Sentiment Analysis: from Classic to Modern Methods

Mohamed Amine Fezzani,[1] Aziz Fritis,[2,*] Fourat Mansouri, [3,*] Yosr Ben Amor, [4,*] Yosr Akrout, [5,*] Mayssa Hmem, [6,*] and Selima Moalla[7]

[1]*The Private Higher School of Engineering and Technologies (ESPRIT), Tunisia*
contact@esprit.ens.tn

Abstract: For decades, humans have been trying to train computers to understand text as we naturally do. And it's fascinating how far we've come in the last twenty years: machines nowadays can achieve a near-flawless, high-level of **'Natural Language Processing'** and can span on different tasks ranging from a simple translation to advanced reasoning. In this work, we dive into Sentiment Analysis with three approaches: classic Machine Learning algorithms, Deep Learning and Large Language Models. We will then compare the results of each method to see the difference.

1. Introduction

Sentiment analysis is a core task in Natural Language Processing (NLP), aiming to determine the emotional tone behind a body of text. It is widely applied in areas such as product reviews, social media monitoring, and customer feedback systems. Over the years, different approaches have been used for sentiment classification, ranging from traditional Machine Learning (ML) techniques to modern Large Language Models (LLMs). Each method comes with its own strengths, trade-offs, and computational requirements.
In this report, we explore and compare these approaches through the lens of a real-world dataset, evaluating their effectiveness and practicality in handling sentiment analysis tasks.

2. Dataset Presentation

The dataset we used in this work is the well-known Amazon Reviews'23 [2] by Julian McAuley *et al.*. It is a large-scale dataset with 571.54M reviews from May 1996 to September 2023 with different categories such as 'Books' and 'Electronics'. We refer you to the website of the authors for more detailed information here.
For the rest of this work, We will be proceeding with the 'All_Beauty' category, there is no particular reason for the choice though. This category contains 701.5K ratings from 632.0K users. The data fields for User Reviews are as follows:

- rating (float): represents the actual rating of the product from 1 to 5.
- title (string): the title of the review.
- text (string): the actual review.
- images (list): a list of images that the user attached to his/her review. It has small-sized, medium-sized and large-sized images.
- asin (string): the product ID.
- parent_asin (string): the parent ID of the product, which can be found in the metadata.
- user_id (string): the user ID.
- timestamp (int): the unix time at which the review was posted.
- verified_purchase (boolean): indicates wether a purchase was verified or not.
- helpful_vote (int): the number of users who found the review helpful.

3. Data Preparation

Data preparation is a crucial step in any task, especially in NLP to ensure normalization and guarantee robust performance of our models. The following pipeline will be run on a labeled and balanced version of the dataset which we will explore in the following section.

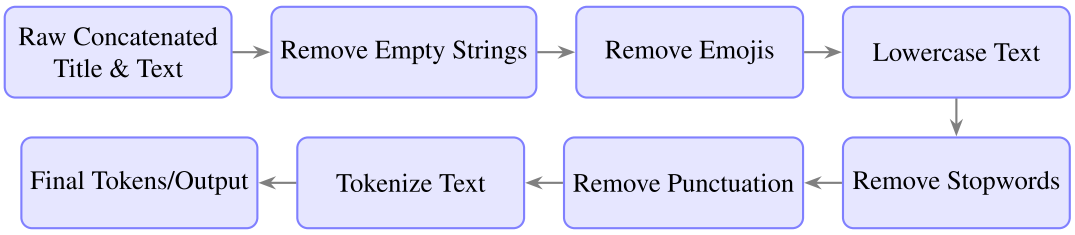

Fig. 1: Text Preprocessing Pipeline for NLP Tasks

3.1. Remove Empty Strings

Some entries in the data lack the review. We can check for them and remove them using the following code:

Listing 1: Removing empty strings code

```python
print(df_balanced[df_balanced["review_text"] == ""].head()) # We got 73 rows
df_balanced = df_balanced[df_balanced["review_text"] != ""]
```

3.2. Remove Emojis

A lot of reviews use emojis to convey their emotions. While they carry a lot of weight, they add complexity to the task at hand. For the sake of keeping it simple, we will remove them.

Listing 2: Removing emojis

```python
def remove_emojis(data: str) -> str:
  emoj = re.compile("["
      u"\U0001F600-\U0001F64F"  # emoticons
      u"\U0001F300-\U0001F5FF"  # symbols & pictographs
      u"\U0001F680-\U0001F6FF"  # transport & map symbols
      u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
      u"\U00002500-\U00002BEF"  # chinese char
      u"\U00002702-\U000027B0"
      u"\U000024C2-\U0001F251"
      u"\U0001f926-\U0001f937"
      u"\U00010000-\U0010ffff"
      u"\u2640-\u2642"
      u"\u2600-\u2B55"
      u"\u200d"
      u"\u23cf"
      u"\u23e9"
      u"\u231a"
      u"\ufe0f"  # dingbats
      u"\u3030"
                  "]+", re.UNICODE)
  return re.sub(emoj, '', data)

df_balanced["review_text"] = df_balanced["review_text"].apply(remove_emojis)
```

3.3. Remove Stopwords

Usually, stopwords like 'this', 'is' and 'are' do not add any information regarding sentiments. We will remove them while retaining some using NLTK's English 'stopwords'.

Listing 3: Removing stopwords code

```
import nltk

nltk.download('stopwords')
stop_words = set(stopwords.words('english')) # Faster search, sets are hash-
    based
stop_words.discard('not') # A few stopwords are important to keep
stop_words.discard('such')
stop_words.discard('but')
stop_words.discard('all')
stop_words.update(['it', 'br', "it's"]) # Stopwords that are not in the dict

def remove_stopwords(data: str, stop_words: set) -> str:
  return ' '.join([word for word in data.split() if word not in stop_words])

df_balanced["review_text"] = df_balanced["review_text"].apply(lambda x:
    remove_stopwords(x, stop_words))
```

3.4. *Remove Punctuation*

While they may better emotions better, punctuation certainly adds more complexity to the text. We will also be removing them using string's punctuation.

Listing 4: Removing punctuation code

```
from string import punctuation

def remove_punctuation(data: str) -> str:
  clean_text = data.translate(str.maketrans('', '', punctuation))
  return clean_text

df_balanced["review_text"] = df_balanced["review_text"].apply(lambda x:
    remove_punctuation(x))
```

3.5. *Tokenize Text*

Now, all that's left is to tokenize our reviews. For this, we will use the pre-defined 'word_tokenize' function from NLTK.

Listing 5: Tokenizing code

```
from nltk.tokenize import word_tokenize

df_balanced['tokens'] = df_balanced['review_text'].apply(word_tokenize)
```

4. Exploratory Data Analysis

It's time to explore our dataset. It's important to have a good grasp of the data, even if we'll end up using the textual columns only down the line.

4.1. *Text-Related EDA*

4.1.1. Word Frequency Analysis

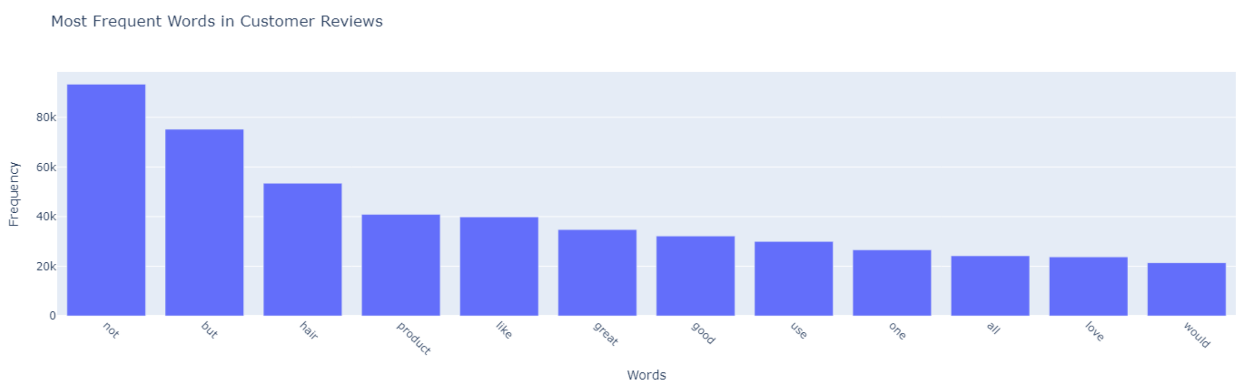

Fig. 2: Top 12 Words

4.1.2. Tri-grams Analysis

Listing 6: Tri-grams code

```
    from nltk.util import ngrams
    from collections import Counter

    words = ' '.join(df_balanced['review_text']).split()
    bigrams = Counter(ngrams(words, 3))
    bigrams.most_common(10)

>>> [(('would', 'not', 'recommend'), 999),
     (('five', 'stars', 'love'), 894),
     (('not', 'worth', 'money'), 832),
     (('five', 'stars', 'great'), 761),
     (('don''t', 'waste', 'money'), 649),
     (('three', 'stars', 'not'), 623),
     (('would', 'not', 'buy'), 507),
     (('one', 'star', 'not'), 462),
     (('love', 'love', 'love'), 458),
     (('waste', 'money', 'not'), 392)]
```

- The most frequently occurring tri-gram, ‘would not recommend’, appears 999 times and strongly reflects negative sentiment. Similarly, phrases such as ‘not worth money’, ‘don’t waste money’, and ‘would not buy’ indicate user dissatisfaction and caution, often aligned with low product ratings. These n-grams typically express regret, disapproval, or negative evaluation.
- Conversely, tri-grams such as ‘five stars love’, ‘five stars great’, and the emphatic ‘love love love’ are indicative of highly positive sentiment, with users explicitly affirming their satisfaction. These patterns often accompany maximum rating scores and represent strong endorsements.
- Some tri-grams, such as ‘three stars not’, appear to occupy an intermediate sentiment space, possibly suggesting moderate satisfaction or nuanced criticism. The presence of negation terms (‘not’, ‘don’t’, ‘waste’) in multiple high-frequency tri-grams further supports their utility as features for sentiment classification.

4.2. Other EDAs

4.2.1. Ratings Distribution

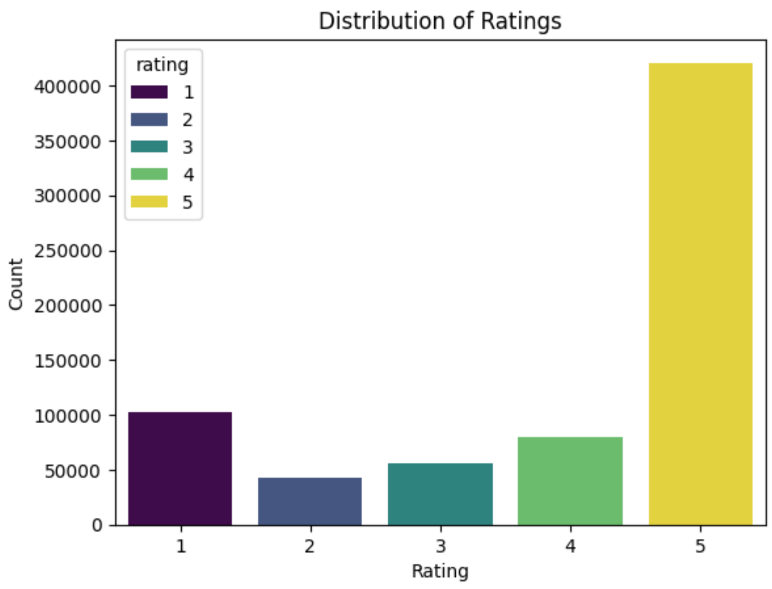

Fig. 3: Ratings Distribution

4.2.2. Ratings Distribution by Year

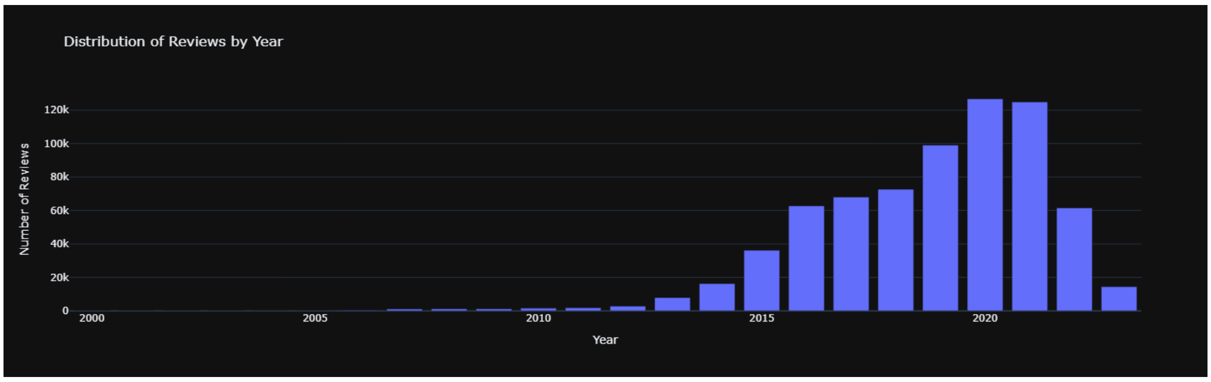

Fig. 4: Ratings Distribution

- With the rise of internet and online stores, many people around the world started shopping online. The numbers gradually increased over the years and so did the reviews. 2020, marked by the global pandemic SARS-CoV-2 (Covid 19), saw the highest number of Amazon reviewers at around 126K, followed by 2021 at near 125K. This is only logical since about everyone was confined to their homes and weren't allowed to go out, local stores and retailers ran out of stock and/or closed up.
- Although Amazon nowadays is notorious for botted reviews, the objective of today is to classify the reviews based on sentiments, not detect wether a review is real or not.

4.2.3. Verified Purchases Pie Chart

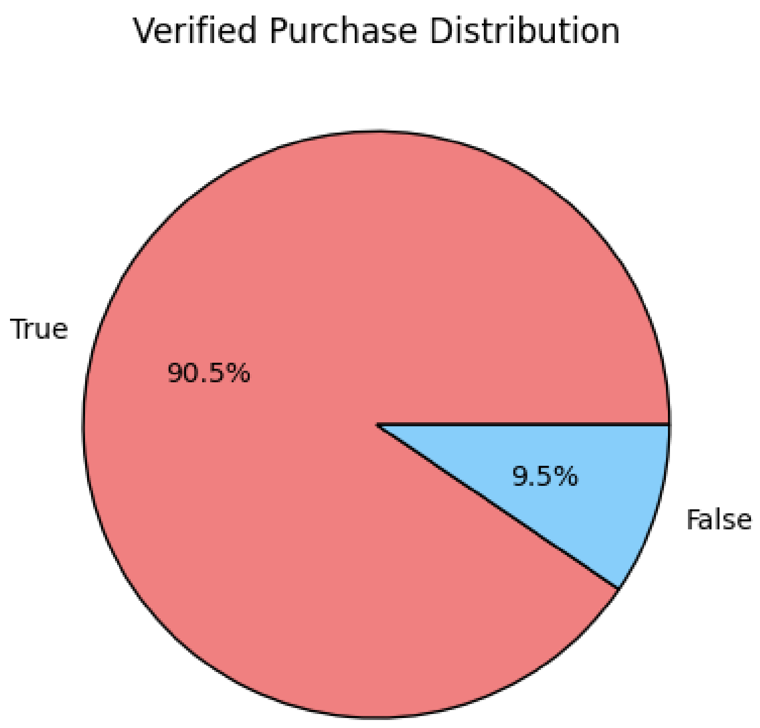

Fig. 5: Verified Purchase Pie Chart

The pie chart illustrates the proportion of verified versus unverified purchases within the dataset. A majority of the reviews correspond to verified purchases, indicating that most feedback comes from users who actually bought the product through Amazon. This enhances the credibility of the dataset for sentiment analysis tasks, as verified purchase reviews are generally more reliable indicators of genuine user experience compared to unverified ones.

5. Machine Learning Models

The training will be done on the following hardware:

- CPU: AMD Ryzen 5 5600h 3.3Ghz / 4.2Ghz Turbo six-core processor
- GPU: nVIDIA RTX 3050 Ti 4GB
- RAM: 16GB DDR4 3200 Mhz
- Storage: OEM Samsung M.2 NVMe
- OS: Windows 10 Pro 22H2 build 19045.5608

5.1. TF-IDF Approach

TF-IDF, which stands for **Term Frequency-Inverse Document Frequency**, is a statistical approach that is used in NLP tasks to evaluate the importance of a word in a document relative to a corpus (a collection of documents). The formulas are as follows:

$$\text{TF}(t, d) = \frac{\text{Number of times term t appears in document d}}{\text{Total number of terms in document d}}$$

$$\text{IDF}(t, D) = \log \frac{\text{Total number of documents in corpus D}}{\text{Number of documents containing term t}}$$

Fig. 6: TF and IDF formulas

- **TF** aims to determine the frequency of a term in a document that, alone, does not account for its global importance across the corpus. Also, common stopwords like *the* or *and* may have high scores but are not meaningful.
- **IDF** highlights the importance of rare words by reducing the weight of common words. And through the logarithmic formula, the IDF scores scale appropriately. However, it should be noted that a term might be rare across the corpus but irrelevant in a specific document.

We will start by creating our TF-IDF model first. Then, we will split our data into training and test sets as follows:

Listing 7: TF-IDF code

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# Model creation
tfidf = TfidfVectorizer(max_features=7500)
X_tfidf = tfidf.fit_transform(df_balanced['review_text'])

X_tfidf_df = pd.DataFrame(X_tfidf.toarray(), columns=tfidf.
    get_feature_names_out())
y = df['label']

# Data splitting (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X_tfidf_df, y, test_size
    =0.2, random_state=42)
```

5.1.1. XGBoost

Our first model of choice is **eXtreme Gradient Boosting**, a machine learning algorithm based on gradient boosting, which combines the result of weak learners, typically decision trees, to create a strong predictive model. It improves upon standard gradient through performance optimization.

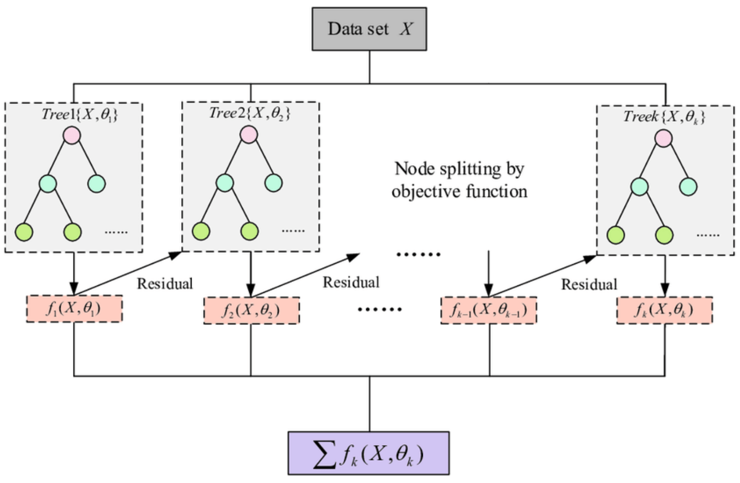

Fig. 7: XGBoost Architecture

Some of its hyperparameters include:

- ‘tree_method‘: The algorithm used for constructing decision trees (‘exact‘ for greedy construction, ‘approx‘, ‘hist‘ and ‘gpu_hist‘).
- ‘n_estimators‘: The number of trees to train.
- ‘learning_rate‘: The step size for each tree’s contribution to the final prediction.
- ‘max_depth‘: The maximum depth of each tree.
- ‘eval_metric‘: The evaluation metric used for validation (‘rmse‘ for regression, ‘logloss‘ and ‘mlogloss‘ for classification depending on the number of classes).
- ‘random_state‘: The seed for the random number generation, a fixed number ensures the same results can be reproduced every time.

This is how we’ll define our model:

Listing 8: XGBoost model code

```
import xgboost as xgb

xgb_tfidf_model = xgb.XGBClassifier(
    tree_method="gpu_hist", # Use GPU acceleration
    max_depth=6,
    learning_rate=0.1,
    n_estimators=300,
    random_state=42,
    verbosity=2, # Controls general logs
    eval_metric="mlogloss" # Multi-class log loss
)

xgb_tfidf_model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=True
    )
```

5.1.2. Random Forest

Random Forest is a derivative of Decision Tree, an ensemble learning algorithm that builds multiple decision trees and combines their predictions to improve accuracy and reduce overfitting. It trains each tree on a random subset of the data, also known as ‘bagging‘. The final prediction is made by majority voting (classification) or averaging (regression), making it robust and less prone to overfitting than individual decision trees. Its hyperparameters are similar to those of the XGBoost model.

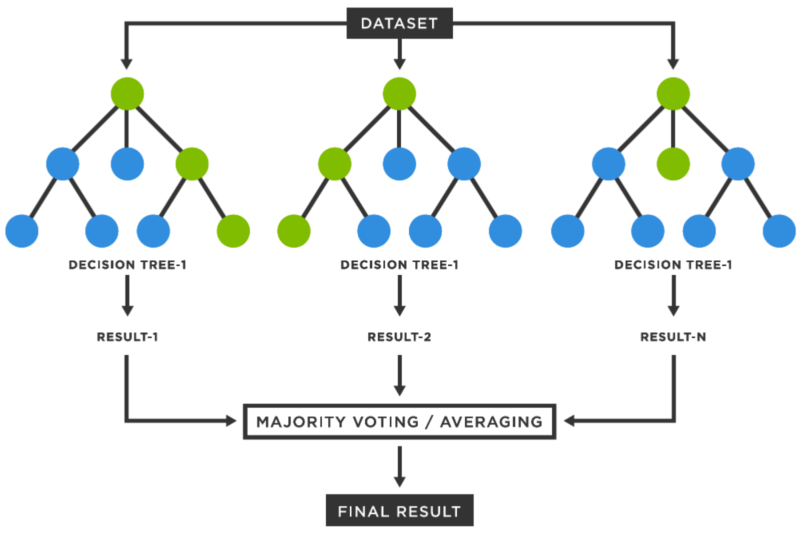

Fig. 8: Random Forest Architecture

Listing 9: RF model code

```
from sklearn.ensemble import RandomForestClassifier

rf_tfidf_model = RandomForestClassifier(n_estimators=300, max_depth=6,
    random_state=42)
rf_tfidf_model.fit(X_train, y_train)
```

5.1.3. Logistic Regression

A statistical model usually used for binary classification, logistic regression, has been adapted for multinomial tasks (multiclass classification). You can think of it as linear regression, but for classification tasks. It works by outputting the probability of a class. The following is the mathematical formula:

$$P(y=1|x) = \frac{1}{1+e^{-(w*x+b)}} \tag{1}$$

Where:

- $P(y=1|x)$ is the probability of class 1.
- x represent the input features.
- w are the weights.
- b is the bias.

For a multiclass task, the formula is adapted as follows:

$$P(y=k|x) = \frac{e^{w_k^T x + b_k}}{\sum_{j=1}^{K} e^{w_j^T x + b_j}} \tag{2}$$

Where:

- x is a feature vector.
- w_k, b_k are the weights and bias for class k
- K is the total number of classes.

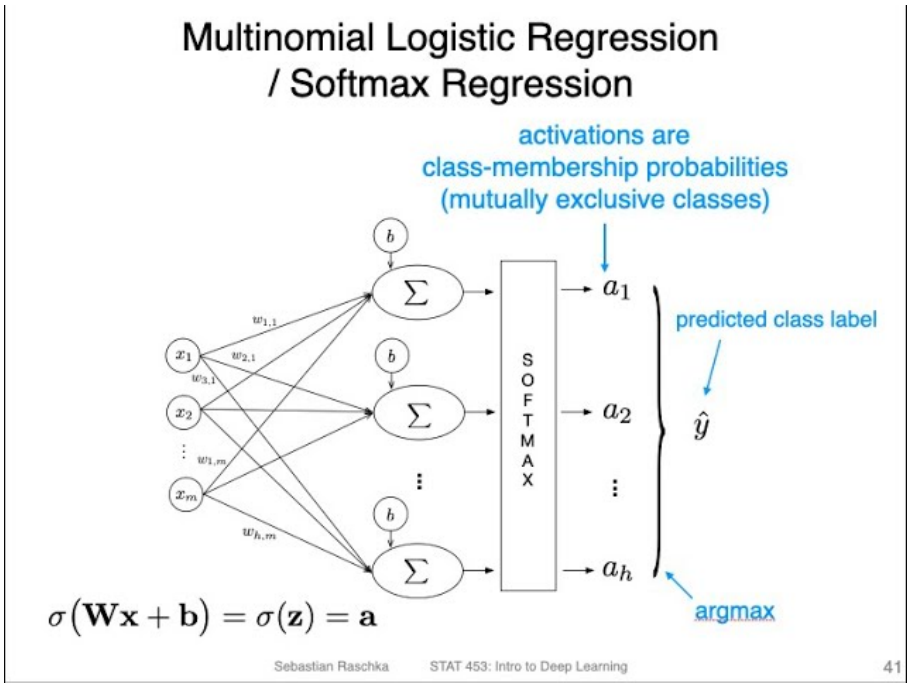
Multinomial Logistic Regression
/ Softmax Regression
activations are
class-membership probabilities
(mutually exclusive classes)
predicted class label
$\hat{y}$
argmax
x_1
x_2
x_m
$w_{1,1}$
$w_{2,1}$
$w_{3,1}$
$w_{1,m}$
$w_{h,m}$
b
Σ
SOFTMAX
a_1
a_2
a_h
$\sigma(\mathbf{Wx}+\mathbf{b}) = \sigma(\mathbf{z}) = \mathbf{a}$
Sebastian Raschka
STAT 453: Intro to Deep Learning
41

Fig. 9: Logistic Regression Architecture

Listing 10: LR model code

```
from sklearn.linear_model import LogisticRegression

lr_tfidf_model = LogisticRegression(max_iter=300, random_state=42)
lr_tfidf_model.fit(X_train, y_train)
```

5.1.4. Multinomial Naive Bayes

Multinomial Naive Bayes is a probabilistic classification algorithm used primarily for discrete count data, like word counts in text classification. It assumes that **features follow a multinomial distribution given its class and that said features are conditionally independent given the class**. Let's break it down for our case.
Say we have the following review (document):

- The document: ”great product great”
- Feature vector: ”great”: 2, ”product”: 1

You are in the **Positive** class. This assumes that there is a probability for ”great” independent from the probability of ”product” for the class. It also assumes that the document's word counts were generated based on those probabilities.
This is why this should work well with a TF-IDF approach. Let's try it below!

Listing 11: NB model code

```
from sklearn.naive_bayes import MultinomialNB

nb_tfidf_model = MultinomialNB()
nb_tfidf_model.fit(X_train, y_train)
```

5.1.5. K-Means

This algorithm will divide the dataset into a pre-defined number of classes (clusters to be more technical). It allows us to group data point such that points within each cluster are as similar as possible.

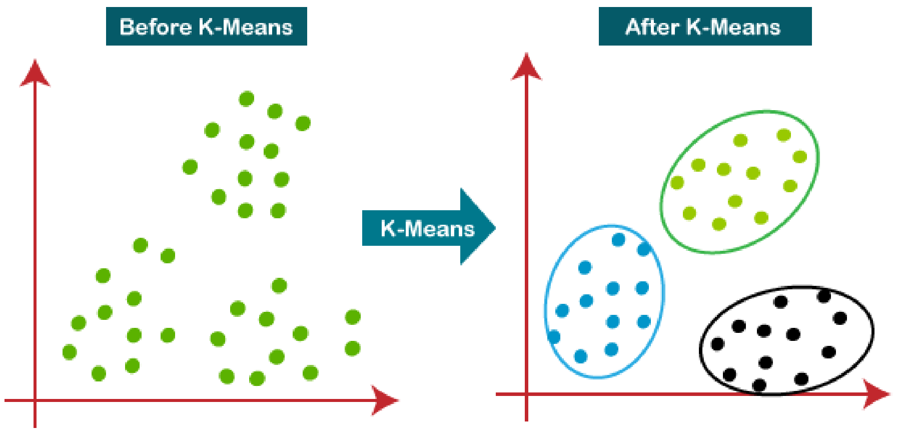

Fig. 10: K-Means Clustering

These are the steps for this algorithm:

1. Select random k initial centroids.
2. Assign each datapoint to the nearest centroid (it calculates euclidian distance by default).
3. Recompute the k centroids as the average of the points assigned to each cluster.
4. Repeat steps 2 and 3 until no major changes occur in the clusters or a certain number of iterations is reached.

Listing 12: K-Means model code

```
from sklearn.cluster import KMeans

num_clusters = 3
kmeans_tfidf_model = KMeans(n_clusters=num_clusters, random_state=10, n_init
    =10)
kmeans_tfidf_model.fit(X_train)
```

5.2. *Word2Vec Approach*

Instead of relying on term frequency, Word2Vec transforms words into dense vector representations that capture the semantic meaning. It uses a neural network trained on a large corpus to achieve this.
There are two main architectures of Word2Vec:

- CBOW (Continuous Bag of Words) which predicts a word from surrounding context.
- Skip-Gram which perdicts surrounding words from a target word.

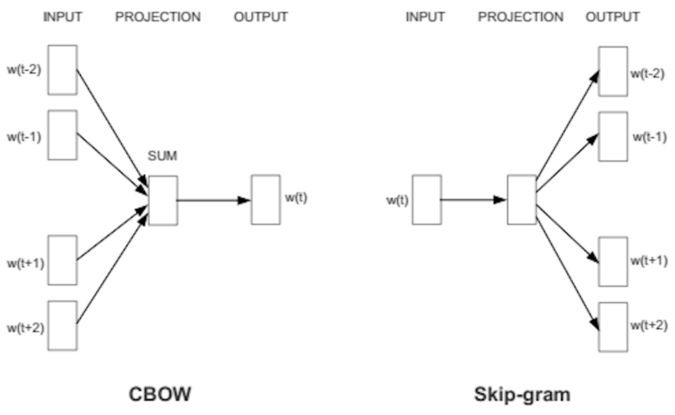

Fig. 11: Word2Vec Workflow

Listing 13: Word2Vec model code

```
from gensim.models import Word2Vec #!pip3 install gensim

sentences = df['tokens'].tolist()
word2vec_model = Word2Vec(sentences=sentences, vector_size=100, window=16,
    min_count=1, workers=4, epochs=10)
```

We can get the embeddings of a word or sentence afterwards, or run a similarity test. Let's give it a try:

Listing 14: Similarity test code

```
test_words = ["nice", "dislike", "okay",]

for word in test_words:
similar_words = word2vec_model.wv.most_similar(word, topn=2)
print(f"Similar word of '{word}': {similar_words}")

>>> Similar word of 'nice': [('good', 0.6386290192604065), ('great',
    0.5838353037834167)]
    Similar word of 'dislike': [('disliked', 0.5451518893241882), ('lacker',
        0.5166812539100647)]
    Similar word of 'okay': [('ok', 0.8634284734725952), ('alright',
        0.7491957545280457)]
```

As we can see above, the code returns the top two similar words based on their scores. The only issue is when a word is not present in the vocabulary, that would raise a KeyError. We will create a function to vectorize our sentences and return a numpy array, even if the words do not exist.

Listing 15: Vectorization function code

```
import numpy as np
from typing import Union

def get_sentence_vector(sentence: Union[str, list], model: Word2Vec) -> np.
    ndarray:
"""
Takes a sentence and created a vector of embeddings,
this function assumes that the sentence is already
pre-processed (e.g. lower text, removed emojis, stopwords, punctuation, etc.).

Args:
  sentence: either a list of tokens or a string.
  model: the Word2Vec model.

Returns:
  a numpy array representing the embedding vector
"""

if not isinstance(sentence, list):
    words = sentence.split()
else:
    words = sentence

# Collect word vectors for each word in the sentence
word_vectors = [model.wv[word] for word in words if word in model.wv]

# If no words found in model, return a vector of zeros
if len(word_vectors) == 0:
    return np.zeros(model.vector_size)

# Return the mean of the word vectors
return np.mean(word_vectors, axis=0)
```

Now, we are ready to embed our data and train our models. The same previous algorithms will be used.

Listing 16: Embedding data code

```
sentences = df['tokens'].tolist()
labels = df['label'].tolist()

X = np.array([get_sentence_vector(sentence, word2vec_model) for sentence in
    sentences])
y = np.array(labels)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
```

5.3. *Results & Evaluation*

In this section, we will compare the performance of our models with both TF-IDF and Word2Vec approaches by reporting the classification metrics and confusion matrices and then trying to classify unforeseen reviews correctly. The following reviews are to be used for prediction:

- Review 1 (Positive): I wasn't expecting much, but this exceeded my expectations. The build quality feels premium, and the instructions were easy to follow. Setup took under 10 minutes, and it's been working flawlessly ever since. I've used it daily without any issues. Definitely worth the price and would happily recommend it to anyone looking for a reliable option in this category.

- Review 2 (Neutral): It works as described, but there's nothing particularly special about it. The performance is fine for basic tasks, though I did notice some minor lag during more demanding use. Packaging was a bit flimsy, but the item arrived intact. If you're looking for something simple and functional, this will do, just don't expect premium quality or standout features.
- Review 3 (Negative): Pretty disappointed overall. The product didn't match the pictures and felt cheaply made. It stopped working properly after just a few uses, and customer service was slow to respond. I tried troubleshooting it myself, but nothing helped. For the price, I expected a much better experience. Wouldn't buy again, and I'd recommend looking elsewhere if you need something dependable.

Below is a table summarizing the weighted precision, recall and f1-score metrics for all our models:

Table 1: Weighted Precision, Recall & F1-Score Averages Across TF-IDF & Word2Vec

Model / Approach	**TF-IDF**	**Word2Vec**
XGBoost	(0.76, 0.76, 0.76)	(0.76, 0.76, 0.76)
Random Forest	(0.71, 0.71, 0.71)	(0.70, 0.70, 0.70)
Logistic Regression	(0.78, 0.78, 0.78)	(0.75, 0.75, 0.75)
Multinomial Naive Bayes	(0.71, 0.71, 0.71)	(N/A)

- Each model has shown decent overall performance, they seem to generalize well. The LR model scores the most across all metrics at 0.78 with TF-IDF, followed by XGBoost at 0.76 in both approaches. The scorE drops to 0.75 for LR with Word2Vec. However, it's still higher than RF and NB models.
- Note: Multinomial Naive Bayes is incompatible with dense vectors, especially if they have negative values, as is often the case with Word2Vec embeddings.

Table 2: Model Predictions for Unseen Reviews

Model / Approach	**TF-IDF**			**Word2Vec**		
	Review 1	Review 2	Review 3	Review 1	Review 2	Review 3
XGBoost	Positive	Positive	Positive	Positive	Neutral	Neutral
Random Forest	Positive	Neutral	Negative	Positive	Neutral	Neutral
Logistic Regression	Positive	Neutral	Negative	Positive	Neutral	Negative
Multinomial NB	Positive	Neutral	Negative	N/A	N/A	N/A

Each model was tasked with predicting the sentiment of three previously unseen reviews, representing positive, neutral, and negative sentiments respectively.

- XGBoost showed a bias toward positive sentiment, predicting "Positive" for all TF-IDF inputs and only partially adjusting with Word2Vec. Despite the shift to "Neutral" in Reviews 2 and 3 with Word2Vec, it failed to identify the negative sentiment.
- Random Forest and Logistic Regression captured the progression from Positive $\rightarrow$ Neutral $\rightarrow$ Negative more effectively under TF-IDF, aligning well with the actual order of sentiments.
- With Word2Vec, RF lost some sensitivity to negativity, with the model never predicting "Negative" at all.
- Multinomial Naive Bayes, compatible only with TF-IDF, performed quite well here, perfectly capturing the intended sentiment progression.

6. Deep Learning

In recent years, deep learning has revolutionized natural language processing (NLP) by enabling models to learn complex linguistic patterns directly from raw text. Unlike traditional machine learning methods, which rely heavily on handcrafted features, deep learning models—especially those based on neural networks—automatically learn high-level representations that capture the nuanced structure of language.
One of the most impactful advancements in this area is BERT [1] (Bidirectional Encoder Representations from Transformers), introduced by Devlin et al. (2018). BERT leverages a deep bidirectional Transformer architecture

to pre-train language representations on large corpora such as BooksCorpus and English Wikipedia. Its key innovation lies in its ability to consider the context from both the left and right of each word simultaneously, which significantly improves performance on a wide range of NLP tasks. Once pre-trained, BERT can be fine-tuned with just one additional output layer to achieve state-of-the-art results on tasks such as sentiment classification, question answering, and named entity recognition.

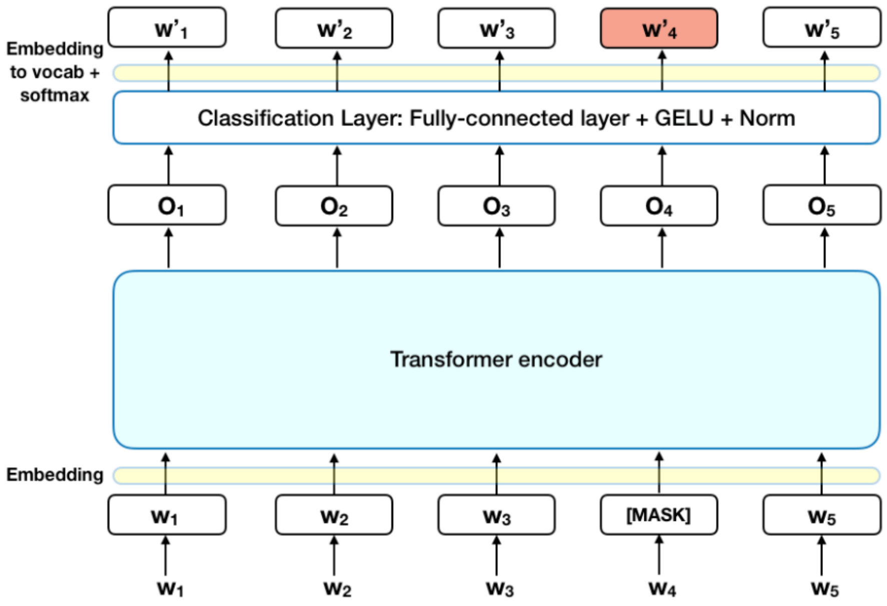
Embedding to vocab + softmax
w'1
w'2
w'3
w'4
w'5
Classification Layer: Fully-connected layer + GELU + Norm
O1
O2
O3
O4
O5
Transformer encoder
Embedding
w1
w2
w3
[MASK]
w5
w1
w2
w3
w4
w5

Fig. 12: BERT Architecture

6.1. Fine-Tuning BERT

To move beyond traditional machine learning pipelines, we fine-tuned a pretrained ”bert-base-uncased model” from HuggingFace Transformers library on our sentiment-labeled dataset. Fine-tuning allows the model to adjust its internal representations specifically to our task with minimal architectural changes.

1. Training Setup:
   - Model: bert-base-uncased with a classification head (3 output classes: Positive, Neutral, Negative)
   - Loss function: Cross-Entropy Loss
   - Optimizer: Adam
   - Batch Size: 32
   - Epochs: 5
   - Device: nVIDIA P100 16GB (Kaggle)
2. Results:
   - Validation Accuracy: 0.877
   - Validation Loss: 0.0217

The model achieved high performance compared to traditional ML models like Logistic Regression and Random Forest. It generalized well to unseen examples, accurately predicting sentiments even when phrasing or wording changed.

- Review 1 predicted sentiment: Positive
- Review 2 predicted sentiment: Neutral
- Review 3 predicted sentiment: Negative

7. Large Language Models

The model of choice for our work will be from Meta's Llama family, precisely Llama 3.2 3b Instruct [3]. It's a solid lightweight LLM with good out-of-the-box performance that can follow instructions properly and can run locally with no issues on our system. We will be applying few-shot prompting to classify our reviews.
We started by loading the model and its tokenizer from HuggingFace:

Listing 17: Model loading code

```
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-3.2-3B-Instruct")
model = AutoModelForCausalLM.from_pretrained(
"meta-llama/Llama-3.2-3B-Instruct",
torch_dtype=torch.float16,
device_map="auto" # automap to GPU if available
)
tokenizer.pad_token = tokenizer.eos_token #
```

We then build our prompt template as follows:

Listing 18: Few-shot prompt template

```
def build_prompt(review):
    return f"""
Classify the sentiment of the following review as Positive, Neutral, or
Negative.
Review: "This product is terrible and broke after one use."
Sentiment: Negative
Review: "It works as expected, nothing special."
Sentiment: Neutral
Review: "I love this! It's amazing and works perfectly."
Sentiment: Positive
Review: "{review}"
Sentiment:"""
```

Now, we create our classification function:

Listing 19: Classification function

```
def classify_review(review, max_new_tokens=10):
    prompt = build_prompt(review)
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)

    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=max_new_tokens,
            do_sample=True,
            pad_token_id=tokenizer.pad_token_id,
        )

    generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
    after_sentiment = generated_text.split("Sentiment:")[-1].strip()
    # Stop at the first line break (or anything resembling a new Review)
    sentiment = after_sentiment.split("\n")[0].split("Review:")[0].strip()
    return sentiment
```

We will be using the same reviews as before. These were the results:

- Review 1 predicted sentiment: Positive
- Review 2 predicted sentiment: Neutral
- Review 3 predicted sentiment: Negative

Aspect	Traditional ML	Deep Learning	LLMs
Preprocessing	High	Moderate	Minimal
Compute Needs	Low to Moderate	Moderate to High	High
Performance on Complex Sentiment	Moderate	High	Very High
Interpretability	High	Medium	Low
Training Time	Fast	Moderate	Slow

Table 3: Comparison between Traditional ML, Deep Learning, and LLMs for Sentiment Analysis

All these result were correct. The main takeaway from this is that we didn't need to pre-process our text as we previously did and no training is required. It performs well even with text containing emojis. You can check out the notebook on Kaggle here. There are some pre-requisites to use certain LLMs with HuggingFace, especially the Llama family:

1. Request access to the model.
2. Once granted, create an API.
3. If you're working on Kaggle, add the API to your secrets by pressing 'Add-ons'. You can also save it to your machine as an environment variable.
4. Authenticate using the following code:

Listing 20: HF authentication

```
#################################
            Kaggle
#################################

from kaggle_secrets import UserSecretsClient
from huggingface_hub import login

user_secrets = UserSecretsClient()
login(user_secrets.get_secret("HUGGINGFACE_TOKEN"))

#################################
          Local Machine
#################################

import os
from huggingface_hub import login

hf_token = os.environ["HUGGINGFACE_TOKEN]
login(hf_token)
```

8. Conclusion

This paper presented a comparative analysis of traditional Machine Learning (ML) models, classic Deep Learning architectures and Large Language Models (LLMs) for sentiment analysis tasks in Natural Language Processing (NLP). Our experiments highlight that while ML models offer speed, simplicity, and interpretability, they often require extensive feature engineering and fall short in handling nuanced or context-rich inputs.

On the other hand, LLMs such as LLaMA and many others demonstrate superior performance in understanding complex sentiment, thanks to their ability to capture contextual semantics. However, they come with significantly higher computational demands and lower interpretability.

Ultimately, the choice between ML, DL, and LLM approaches depends on the application context: ML models remain suitable for resource-constrained or real-time environments, whereas LLMs are ideal when accuracy and deep language understanding are paramount. Future work could explore hybrid approaches or distillation techniques to bridge this performance-efficiency gap.

References

1. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
2. Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. Bridging language and items for retrieval and recommendation. *arXiv preprint arXiv:2403.03952*, 2024.
3. Meta. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models, September 2024. Accessed: 2025-04-25.