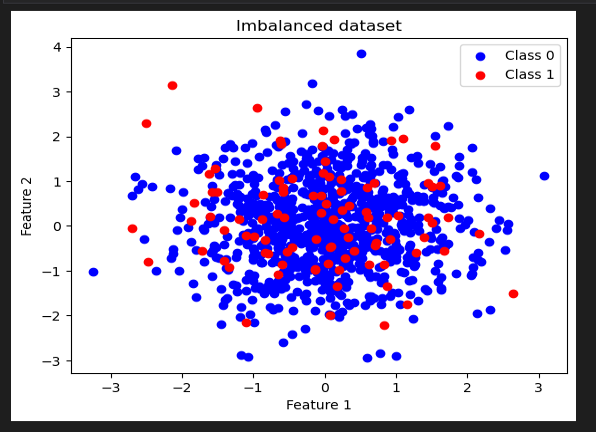
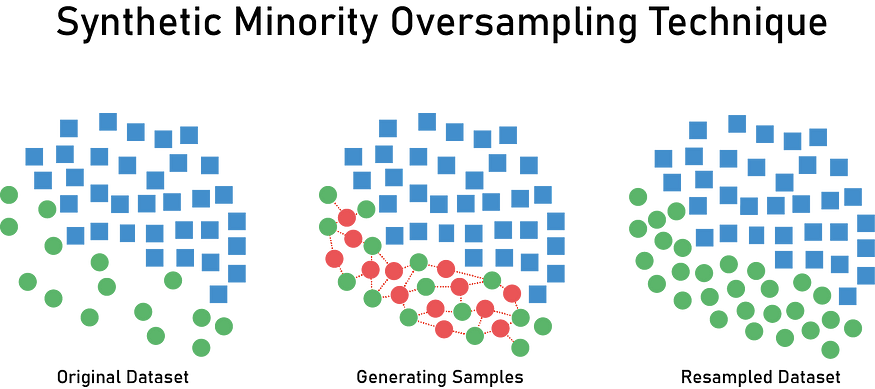
# **Synthetic Minority Over-sampling Technique**

SMOTE stands for Synthetic Minority Over-sampling Technique. It is a data augmentation technique commonly used in machine learning to deal with imbalanced datasets. Imbalanced datasets are those in which the classes to be predicted are not represented equally. For example, the dataset is imbalanced if we have a binary classification problem in which 95% of the samples belong to class A, and only 5% belong to class B. In such cases, the minority class (class B) is often underrepresented in the training data, making it difficult for the machine learning algorithm to learn to classify the minority class correctly. SMOTE is a technique that generates synthetic samples of the minority class by interpolating new data points between the existing minority class samples.



An imbalanced dataset is one in which the predicted classes are represented in different ways. In other words, one or more classes have a much smaller number of samples than the others. For example, in a binary classification problem, if 90% of the samples belong to class A and only 10% belong to class B, the dataset is imbalanced. Similarly, in a multi-class classification problem, the dataset is also considered imbalanced if one or more classes have a much smaller number of samples than the others.

Imbalanced datasets are standard in many real-world applications, such as fraud detection, disease diagnosis, and anomaly detection. However, they can challenge machine learning algorithms because they tend to be biased toward the majority class. This means that the algorithm may have higher accuracy for predicting the majority class but perform poorly on the minority class. To address this issue, SMOTE is used. The SMOTE algorithm works by randomly selecting a minority class sample, finding its k nearest neighbors, and generating new synthetic samples by linearly interpolating between the chosen sample and its k nearest neighbors. The user typically specifies the number of synthetic samples to be generated.



Applying SMOTE is a more balanced augmented dataset that provides the machine learning algorithm with more examples of the minority class to learn from, thereby improving its performance on the imbalanced dataset.

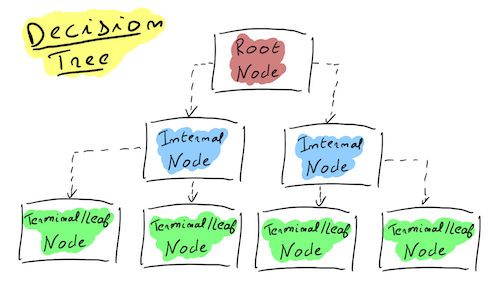
Sure, here is an example code using Python and the imbalanced-learn library to apply SMOTE on an imbalanced dataset:

import os  
from imblearn.over\_sampling import SMOTE  
import pandas as pd  
  
# Load dataset  
dataset = '/reditcard.csv'  
df = pd.read\_csv(dataset)  
  
X = df.drop('Class', axis=1)  
y = df['Class']  
  
smote = SMOTE()  
X\_resampled, y\_resampled = smote.fit\_resample(X, y)  
  
print("Number of samples before SMOTE:", len(X))  
print("Number of samples after SMOTE:", len(X\_resampled))  
  
# Save the resampled dataset to a new CSV file  
dirname = os.path.dirname(dataset)  
resampled\_df = pd.concat([pd.DataFrame(X\_resampled), pd.DataFrame(y\_resampled)], axis=1)  
resampled\_df.to\_csv(os.path.join(dirname, 'resampled\_dataset.csv'), index=False)

The code applies the SMOTE algorithm to an imbalanced dataset using the imbalanced-learn library in Python. The dataset used in the example is a credit card fraud dataset, which is a common use case for SMOTE as the fraud class is often underrepresented.

# **Decision Tree :**

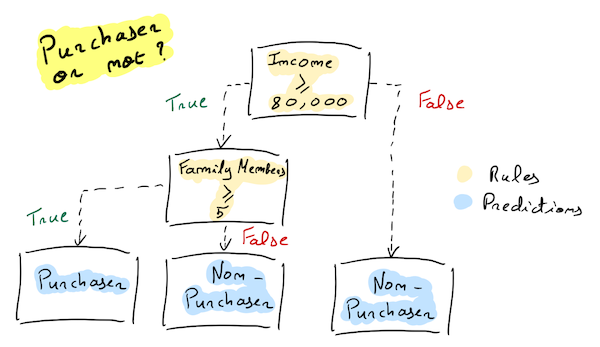
This article is a quick overview of how Decision Trees work.****Decision Trees**** are algorithms composed of nodes which are basically questions returning True are False. The****root node****is the first one and ****internal nodes,****that can also be ****terminal nodes**** (or ****leaf nodes****), are all the others



* The ****Root Node**** is the starting point where the dataset is firstly split in two.
* ****Internal**** ****Nodes**** are all nodes after the Root Node. These nodes also split in two the observations that come into it.
* ****Leaf**** ****Nodes**** or ****Terminal**** ****Nodes**** are Internal Nodes that give an output. These nodes are chosen by the algorithm as decision points because of some characteristics, allowing to get a final answer : 0 or 1 (in classification problems), a value (in regression problems).

The idea behind Decision Trees is getting a prediction by passing through the nodes until reaching a terminal node that returns the estimated output.

The tree starts with the entire training dataset (inputs and outputs). It looks for a ****rule**** that will split the dataset in two new nodes. Then it will find new rules in the new nodes to split the samples in two. It does this until the ****maximum depth**** has been reached or until it is satisfied.



# **Differences Between Supervised and Unsupervised Learning?**

****Supervised learning**** is the ****AI****equivalent of a student learning a subject under the guidance of a teacher. This method relies on a labeled dataset, meaning the inputs are directly associated with the outputs.

## **Key Features:**

* ****Labeled Data:**** Involves datasets where the outcome is already known, allowing the algorithm to learn from past data.
* ****Feedback System:**** Models are trained and corrected using a feedback system, leading to precise predictions over time.
* ****Predictive Tasks:**** Ideal for predictive tasks such as classification and regression.

# **Unsupervised Learning: The Path of Self-Discovery**

Contrarily, unsupervised learning has no labels to guide the algorithm. It must instead detect patterns, structures, or features intrinsically present in the input data.

## **Key Features:**

* ****Without Labels:**** It does not require outcome labeling, dealing with inherently more complex data.
* ****Pattern Discovery:**** It’s used for clustering and association, identifying natural groupings or relationships in data.

# **Linear Regression and Cost Function**

These concepts form the foundation of many machine learning algorithms.

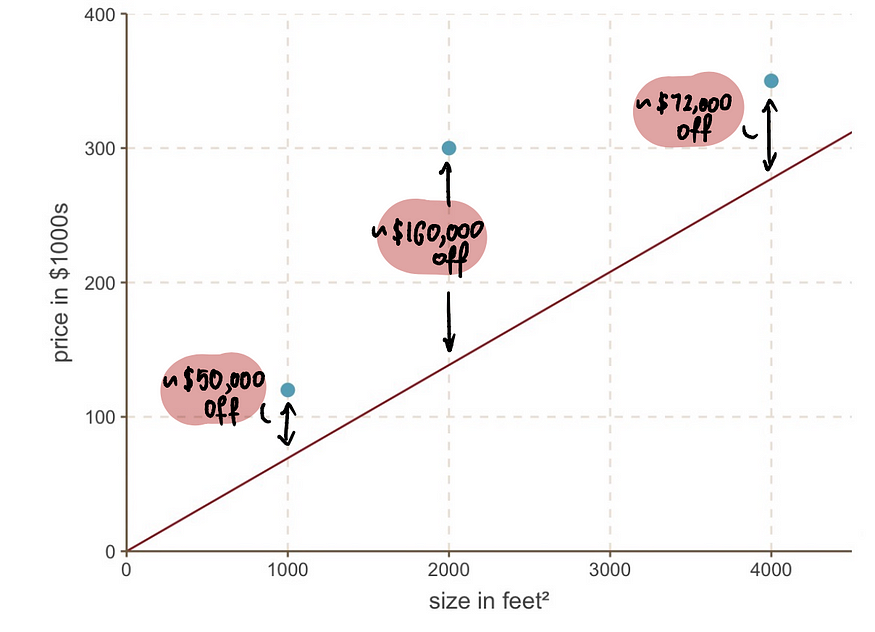


Figure-a

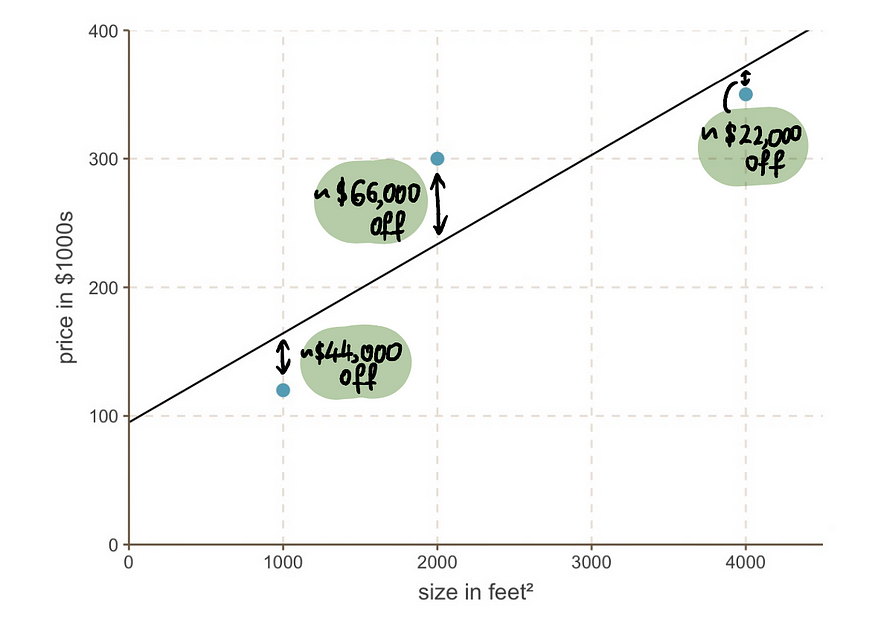
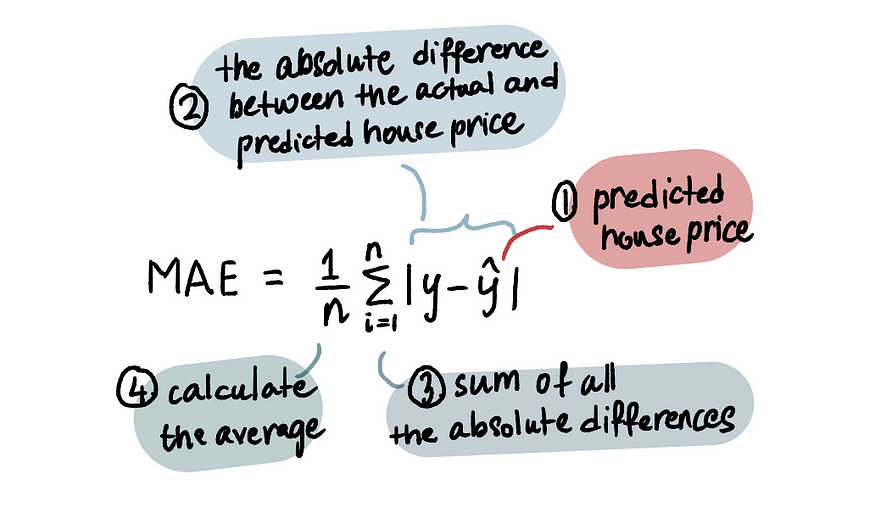


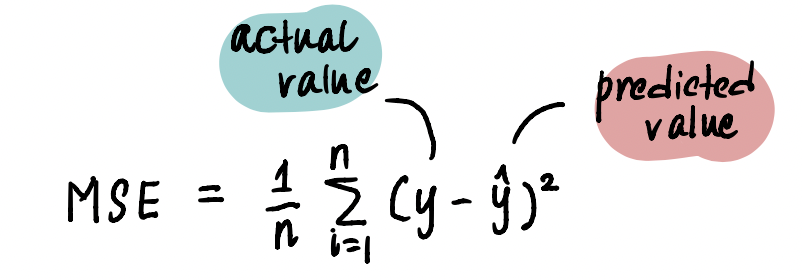
Figure-b

# **Cost Function**

Above, we utilized the ****Mean Absolute Error (MAE)**** cost function to determine the deviation of the actual house prices from the predicted prices. This basically calculates the average of how off the actual house prices (denoted as y, as it represents the value on the y-axis) were from the predicted house prices (denoted as ŷ). We represent ****MAE**** mathematically like this:

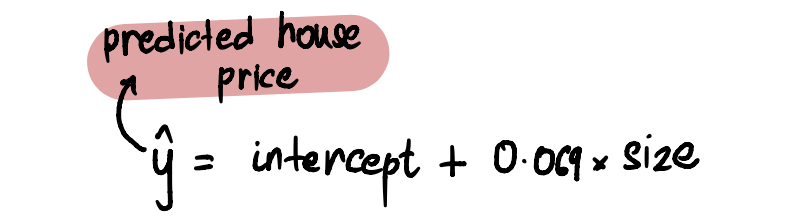


For our problem, instead of using the ****MAE****, we will employ a commonly used method, the ****Mean Squared Error (MSE)****, which calculates the *average of the squares of the difference between the predicted house price and the actual house price.*



To make things even simpler, let’s assume that we somehow magically already have the value of the *slope*, 0.069.

So the equation of our linear regression line is:



NLP software mainly works at the sentence level and it also expects words to be separated at the minimum level.

Our cleaned text data may contain a group of sentences. and each sentence is a group of words. So, first, we need to Tokenize our text data.

* [Tokenization](https://www.geeksforgeeks.org/nlp-how-tokenizing-text-sentence-words-works/)**:** Tokenization is the process of segmenting the text into a list of tokens. In the case of sentence tokenization, the token will be sentenced and in the case of word tokenization, it will be the word. It is a good idea to first complete sentence tokenization and then word tokenization, here output will be the list of lists. Tokenization is performed in each & every NLP pipeline.
* [Lowercasing](https://www.geeksforgeeks.org/python-string-lower/)**:** This step is used to convert all the text to lowercase letters. This is useful in various NLP tasks such as text classification, information retrieval, and sentiment analysis.
* [Stop word removal](https://www.geeksforgeeks.org/removing-stop-words-nltk-python/)**:**Stop words are commonly occurring words in a language such as “the”, “and”, “a”, etc. They are usually removed from the text during preprocessing because they do not carry much meaning and can cause noise in the data. This step is used in various NLP tasks such as text classification, information retrieval, and topic modeling.
* [Stemming or lemmatization](https://www.geeksforgeeks.org/introduction-to-nltk-tokenization-stemming-lemmatization-pos-tagging/)**:** Stemming and lemmatization are used to reduce words to their base form, which can help reduce the vocabulary size and simplify the text. Stemming involves stripping the suffixes from words to get their stem, whereas lemmatization involves reducing words to their base form based on their part of speech. This step is commonly used in various NLP tasks such as text classification, information retrieval, and topic modeling.
* [Removing digit/punctuation](https://www.geeksforgeeks.org/python-remove-punctuation-from-string/)**:** This step is used to remove digits and punctuation from the text. This is useful in various NLP tasks such as text classification, sentiment analysis, and topic modeling.
* [POS tagging](https://www.geeksforgeeks.org/nlp-part-of-speech-default-tagging/)**:** POS tagging involves assigning a part of speech tag to each word in a text. This step is commonly used in various NLP tasks such as named entity recognition, sentiment analysis, and machine translation.

import nltk

from nltk.tokenize import word\_tokenize

from nltk.corpus import stopwords

from nltk.stem import SnowballStemmer, WordNetLemmatizer

from nltk.tag import pos\_tag

from nltk.chunk import ne\_chunk

import string

# sample text to be preprocessed

text = 'GeeksforGeeks is a very famous edutech company in the IT industry.'

# tokenize the text

tokens = word\_tokenize(text)

# remove stop words

stop\_words = set(stopwords.words('english'))

filtered\_tokens = [token for token in tokens if token.lower() not in stop\_words]

# perform stemming and lemmatization

stemmer = SnowballStemmer('english')

lemmatizer = WordNetLemmatizer()

stemmed\_tokens = [stemmer.stem(token) for token in filtered\_tokens]

lemmatized\_tokens = [lemmatizer.lemmatize(token) for token in filtered\_tokens]

# remove digits and punctuation

cleaned\_tokens = [token for token in lemmatized\_tokens

if not token.isdigit() and not token in string.punctuation]

# convert all tokens to lowercase

lowercase\_tokens = [token.lower() for token in cleaned\_tokens]

# perform part-of-speech (POS) tagging

pos\_tags = pos\_tag(lowercase\_tokens)

# perform named entity recognition (NER)

named\_entities = ne\_chunk(pos\_tags)

# print the preprocessed text

print("Original text:", text)

print("Preprocessed tokens:", lowercase\_tokens)

print("POS tags:", pos\_tags)

print("Named entities:", named\_entities)

* [Named Entity Recognition (NER)](https://www.geeksforgeeks.org/named-entity-recognition/)**:** NER involves identifying and classifying named entities in text, such as people, organizations, and locations. This step is commonly used in various NLP tasks such as information extraction, machine translation, and question-answering.

## Feature Engineering:

In Feature Engineering, our main agenda is to represent the text in the numeric vector in such a way that the ML algorithm can understand the text attribute. In NLP this process of feature engineering is known as Text Representation or Text Vectorization.

**There are two most common approaches for Text Representation.**

### 1. Classical or Traditional Approach:

In the traditional approach, we create a vocabulary of unique words assign a unique id (integer value) for each word. and then replace each word of a sentence with its unique id.  Here each word of vocabulary is treated as a feature. So, when the vocabulary is large then the feature size will become very large. So, this makes it tough for the ML model.

#### ****One Hot Encoder:****

[One Hot Encoding](https://www.geeksforgeeks.org/ml-one-hot-encoding-of-datasets-in-python/) represents each token as a binary vector. First mapped each token to integer values. and then each integer value is represented as a binary vector where all values are 0 except the index of the integer. index of the integer is marked by 1.

import nltk

# nltk.download('punkt') # Download 'punkt'

# from nltk if it's not downloaded

from nltk.tokenize import sent\_tokenize

Text = """Geeks For Geeks.

Geeks Learning Together.

Geeks For Geeks is famous for DSA.

Learning DSA"""

sentences = sent\_tokenize(Text)

sentences = [sent.lower().replace(".", "") for sent in sentences]

print('Tokenized Sentences :', sentences)

# Create the vocabulary

vocab = {}

count = 0

for sent in sentences:

for word in sent.split():

if word not in vocab:

count = count + 1

vocab[word] = count

print('vocabulary :', vocab)

# One Hot Encoding

def OneHotEncoder(text):

onehot\_encoded = []

for word in text.split():

temp = [0]\*len(vocab)

if word in vocab:

temp[vocab[word]-1] = 1

onehot\_encoded.append(temp)

return onehot\_encoded

# print('\n',sentences[0])

print('OneHotEncoded vector for sentence : "',

sentences[0], '"is \n', OneHotEncoder(sentences[0]))

Bag of Word(Bow):

 A [bag of words](https://www.geeksforgeeks.org/bag-of-words-bow-model-in-nlp/) only describes the occurrence of words within a document or not. It just keeps track of word counts and ignores the grammatical details and the word order.

Code block

import nltk

#nltk.download('punkt') # Download 'punkt' from nltk if it's not downloaded

from nltk.tokenize import sent\_tokenize

from sklearn.feature\_extraction.text import CountVectorizer

Text = """GeeksForGeeks.

Geeks Learning Together.

GeeksForGeeks is famous for DSA.

Learning DSA"""

# TOKENIZATION

sentences = sent\_tokenize(Text)

sentences = [sent.lower().replace(".","") for sent in sentences]

print('Our Corpus:',sentences)

#CountVectorizer : Convert a collection of text documents to a matrix of token counts.

count\_vect = CountVectorizer()

# fit & transform will represent each sentences as BOW representation

BOW = count\_vect.fit\_transform(sentences)

# Get the vocabulary

print("Our vocabulary: ", count\_vect.vocabulary\_)

#see the BOW representation

print(f"BoW representation for {sentences[0]} {BOW[0].toarray()}")

print(f"BoW representation for {sentences[1]} {BOW[1].toarray()}")

print(f"BoW representation for {sentences[2]} {BOW[2].toarray()}")

# BOW representation for a new text

BOW\_ = count\_vect.transform(["learning dsa from geeksforgeeks"])

print("Bow representation for 'learning dsa from geeksforgeeks':", BOW\_.toarray())

Output:Our Corpus: ['geeksforgeeks', 'geeks learning together',

'geeksforgeeks is famous for dsa', 'learning dsa']

Our vocabulary: {'geeksforgeeks': 4, 'geeks': 3, 'learning': 6,

'together': 7, 'is': 5, 'famous': 1, 'for': 2, 'dsa': 0}

BoW representation for geeksforgeeks [[0 0 0 0 1 0 0 0]]

BoW representation for geeks learning together [[0 0 0 1 0 0 1 1]]

BoW representation for geeksforgeeks is famous for dsa [[1 1 1 0 1 1 0 0]]

Bow representation for 'learning dsa from geeksforgeeks': [[1 0 0 0 1 0 1 0]]

Bag of n-grams:

  In Bag of Words, there is no consideration of the phrases or word order. Bag of n-gram tries to solve this problem by breaking text into chunks of n continuous words.The Bag of n-grams model can capture more context than a Bag of Words by considering adjacent words together. This is useful for preserving some local word order information and detecting phrases or common word pairs that occur in text.

Uses of bag of n-grams:

· **Sentiment analysis**: Identifying common phrases or bigrams like "not good" or "very happy."

· **Text classification**: Capturing context for topics, e.g., "climate change" or "data science" may have specific bigram/trigram patterns.

· **Information retrieval**: Enhancing search algorithms by including multi-word terms.

import nltk

# nltk.download('punkt') # Download 'punkt'

# from nltk if it's not downloaded

from nltk.tokenize import sent\_tokenize

from sklearn.feature\_extraction.text import CountVectorizer

Text = """GeeksForGeeks.

Geeks Learning Together.

GeeksForGeeks is famous for DSA.

Learning DSA"""

# TOKENIZATION

sentences = sent\_tokenize(Text)

sentences = [sent.lower().replace(".", "") for sent in sentences]

print('Our Corpus:', sentences)

# Ngram vectorization example with count

# vectorizer and uni, bi, trigrams

count\_vect = CountVectorizer(ngram\_range=(1, 3))

# fit & transform will represent each sentences

# as Bag of n-grams representation

BOW\_nGram = count\_vect.fit\_transform(sentences)

# Get the vocabulary

print("Our vocabulary:\n", count\_vect.vocabulary\_)

# see the Bag of n-grams representation

print('Ngram representation for "{}" is {}'

.format(sentences[0], BOW\_nGram[0].toarray()))

print('Ngram representation for "{}" is {}'

.format(sentences[1], BOW\_nGram[1].toarray()))

print('Ngram representation for "{}" is {}'.

format(sentences[2], BOW\_nGram[2].toarray()))

# Bag of n-grams representation for a new text

BOW\_nGram\_ = count\_vect.transform(["learning dsa from geeksforgeeks together"])

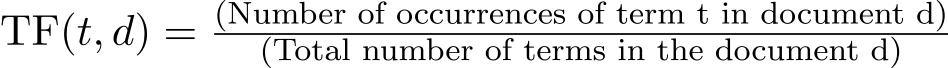
print("Ngram representation for 'learning dsa from geeksforgeeks together' is",

BOW\_nGram\_.toarray())

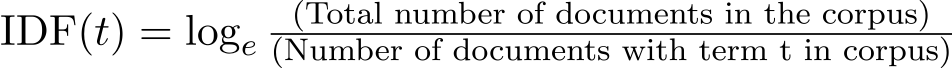
**TF-IDF (Term Frequency – Inverse Document Frequency):**

 In all the above techniques,  Each word is treated equally. [TF-IDF](https://www.geeksforgeeks.org/understanding-tf-idf-term-frequency-inverse-document-frequency/) tries to quantify the importance of a given word relative to the other word in the corpus.  it is mainly used in Information retrieval.

Term Frequency (TF): TF measures how often a word occurs in the given document. it is the ratio of the number of occurrences of a term or word (t ) in a given document (d) to the total number of terms in a given document (d).



Inverse document frequency (IDF): IDF measures the importance of the word across the corpus. it down the weight of the terms, which commonly occur in the corpus, and up the weight of rare terms.



TF-IDF score is the product of TF  and IDF.

Rendered by QuickLaTeX.com

Python3

|  |
| --- |
| import nltk  # nltk.download('punkt') # Download 'punkt'  # from nltk if it's not downloaded  from nltk.tokenize import sent\_tokenize  from sklearn.feature\_extraction.text import TfidfVectorizer  Text = """GeeksForGeeks.  Geeks Learning Together.  GeeksForGeeks is famous for DSA.  Learning DSA"""  # TOKENIZATION  sentences = sent\_tokenize(Text)  sentences = [sent.lower().replace(".", "") for sent in sentences]  print('Our Corpus:', sentences)  # TF-IDF  tfidf = TfidfVectorizer()  tfidf\_matrix = tfidf.fit\_transform(sentences)  # All words in the vocabulary.  print("vocabulary", tfidf.get\_feature\_names())  # IDF value for all words in the vocabulary  print("IDF for all words in the vocabulary :\n", tfidf.idf\_)  # TFIDF representation for all documents in our corpus  print('\nTFIDF representation for "{}" is \n{}'  .format(sentences[0], tfidf\_matrix[0].toarray()))  print('TFIDF representation for "{}" is \n{}'  .format(sentences[1], tfidf\_matrix[1].toarray()))  print('TFIDF representation for "{}" is \n{}'  .format(sentences[2],tfidf\_matrix[2].toarray()))  # TFIDF representation for a new text  matrix = tfidf.transform(["learning dsa from geeksforgeeks"])  print("\nTFIDF representation for 'learning dsa from geeksforgeeks' is\n",  matrix.toarray()) |

Word Embeddings used?

They are used as input to machine learning models.  
Take the words —-> Give their numeric representation —-> Use in training or inference.

1.1. One-Hot Encoding

One-hot encoding is a simple method for representing words in natural language processing (NLP). In this encoding scheme, each word in the vocabulary is represented as a unique vector, where the dimensionality of the vector is equal to the size of the vocabulary.

def one\_hot\_encode(text):

words = text.split()

vocabulary = set(words)

word\_to\_index = {word: i for i, word in enumerate(vocabulary)}

one\_hot\_encoded = []

for word in words:

one\_hot\_vector = [0] \* len(vocabulary)

one\_hot\_vector[word\_to\_index[word]] = 1

one\_hot\_encoded.append(one\_hot\_vector)

return one\_hot\_encoded, word\_to\_index, vocabulary

# sample

example\_text = "cat in the hat dog on the mat bird in the tree"

one\_hot\_encoded, word\_to\_index, vocabulary = one\_hot\_encode(example\_text)

print("Vocabulary:", vocabulary)

print("Word to Index Mapping:", word\_to\_index)

print("One-Hot Encoded Matrix:")

for word, encoding in zip(example\_text.split(), one\_hot\_encoded):

print(f"{word}: {encoding}")

**Output:**

Vocabulary: {'mat', 'the', 'bird', 'hat', 'on', 'in', 'cat', 'tree', 'dog'}  
Word to Index Mapping: {'mat': 0, 'the': 1, 'bird': 2, 'hat': 3, 'on': 4, 'in': 5, 'cat': 6, 'tree': 7, 'dog': 8}  
One-Hot Encoded Matrix:  
cat: [0, 0, 0, 0, 0, 0, 1, 0, 0]  
in: [0, 0, 0, 0, 0, 1, 0, 0, 0]  
the: [0, 1, 0, 0, 0, 0, 0, 0, 0]  
hat: [0, 0, 0, 1, 0, 0, 0, 0, 0]  
dog: [0, 0, 0, 0, 0, 0, 0, 0, 1]  
on: [0, 0, 0, 0, 1, 0, 0, 0, 0]  
the: [0, 1, 0, 0, 0, 0, 0, 0, 0]  
mat: [1, 0, 0, 0, 0, 0, 0, 0, 0]

Note: - While one-hot encoding is a simple and intuitive method for representing words in NLP, it has several disadvantages, which may limit its effectiveness in certain applications.

One-hot encoding results in high-dimensional vectors, making it computationally expensive and memory-intensive, especially with large vocabularies.