diagram shows how many reviews we had for each sentiment. Positive reviews are prevalent, this did not cause any problem with the accuracy but it is always best to have an even amount of data for better training. This , however, with crawlers is impossible since we extract all the data from a website randomly.



In this token frequency diagram we can see the most prevalent words in the dataset. They're all related to movies (movie, film, character) , and some others are related to quality (good, great since we had more positive reviews).

Same as above goes for the word cloud, words like movie, scene, film, character are prevalent here.



Lastly, we have the review length histogram. Most reviews contain few words as expected , however there are some outliers with very long reviews, that people express their opinions in a very detailed way.