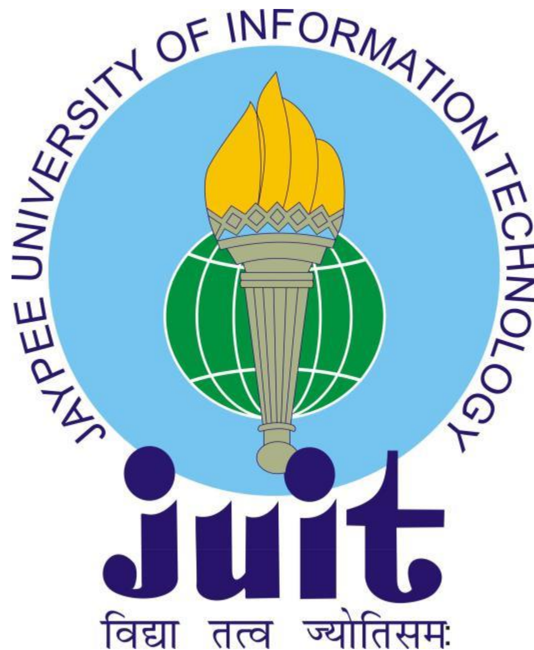
SENTIMENT ANALYSIS USING NLP
(INTERNSHIP REPORT)

Enrol. No. - 181322

Name of Student - Devansh Kaushik

Project Supervisor - Dr. Amol Vasudeva



May - 2022

Submitted in partial fulfilment of the Degree of

Bachelor of Technology

In

Computer Science Engineering

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING &
INFORMATION TECHNOLOGY
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, SOLAN

DECLARATION

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in text.

Date: 30 May 2022	Name:	Enroll. No:
	Devansh Kaushik	181322

CERTIFICATE

This is to certify that Devansh Kaushik was working as an intern with Paxcom since 7th February 2022 on developing different product related feature and also integrating various tools. This letter was issued on request from the employee and the company bears no responsibility or liability on behalf of the employee for any transaction that may arise.

Name of Supervisor: Sakshi Sehgal
Designation: Senior Data Scientist
Date: 27th May

ACKNOWLEDGEMENT

I have taken efforts in this internship. However, it would not have been possible without the kind support and help of many peers and organisation. I would like to extend our sincere thanks to all of them. I am highly indebted to Ms. Sakshi Sehgal for her guidance and constant supervision as well as for providing information regarding the internship and also for their support in completing the projects given during internship. My thanks and appreciations also go to my colleagues in developing the project and people who have willingly helped us out with their abilities.

TABLE OF CONTENTS

Chapter No. Topics	Page No.
Chapter – 1 Introduction	**1-3**

1. General Introduction
2. Problem Statement
3. Solution Approach

Chapter – 2 Libraries and Software Used	**4-7**

1. Python
2. Jupyter
3. Google Colab
4. LabelImg
5. Doccano
6. Pandas
7. NumPy
8. NLTK
9. Scikit-learn
10. Gensim
11. Transformers

Chapter – 3 Task Assigned Details	**8-31**

1. Dataset Description
2. Dataset Labelling
3. Pre-processing & Cleaning
4. Vectorizers
5. Word Embeddings
6. Word2Vec & GLoVe
7. Bert & ELMo
8. ML Algorithms
9. Boosting Algorithms

Chapter – 4 Conclusion	**32**

1. Conclusion
2. Future Work

Chapter – 5 References	**33**

LIST OF ABBREVIATIONS

1. ML – Machine Learning
2. NLP – Natural Language Pre-processing
3. CPU – Central Processing Unit
4. GPU – Graphics Processing Unit
5. BERT - Bidirectional Encoder Representations from Transformers
6. NLTK – Natural Language Toolkit
7. GloVe – GLObal VEctors for word representation
8. RAM – Random Access Memory
9. TF – Term Frequency
10. IDF – Inverse Document Frequency
11. Approx. – Approximately
12. ROBERTa - Robustly Optimized BERT Pretraining Approach
13. POS – Parts of Speech
14. ELMo – Embeddings from Language Model
15. SVM – Support Vector Machines
16. AVL - Adelson, Velski & Landis

LIST OF FIGURES

1. NLP Pipeline Broad Overview
2. Using python to code function which gets performance results and best model
3. LabelImg being used to label barcodes and QR codes
4. Labelling texts using Doccano
5. 2 Datasets (i) Biscuit Feedback (ii) Electronics Feedback
6. Labelling Function
7. Pre-processing Flow Diagram
8. Pre-processing Pipeline Code – 1
9. Stemming vs Lemmatization
10. Pre-processing Pipeline Code – 2
11. Pos Tagging
12. One Hot Encoding
13. TF IDF Formula
14. TF IDF Implementation
15. TF IDF Results
16. Word Embeddings
17. CBOW model
18. Skip-gram model
19. Word2Vec Results
20. GLoVe Embeddings
21. GloVe Results
22. ELMo Embeddings

23. ELMo Results

24. BERT Embeddings

25. BERT Results

26. Learning Curve

27. SVC

28. Pipeline Overview

29. Decision Tree

30. Random Forest

31. Boosting Algorithms

32. XG Boosting

33. Optimised Results

ABSTRACT

Paxcom India Pvt Ltd., is an e-commerce and product-based company with vision of becoming a leader in e-commerce analytics and automation across all key geographies. with the parent company as PaymentUs, an integrated payment platform. Amongst the many teams Paxcom consists of, such as IVR, e-commerce, ChatBot, I work in the ML team, which designs and takes on projects from clients as well as other Paxcom teams, which helps us increasing efficiency and utilizing man hours that can be saved by using a good model. The ML team works together and creates multiple models, like Ensemble Learning, extract either features from the dataset or makes a variety of algorithms as the features, thereby adding effective algorithms just as training samples and increasing model performance metrics (f1, accuracy, precision, depending upon project requirement). In the ML team, with the guidance and support of my team lead and colleagues, I have gained industrial exposure in NLP and have been working on hands-on projects on various datasets. Until now, A generalized Sentiment Analysis pipeline have been made which when given a raw unlabelled dataset as input, will label, pre-process, run across multiple algorithms, and finally give the best hyperparameter-tuned model with results and classification report. I have worked on annotations/labelling, blur classification, sentiment analysis and both supervised and unsupervised learning. With the help of my team, I have strengthened my skillset in the domain of ML and NLP both theoretically and hands-on, and will continue learning with deep neural networks currently. Hence this project will be about how I approached the sentiment analysis problem, what I learnt along the way and the respective performance and results of the same.

Chapter 1. INTRODUCTION

1. General Introduction

Sentiment Analysis, as the name suggests, it means to identify the view or emotion behind a situation. It basically means to analyze and find the emotion or intent behind a piece of text or speech or any mode of communication, whose use cases can vary from a feedback form to being used in automation based human emotion detection.

Polarity precision can be crucial when it comes to various businesses, for instance, when analyzing customer feedback in survey responses and conversations, one can learn to detect emotions and provide services and products accordingly. One of the ways to solve them is by using NLP (Natural Language Processing), a branch of computer science concerned with giving computers to understand vocabulary and grammar of texts.

Since the machine only understand zeroes and ones, we need to convert such grammatical phrases to numerical data while retaining the contextual information of the same, and then apply machine learning algorithms or deep learning networks, hence getting a model which can understand sentiment like humans, and hence, then be used in end user applications. Thus, in this internship, I tried to experience NLP a bit deeper and prepare myself for industry level projects from my learnings of this use case.

The pipeline includes complete generalization, labelling the dataset with state-of-the-art models, a series of preprocessing steps, then vectorizing the corpus or create embeddings, then running a Gridsearch on multiple ML and boosting algorithms with cross-validation, evaluating performance results and classification report and returning the best (non-overfitted) model.

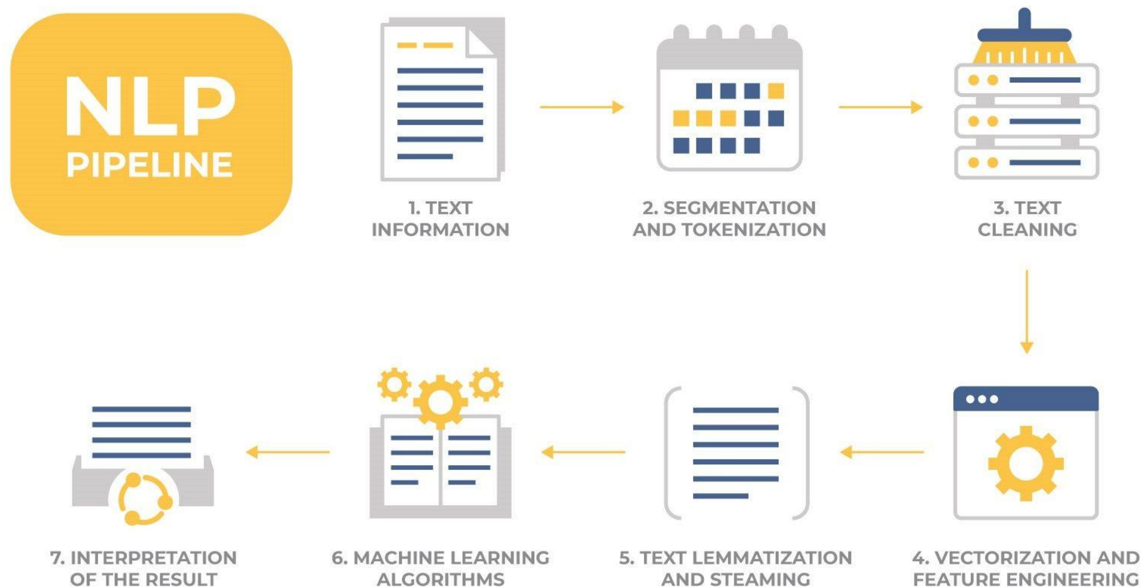


Fig 1. NLP Pipeline Broad Overview

The project includes analysis of the different preprocessing steps and vectorizers and algorithms used, and why specific algorithms and vectorizers/embeddings worked well than others.

2. Problem Statement

The problem statement is to how to parse through thousands of lines of text corpus or feedback survey records and detect human emotion and sentiment out of them, thereby efficiently utilizing time and manpower. Emotion detection has been surprisingly significant in businesses, and helps companies plan and tailor products according to the needs of end user.

3. **Solution Approach**

The solution is to create a generalized ML pipeline which when given an unlabelled / labelled dataset, will preprocess the data vectorize and run ML models on the data, and at the end of pipeline return the best hyperparameter tuned model, completely on its own with no human interference.

Chapter 2. LIBRARIES AND SOFTWARE USED

1. Python

Python is a high-level interpretable language, used widely when it comes to ML and API making. Python unlike C and Java, and minimize long codes into few lines with its rich libraries and functions with the use of space indentation.

```python
def save_models_and_results(root_dir: str, save_dict: dict = {}) -> None:
  os.mkdir(root_dir)
  for filename in save_dict.keys():
    with open(root_dir + filename, 'wb') as fid:
     pickle.dump(save_dict[filename], fid)

def save_textual_data(root_dir:str, filename:str, text):
  with open(root_dir + filename, 'w') as f:
    f.write(text)

def get_performance_results_and_best_model(transformed_X_train, y_train, transformed_X_test, y_test, classifier_dict = {}):
  max_train_acc, max_test_acc, best_model, bestmodelname, results_dict  = 0, 0, None, "", {}
  for classifier_class in classifier_dict.keys():
    modelname = str(classifier_class).split('.')[-1][:-2]
    print("Training set")
    train_acc, train_report, train_conf_matrix  = model_evaluation(transformed_X_train, y_train, classifier_dict[classifier_class])
    print("Testing set")
    test_acc,  test_report, test_conf_matrix = model_evaluation(transformed_X_test, y_test, classifier_dict[classifier_class])
    print(modelname + " Training acc:",train_acc)
    print(modelname + " Test acc:",test_acc)
    if train_acc>= 0.98 and test_acc <0.90:
      print(modelname + "(OVERFIT):\t" + str(train_acc))
    elif test_acc > max_test_acc:
      max_train_acc, max_test_acc, bestmodelname, best_model = train_acc, test_acc, modelname ,classifier_dict[classifier_class]
      print("Performance results, Classifier dictionary:\t", classifier_dict[classifier_class])
    results_dict[modelname] = {"train_acc": train_acc, "train_report": train_report, "train_conf_matrix": conf_matrix_to_string(train_conf_matrix),
                               "test_acc": test_acc, "test_report": test_report, "test_conf_matrix": conf_matrix_to_string(test_conf_matrix)}

  return best_model, bestmodelname, results_dict
```

Fig 2. Using python to code function which gets performance results and best model

All pipelines and pre-processings have been done on Python as it's tools and integrated systems help in analyzing models and visualizing performances easily and saves time.

2. Jupyter

Jupyter notebooks is a web based local host platform with variable coding environments for coding and data. It supports both CPU and GPU, hence one can use their remote PC's system and RAM to run models here.

3. Google Colab

While Jupyter notebooks are good when it comes to accessing local RAM and saving remote data, Colab notebooks is also a web-based computing platform, but one can use Google's GPU and RAM for inferences. Notebooks and datasets can be either imported or saved at google drive and code data can be easily shared.

4. LabelImg

LabelImg is an open-source graphical image annotation tool, where you can emphasize meaningful areas of images that you want to label for classification.

Fig 3. LabelImg being used to label barcodes and QR codes

5. Doccano

Doccano is an open-source text annotation tool, analogous to LabelImg, just used in emphasizing texts and later indexed in numerical data for NLP models.

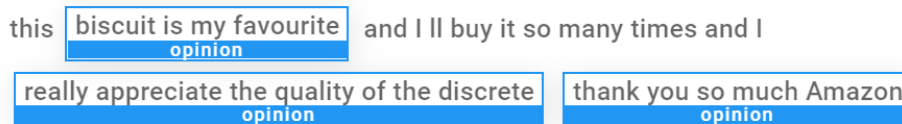


Fig 4. Labelling texts using Doccano

6. Pandas

Pandas is a python library used to structure and manipulate data using for instance, data frame and series. In this project, pandas will come in handy to tabulate data and apply operations.

7. Numpy

Numpy is a python library used for applying mathematical operations and transformations to n-dimensional data structures.

8. NLTK

NLTK (Natural Language Tool Kit) is a textual data pre-processing library used to structure and organize raw textual data for text analysis and visualizations. Functions such as tokenizer, lemmatizator and stopwords copurs are some of the NLTK functions that come in handy whehn pre-processing text.

9. Scikit-Learn

Scikit-Learn is a machine learning library for Python providing powerful preprocessing tools, various algorithms for regression and classification, feature selection and extraction, and dimensionality reduction functions.

10. Gensim

Gensim is an open-source library used for document indexing, and finding phrase/document similarity. It also consists of Word2Vec function, which as the name suggests, converts words and phrases to n-dimensional vectors, attempting to capture the context of phrases in a sentence or a document. These came as an improvement of count and TF-IDF vectorizers.

11. Transformers

Transformers is an open-source library by HuggingFace widely know for it’s state of the art NLP classification models for PyTorch, TensorFlow and JAX. The major use of transformers in our pipeline will be for BERT, a state-of-the-art model which will help us in labelling unlabelled data and providing word embeddings for corpus.

These were the libraries that were majorly used, other mentions could be of collections, xgboost, catboost, tensorflow and more.

Chapter 3. TASK ASSIGNED DETAILS

1. Dataset Description

There are 2 unlabelled datasets merged together for this situation, first has people who bought biscuits online feedback entries, second is of people who bought electronics online feedback entries.

Text
good material
nice texture
favourite biscuit
broken biscuit part biscuit opposite placed cream inside product delivery fail
taste
taste fresh
mummy
product good price high printed packet
sweetness
sugar make whole thing painless item mislead consumer pretext healthy
good quality
fibrous perfect go morning tea
like
now delicious
for taste crisp healthy
nice
taste healthy

review
The sound quality is good
The sound quality is good
Ok with cost but washing is not as expected
Better to purchase from flipkart bcos of good price.
Voice search option is not available which surprised me as my 32 inch Samsung smart TV remote has that feature. Otherwise, worth the money we pay for this TV.
Not for fast cooling
Nice
Supar
Better to purchase from flipkart bcos of good price.
Better to purchase from flipkart bcos of good price.
Nice
Ok with cost but washing is not as expected
Excellent product and easy to use. Love ?? it. Inverter sound is very less as compared to other product in this price range. Thanks Samsung & Flipkart.
Super

Fig 5. 2 Datasets (i) Biscuit Feedback (ii) Electronics Feedback

In total removing duplicates, there are 5000 text samples with 3 labels to classify: Positive (1), Negative (-1) and Neutral (0). The entries are mostly in English, with some outliers of Hindi and Tamil.

The problem with this dataset is, it is highly imbalanced with approx. 3500 entries of Positive labelled data, 1400 of negative and a 100 of neutral.

Hence instead of accuracy, this situation calls for focussing on f1 as a performance metric since even if the accuracy is high, that might also occur when our model is perfectly predicting the positive ones but wrongly predicting the neutral entries by a large difference.

2. Dataset Labelling

As the dataset is unlabelled, our first step should be labelling the data with a very strong model. This is where RoBERTa steps in, A Robustly Optimized BERT Pretraining Approach, which is based on Google's Bert model released in 2018.

It builds on BERT and modifies key hyperparameters, removing the next-sentence pretraining objective and training with much larger mini-batches and learning rates.

```
def label(df):
  df.drop_duplicates(keep = 'first', inplace = True)
  df.dropna(inplace=True)
  print("Dataframe stats")
  get_words_stats(df)

  print("Creating Label column..")
  df = df.assign(Label = [None]*len(df))

  print("Import transformers..")
  for i in range(len(df)):
    predicted_label = sentiment_analysis(df.iloc[i,0])[0]
    if predicted_label['label'] == 'NEGATIVE' and predicted_label['score']>=0.75:
      df.iloc[i,1] = -1
    elif predicted_label['label'] == 'POSITIVE' and predicted_label['score']>=0.75:
      df.iloc[i,1] = 1
    else:
      df.iloc[i,1] = 0
    if i%100 == 0:
      print("progress: ", i/len(df))
  return df
```

Fig 6. Labelling function

This pre-trained model itself takes care of stemming, lemmatizing, removing unknown symbols and stopwords. One might think of straight away use the given model for all kinds of problems, as honestly, it is the best. But the point to note is it is still a pretrained and a very large model, present on HuggingFace's cloud and remotely available for us, thus it takes a good amount of time to use ROBERTa and consequently, requires the need for making our own model tailored to our requirements.

Finally, the dataset is labelled and now requires manual review, as the model is best but migt not be matching our requirements of labelling.

3. Preprocessing & Cleaning

Preprocessing a textual dataset requires a series of steps for it to be prepared for vectorizing or in other words, converting it to numerical data.

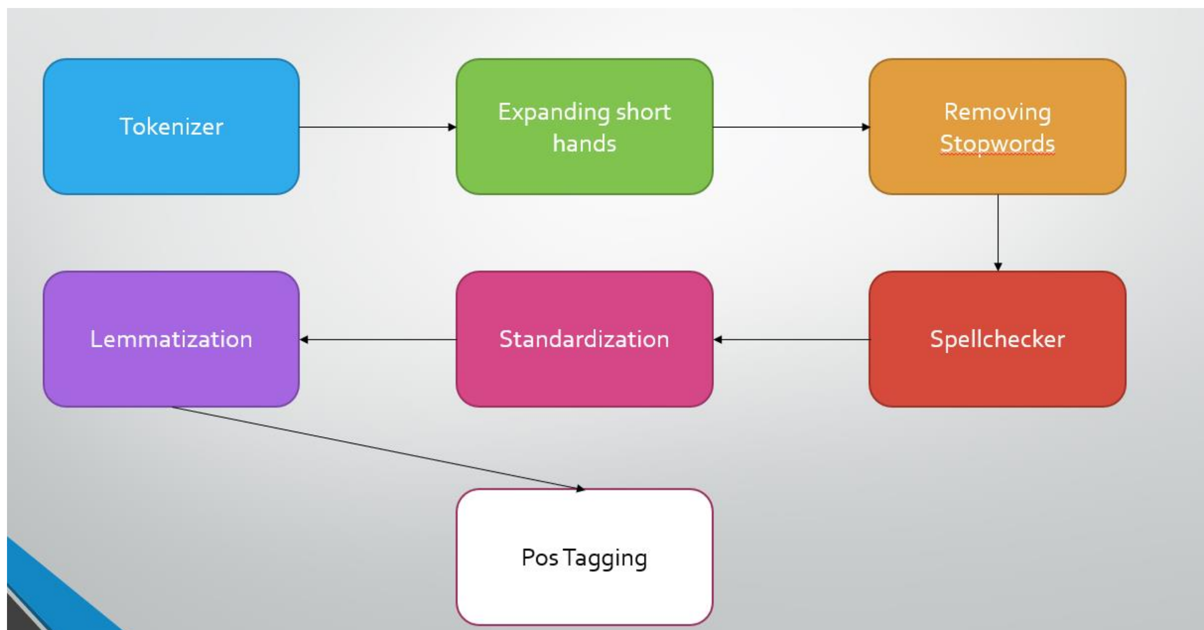


Fig 7. Pre-processing Flow Diagram

1. Removal of unwanted symbols: This can be done by limiting the ASCII numbers to small letters, capital letters and digits.

2. Tokenizer: Tokenizer creates a list of all words and letters from a sentence or paragraph, disregarding all punctuations and special characters. Tokenizer provided by NLTK can be of 2 types, sent_tokenize and word_tokenize. sent_tokenize is used to split a document or paragraph into sentences while word_tokenize is used to split a sentence into tokens.

Preprocess

```
[ ] def cont_expand(a, cont):
      expText = cont.expand_texts(a, precise=False)
      return expText

    def preprocess(df):
      print("Removing duplicates..")
      df['Text'].dropna(inplace=True)

      lem = WordNetLemmatizer()
      tokenizer = RegexpTokenizer(r'\w+')
      # print("Updating java..")

      # ("Installing pycontractions..")
      # !pip install pycontractions
      # ("Installing pycontractions..DONE")
      print("Importing glove-twitter-100..")
      cont = Contractions(api_key="glove-twitter-100")
      print("Importing glove-twitter-100.. DONE")

      print("Expanding shorthands.. DONE")
      print("Initializing nltk.corpus.stopwords dictionary..")
      sw = stopwords.words('english')
      re = ["no", "not", "nor", "same", "so", "some", "too"]
      for c in re:
        sw.remove(c)
      print("Initializing nltk.corpus.stopwords dictionary.. DONE")

      print("Expanding shorthands, Removing symbols..")
      for i in range(len(df)):
        df.iloc[i,0] = list(cont_expand([df.iloc[i,0]], cont))[0]
        tokens = tokenizer.tokenize(df.iloc[i,0])
        df.iloc[i,0] = " ".join(tokens)
        # print(df.iloc[i,0])
      print("Expanding shorthands, Removing symbols.. DONE")
```

Fig 8. Preprocessing Pipeline Code – 1

3. Expanding short hands: Expanding words such as don’t, can’t, it’s is essential as it helps in recognizing them as stopwords and getting “not” word as a token, for negative entries.
4. Removing stopwords: Now as the words have been expanded, NLTK’s stopwords library can be used to recognize and remove the same.
5. Spellchecker: A good spellcheck such as TextBlob is used here to correct wrongly spelled words.
6. Standardization: People might use slangs obtained from social networking sites in their textual vocabulary, writing “good” as “gud”, “osm” for “awesome” are some of the common ones.
7. Lemmatization: Lemmatization is bringing the token back to its actual form, for example, “studies” to “study”. While stemming is also widely used, it has not been

applied here as stemming doesn't give the actual word, it removes affixes out of words, for example, "studies" to "studi", which is not base word. Lemmatization gives us the base word.

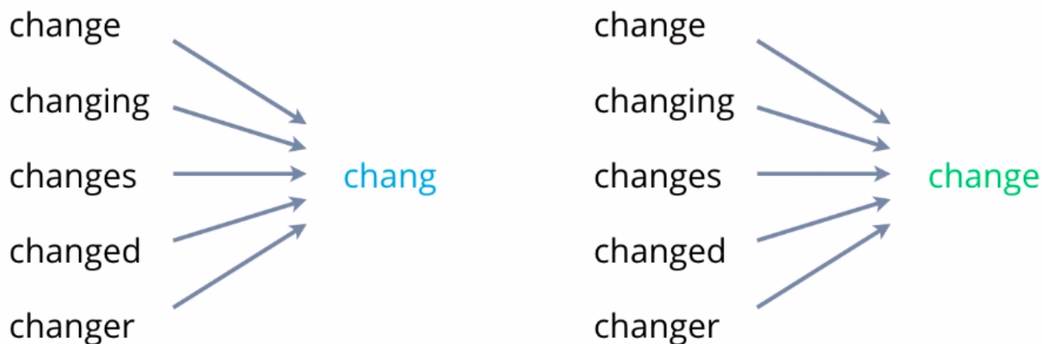


Fig 9. Stemming vs Lemmatization

```python
print("Initiating Spellcheck.. ")
df.Text = df.Text.apply(lambda txt: ''.join(TextBlob(txt).correct()))
print("Initiating Spellcheck.. DONE")

print("Standardizing.. ")
lookup_dict = { 'bikis': 'biscuits', 'buiscuit': 'biscuit', 'buiscuits': 'biscuits', "nyc": "nice", "osm": "awesome", "ny": "nice", "bed": "bad",
               'gud': "good", 'okay':'ok', 'favorite':'favourite', "test": "taste", "luv" :"love", "testy" : "tasty", "flavour": "flavor", "tast": "taste"}
for i in range(len(df)):
  # sw  = set(sw)
  tokens = tokenizer.tokenize(df.iloc[i,0])
  # tokens = [w for w in tokens if not w.lower() in sw]
  tokens = [lookup_dict[w] if w in lookup_dict.keys() else w for w in tokens ]
  df.iloc[i,0] = " ".join(tokens)
  # print(df.iloc[i,0])
print("Standardizing.. DONE ")
df['Text'].dropna(inplace=True)

print("Initiating Lemmatization and PosTagging.. ")
li = ["RB", "NN", "NNS", "JJ", "VBD", "VBN", "VB", "VBZ", "IN", "RBR", "VBP", "VBG", "JJS", "RP", "JJR", "RBS", "UH"]
for i in range(len(df)):
  s = df.iloc[i,0]
  # print(s)
  tokens = tokenizer.tokenize(s)
  # tokens = tokenizer.tokenize(s)
  tokens = [lem.lemmatize(word) for word in tokens]
  df.iloc[i,0] = " ".join(tokens)
  tags = pos_tag(tokens)
  # print(df.iloc[i,0])
  ans = ""
  for j in range(len(tags)):
    if tags[j][1] in li:
      ans += tags[j][0] + " "
  ans = ans[:-1]
  df.iloc[i,0] = ans
print("Initiating Lemmatization and PosTagging.. DONE")

df['Text'].dropna(inplace=True)
return df
```

Fig 10. Preprocessing Pipeline Code - 2

8. POS Tagging: Finally, POS-tagging, recognizing and filtering words based on their categories of phrases, like removing proper nouns and keeping adjectives.

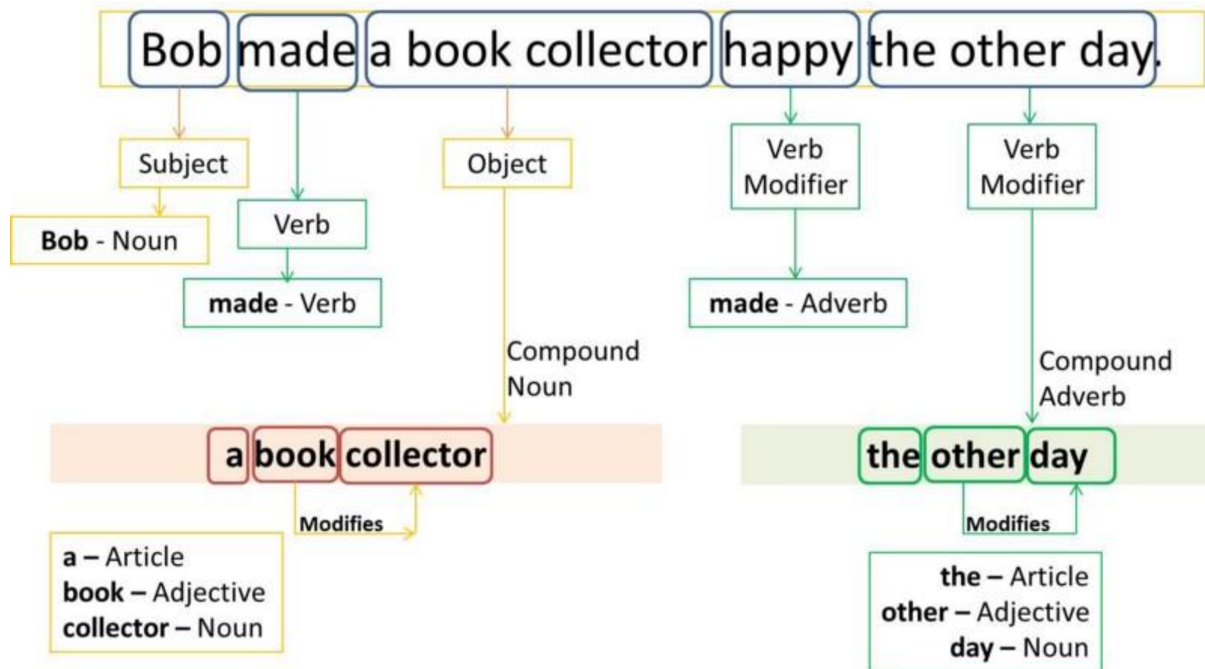


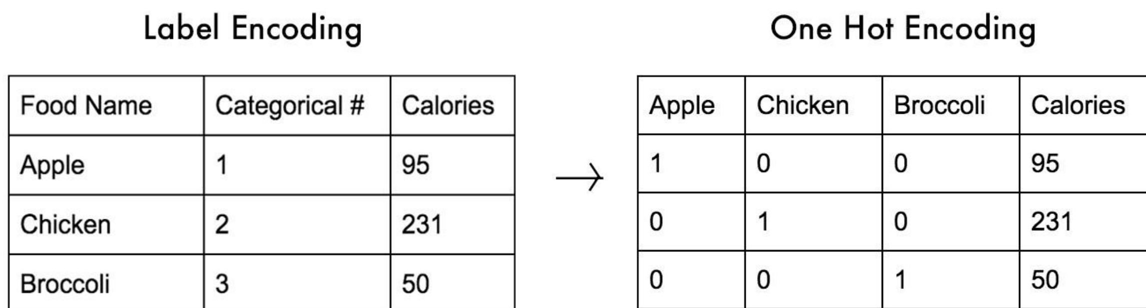
Fig 11. Pos Tagging

Thus, preprocessing is done, and now can remove empty and duplicate entries in the dataset, and move om to the next step, i.e., vectorizing.

4. Vectorizers

There are 2 types of vectorizers mainly used, Count Vectorizer and TF IDF Vectorizer.

1. One Hot Encoding: Giving each word a one while all others are giving index 0. Easiest implementation by far, but highly inefficient when it comes to dimensionality of matrix and memory usage.



Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

Fig 12. One Hot Encoding

2. Count Vectorizer: A simple vectorizer that provides indexes to all unique words, the count of words will accordingly be given higher or lower priority by the model.

   Again, a simple approach but with a lot of loopholes as it is incapable of identifying relationships between words and works purely based on number of occurrences in the whole document.

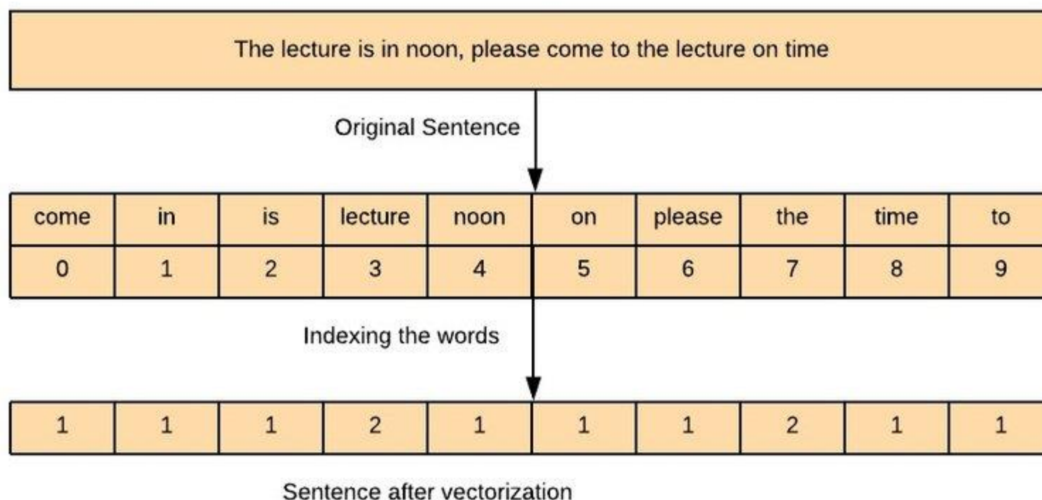


Fig 13. Count Vectorizer

3. TF-IDF Vectorizer: Term Frequency - Inverse Document Term frequency is based on not only on count of word in the document, but also takes in account number of times the word is occurring in a sentence.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Fig 14. TF – IDF Formula

This is a simple yet powerful tool, and in some cases work even better than Word2Vec, which takes contextual information in account too. But it has 2 edge cases to it:

ZERO VALUE ISSUE: which occurs because of the IDF calculation. Imagine we have three entries, and the word we are considering here is “cat”. In document A, cat makes up 70% of

the words. If cat appears once in document B and none times in document C, the TF-IDF value would be high. But if cat appeared once in C too, tf-idf would fall all the way to zero, which is a way too big contrast for just one occurrence change.

This is an undesirable feature as even though cat is a very important word here, TF IDF says otherwise.

EXTENSIVE MARGIN ISSUE: again because of the IDF portion. This occurs because TF IDF does take account of number of times the word cat occurs in the document, but doesn't take in account number of times it occurs in other documents.

Consider the same scenario taken above. Now suppose cat makes up 5% of words in document A. If cat didn't occur at all in document B, TF IDF value would be high, but if it came once in B, TF IDF would again, fall down like a hero to zero.

Hence, even in this situation even though it condition is almost identical, TF IDF shows complete change of behavior just by ne word difference. Hence, TF IDF is sometimes not desirable when it comes to vectorizing.

The traits of TF IDF can be changed by giving proportional importance to words, but why do that when we can move to contextual words vectorizing in methods such as Word2Vec and Glove?

```
[ ] X = df.iloc[:,0]
    y = df.iloc[:,1]

    print(X.shape,y.shape)
    X_train, X_test = train_test_split(df, test_size=0.2, random_state=42, stratify = y)

    y_train = X_train["Label"].values
    y_test = X_test["Label"].values

    (4979,) (4979,)

[ ] print(type(X_train["Text"]), len(X_train["Text"]), len(X_test["Text"]))

    <class 'pandas.core.series.Series'> 3983 996

▾ TfIdf

[ ] from sklearn.feature_extraction.text import TfidfVectorizer
    vectorizer = TfidfVectorizer(max_features=5000, ngram_range=(1,2))
    corpus =  X_train["Text"].values.astype('U')
    vectorizer.fit(corpus)

    TfidfVectorizer(max_features=5000, ngram_range=(1, 2))

[ ] train = vectorizer.transform(corpus)
    dic_vocabulary = vectorizer.vocabulary_
```

Fig 15. TF IDF Implementation

Model	Training Acc	Test Acc
Logistic Regression	93.79	88.75
Weighted Logistic Regression	98.4	87.9
SVM	95.1	89.2
Naïve Bayes	92.5	86.6
Decision Tree	98.5	82.1
Random Forest	98.4	88.5
Gradient Boosting	91.4	86.1
XGBoost	87.2	85.2
LightGBM	92.8	87.3
CatBoost	89.6	85.8

Without gridSearchCV
Best Model
Great Accuracies
Overfitting

Fig 16. TF IDF Results

In the case TF IDF, SVM showed the best results a possible explanation for that is **SVM performs well at small datasets** whereas the other algorithms such as boosting and

ensemble learning algorithms (decision tree & random forest), they are data hungry and hence perform great at large datasets.
SVM works well when there is a clear set of separation and it uses a subset of training points in the decision function (called support vectors), so it is also memory efficient. We will talk about models later.

5. Word Embeddings

Word Embeddings basically, are n-dimensional (n specified by the programmer) vectors used for representation of words, another way to convert text to numerical data. Before this even though vectorizers had no dimensionality and just single digit values, they were completely unknown to the context. Word embeddings can be used to take context into account and hence, making the model altogether better.

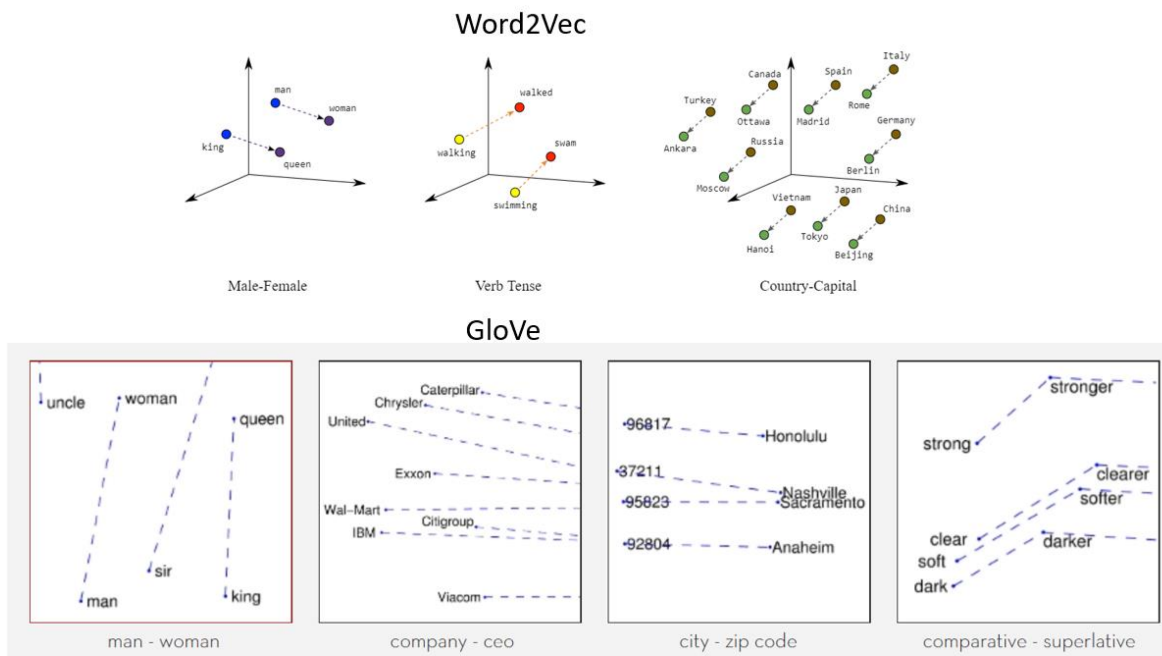


Fig 17. Word Embeddings

The most famous example of contextual embeddings is:

$$Queen = King - Male + Female$$

There are 4 types of embeddings analysed on the pipeline:

1. Word2Vec
2. Glove
3. Bert
4. ELMo

Bert and ELMo are currently the best working embeddings currently, and are available as pretrained models. But we for our use, will only use their embeddings in this case.

6. Word2Vec & Glove Embeddings

Word2Vec is a semi-supervised learning technique, like every other neural network. It is supervised if you consider that the network has to learn from backpropagation, unsupervised if you consider that no human expert makes labels. For instance, KNN for K-nearest neighbors is a classification algorithm that in order to determine the classification of a point combines the classification of K-nearest points. It is supervised because if you are trying to classify a point based on the known classification of other points.

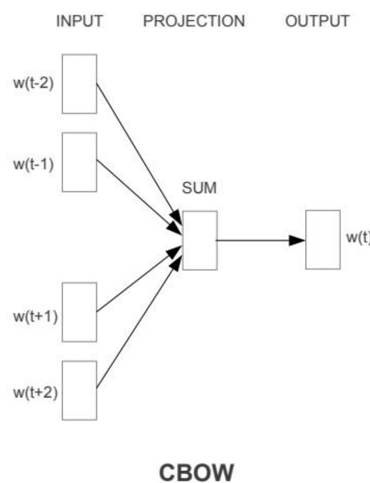


Fig 18. CBOW model

Word2Vec produces a produces a vector space, typically of several hundred dimensions, with each unique word in the corpus such that words that share common contexts in the corpus are located close to one another in the space. That can be done using 2 different approaches: starting from a single word to predict its context (Skip-gram) or starting from the context to predict a word (Continuous Bag-of-Words).

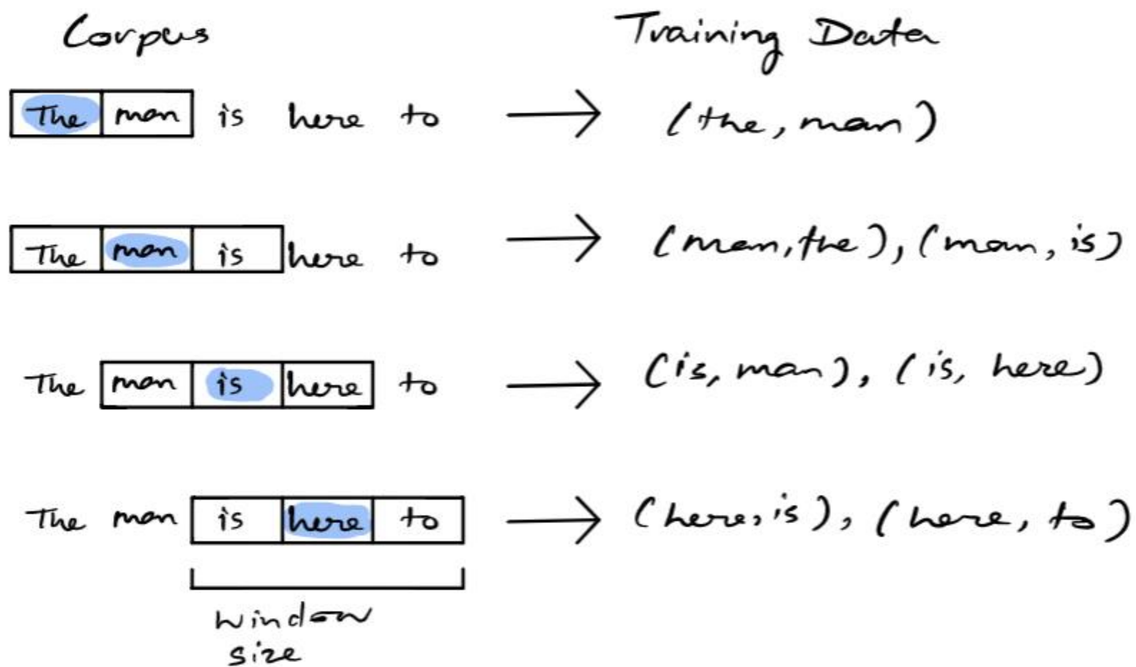


Fig 19. Skip-gram model

Hence, the neural network is fully used to predict the next word or predict the words around it, either way we will only be taking the network weights as embeddings and don't require the full model.

Also, we will be preferring CBOW over Skip-gram here because firstly, CBOW trains faster and can better represent frequent words, mainly because it works on context and maximizes probability of target word by looking at context hence, making it easier to work on frequently occurring words than the rarely occurring ones. And secondly, CBOW gave better results on analysis.

Gradient Boosting	0.82	0.7

Training Set Conf Matrix	precision	recall	f1-score	support
-1	0.5	0.91	0.65	637
0	0.46	0.93	0.62	42
1	0.98	0.8	0.88	3095
accuracy			0.82	3774
macro avg	0.65	0.88	0.72	3774
weighted avg	0.89	0.82	0.84	3774

Test Set Conf Matrix	precision	recall	f1-score	support
-1	0.65	0.53	0.58	156
0	0	0	0	3
1	0.75	0.81	0.78	261
accuracy			0.7	420
macro avg	0.46	0.45	0.45	420
weighted avg	0.7	0.7	0.7	420

XG Boosting	0.81	0.69

Training Set Conf Matrix	precision	recall	f1-score	support
-1	0.58	0.81	0.67	821
0	0.13	0.92	0.23	12
1	0.94	0.81	0.87	2941
accuracy			0.81	3774
macro avg	0.55	0.85	0.59	3774
weighted avg	0.86	0.81	0.83	3774

Test Set Conf Matrix	precision	recall	f1-score	support
-1	0.53	0.52	0.52	132
0	0	0	0	0
1	0.78	0.77	0.78	288
accuracy			0.69	420
macro avg	0.44	0.43	0.43	420
weighted avg	0.7	0.69	0.7	420

Fig 20. Word2Vec results

As you can observe, Word2Vec results are not as good as TF IDF, mainly because of TF IDF takes care of rarely occurring words as well, while Word2Vec CBOW doesn't predict well at rarely occurring words.

Also, once Word2Vec is applied or trained, it can't be trained again. Hence every time a new word comes in, Word2Vec will try to give an embedding accordingly in the n-dimensional vector space to the word.

Word2vec relies only on local information of language words and proves suboptimal, we will see how other embeddings prove over Word2Vec in this case. Also, it might be possible that in our case the occurrences of words had a good effect on the model, thereby TF-IDF giving good results and Word2Vec unable to.

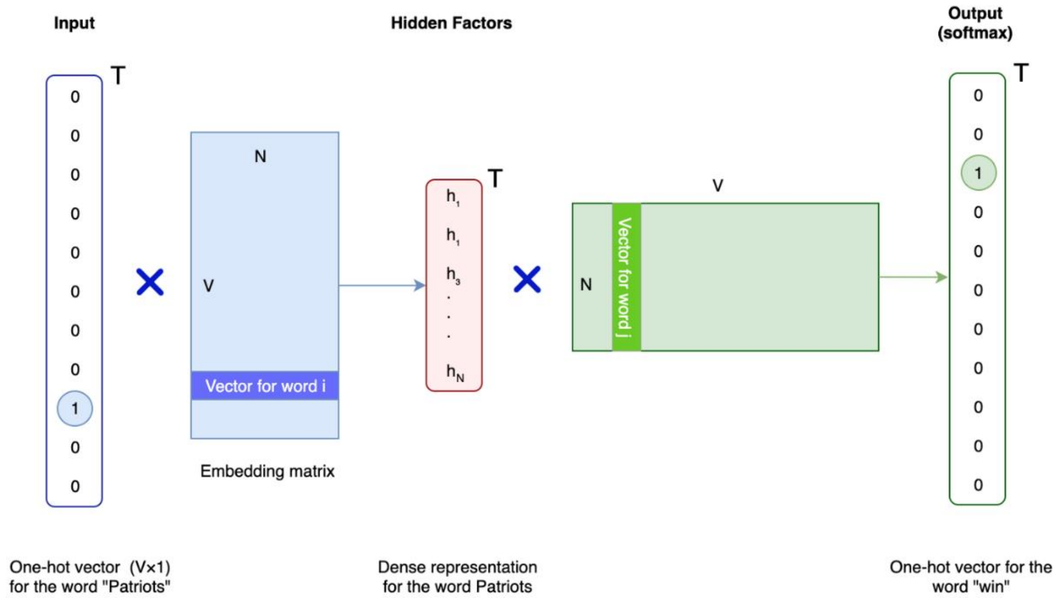


Fig 21. GloVe Embeddings

Enter **GLoVe**, the model which considers both contextual and statistical information of a corpus. In other words, it tries to capture both the count vectorizer technique (or co-occurrence matrix) and Word2Vec's prediction-based technique (word embeddings).

Dataset: ABSA											
Loigstic Regression	0.928	0	0.9	0.92	0.91		0.91	0	0.89	0.9	0.89
		1	0.95	0.93	0.94			1	0.93	0.92	0.93
SVM	0.8	0	0.54	0.96	0.69		0.88	0	0.95	0.81	0.88
		1	0.98	0.76	0.85			1	0.85	0.96	0.9
Multinomial NB	0.62	0	0.38	0.55	0.45		0.41	0	1	0.41	0.58
		1	0.79	0.65	0.71			1	0	1	0.01
Decision Tree	0.99	0	0.99	0.99	0.99		0.75	0	0.64	0.73	0.68
		1	0.99	0.99	0.99			1	0.84	0.77	0.8
Random Forest	0.99	0	0.99	0.99	0.99		0.72	0	0.57	0.69	0.63
		1	1	0.99	0.99			1	0.82	0.74	0.78
Gradient Boosting	0.95	0	0.94	0.95	0.94		0.85	0	0.73	0.89	0.8
		1	0.97	0.96	0..96			1	0.94	0.83	0.88
XG Boosting	0.99	0	0.99	0.99	0.99		0.88	0	0.82	0.88	0.85
		1	1	0.99	0.99			1	0.93	0.88	0.9
CatBoost	0.94	0	0.92	0.94	0.93		0.87	0	0.83	0.86	0.85
		1	0.96	0.95	0.95			1	0.91	0.88	0.9

Fig 22. GLoVe Results

And look at what we have here, better results as soon as occurrence of words was taken into account. Thus, the reasons were correct, number of occurrences is important in this dataset and GLoVe improved over Word2Vec using the same. Even the overfitted results can be improved over by some Hyper-parameter Tuning and vectorization.

Also, GLoVe is already trained over a big corpus, so just download the embeddings and you have shorter training time compared to Word2Vec. This is great, but what if the embeddings didn't only rely on the neighbours?

7. BERT & ELMo Embeddings

Until now in Word2vec and Glove embeddings, we had to get sentence embeddings out of word embeddings by adding and averaging them out. An effective method, but not an ethical one as it doesn't represent the sentence at all.

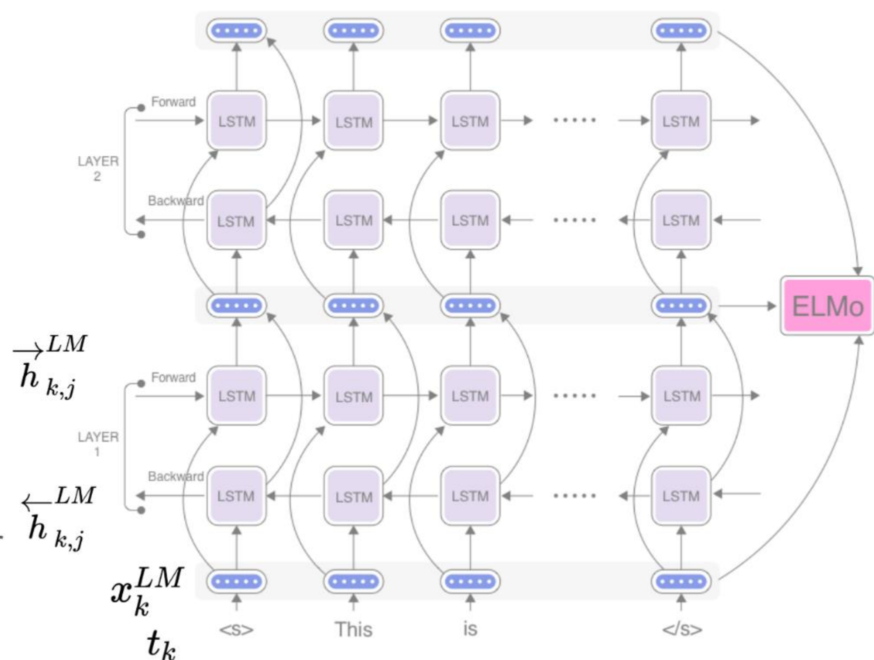


Fig 23. ELMo Embeddings

ELMo embeddings, tries to solve the disadvantages of Word2Vec and Glove by bringing in contextual information in embeddings too, hence the weighted sum of the word vector and 2 intermediate word vectors gives us the resultant word vector.

Gradient Boosting	0.95	0	0.94	0.95	0.95		0.89	0	0.86	0.87	0.86
		1	0.97	0.96	0.96			1	0.91	0.9	0.91
XG Boosting	0.95	0	0.94	0.95	0.94		0.89	0	0.87	0.86	0.87
		1	0.96	0.96	0.96			1	0.91	0.91	0.91
CatBoost	0.94	0	0.92	0.94	0.93		0.89	0	0.88	0.86	0.87
		1	0.96	0.95	0.95			1	0.9	0.92	0.91

Fig 24. ELMo Results

The results are still good, contextualized embeddings are better then isolated embeddings as they do not grasp the environment around the same.

BERT on the other hand, has its own way of doing things. BERT creates a mask first, SEP mask for separating lines and CLS at the beginning of text.

For UNK or unknown tag, BERT doesn't use it this time. Rather, it will split the unknown word into small pieces, and make embeddings out of it. For instance, if there was an embedding present for the word "embed" and a new "embedding" word has come now, it will create the embeddings for "ding" and "embed#####", showing that it is similar to embed but might have been used differently, look out.

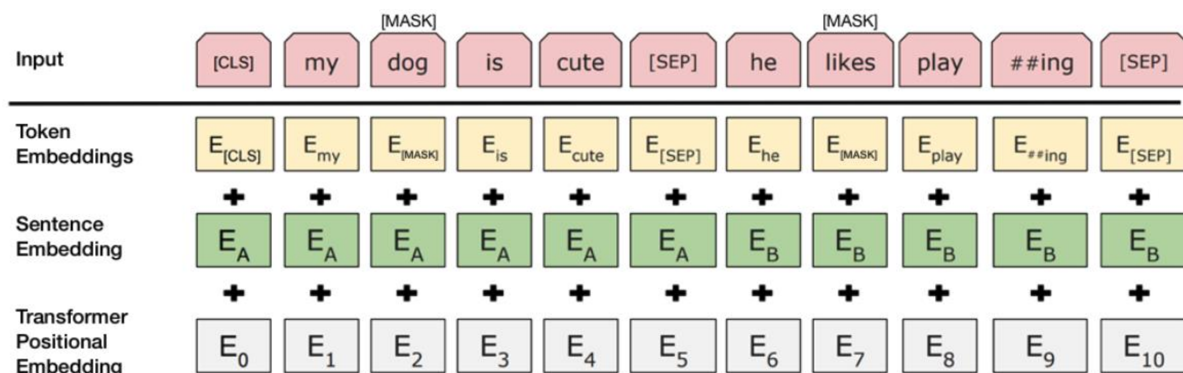


Fig 25. Bert Embeddings

That’s why BERT is called state-of-the-art, it prepares itself for all the edge cases it can out there.

Model	Training Set Acc.		precision	recall	f1-score		Test Set Acc.		precision	recall	f1-score
Dataset: ABSA											
Logistic Regression	0.72	0	35	1	0.52		0.75	0	0.42	0.97	0.58
		1	1	0.68	0.74			1	0.99	0.7	0.82
SVM	0.68	0	0.98	0.57	0.72		0.63	0	0.95	0.54	0.69
		1	0.47	0.97	0.63			1	0.41	0.92	0.56
Multinomial NB	0.61	0	0.51	0.55	0.53		0.63	0	0.51	0.57	0.54
		1	0.7	0.66	0.68			1	0.72	0.67	0.7
Decision Tree	0.99	0	0.99	1	0.99		0.9	0	0.88	0.88	0.88
		1	1	0.99	1			1	0.91	0.91	0.91
Random Forest	0.99	0	0.99	1	0.99		0.93	0	0.92	0.92	0.92
		1	1	0.99	1			1	0.94	0.94	0.94
Gradient Boosting	0.99	0	0.99	1	0.99		0.92	0	0.9	0.9	0.9
		1	1	0.99	0.99			1	0.93	0.93	0.93
XG Boosting	0.99	0	0.99	1	0.99		0.935	0	0.93	0.92	0.92
		1	1	0.99	1			1	0.94	0.95	0.94
CatBoost	0.983	0	0.97	0.99	0.98		0.915	0	0.94	0.87	0.9
		1	0.99	0.98	0.99			1	0.9	0.95	0.92

Fig 26. BERT Results

The results are better than ever, thus in our case. BERT performed the best with many models almost to the brim of perfection. Almost because if we observe, there are still some models with overfitting present.

In my duration of internship, I used to think that 90 training acc. and 80 testing acc. (also known as validation accuracy) means overfitting, but when it comes to deeper observation, here catboost is overfitting as well, with 98 training acc. and 91.5 testing acc.

Thus, now we will go deep into all algorithms and fine tune them, and prevent them for overfitting.

8. ML Algorithms

Logistic Regression

Logistic Regression in our case, didn't had any overfitting, but if it needed, L1 (Lasso), L2 (Ridge) and Elastic Net (combined Ridge and Lasso) are the parameters used for regularization.

- Lasso (L1): $\lambda \cdot |w|$
- Ridge (L2): $\lambda \cdot w^2$
- Elastic Net (L1+L2): $\lambda_1 \cdot |w| + \lambda_2 \cdot w^2$

Not all of them are compatible with all kinds of solver Logistic Regression has. For instance, LBFGS solver is only compatible with L2, not with L1 and Elastic Net. So, need to keep a track on that. The difference between L1 and L2 is that one is absolute lambda and other is squared lambda.

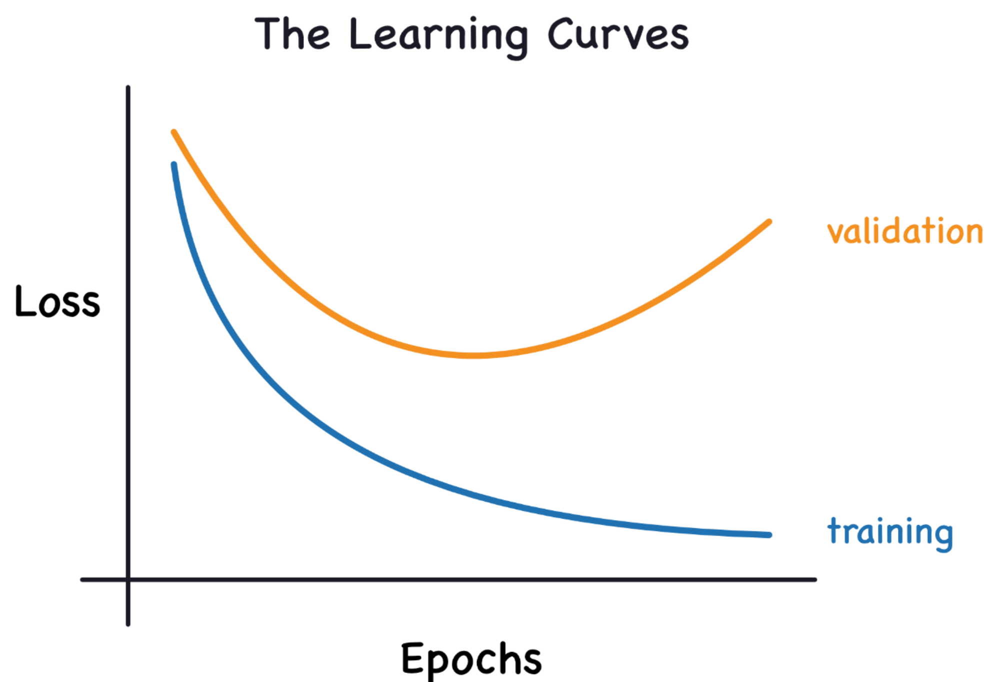


Fig 27. Learning Curve

Logistic and SVM were also underperforming because of MultinomialNB. When sending training data, whenever the pipeline got MultinomialNB in the models list, MultinomialNB has the requirement to only allow zero or positive values.

Hence when the dataset was scaled to positive exclusively for MultinomialNB and not for other models, the accuracy of both SVM and Logistic Regression rose up.

Hence, SVM didn't require any hyperparameter tuning. And MultinomialNB being purely based on probability, didn't have any parameters to tune.

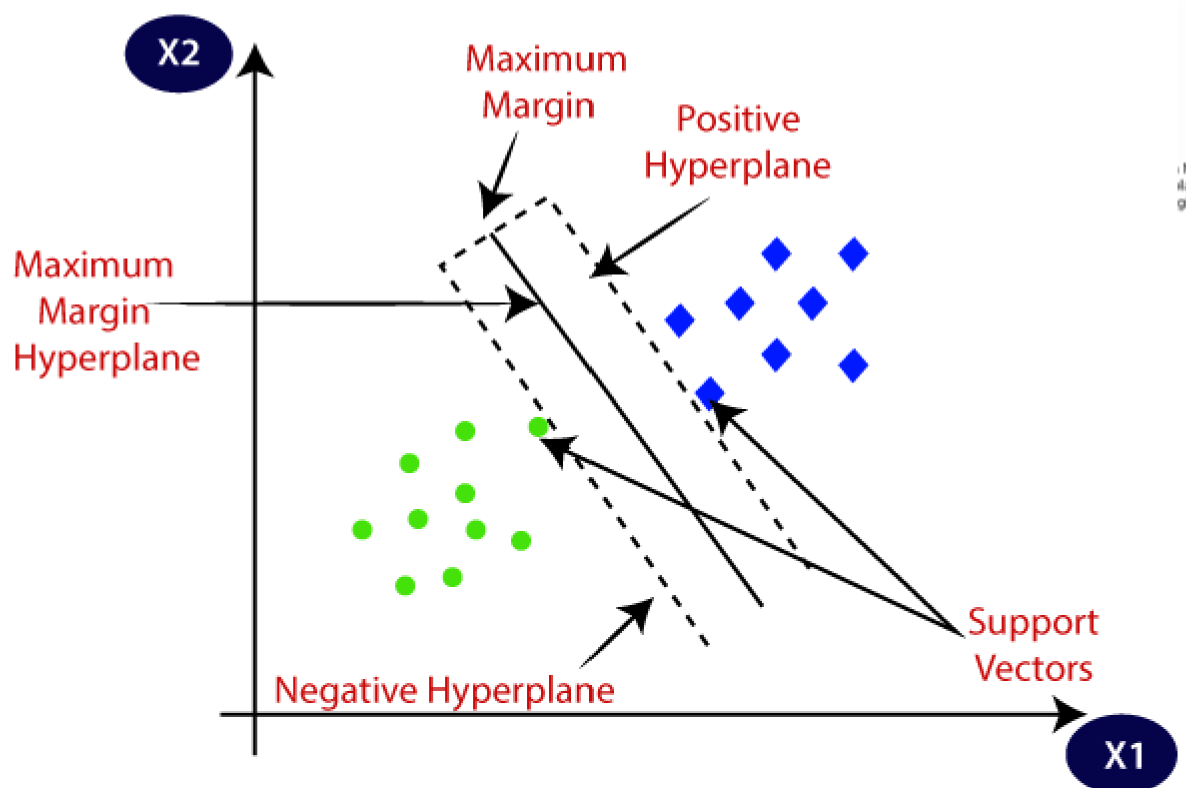


Fig 28. SVC

Decision Tree and **Random Forest,** always tend to overfit. This is because by default the scikit-learn's tree depth is mentioned as "None", so the tree naturally keeps splitting the nodes (training samples) until all of them aren't segregated.

```
#Train splitting parameters
test_size = 0.2
random_state = 42
classifier_inst_dict = { #LogisticRegression: {"penalty": ('l2', 'elasticnet', 'none'), "tol" : (None, 0.0001, 0.00001, 0.0005),
                           #                      'class_weight': ('balanced', None, {0:50,1:25,-1:25}, {0:20,1:40,-1:40})},
                        SVC: {"kernel": ("linear", "poly", "sigmoid"), "C" : (1, 5, 10)},
                        MultinomialNB: {"alpha": (0, 1, 2, 5, 10), "fit_prior" : (False, True)},
                        # DecisionTreeClassifier: {"criterion": ("gini", "entropy"), "max_depth": (None, 2, 5), "min_samples_split": (2, 3, 5)},
                        # RandomForestClassifier:  {"criterion": ("gini", "entropy"), "max_depth": (None, 2, 5), "min_samples_split": (2, 3, 5), "max_features": ("auto","sqrt")},
                        # GradientBoostingClassifier: {"loss": ("deviance", "exponential"), "learning_rate": (0.1, 0.5, 0.01), "n_estimators": (100, 200),
                                                        # "criterion": ("friedman_mse", "squared_error", "mse", "mae")},
                        # XGBClassifier: {"booster": ("gbtree", "gblinear", "dart"), "binary": ("logistic", "logitraw"), "lambda": (0, 0.01, 0.5), "alpha": (0, 0.01, 0.5)},
                        # CatBoostClassifier: {"iterations": (50), "learning_rate": (None, 0.01, 0.05, 0.1)}
                        }

classifier_inst_parameters = { #LogisticRegression: {"max_iter": 1000},
                              SVC: {"cache_size": 500},
                              MultinomialNB: {},
                              # DecisionTreeClassifier: {},
                              # RandomForestClassifier: {},
                              # GradientBoostingClassifier: {},
                              # XGBClassifier: {},
                              # CatBoostClassifier: {}
}
```

Fig 29. Pipeline Overview

Decision Tree has multiple parameters that can be tuned:

- Criterion: Gini or Entropy
- Max_depth: Change “None” to a value and observe how it stops overfitting
- Min_samples_split: Just like an AVL tree, the node will simple into more only if the training samples in the node are equal or above the given criteria.
- Min_samples_leaf: Minimum number of training samples that should be a present for a node to be present.

These were the criterions I tuned and got appropriate results, there are more that can be looked into.

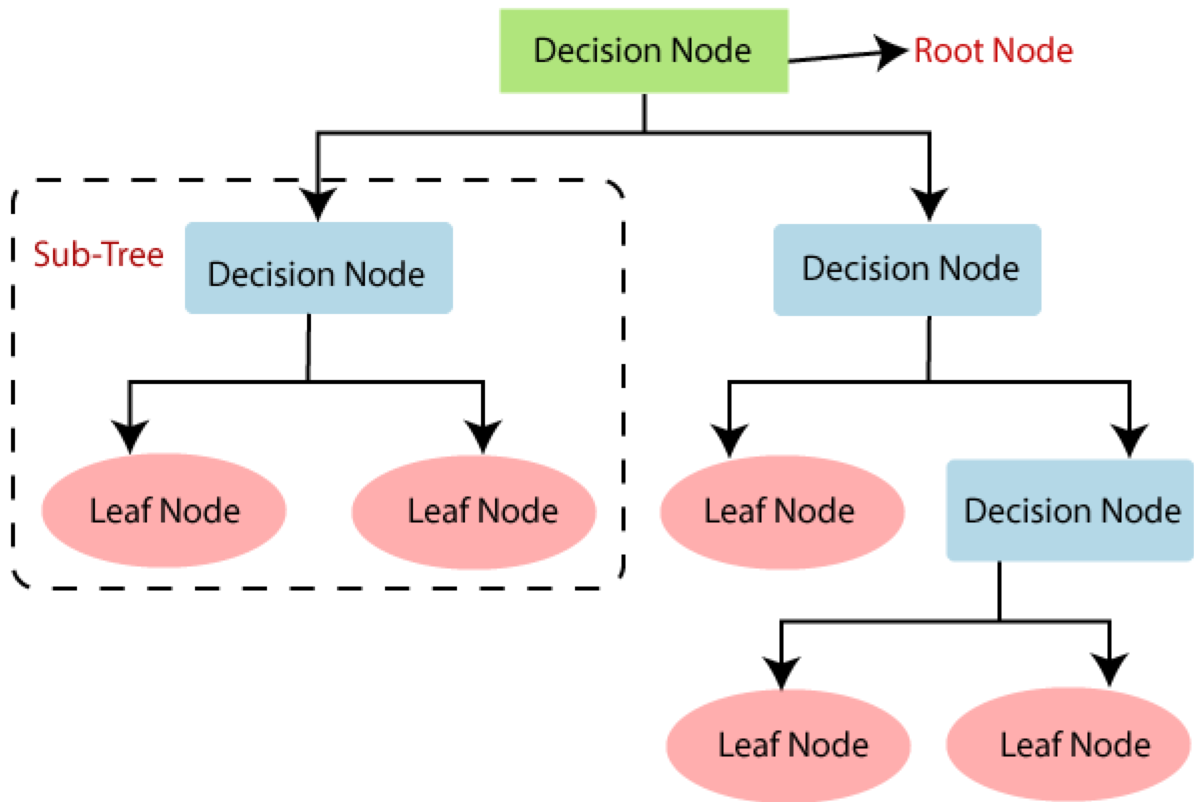


Fig 30. Decision Tree

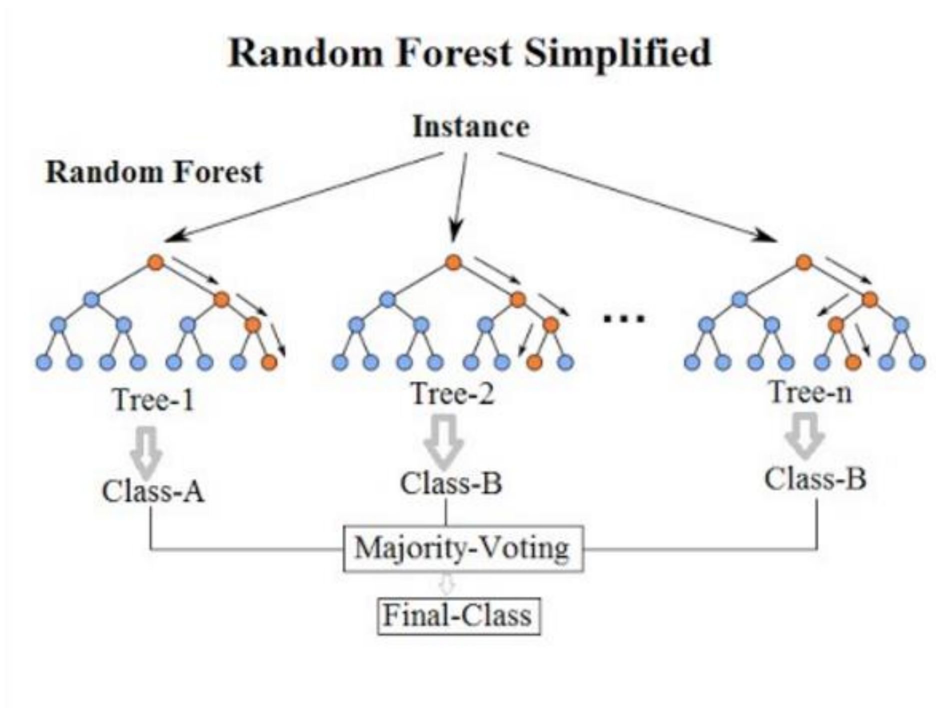


Fig 31. Random Forest

9. Boosting Algorithms

Boosting Algorithms are said to take weak learns and combine them into a strong one, but what is the difference between Decision Tree algorithms and Boosting algorithms?

Ensemble Learning is joining multiple models to solve one problem, Bagging is a way to decrease the variance in the prediction by generating additional data for training from the dataset using combinations with repetitions to produce multi-sets of the original data. Boosting is an iterative technique which adjusts the weight of an observation based on the last classification. If an observation was classified incorrectly, it tries to increase the weight of this observation.

Random Forest comes under bagging, while XGboost, Catboost and more come under boosting algorithms.

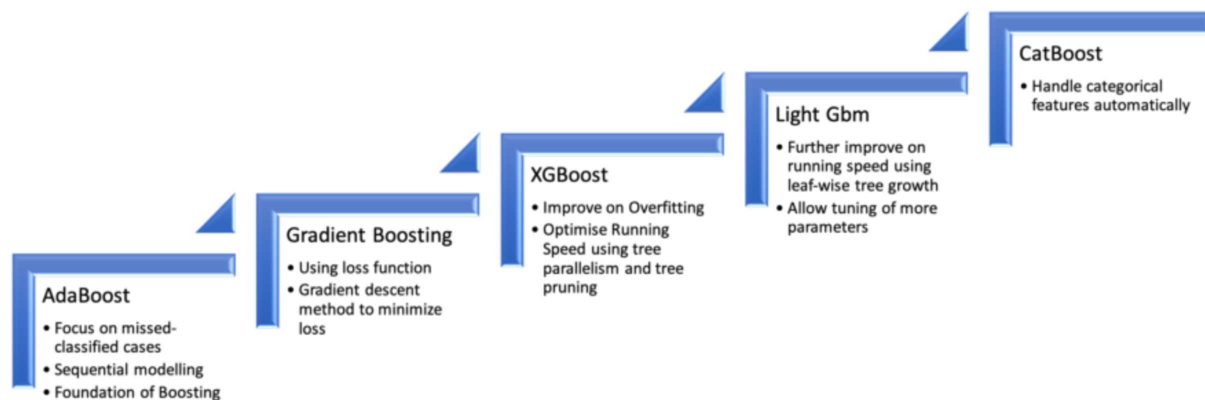


Fig 32. Boosting Algorithms

Even though Boosting Algorithms have trees, they have a different way to make trees compared to Decision Tree. Surely, they do have decision-based trees, yet they have good optimisation techniques to the loss function. While Random Forest randomly takes random subsets and work on them and returns the best subset of trees.

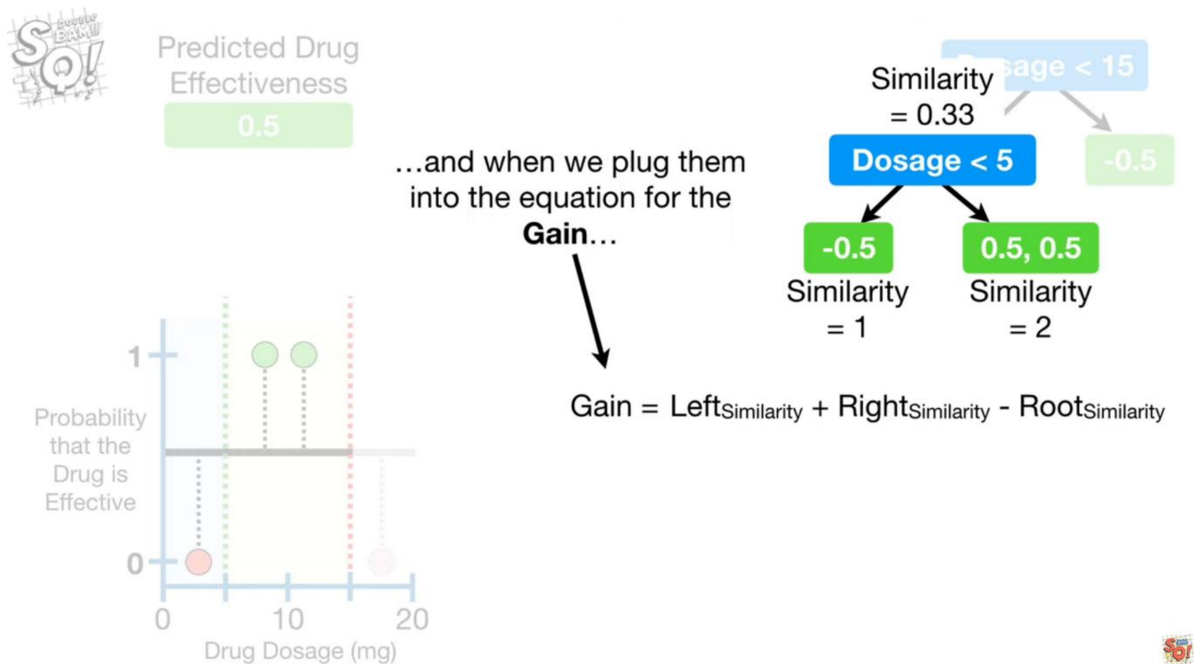


Fig 33. XG Boosting

Moving ahead, I had used Gradient Boosting, XG boosting and Catboost in my pipeline. Gradient Boosting worked better when n_estimators were reduced from 100 to 50, reduced overfitting (the number of sequential trees to be 40odelled).

XG Boost got optimized when regularization was Added and its learning ratewas tuned. Unlike Decision trees XG Boost had reguliarization parameters (xgb: lambda = L2 regularization, alpha = L1 reg, learning_rate a.k.a eta)

And Catboost too, was optimised with the help of Added L2 regularization and tuned no_of_iterations (the number of trees in the model).

Chapter 5. CONCLUSION

1. Conclusion

Logistic Regression	0.95	0	0.96	0.94	0.95	781		0.92	0	0.95	0.87	0.91	84	training set was transformed before due to multinomialNB (no negative values were sent), now training set is temporarily tranformed only for MultinomialNB and not for other models, which increased accuracies in Logistic Reg and SVC.
		1	0.95	0.97	0.96	1019			1	0.9	0.96	0.93	116	Had already added regularization and class_weights
SVM	0.956	0	0.94	0.95	0.95	752		0.925	0	0.93	0.9	0.91	83	same as logistic reg
		1	0.96	0.96	0.96	1048			1	0.92	0.95	0.93	117	
Multinomial NB	0.77	0	0.75	0.73	0.74	764		0.795	0	0.77	0.75	0.76	81	initially to make the training matrix to positive, the minimum negative value present in the training set was being added. Replaced that method with sklearn's MinMaxScaling from [0,1]
		1	0.8	0.81	0.81	1036			1	0.81	0.83	0.82	119	
Decision Tree	0.91	0	0.94	0.86	0.9	764		0.88	0	0.94	0.81	0.87	81	Hyperparameter Tuning, changed tree_depth from None to an integer (7) (None makes the depth of the tree continue until either classifies all of them correct or until all leaves contain less than min_samples_split sample. Added class_weights options to grid search, min_samples_split (nodes will leave expand until leaves contain less than min_samples_split) and min_samples_leaf (min_samples per leaf).
		1	0.89	0.96	0.92	1036			1	0.84	0.95	0.89	119	
Random Forest	0.964	0	0.95	0.98	0.96	759		0.93	0	0.92	0.92	0.92	80	same as decision tree
		1	0.98	0.97	0.97	1041			1	0.94	0.94	0.94	120	
Gradient Boosting	0.976	0	0.96	0.98	0.97	748		0.92	0	0.92	0.92	0.92	75	reduced n_estimators from 100 to 50, reduced overfitting (the number of sequential trees to be modeled.)
		1	0.99	0.97	0.98	1052			1	0.94	0.94	0.94	125	
XG Boosting	0.968	0	0.95	0.97	0.96	750		0.915	0	0.9	0.89	0.9	78	Added regularization and tuned learning rate. xgb: lambda = L2 regularization, alpha = L1 reg, learning_rate a.k.a eta
		1	0.98	0.97	0.97	1050			1	0.92	0.93	0.93	122	
CatBoost	0.977	0	0.98	0.96	0.97	773		0.965	0	0.98	0.94	0.96	87	Added L2 regularization and tuned no_of_iterations (the number of trees in the model.)
		1	0.99	0.99	0.98	1027			1	0.96	0.98	0.97	113	

Fig 33. Optimised Results

Hence, we learnt how to apply Supervised learning ML algorithms, input a raw dataset and return the best hyperparameter tuned model with good embeddings.

2. Future Work

In the next part, an entire segment of Deep Learning is still left to apply and explore, hence looking forward to that. There is still a lot of space where these models can be optimised and embeddings can be made better. BERT embeddings are a black box, even though we know how it works we are not sure how the weights of the neural network can relate to each other.

Multiple Researches are going into this and new aspects of learning are constantly flowing in, so can look into that as well.

References

1. https://scikit-learn.org/stable/index.html
2. https://www.geeksforgeeks.org/
3. https://medium.com/
4. https://towardsdatascience.com/
5. https://stats.stackexchange.com/
6. https://stackoverflow.com/