**Downloading stop words using NLTK library**

import nltk

nltk.download('stopwords')

Displaying the downloaded stop words:

from nltk.corpus import stopwords

stopwords.words('english')

stopwords.words('german')

**Obtaining the root word**

**Stemming**

from nltk.stem import PorterStemmer

porter = PorterStemmer()

porter.stem('barking')

This will return bark. A rather crude method whose output at times can be difficult to interpret

**Lemmatization**

from nltk.stem import WordNetLemmatizer

from nltk.corpus import wordnet

nltk.download('wordnet')

lemmatizer = WordNetLemmatizer()

lemmatizer.lemmatize('mice')

* We obtain mouse as a result.
* We can additionally specify **pos** (parts of speech) to obtain the root. By default, it is set to noun, which might yield us the desired result at times. For example in the case below, not mentioning **pos**  would have resulted in the output being ‘threw’ instead of ‘throw’.

lemmatizer.lemmatize('threw', pos = wordnet.VERB)

**Lemmatizing an entire sentence**

* While lemmatizing a sentence, we need to have pos tagging for every word
* While we can use nltk library to obtain this mapping, the outputs aren't compatible with the **WordNetLemmatizer()**
* Hence, we will need a separate / intermediate function to map tags from one format to another

def get\_wordnet\_pos(treebank\_tag):

    if treebank\_tag.startswith('J'):

        return wordnet.ADJ

    if treebank\_tag.startswith('V'):

        return wordnet.VERB

    if treebank\_tag.startswith('N'):

        return wordnet.NOUN

    if treebank\_tag.startswith('R'):

        return wordnet.ADV

    else:

        return wordnet.NOUN

**Obtaining pos tags from NLTK library**

nltk.download('averaged\_perceptron\_tagger')

Tokenize the sentence:

sentence = "Bolsonaro has a cult following".split()

Obtain pos tags:

words\_and\_tags = nltk.pos\_tag(sentence)

Pass the word and mappings of these tags to the lemmatizer:

for word,tag in words\_and\_tags:

    print(lemmatizer.lemmatize(word,pos=get\_wordnet\_pos(tag)),end=" ")

**Implementing a Count Vectorizer**

We first import the requisite libraries and datasets and split them into inputs and output. We then proceed with test-train split for this dataset.

**Using normal Count vectorizer**

Initialise the count vectorizer and obtain the count vector representations of the inputs

vectorizer = CountVectorizer()

X\_train = vectorizer.fit\_transform(inp\_train)

X\_test = vectorizer.transform(inp\_test)

The obtained vectors are sparse (very small number of non-zero entries). In order to verify this, we can check the percentage of non-zero entries in the vector.

(X\_train != 0).sum()

((X\_train != 0).sum()) / np.prod(X\_train.shape) \* 100

We now initialise a ML model of our choice(multinomial naïve bayes in this case) and proceed to train the model on this data. The train and test score are the calculated.

model = MultinomialNB()

model.fit(X\_train,y\_train)

print('train score: ',model.score(X\_train,y\_train))

print('test score: ',model.score(X\_test,y\_test))

**Implementing a Count Vectorizer with stopwords defined**

Identical to the approach before , just that stop words are defined which brings down the dimensionality of the count vector by a little (frequently occurring words which offer little insights are not monitored anymore)

vectorizer = CountVectorizer(stop\_words='english')

X\_train = vectorizer.fit\_transform(inp\_train)

X\_test = vectorizer.transform(inp\_test)

model = MultinomialNB()

model.fit(X\_train,label\_train)

print(f'Dimensionality : {X\_train.shape}')

print('train score: ',model.score(X\_train,label\_train))

print('test score: ',model.score(X\_test,label\_test))

**Implementing a Count Vectorizer with Lemmatization(** **tokenizer = LemmaTokenizer())**

This involves two steps:

* Defining a mapping function which maps the NLTK pos tags to wordnet pos tags
* Defining a lemmatizer class which takes in document as input, tokenizes it, obtains it’s NLTK pos tags and this information is passed on to the wordnet lemmatizer and the output is obtained

**Mapping function**

def get\_wordnet\_tags(treebank\_tag):

    if treebank\_tag.startswith('J'):

        return wordnet.ADJ

    elif treebank\_tag.startswith('V'):

        return wordnet.VERB

    elif treebank\_tag.startswith('N'):

        return wordnet.NOUN

    elif treebank\_tag.startswith('R'):

        return wordnet.ADV

    else:

        return wordnet.NOUN

**Lemmatizer class**

class LemmaTokenizer:

    def \_\_init\_\_(self):

        self.wnl = WordNetLemmatizer()

    def \_\_call\_\_(self, doc):

        tokens = word\_tokenize(doc)

        words\_and\_tags = nltk.pos\_tag(tokens)

        return [self.wnl.lemmatize(word,pos=get\_wordnet\_tags(tag)) for word,tag in words\_and\_tags]

**Training the model**

vectorizer = CountVectorizer(tokenizer = LemmaTokenizer())

X\_train = vectorizer.fit\_transform(inp\_train)

X\_test = vectorizer.transform(inp\_test)

model = MultinomialNB()

model.fit(X\_train,label\_train)

print(f'Dimensionality : {X\_train.shape}')

print('train score : ', model.score(X\_train,label\_train))

print('test score :' , model.score(X\_test,label\_test))

**Implementing a Count Vectorizer with Stemming (tokenizer = StemTokenizer())**

No mapping function needed in this case. However we still need to stemming class which tokenizes the input document and returns the stemmed output

class StemTokenizer:

    def \_\_init\_\_(self):

        self.portstem = PorterStemmer()

    def \_\_call\_\_(self,doc):

        tokens = word\_tokenize(doc)

        return [self.portstem.stem(token) for token in tokens]

**Training the model**

vectorizer = CountVectorizer(tokenizer = StemTokenizer())

X\_train = vectorizer.fit\_transform(inp\_train)

X\_test = vectorizer.transform(inp\_test)

model = MultinomialNB()

model.fit(X\_train,label\_train)

print(f'Dimensionality : {X\_train.shape}')

print('train score : ', model.score(X\_train,label\_train))

print('test score :' , model.score(X\_test,label\_test))

**Implementing a Count Vectorizer with Simple input split**

def SimpleSplit(doc):

    return doc.split()

vectorizer = CountVectorizer(tokenizer = SimpleSplit)

X\_train = vectorizer.fit\_transform(inp\_train)

X\_test = vectorizer.transform(inp\_test)

model = MultinomialNB()

model.fit(X\_train,label\_train)

print('train score : ', model.score(X\_train,label\_train))

print('test score :' , model.score(X\_test,label\_test))

While lemma tokenizer was the most intricate amongst the methods used, it was normally expected to perform the best. But in this case, it is found to have the worst performance. Therefore, we must choose the approach based on the problem in hand.

**TF- IDF Recommendation System**

* In the dataset being used, we have information related to movie genres and keywords associated with that movie along with other information. The catch however is that the data to be fed to vectorizer must be a string.
* The data that we have currently has json elements, which needs to pre-processed and then fed to the tf-idf vectorizer.

**The following function takes in a row of data as input and returns the required data as a string.**

def genres\_and\_keywords\_to\_string(row):

    g = json.loads(row['genres'])

    genres = ' '.join(''.join(genre['name'].split()) for genre in g)

    k = json.loads(row['keywords'])

    keywords = ' '.join(''.join(kw['name'].split()) for kw in k)

    return '%s %s'%(genres, keywords)

* We are considering the **genres** and **keywords** column for our current use case. The **json.loads** command is used in converting json elements into dictionaries.
* Since our vectorizer expects a string input, we need to concatenate the info into a single string and pass it as a string
* There may be entries where a space between the same genre might result in two separate words (science fiction for example). The **''.join(genre['name].split())** takes care of those examples.

**Creating a new string representation for each movie using this function**

df['string'] = df.apply(genres\_and\_keywords\_to\_string,axis=1)

**Initialize tf-idf vectorizer object**

tfidf = TfidfVectorizer(max\_features=2000)

**Creating a data matrix by transforming the input data using the tf-idf vectorizer**

X = tfidf.fit\_transform(df['string'])

**Title index mapping**

* By now we have seen the need for word-index mapping ( since it is easier for programs to deal with numbers) . In this case we generate a mapping between movie titles to their index.

movie2idx = pd.Series(df.index,index=df['title'])

**Getting the recommendation**

Getting the index for the movie based on which recommendations are sought:

idx = movie2idx['The Secret Life of Walter Mitty']

Obtaining the corresponding query from the transformed input matrix:

query = X[idx]

Converting this to an array (for computational reasons):

query.toarray()

Calculating cosine similarity between the query and all other vectors in X:

scores = cosine\_similarity(query,X)

The array is currently 1xN. We need to make it a 1-D array

scores = scores.flatten()

We shall proceed with sorting these scores in descending order so that we can obtain the titles most similar to the Secret life of walter mitty. Since we need to order and not the order itself, we use **argsort()** which returns the indices of elements sorted in ascending order. (-scores is for descending order)

(-scores).argsort()

Getting top K matches:

recommended\_idx = (-scores).argsort()[1:K+1]

Obtaining the titles corresponding to these indices:

df['title'].iloc[recommended\_idx]

The steps can be aggregated into a function, which takes in a movie name as input and returns titles similar to it.

def recommend(title):

    #get the row from the dataframe for this movie

    idx = movie2idx[title]

    #The following code combats cases where than can be multiple rows corresponding to a title

    if type(idx) == pd.Series:

        idx = idx.iloc[0]

    #Calculating pairwise similarities for this movie

    query = X[idx]

    scores = cosine\_similarity(query,X)

    #Currently array is 1xN, we need to make it a 1D array

    scores = scores.flatten()

    #get the index of highest scoring elements by sorting in descending order

    #then obtain the K highest recommendations and store their indices

    #Obtain the corresponding titles for these indices

    recommended\_idx = (-scores).argsort()[1:6]

    #return the titles of these recommendations

    return df['title'].iloc[recommended\_idx]

**TF-IDF implementation from scratch**

Start by importing the requisite libraries and datasets.

**Implement word to index mapping**

* Initialise index as 0 and word to index as an empty dictionary
* Iterate through the column where the text is present and tokenize each document
* For each token, check if it is already present in the word to index dictionary and if not, update it using the current index and increment current index.
* Convert each doc into their indexed representations

curr\_idx = 0

word\_idx = {}

tokenized\_docs = []

for doc in df['text']:

    words = word\_tokenize(doc)

    doc\_as\_int = []

    for word in words:

        if word not in word\_idx:

            word\_idx[word] = curr\_idx

            curr\_idx += 1

        doc\_as\_int.append(word\_idx[word])

    tokenized\_docs.append(doc\_as\_int)

**Reverse Mapping**

Later we will need to map the indices back to the words to obtain meaningful results.

idx2word = {k:v for v,k in word\_idx.items()}

Alternatively, we can store it as a list too.

idx2word\_l = [0]\*len(word\_idx)

for v,k in word\_idx.items():

    idx2word\_l[k] = v

**Building the TF (term frequency matrix)**

Obtain number of documents:

N = len(df['text'])

Number of words:

V = len(word\_idx)

**Initialise the matrix:**

* Size of the matrix is (N x V)
* We will be manually counting the occurrences of the terms here. Instead, we could have used a count vectorizer too

tf = np.zeros((N,V))

**Populating the term frequency matrix**

In the wordidx function defined earlier, we have stored tokens represented by their corresponding index which we count to obtain the term frequency for every term.

for i,doc\_as\_int in enumerate(tokenized\_docs):

    for j in doc\_as\_int:

        tf[i,j] += 1

**Inverse Document Frequency term (IDF)**

IDF = log (N/N(t)) where N(t) is frequency of occurrence of the term t over all documents

document\_freq = np.sum(tf>0,axis =0) #Summing over each word. Output (Nx1)

idf = np.log(N/document\_freq)

Multiply TF and IDF

tf\_idf = tf\*idf

**Picking a random document and displaying the top 5 terms relating to that document**

* The row corresponding to the document is chosen and the scores in that row are obtained.
* Like previously, we then sort the scores in descending order and the words with highest similarity are displayed.

i = np.random.choice(N)

#Choose the row for the doc selected at random

row = df.iloc[i]

print('Label: ',row['labels'])

print("Text:", row['text'].split("\n", 1)[0])

print("Top 5 terms:")

scores = tf\_idf[i]

indices = (-scores).argsort()

for j in indices[:5]:

    print(idx2word[j])

**Word Embeddings Demo**

**Text Classifier**

We make use of Markov property to achieve this.

**Pre-processing the data**

* Import the dataset

input\_files = ['edgar\_allan\_poe.txt','robert\_frost.txt']

**Collect this data into lists**

* We also assign the labels (poet name in this case) to the lines of the poem.
* **Translate()** method is used to remove punctuations

input\_texts = []

labels = []

for label,f in enumerate(input\_files):

    print(f'{f} is assigned label {label}')

    for line in open(f):

        line = line.rstrip().lower() #rstrip to remove newline character at the end

        #Only include non empty lines

        if line:

            #remove punctuation

            line = line.translate(str.maketrans('','',string.punctuation))

            input\_texts.append(line)

            labels.append(label)

**Split into train-test sets before word-to-index mapping**

train\_text, test\_text, y\_train, y\_test = train\_test\_split(input\_texts,labels)

**Word to index mapping**

* Unknown tokens are assigned the 0th index and the words that follow are assigned indexes in a sequential manner.

currIdx = 1

word2idx = {'<unk>':0}

for text in train\_text:

    tokens = text.split() #We used inbuilt word tokenizers initally. Now, just split will suffice

    for token in tokens:

        if token not in word2idx:

            word2idx[token] = currIdx

            currIdx += 1

We now use this function to obtain the integer form of the train and test sets.

train\_text\_int = []

test\_text\_int = []

for text in train\_text:

    tokens = text.split()

    line\_text = [word2idx[token] for token in tokens]

    train\_text\_int.append(line\_text)

for text in test\_text:

    tokens = text.split()

    line\_text = [word2idx.get(token,0) for token in tokens]

    test\_text\_int.append(line\_text)

While converting the test set into integer, we need to account for words which might not have appeared in the training set and hence won’t have mappings. For those values we set a default value of 0 which corresponds to unknown token.

**Initiate transition state(A) and initial state(pi) matrices**

V = len(word2idx)

A0 = np.ones((V,V))

pi0 = np.ones(V)

A1= np.ones((V,V))

pi1= np.ones(V)

We will be modelling a Markov model each for edgar allen poe poems and for Robert croft.

**Computing the counts of terms for the transition and the initial state**

def ComputeCounts(text\_as\_int,A,pi):

    for line\_int in text\_as\_int:

        last\_idx = None

        for idx in line\_int:

            #If it is first index then update the pi distribution

            if last\_idx is None:

                pi[idx] += 1

            else:

                A[last\_idx,idx] += 1

            last\_idx = idx

* For A0, we will need lines of edgar allen poe and for A1 we will need Robert frost. But these are are all mixed in **train\_test\_int**. Therefore, we combine **train\_text\_int** with their corresponding labels and then choose the lines corresponding to each author using the labels.

ComputeCounts([t for t,y in zip(train\_text\_int,y\_train) if y == 0],A0,pi0)

ComputeCounts([t for t,y in zip(train\_text\_int,y\_train) if y == 1],A1,pi1)

We have the counts now but we need the probabilities. This can be achieved by normalizing the counts with their row sums.

A0 /= A0.sum(axis=1,keepdims=True)

pi0 /= pi0.sum()

A1 /= A1.sum(axis=1,keepdims=True)

pi1 /= pi1.sum()

Since we will be dealing with log probabilities.

logA0 = np.log(A0)

logpi0 = np.log(pi0)

logA1= np.log(A1)

logpi1 = np.log(pi1)

**Computing prior**

This will help us determine whether the classes are imbalanced (MAP) or balanced (maximum likelihood). In this case we are finding **p(author = frost|edgar)**

count0 = sum(y == 0 for y in y\_train)

count1 = sum(y == 1 for y in y\_train)

total\_count = len(y\_train)

p0 = count0 / total\_count

p1 = count1 / total\_count

logp0 = np.log(p0)

logp1 = np.log(p1)

Clear that we have an unbalanced class situation and hence can't go for maximum likelihood solution.

class Classifier:

    def \_\_init\_\_(self, logAs, logPis, logPriors):

        self.logAs = logAs

        self.logPis = logPis

        self.logPriors = logPriors

        self.K = len(logPriors)

    def compute\_likelihoood(self,inputs,class\_):

        logA = self.logAs[class\_]

        logPi = self.logPis[class\_]

        last\_idx = None

        logprob = 0

        for idx in inputs:

            if last\_idx is None:

                logprob += logPi[idx]

            else:

                logprob += logA[last\_idx,idx]

            last\_idx = idx

        return logprob

    def predict(self, inputs):

        predictions = np.zeros(len(inputs))

        for i,input in enumerate(inputs):

            posteriors = [self.compute\_likelihoood(input,c) + self.logPriors[c] for c in range(self.K)]

            pred = np.argmax(posteriors)

            predictions[i] = pred

        return predictions

The function comprises of 3 parts:

* In the initialize function, log of the transition matrix, initial state matrix and priors are initialized.
* We then compute the log-likelihood. For the first index we add the initial state value for the index and for the rest, we add the transition state matrix values. The **last\_idx** is updated in every iteration.
* The prediction function calculates the posterior by adding up the log-likelihood and the posterior.

clf = Classifier([logA0,logA1],[logpi0,logpi1],[logp0,logp1])

Further we can compute performance metrics to evaluate the performance.

**Summary**

* Pre-process data and assign labels to the text
* Split data into train and test sets
* Use word to index mapping and convert train and test data into integer equivalents
* Initialise initial state(pi) and transition matrix(A) for both sets of text
* Compute counts of terms for the transition and the initial state and normalize them to obtain the probabilities
* Compute the prior which will help us determine if the classes are imbalanced or not.
* Calculate the log likelihood by summing up the logA and log(pi) values. Obtain the predictions by computing the posteriors (log likelihood + posterior)

**Poetry Generator**

In the text classifier case, we had strictly adhered to the Markov property which states that the current word will depend on the previous word only. Here , we are extending that concept and will be using the previous two words.

* Initialise dictionaries for initial (first word), first order (second word) and second order(other words).

initial = {} # start of a phrase

first\_order = {} # second word only

second\_order = {}

**Subsidiary functions**

* We will be making use of a few helper functions

**Remove punctuations**

def remove\_punctuation(s):

    return s.translate(str.maketrans('','',string.punctuation))

**Add items to dictionary**

def add2dict(d, k, v):

  if k not in d:

    d[k] = []

  d[k].append(v)

**Process input text and populate the three dictionaries initialized earlier**

for line in open('robert\_frost.txt'):

  tokens = remove\_punctuation(line.rstrip().lower()).split()

  T = len(tokens)

  for i in range(T):

    t = tokens[i]

    if i == 0:

      # measure the distribution of the first word

      initial[t] = initial.get(t, 0.) + 1

    else:

      t\_1 = tokens[i-1]

      if i == T - 1:

        # measure probability of ending the line

        add2dict(second\_order, (t\_1, t), 'END')

      if i == 1:

        # measure distribution of second word

        # given only first word

        add2dict(first\_order, t\_1, t)

      else:

        t\_2 = tokens[i-2]

        add2dict(second\_order, (t\_2, t\_1), t)

* For the last word , we are adding an ‘END’ token which will let our program know when to end a current line.

**Normalise the distributions to obtain probabilities**

initial\_total = sum(initial.values())

for t, c in initial.items():

    initial[t] = c / initial\_total

**Turn each list of possibilities into a dictionary of probabilities**

# convert [cat, cat, cat, dog, dog, dog, dog, mouse, ...]

# into {cat: 0.5, dog: 0.4, mouse: 0.1}

def list2pdict(ts):

  d = {}

  n = len(ts)

  for t in ts:

    d[t] = d.get(t, 0.) + 1

  for t, c in d.items():

    d[t] = c / n

  return d

**Applying this to first and second order**

for t\_1, ts in first\_order.items():

  first\_order[t\_1] = list2pdict(ts)

for k, ts in second\_order.items():

  second\_order[k] = list2pdict(ts)

**Sample word from probability dictionary**

def sample\_word(d):

  # print "d:", d

  p0 = np.random.random()

  # print "p0:", p0

  cumulative = 0

  for t, p in d.items():

    cumulative += p

    if p0 < cumulative:

      return t

  assert(False) # should never get here

We use a random probability and check it with the cumulative probability.

**Generate poetry**

def generate():

  for i in range(5): # generate 4 lines

    sentence = []

    # initial word

    w0 = sample\_word(initial)

    sentence.append(w0)

    # sample second word

    w1 = sample\_word(first\_order[w0])

    sentence.append(w1)

    # second-order transitions until END

    while True:

      w2 = sample\_word(second\_order[(w0, w1)])

      if w2 == 'END':

        break

      sentence.append(w2)

      w0 = w1

      w1 = w2

    print(' '.join(sentence))

* We first generate the first word, which is used to generate the second word and so on
* For the other words, we keep generating the words until an ‘END’ token in encountered

**Article Spinner**

An application of the language model discussed earlier. It uses the N-gram approach where the context word pairs are used to come up with replacements for existing words.

* Import dataset

df = pd.read\_csv('bbc\_text\_cls.csv')

* We will be choosing just one label and training our model on it

label = 'business’

texts = df[df['labels'] == label]['text']

**Collecting key values pairs of context pairs and middle word with its count**

* In accordance with the N-gram approach being used, we obtain a dictionary of dictionaries where the keys are the context words(preceding and next word) while the value is the possible middle word with the count of it’s occurrences

**key: (w(t-1),w(t+1) value : {w(t):count(w(t))}**

probs = {}

for doc in texts:

    lines = doc.split('/n')

    for line in lines:

        tokens = word\_tokenize(line)

        for i in range(len(tokens)-2):

            t\_0 = tokens[i]

            t\_1 = tokens[i+1]

            t\_2 = tokens[i+2]

            key = (t\_0,t\_2)

            if key not in probs:

                probs[key] = {}

            if t\_1 not in probs[key]:

                probs[key][t\_1] = 1

            else:

                probs[key][t\_1] += 1

**Normalize the probabilities**

* What we have currently are counts, we need to convert that into probability by normalizing them.

for t\_c,t in probs.items():

    total\_value = sum(t.values())

    for k,v in t.items():

        t[k] = v/total\_value

* We will be using the Detokenizer function to aggregate our obtained result into user comprehendible form

detokenizer = TreebankWordDetokenizer()

detokenizer.detokenize(word\_tokenize(texts.iloc[0].split("\n")[2]))

**Processing the paragraphs**

* The current dataset is a collection of paragraphs which will be splitting using the newline character before going through it line by line.

def SpinDocument(docs):

    lines = docs.split('\n')

    outputs = []

    for line in lines:

        if line:

            new\_line = SpinLine(line)

        else:

            new\_line = line

        outputs.append(new\_line)

    return '\n'.join(outputs)

**Function to sample words from probability dictionary**

def SampleWords(d):

    rand\_prob  = np.random.random()

    cumulative = 0

    for k,v in d.items():

        cumulative += v

        if rand\_prob < cumulative:

            return k

    assert(False)

**Function to tokenize the sentences and spin words**

def SpinLine(line):

    tokens = word\_tokenize(line)

    i = 0

    output = [tokens[0]]

    while i < len(tokens)-2:

        t\_0 = tokens[i]

        t\_1 = tokens[i+1]

        t\_2 = tokens[i+2]

        key = (t\_0,t\_2)

        prob\_dist = probs[key]

        if len(prob\_dist) > 1 and np.random.random() < 0.3:

            middle = SampleWords(prob\_dist)

            output.append(t\_1)

            output.append('<'+middle+'>')

            output.append(t\_2)

            i += 2

        else:

            output.append(t\_1)

            i += 1

    #append the final word only if there was no replacement

    if i == len(tokens) - 2:

        output.append(tokens[-1])

    return detokenizer.detokenize(output)

**Key considerations:**

* We don't spin the first word since there is no word preceding it. There we append the first word directly into the output list.
* For a given ‘i’ , we obtain ‘i+1’ and ‘i+2’th words. Here we will be using 2-gram which means ‘i’ and ‘i+2’th words will be forming context for the ‘i+1’ th word
* From the probability dictionary formed earlier, we obtain the probability distribution for the current key pair
* We make sure there are multiple word choices available by checking **len(prob\_dist) >1**. We then assign a 30% chance that a given word will be replaced using **np.random.random()**
* We sample a word from the probability distribution obtained and append it to the output dictionary. Since here we aren’t going for successive word replacements, we append the next word too and increment **i** by 2
* Finally, we only append the last word if there has been no replacement for it and then use detokenizer to obtain a meaningful paragraph/line.

**Generating spun articles**

i = np.random.choice(texts.shape[0])

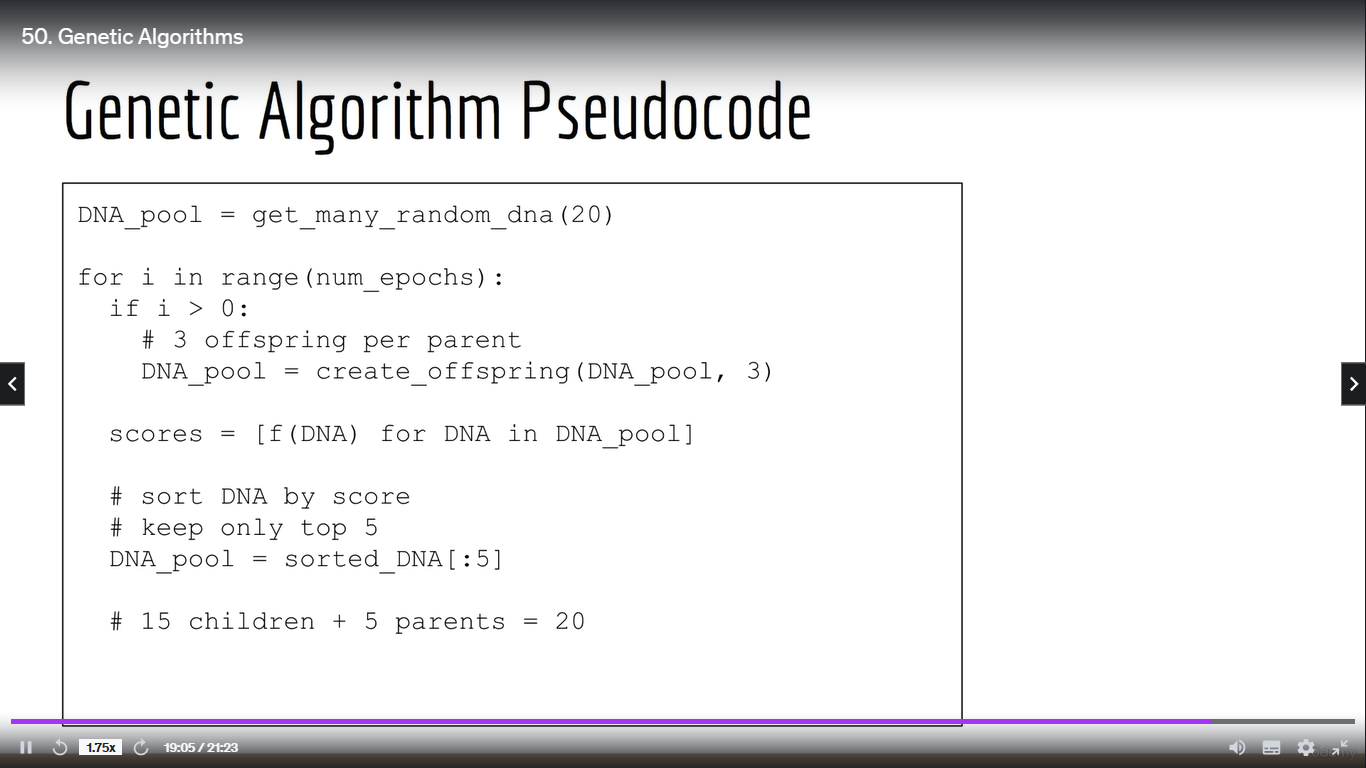
doc = texts.iloc[i]

new\_doc = SpinDocument(doc)

print(textwrap.fill(new\_doc,replace\_whitespace=False,fix\_sentence\_endings=True))

**Cipher Decryption**

**Genetic algorithm**



**Implementation**

**Creating the substitution cipher**

Initialise characters in lists, shuffle one of them and map them in a dictionary. Later, these mapping will be generated for each DNA string during decoding.

#One will act as the key, other will be the value

letters1 = list(string.ascii\_lowercase)

letters2 = list(string.ascii\_lowercase)

#Shuffle second set of letters

random.shuffle(letters2)

true\_mapping = {}

#Populate the map

for k,v in zip(letters1,letters2):

    true\_mapping[k] = v

**Language model**

**Initialize the Markov matrix (for transition) and initial state distribution.**

#Initialise Markov matrix to store bigram probabilities

M = np.ones((26,26)) #26 because of number of alphabets in english language

#Initialise initial state distribution

pi = np.zeros(26)

**Function to update the Markov matrix**

def update\_transition(ch1,ch2):

    i = ord(ch1) - 97

    j = ord(ch2) - 97

    M[i,j] += 1

**Function to update the initial state distribution**

def update\_pi(chr):

    i = ord(chr) - 97

    pi[i] +=1

**Getting log probability for each word**

The first alphabet moves into the initial state distribution whereas the subsequent transitions populate the Markov matrix.

def get\_word\_probability(word):

    i = ord(word[0]) - 97

    logp = np.log(pi[i])

    for ch in word[1:]:

        j = ord(ch) - 97

        logp += np.log(M[i,j]) #Update log prob

        i = j #Update j

    return logp

**Getting log probability for a sequence of words**

def get\_prob\_sequence(word):

    if type(word) == str:

        words = word.split()

    logp = 0

    for word in word:

        logp += get\_word\_probability(word)

    return logp

**Importing dataset and pre-processing data**

if not os.path.exists('moby\_dick.txt'):

  print("Downloading moby dick...")

  r = requests.get('https://lazyprogrammer.me/course\_files/moby\_dick.txt')

  with open('moby\_dick.txt', 'w') as f:

    f.write(r.content.decode())

#Replacing the non alphabet characters

regex = re.compile('[^a-zA-Z]')

**Creating a Markov Model**

#load the words

for line in open('moby\_dick.txt'):

    lines = line.rstrip()

    if line:

        line = regex.sub(' ',line) #Replace non alpha characters with space

        tokens = line.lower.split()

        for token in tokens:

            #First letter

            ch0 = token[0]

            update\_pi(ch0)

            #Other letters

            for ch in token[1:]:

                update\_transition(ch0,ch)

                ch0 = ch

**Normalize the count to obtain probabilities**

pi /= pi.sum()

M /= M.sum(axis=1,keepdims=True)

**Encode and decode functions**

def encode\_message(msg):

    msg = msg.lower()

    msg = regex.sub(' ',msg)

    coded\_msg = []

    for ch in msg:

        coded\_ch = ch

        if ch in true\_mapping:

            coded\_ch = true\_mapping[ch]

        coded\_msg.append(coded\_ch)

    return ''.join(coded\_msg)

encoded\_message = encode\_message(original\_message)

def decode\_message(msg,word\_map):

    decoded\_msg = []

    for ch in msg:

        decoded\_ch = ch

        if ch in word\_map:

            decoded\_ch = word\_map[ch]

        decoded\_msg.append(decoded\_ch)

    return ''.join(decoded\_msg)

The word map used as input to the decode function will be generated later as part of the evolutionary algorithm.

**Running an evolutionary algorithm to decode the message**

**Generating random DNA strings**

dna\_pool = []

for i in range(20):

    dna = list(string.ascii\_lowercase)

    random.shuffle(dna)

    dna\_pool.append(dna)

**Function to generate offsprings off given DNA strings**

* We generate offsprings by introducing mutations which like we have discussed can only be character swaps

def evolve\_offspring(dna\_pool,n\_children):

    #make n\_children per offspring

    offspring =[]

    for dna in dna\_pool:

        for i in n\_children:

            copy = dna.copy()

            i = np.random.randint(len(copy))

            j = np.random.randint(len(copy))

            #swapping the ith and jth characters

            copy[i],copy[j] = copy[j],copy[i]

            offspring.append(copy)

    return offspring + dna\_pool

* In every iteration, random indices within the DNA string are chosen and the elements are swapped . The offsprings are appended to the exsiting DNA pool

**Running the evolutionary algorithm**

num\_iters = 100

best\_score = float('-inf')

best\_dna = None

best\_map = None

scores = np.zeros(num\_iters)

for i in range(num\_iters):

    if i > 0:

        dna\_pool = evolve\_offspring(dna\_pool,3)

    #Calculate score for each dna

    dna2score = {}

    for dna in dna\_pool:

        #Populate word mapping for current dna string

        current\_map = {}

        for k,v in zip(letters1,dna):

            current\_map[k] = v

        decoded\_message = decode\_message(encoded\_message,current\_map)

        score = get\_prob\_sequence(decoded\_message)

        #store the current score

        #we can't have lists as dictionary keys, hence we convert it to a string

        dna2score[''.join(dna)] = score

        if score > best\_score:

            best\_dna = dna

            best\_map = current\_map

            best\_score = score

        #average score for this generation

        scores[i] = np.men(list(dna2score.values()))

* During every iteration, a DNA string is chosen and a word map is created based on that string. This word map is used to decode the encoded message and the probability score is calculated for this decoded message
* If this score is better than the previously stored best score, the current decoded message is considered as the best decoding result for the encoded message and the parameters are updated accordingly.

**Summary**

* Creating an initial word mapping based on which our message is encoded
* Initialising Markov matrix and initial state distribution which are populated once we have our input data
* Functions to get the probability score for a word where the first word prob goes into the initial state distribution whereas the subsequent letter transitions populate the Markov matrix. We also have a function to get the probability for a sequence of words
* Encode and decode functions. The latter uses the word maps generated for each dna string while trying to best decipher the code.
* Offsprings are created for existing DNA strings by randomly swapping characters within the DNA string.
* Running the evolutionary algorithm where at every iteration, a decoded message is obtained and it’s score is compared with the best score in order to decipher what the most likely decoded message might be.

**Spam Detection**

* Import and pre-process the dataset and split it into train and test set.

df = pd.read\_csv('spam.csv',encoding="ISO-8859-1")

Drop the unnecessary columns

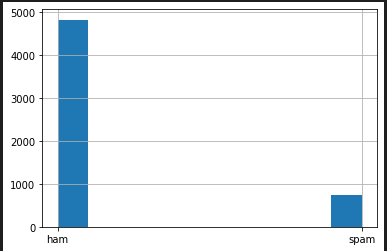
df = df.drop(['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'], axis=1)

Rename the columns

df.columns = ['Labels','Data']

Check if the dataset is imbalanced by plotting a histogram for the target label values.

df['Labels'].hist()



Create binary labels and extract the column as a numpy array

df['bin\_labels'] = df['Labels'].map({'ham':0,'spam':1})

Y = df['bin\_labels'].to\_numpy()

Split the data into train and test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Data'],Y,test\_size=0.33)

**Converting text into integers using Count vectorizer**

counter = CountVectorizer(decode\_error='ignore')

X\_train = counter.fit\_transform(X\_train)

X\_test = counter.transform(X\_test)

**Initialize and train Naïve bayes classifier**

nvb = MultinomialNB()

nvb.fit(X\_train,y\_train)

print('train score: ',nvb.score(X\_train,y\_train))

print('test score: ',nvb.score(X\_test,y\_test))

**Since the dataset is unbalanced, we go for calculating additional metrics:**

**F1 score**

P\_train = nvb.predict(X\_train)

P\_test = nvb.predict(X\_test)

print('train F1: ',f1\_score(y\_train,P\_train))

print('test F1: ',f1\_score(y\_test,P\_test))

**AUC score**

We are only considering the 2nd column(index 1) because that is what we are interested in the positive cases while dealing with ROC AUC Score.

Prob\_train = nvb.predict\_proba(X\_train)[:,1]

Prob\_test = nvb.predict\_proba(X\_test)[:,1]

print('train roc\_auc\_score: ',roc\_auc\_score(y\_train,Prob\_train))

print('test roc\_auc\_score: ',roc\_auc\_score(y\_test,Prob\_test))

**Confusion matrix**

cm = confusion\_matrix(y\_train,P\_train)

**Wordcloud representation to visualize most frequently occurring words**

def visualize(label):

  words = ''

  for msg in df[df['Labels'] == label]['Data']:

    msg = msg.lower()

    words += msg + ' '

  wordcloud = WordCloud(width=600, height=400).generate(words)

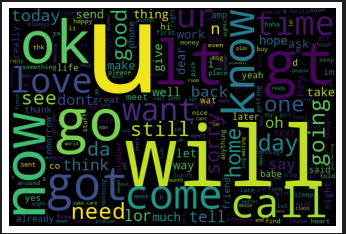
  plt.imshow(wordcloud)

  plt.axis('off')

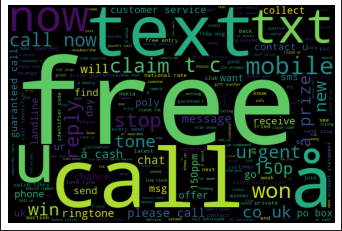
  plt.show()

**Non-spam words**

visualize('ham')



visualize('spam')



Things that should be spam but haven't been classified as spam.

sneaky\_spam = df[(df['bin\_labels'] == 1) & (df['predictions'] == 0)]['Data']

for msg in sneaky\_spam:

    print(msg)

Things that should not be spam but have been mislabelled

wrongly\_labelled\_spam = df[(df['bin\_labels'] == 1) & (df['predictions'] == 0)]['Data']

for msg in wrongly\_labelled\_spam:

    print(msg)

**Sentiment Analysis**

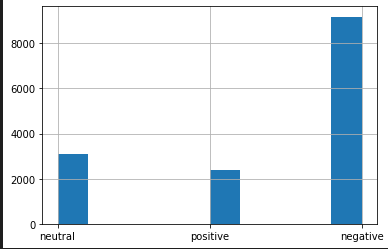
Logistic regression classifier will be used in this task . Import the dataset and choose only the columns that will aid us in our sentiment analysis task.

df\_ = pd.read\_csv('AirlineTweets.csv')

df = df\_[['airline\_sentiment','text']]

We then check if our class representations are imbalanced by plotting the histogram for the target label.

df['airline\_sentiment'].hist()



We have an imbalanced class situation and therefore we must go for performance metrics other than accuracy to obtain a clear picture.

**Assigning integer values to the categorical label using the map function**

target = {'positive':1,'negative':0,'neutral':2}

df['target'] =df['airline\_sentiment'].map(target)

**Train -test split**

df\_train,df\_test = train\_test\_split(df,test\_size=0.33)

**Obtaining integer representation for the text data using Tfidf vectorizer**

tfidf = TfidfVectorizer()

X\_train = tfidf.fit\_transform(df\_train['text'])

X\_test = tfidf.transform(df\_test['text'])

y\_train = df\_train['target']

y\_test = df\_test['target']

**Training the logistic regression model and obtaining the accuracy score**

model = LogisticRegression(max\_iter=500)

model.fit(X\_train,y\_train)

print(f'Train score is {model.score(X\_train,y\_train)}')

print(f'Test score is {model.score(X\_test,y\_test)}')

Since our dataset is imbalanced, we then proceed to the roc\_auc\_score. In case of binary classification we had only chosen the class corresponding to one of the target labels but in case of multiclass classification we use the entire probability prediction matrix.

Prob\_train = model.predict\_proba(X\_train)

Prob\_test = model.predict\_proba(X\_test)

print("Train roc auc score is: " ,roc\_auc\_score(y\_train,Prob\_train,multi\_class='ovo'))

print("Test roc auc score is: " ,roc\_auc\_score(y\_test,Prob\_test,multi\_class='ovo'))

**‘ovo’: One vs One :**This splits the multi class classification problem into multiple binary classification problems (every class against every other class)

**Confusion matrix**

p\_train = model.predict(X\_train)

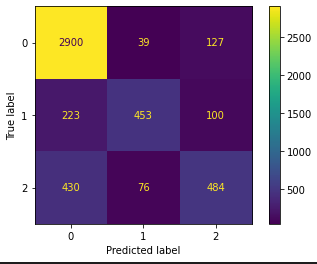
p\_test = model.predict(X\_train)

cm = confusion\_matrix(y\_train,p\_train,normalize='true')

For test set:

from sklearn.metrics import plot\_confusion\_matrix

plot\_confusion\_matrix(model,X\_test,y\_test)



For train set:

plot\_confusion\_matrix(model,X\_train,y\_train,normalize='true')

For the same problem statement, we will now proceed by building a binary classifier by only considering the positive and negative sentiment examples.

**Building a binary model and interpreting model coefficients**

Choose only the positive and negative tweets (leaving out the neutral).

interested =['positive','negative']

X = df[df['airline\_sentiment'] .isin(interested)]['text']

y = df[df['airline\_sentiment'] .isin(interested)]['target']

Train the model and obtain accuracy scores:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.33)

X\_train = tfidf.fit\_transform(X\_train)

X\_test = tfidf.transform(X\_test)

model.fit(X\_train,y\_train)

print(f'train score is {model.score(X\_train,y\_train)}')

print(f'test score is {model.score(X\_test,y\_test)}')

**Obtaining the model weights:**

model.coef\_

Obtaining the word mapping formed by the vectorizer based on the text corpus fed to it:

word\_map = tfidf.vocabulary\_

* Now using thresholds, lets try to display the most positive and negative words.

**To get the most positive word, choose a high positive threshold**

threshold = 2

for word,index in word\_map.items():

    weight = model.coef\_[0][index]

    if weight > threshold:

        print(word,weight)

**Next, we try to obtain the most negative words as per the classifier:**

threshold = -2

for word,index in word\_map.items():

    weight = model.coef\_[0][index]

    if weight < threshold:

        print(word,weight)

**Deep Learning**

**The Neuron**

**Best fit line**

**Text classifier using neurons**

**Feedforward neural networks**

Universal approximation theorem: Any mathematical function can be modelled using a neural network with a single hidden layer.