Author Writings Identification with NLP and Topic Modelling

Introduction

In this notebook, I will attempt a basic topic modeling analysis of the Spooky Author dataset. Topic modeling involves uncovering abstract themes or 'topics' within a corpus of text based on its underlying documents and words. I will introduce two standard topic modeling techniques: the first technique is known as Latent Dirichlet Allocation (LDA), and the second is Non-negative Matrix Factorization (NMF). Additionally, I will cover some fundamental concepts in Natural Language Processing (NLP) such as Tokenization, Stemming, and the vectorization of raw text. These concepts will be helpful when making predictions with learning models.

The outline of this notebook is as follows:

1. Exploratory Data Analysis (EDA) and Wordclouds involve analyzing the data by generating simple statistics, such as word frequencies across different authors, and creating word clouds, some of which may incorporate image masks.

1. Natural Language Processing (NLP) using NLTK (Natural Language Toolkit) involves introducing fundamental text processing methods like tokenization, stop-word removal, stemming, and text vectorization using techniques such as term frequencies (TF) and inverse document frequencies (TF-IDF).

2. Implementing two topic modeling techniques, Latent Dirichlet Allocation (LDA) and Non-negative Matrix Factorization (NMF).

Import Libraries

In [7]:
```
import base64
import numpy as np
import pandas as pd

# Plotly imports
import plotly.offline as py
py.init_notebook_mode(connected=True)
import plotly.graph_objs as go
import plotly.tools as tls

# Other imports
from collections import Counter
from scipy.misc import imread
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import NMF, LatentDirichletAllocation
from matplotlib import pyplot as plt
%matplotlib inline
```

In [8]:
```
# Load in the training data
train = pd.read_csv("../input/spooky-author-identification/train.csv")
```

EDA

Take a look at what the first three rows in the data has in store for us and who exactly are the authors

In [9]:
```
train.head()
```

Out[9]:

	id	text	author
0	id26305	This process, however, afforded me no means of...	EAP
1	id17569	It never once occurred to me that the fumbling...	HPL
2	id11008	In his left hand was a gold snuff box, from wh...	EAP
3	id27763	How lovely is spring As we looked from Windsor...	MWS
4	id12958	Finding nothing else, not even gold, the Super...	HPL

Take a look at how large the training data is:

In [10]:
```
print(train.shape)
```

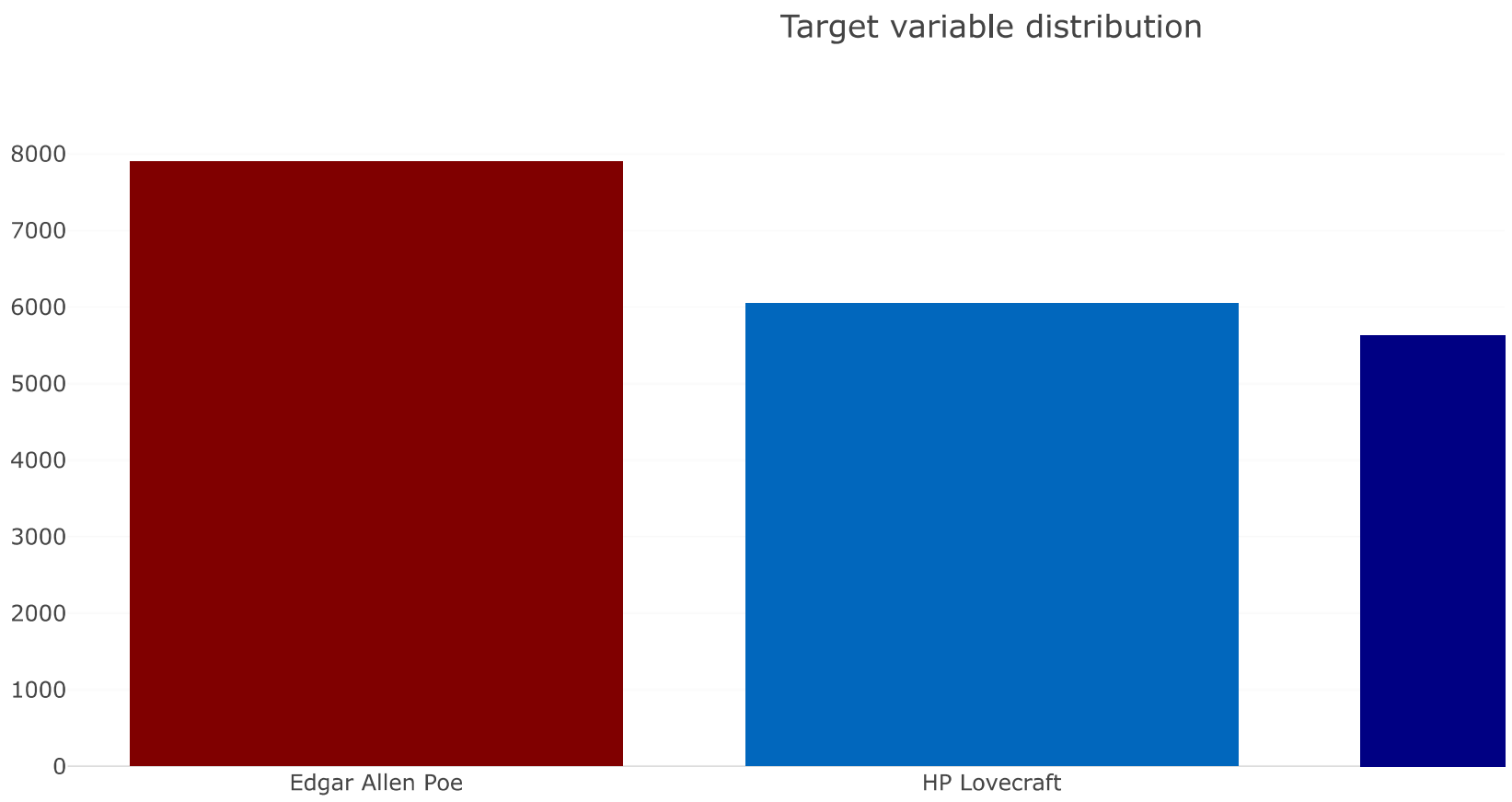
```
(19579, 3)
```

Summary statistics of the training set

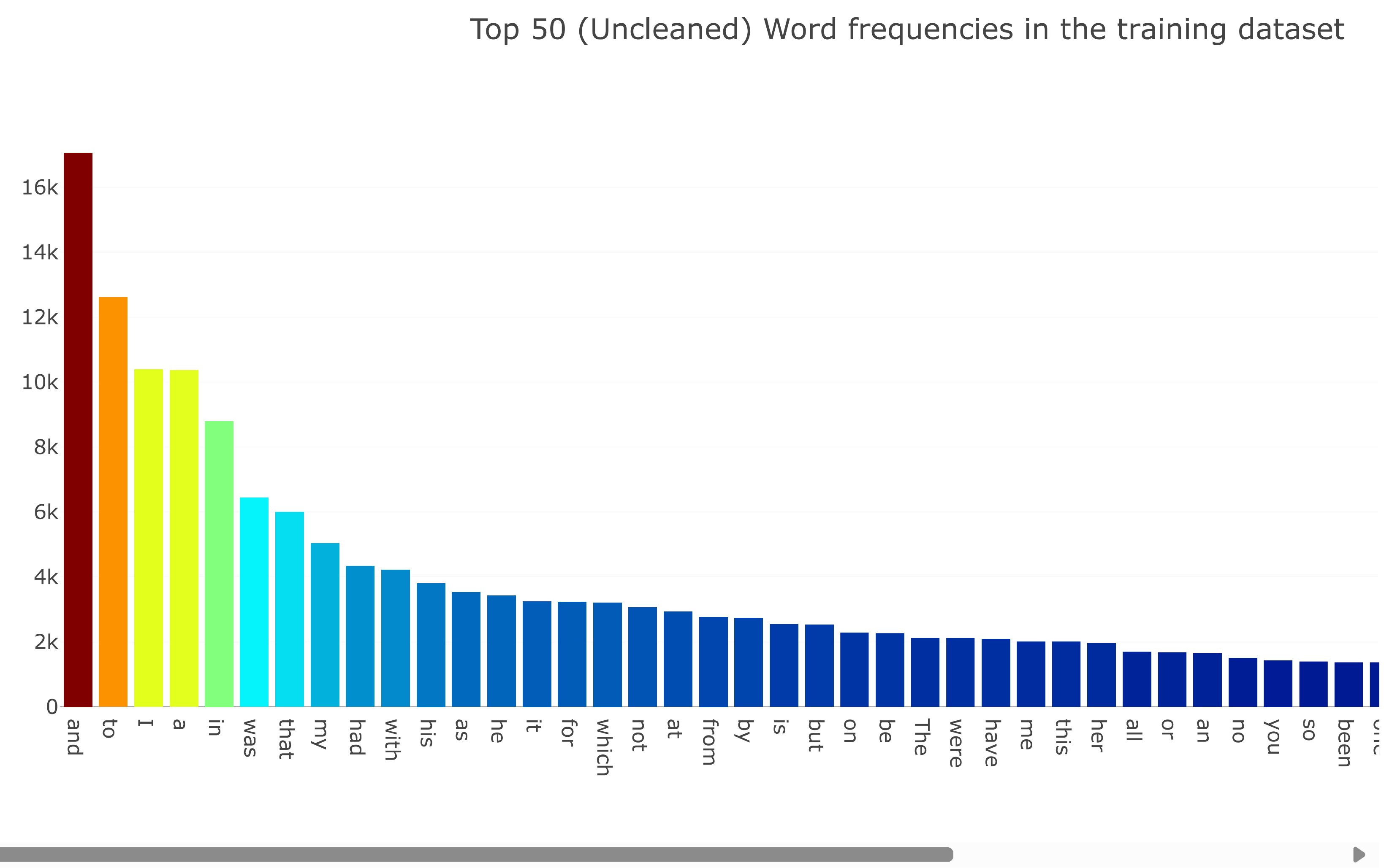
we'll visualize basic statistics within the data, such as the distribution of entries for each author. To do this, I'll use the Plot.ly visualization library to create simple bar plots. Feel free to unhide the cell below if you'd like to see the Plot.ly code.

In [11]:
```
z = {'EAP': 'Edgar Allen Poe', 'MWS': 'Mary Shelley', 'HPL': 'HP Lovecraft'}
data = [go.Bar(
            x = train.author.map(z).unique(),
            y = train.author.value_counts().values,
            marker= dict(colorscale='Jet',
                         color = train.author.value_counts().values
                        ),
```

```
            text='Text entries attributed to Author'
    )]

layout = go.Layout(
    title='Target variable distribution'
)

fig = go.Figure(data=data, layout=layout)

py.iplot(fig, filename='basic-bar')
```



In [12]:
```
all_words = train['text'].str.split(expand=True).unstack().value_counts()
data = [go.Bar(
            x = all_words.index.values[2:50],
            y = all_words.values[2:50],
            marker= dict(colorscale='Jet',
                         color = all_words.values[2:100]
                        ),
            text='Word counts'
    )]

layout = go.Layout(
    title='Top 50 (Uncleaned) Word frequencies in the training dataset'
)

fig = go.Figure(data=data, layout=layout)

py.iplot(fig, filename='basic-bar')
```



Notice anything peculiar about the words depicted in this word frequency plot? Do these words truly convey the themes and concepts that Mary Shelley aims to convey in her stories?

These words are quite common and appear not only in spooky stories and novels by our three authors but also in newspapers, children's books, religious texts, and nearly every other English text. Consequently, we need to preprocess our dataset initially to eliminate these commonly occurring words that contribute little to our analysis.

WordClouds to visualise each author's work

An invaluable visualization tool for data scientists in natural language processing is the 'Word Cloud.' This graphical representation, as the name implies, forms an image composed of distinct words extracted from a text or book, where the size of each word corresponds to its frequency within the text (i.e., the number of times it appears). In this case, we'll be extracting words from the 'text' column.

To begin, we'll store the text of each author in a Python list. Initially, we'll create three separate Python lists to contain the texts of Edgar Allan Poe, H.P. Lovecraft, and Mary Shelley, respectively.

In [13]:
```
eap = train[train.author=="EAP"]["text"].values
hpl = train[train.author=="HPL"]["text"].values
mws = train[train.author=="MWS"]["text"].values
```

Import the python module "Wordcloud".

In [14]:
```
from wordcloud import WordCloud, STOPWORDS
```

However, a typical word cloud might seem mundane. Hence, I'd like to introduce an exciting technique that involves importing relevant pictures and using their outlines as masks for our word clouds. For this purpose, I've chosen images that I believe best represent each author:

The Raven for Edgar Allan Poe

Octopus Cthulhu-inspired illustration for H.P. Lovecraft

Frankenstein's monster for Mary Shelley

First, I derive the Base64 encoding of the chosen images and then reintroduce these images into the notebook using that particular encoding. The cell below contains the Base64 encoding of the three images I'll be using, which I've hidden to avoid cluttering this notebook with lengthy text. Feel free to unhide them if you're interested in viewing the encoding.

In [15]:

```
eap_64 = b'iVBORw0KGgoAAAANSUhEUgAAAoAAAAGHCAYAAAAgIOMGAAB+7klEQVR42u3deXhV1b0+8HcNG2QKQ04YnFGrVisZGB3A2hZBIThACFasE3pl
```


In [16]:

```
hpl_64 = b'iVBORw0KGgoAAAANSUhEUgAAAgAAAAIACAYAAAD0eNT6AAAACXBIWXMAASdHAAEnRwEEDsU+AAAKT2lDQ1BQaG90b3Nob3AgSUNDIHByb2Zpb
```


In [17]:

```
mws_64 = b'iVBORw0KGgoAAAANSUhEUgAAAoAAAAJOCAYAAAAnL7bnAAB2xElEQVR42u3dd5hU1f0G8PeUS1lAygKCIEhH0VgoYgkgIAoqCNbYsCGiIJoY
```


To decode the image, I import the 'codecs' module and save it as a new image within this notebook. Once the image is saved, I can effortlessly load it as a mask using the following steps:

In [18]:
```
import codecs
# Generate the Mask for EAP
f1 = open("eap.png", "wb")
```

```
f1.write(codecs.decode(eap_64,'base64'))
f1.close()
img1 = imread("eap.png")
# img = img.resize((980,1080))
hcmask = img1

f2 = open("mws.png", "wb")
f2.write(codecs.decode(mws_64,'base64'))
f2.close()
img2 = imread("mws.png")
hcmask2 = img2

f3 = open("hpl.png", "wb")
f3.write(codecs.decode(hpl_64,'base64'))
f3.close()
img3 = imread("hpl.png")
hcmask3 = img3;
```

```
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:6: DeprecationWarning:

`imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:13: DeprecationWarning:

`imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:19: DeprecationWarning:

`imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

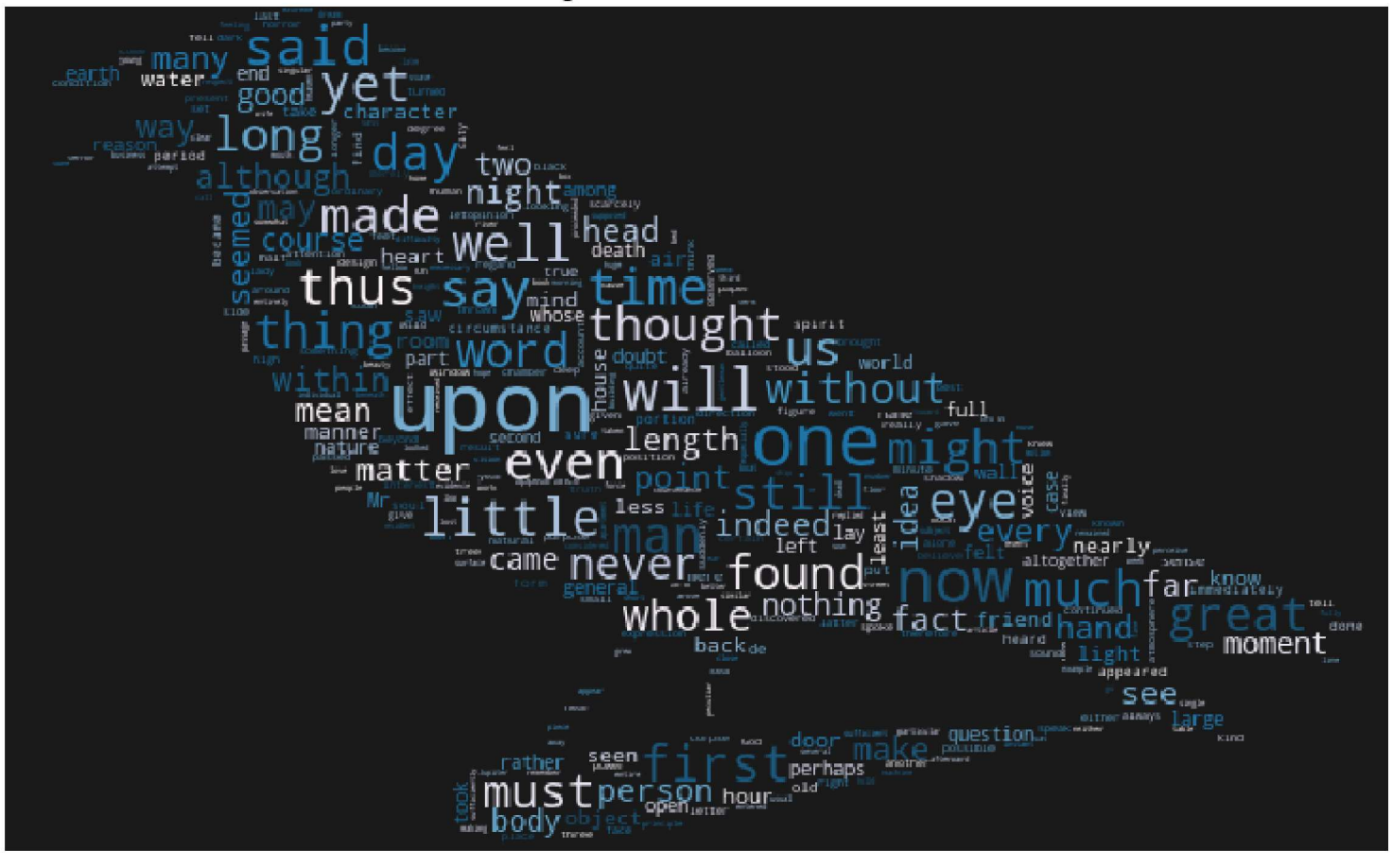
Finally, the word clouds are plotted using the following lines

In [19]:
```
# The wordcloud of Cthulhu/squidy thing for HP Lovecraft
plt.figure(figsize=(16,13))
wc = WordCloud(background_color="black", max_words=10000,
               mask=hcmask3, stopwords=STOPWORDS, max_font_size= 40)
wc.generate(" ".join(hpl))
plt.title("HP Lovecraft (Cthulhu-Squidy)", fontsize=20)
# plt.imshow(wc.recolor( colormap= 'Pastel1_r' , random_state=17), alpha=0.98)
plt.imshow(wc.recolor( colormap= 'Pastel2' , random_state=17), alpha=0.98)
plt.axis('off')
```

Out[19]:
```
(-0.5, 511.5, 511.5, -0.5)
```

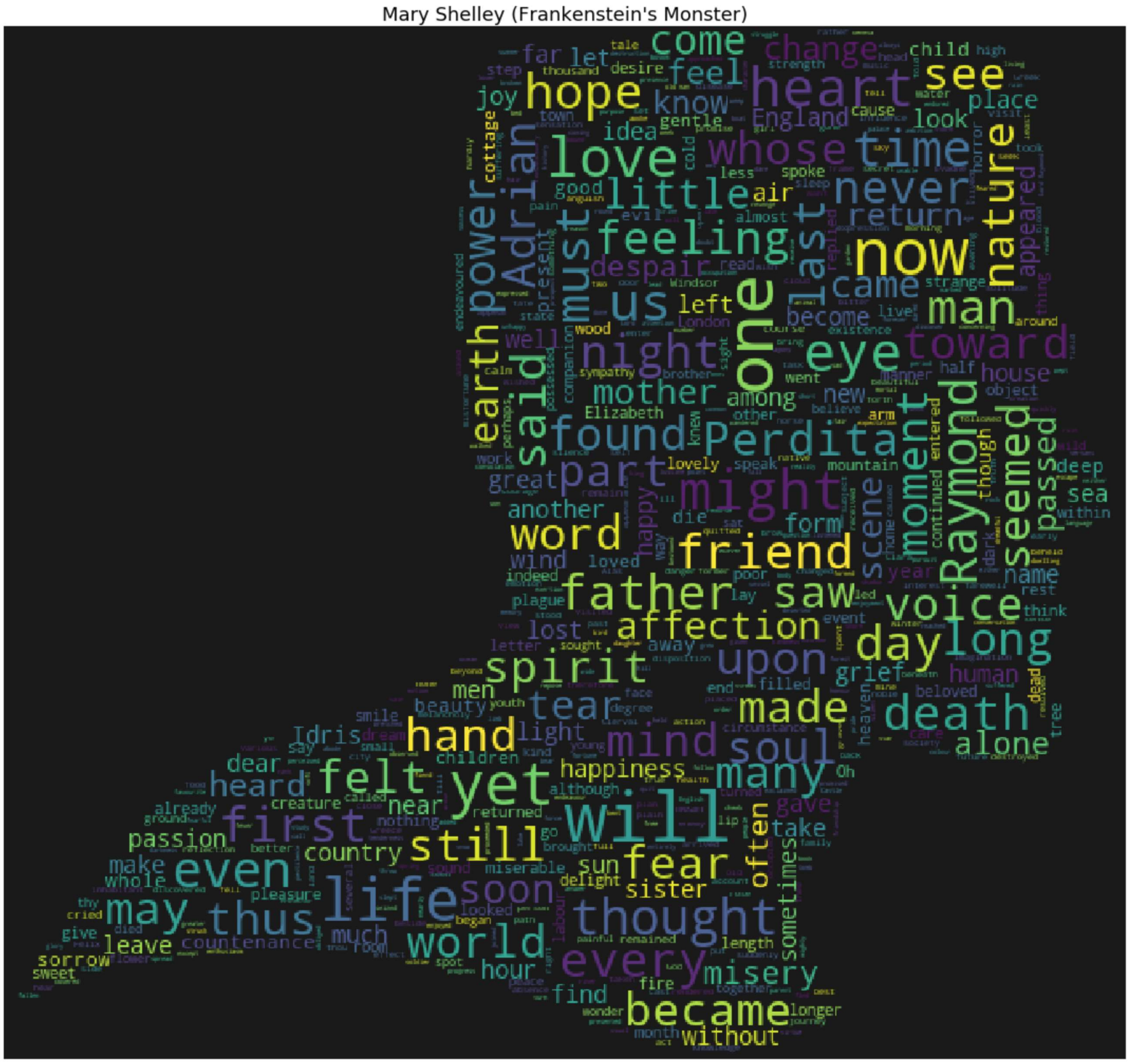
In [20]:
```
plt.figure(figsize=(20,18))
# The wordcloud of the raven for Edgar Allen Poe
plt.subplot(211)
wc = WordCloud(background_color="black",
               max_words=10000,
               mask=hcmask,
               stopwords=STOPWORDS,
               max_font_size= 40)
wc.generate(" ".join(eap))
plt.title("Edgar Allen Poe (The Raven)")
plt.imshow(wc.recolor( colormap= 'PuBu' , random_state=17), alpha=0.9)
plt.axis('off')
```

Out[20]:
```
(-0.5, 639.5, 390.5, -0.5)
```



In [21]:
```
plt.figure(figsize=(20,18))
wc = WordCloud(background_color="black",
               max_words=10000,
               mask=hcmask2,
               stopwords=STOPWORDS,
               max_font_size= 40)
wc.generate(" ".join(mws))
plt.title("Mary Shelley (Frankenstein's Monster)", fontsize= 18)
plt.imshow(wc.recolor( colormap= 'viridis' , random_state=17), alpha=0.9)
plt.axis('off')
```

Out[21]:
```
(-0.5, 639.5, 589.5, -0.5)
```



There you have it—three distinct word clouds, one for each of our spooky authors. From these word clouds, we immediately discern some of the favored words used by each author.

For instance, H.P. Lovecraft's cloud highlights words like 'dream,' 'time,' 'strange,' 'past,' and 'ancient,' resonating with themes prominent in the author's work—themes surrounding the hidden psyche, esoteric fate and chance, the infamous creature Cthulhu, and mentions of ancient cults and rituals associated with it.

Conversely, Mary Shelley's word cloud revolves around primal instincts and a spectrum of morality—from positive to negative—featuring words like 'friend,' 'fear,' 'hope,' and 'spirit.' These themes echo in her notable work, such as Frankenstein.

However, upon examining the word clouds, some words appear out of place, such as 'us,' 'go,' and 'he,' which commonly appear across various texts.

Natural Language Processing

In nearly all Natural Language Processing tasks—whether it's topic modeling, word clustering, document-text classification, and so on—certain preprocessing steps are almost always necessary. These steps aim to transform raw input text into a format understandable by both your model and the machine. You can't simply input a paragraph of words into a Random Forest model and expect it to accurately predict the paragraph's authorship. Behind the scenes, text preprocessing essentially involves these fundamental steps:"

Tokenization: Involves the segregation of text into its individual constituent words.

Stopwords: In this step, we discard words that occur too frequently, as their high frequency doesn't contribute much to detecting relevant texts. As an additional note, it's worth considering the removal of words that occur very infrequently.

Stemming: This process combines variants of words into a single parent word that still conveys the same meaning.

Vectorization: This step involves converting text into vector format. One of the simplest approaches is the famous bag-of-words method, where a matrix is created for each document or text in the corpus. In its simplest form, this matrix stores word frequencies (word counts) and is often referred to as the vectorization of raw text.

Natural Language Toolkit (NLTK): To enhance our Natural Language Processing tasks, let me introduce you to one of the most convenient toolkits in NLP—the Natural Language Toolkit, commonly known as the NLTK module. Importing the toolkit is as simple as:

In [22]:
```
import nltk
```

Tokenization

The concept of tokenization involves taking a sequence of characters (akin to Python strings) in a given document and segmenting it into its individual constituent pieces, known as 'tokens.' These tokens can be loosely compared to singular words in a sentence. A naïve approach might involve using the 'split()' method on a string, which divides it into a Python list based on the identifier in the argument. However, it's not as straightforward as it seems.

For instance, let's split the first sentence of the text in the training data using a space:

In [23]:
```
# Storing the first text element as a string
first_text = train.text.values[0]
print(first_text)
print("="*90)
print(first_text.split(" "))
```

```
This process, however, afforded me no means of ascertaining the dimensions of my dungeon; as I might make its circuit,
and return to the point whence I set out, without being aware of the fact; so perfectly uniform seemed the wall.
==========================================================================================
['This', 'process,', 'however,', 'afforded', 'me', 'no', 'means', 'of', 'ascertaining', 'the', 'dimensions', 'of', 'm
y', 'dungeon;', 'as', 'I', 'might', 'make', 'its', 'circuit,', 'and', 'return', 'to', 'the', 'point', 'whence', 'I', 's
et', 'out,', 'without', 'being', 'aware', 'of', 'the', 'fact;', 'so', 'perfectly', 'uniform', 'seemed', 'the', 'wall.']
```

However, as evident from this initial attempt at tokenization, the segmentation of the sentence into its individual elements or terms is not entirely accurate. For instance, observe the second element of the list, which contains the term 'process,'. The punctuation mark (comma) is included and treated as a part of the word 'process,' forming a single term. Ideally, we'd prefer the comma and the word to be in two distinct elements of the list. Achieving this with pure Python list operations would be rather complex. This is where the NLTK library becomes invaluable. The NLTK offers a convenient method, 'word_tokenize()' (TreebankWord tokenizer), which automatically separates individual words and punctuations into distinct elements, as shown below:

In [24]:
```
first_text_list = nltk.word_tokenize(first_text)
print(first_text_list)
```

```
['This', 'process', ',', 'however', ',', 'afforded', 'me', 'no', 'means', 'of', 'ascertaining', 'the', 'dimensions', 'o
f', 'my', 'dungeon', ';', 'as', 'I', 'might', 'make', 'its', 'circuit', ',', 'and', 'return', 'to', 'the', 'point', 'wh
ence', 'I', 'set', 'out', ',', 'without', 'being', 'aware', 'of', 'the', 'fact', ';', 'so', 'perfectly', 'uniform', 'se
emed', 'the', 'wall', '.']
```

Stopword Removal

As mentioned earlier, stopwords refer to words that appear frequently in a corpus but hold little discriminatory value in distinguishing texts for learning or predictive purposes. These stopwords encompass terms like 'to' or 'the,' which offer minimal contribution to the learning process. Hence, it's advantageous to exclude them during the pre-processing phase. Conveniently, NLTK provides a predefined list of 153 English stopwords. You can reveal this list by unhiding the second cell below.

In [25]:
```
stopwords = nltk.corpus.stopwords.words('english')
len(stopwords)
```

Out[25]:
```
179
```

In [26]:
```
print(stopwords)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yo
urs', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "tha
t'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do',
'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'b
y', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 't
o', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'ther
e', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no',
'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'sho
uld', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn',
"didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn',
"mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'were
n', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

To filter out stop words from our tokenized list of words, we can simply use a list comprehension as follows:

In [27]:
```
first_text_list_cleaned = [word for word in first_text_list if word.lower() not in stopwords]
print(first_text_list_cleaned)
print("="*90)
print("Length of original list: {0} words\n"
      "Length of list after stopwords removal: {1} words"
      .format(len(first_text_list), len(first_text_list_cleaned)))
```

```
['process', ',', 'however', ',', 'afforded', 'means', 'ascertaining', 'dimensions', 'dungeon', ';', 'might', 'make', 'c
ircuit', ',', 'return', 'point', 'whence', 'set', ',', 'without', 'aware', 'fact', ';', 'perfectly', 'uniform', 'seeme
d', 'wall', '.']
============================================================================================
Length of original list: 48 words
Length of list after stopwords removal: 28 words
```

As you can see, our list, with stopwords removed, is now substantially shorter. Commonly occurring words like 'I,' 'me,' 'to,' and 'the' have been eliminated.

Stemming and Lemmatization

After the removal of stopwords, the subsequent stage in NLP that I'd like to introduce is stemming. This step aims to condense different variations of similar words into a single term (reducing different branches into a single word stem). For instance, 'running,' 'runs,' and 'run' ideally collapse into the word 'run,' although this process sacrifices the granularity of past, present, or future tense.

We can once again turn to NLTK, which offers various stemmers like the Porter stemming algorithm, the Lancaster stemmer, and the Snowball stemmer. In the following example, I'll create a Porter stemmer instance:

In [28]:
```
stemmer = nltk.stem.PorterStemmer()
```

Now, let's use the stemmer to check if it can reduce these test words ('running', 'runs', 'run') into a single stemmed word. Conveniently, we can test the stemmer on the fly, like this:

In [29]:
```
print("The stemmed form of running is: {}".format(stemmer.stem("running")))
print("The stemmed form of runs is: {}".format(stemmer.stem("runs")))
print("The stemmed form of run is: {}".format(stemmer.stem("run")))
```

```
The stemmed form of running is: run
The stemmed form of runs is: run
The stemmed form of run is: run
```

We can see that the stemmer has effectively reduced the provided words into their base forms. This will significantly assist in reducing the size of our dataset of words during learning and classification tasks.

However, there is a flaw with stemming—it relies on a rather crude heuristic of chopping off word endings in an attempt to reduce each word to a recognizable base form. As a result, this process disregards vocabulary and word forms, as illustrated in the following example:

In [30]:
```
print("The stemmed form of leaves is: {}".format(stemmer.stem("leaves")))
```

```
The stemmed form of leaves is: leav
```

Lemmatization to the rescue

Hence, we explore an alternative to stemming known as lemmatization. Unlike stemming, lemmatization aims to achieve a similar effect while considering an actual dictionary or vocabulary (the Lemma). It doesn't truncate words into stemmed forms that lack lexical meaning. We can once again use NLTK to initialize a lemmatizer (WordNet variant) and observe how it handles word reduction:

In [31]:
```
from nltk.stem import WordNetLemmatizer
lemm = WordNetLemmatizer()
print("The lemmatized form of leaves is: {}".format(lemm.lemmatize("leaves")))
```

```
The lemmatized form of leaves is: leaf
```

We can observe that our lemmatizer effectively collapses words into forms that hold more lexical sense.

Vectorizing Raw Text

In the vast expanse of NLP literature, various purposes drive the analysis of raw text. Some cases involve comparing the similarity between bodies of text (utilizing clustering techniques or distance measurements), text classification (the objective of this competition), and discovering the underlying topics within a body of text (the focus of this notebook). With the goal of uncovering topics in mind, we now need to consider how to input raw text into a machine learning model.

Having previously discussed tokenization, stopword removal, and stemming (or perhaps lemmatization), we've managed to refine our text dataset considerably from its original state. However, at this stage, our raw text, while readable to humans, unfortunately remains incomprehensible to machines. Since machines comprehend bits and numbers, our next step involves converting our text into numerical form. For this purpose, we'll employ a commonly used approach known as the Bag-of-Words.

The Bag of Words approach

This approach builds upon word counts as its foundation, recording the occurrence of each word from the entire text in a vector specific to that word. For instance, given these two sentences: 'I love to eat Burgers' and 'I love to eat Fries,' we first tokenize the text to create a vocabulary of six words: [I, love, to, eat, Burgers, Fries].

Implementing the Bag-of-Words approach involves creating six distinct vectors, one for each word. Now, you might wonder, with rows representing numbers instead of text, what forms the columns or features? Each word in this transformed dataset becomes an individual feature or column.

To exemplify this, I'll utilize the CountVectorizer method from the Scikit-learn library to implement a vectorizer that generates word count vectors (term frequencies), as shown below:

In [32]:
```
# Defining our sentence
sentence = ["I love to eat Burgers",
            "I love to eat Fries"]
vectorizer = CountVectorizer(min_df=0)
sentence_transform = vectorizer.fit_transform(sentence)
```

Fitting the vectorizer to the dataset

Here, we initialize and create a basic term frequency object using the CountVectorizer function, which we simply name 'vectorizer.' The parameters I've explicitly provided (with the rest set as default) represent the bare minimum. In this context, 'min_df' refers to the minimum document frequency, indicating that the vectorizer will discard all words occurring less frequently than the specified value (either an integer or a fraction).

Finally, the 'fit_transform' method consists of two steps. Firstly, the 'fit' method maps the vectorizer to the dataset you provide. Subsequently, the actual transformation occurs via the 'transform' method, converting the raw text into its vector form, as demonstrated below:

In [33]:
```
print("The features are:\n {}".format(vectorizer.get_feature_names()))
print("\nThe vectorized array looks like:\n {}".format(sentence_transform.toarray()))
```

```
The features are:
 ['burgers', 'eat', 'fries', 'love', 'to']

The vectorized array looks like:
 [[1 1 0 1 1]
 [0 1 1 1 1]]
```

In [34]:
```
sentence_transform
```

Out[34]:
```
<2x5 sparse matrix of type '<class 'numpy.int64'>'
        with 8 stored elements in Compressed Sparse Row format>
```

Sparse matrix vector ouptuts

From the output of the vectorized text, we observe that the features comprise the words present in the text corpus we provided (in this case, the two sentences we defined earlier). Simply calling the 'get_feature_names' attribute from the vectorizer allows us to inspect these features.

Regarding the transformed text, attempting to inspect its values directly yields a message stating, 'sparse matrix of type class 'numpy.int64' with 8 stored elements in Compressed Sparse Row format.' This signifies that the vectorizer returns the transformed raw text as a matrix with most values being zero or nearly negligible—hence, the term 'sparse.' Considering this, it's reasonable that our resulting matrices exhibit a high degree of sparsity due to.

Topic modelling

Implement two different topic modelling techniques as follows:

1. **Latent Dirichlet Allocation** - A probabilistic generative model that reveals latent topics within a dataset by assigning weights to words in a corpus. Each topic assigns varying probability weights to individual words.
2. **Non-negative Matrix Factorization** - An approximation method that accepts an input matrix and approximates the factorization of this matrix into two other matrices, with the condition that the values in the matrix remain non-negative.

In [35]:
```
# Define helper function to print top words
def print_top_words(model, feature_names, n_top_words):
    for index, topic in enumerate(model.components_):
        message = "\nTopic #{}:".format(index)
        message += " ".join([feature_names[i] for i in topic.argsort()[:-n_top_words - 1 :-1]])
        print(message)
        print("="*70)
```

Putting all the preprocessing steps together

This presents the perfect opportunity to consolidate all the text preprocessing steps mentioned in the previous section. You might wonder if it's necessary to go through the entire process again—defining tokenization, stopword removal, stemming/lemmatizing, and so forth.

Fortunately, there's no need to repeat all of that. I deliberately left out a crucial detail about Sklearn vectorizers, which I'll clarify now. When you use CountVectorizer to vectorize raw text, the two stages of tokenizing and stopword filtering are automatically integrated as high-level components. Unlike the NLTK tokenizer introduced earlier in Section 2a, Sklearn's tokenizer automatically discards all single-character terms ('a', 'w', etc.) and lowercases all terms by default. For stopword filtering in Sklearn, simply passing the value 'english' into the 'stop_words' argument triggers the use of a built-in English stopword list.

However, there's no built-in lemmatizer in the vectorizer. We have a couple of options: either implementing it separately before feeding the data for vectorizing or extending the Sklearn implementation to include this functionality. Fortunately, the latter option is available to us by extending the CountVectorizer class and overwriting the 'build_analyzer' method, as shown below:

Extending the CountVectorizer class with a lemmatizer

In [36]:
```
lemm = WordNetLemmatizer()
class LemmaCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(LemmaCountVectorizer, self).build_analyzer()
        return lambda doc: (lemm.lemmatize(w) for w in analyzer(doc))
```

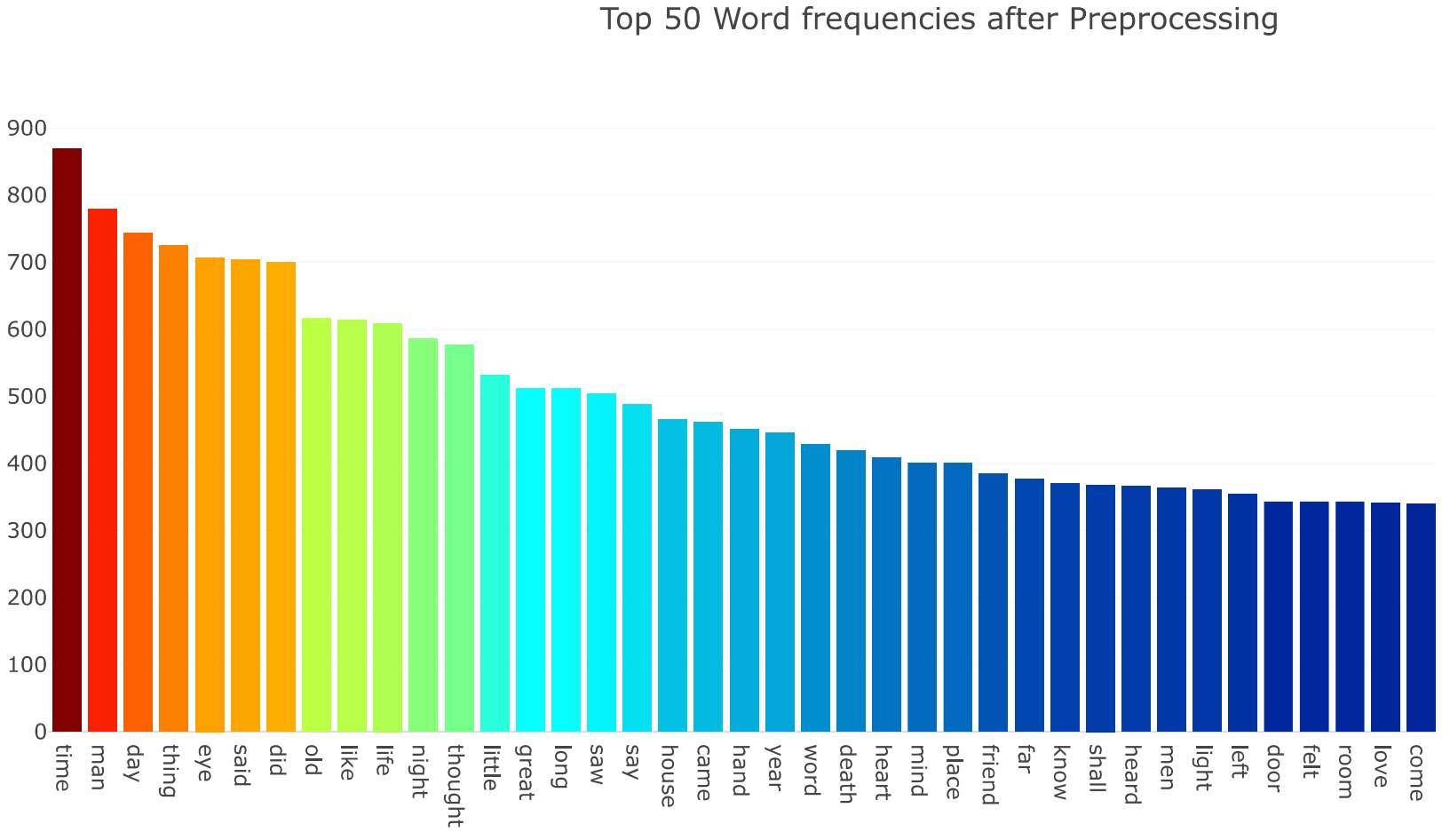
In this case, we've applied some nuanced concepts from Object-Oriented Programming (OOP). Essentially, we've inherited and subclassed Sklearn's original CountVectorizer class, then overwritten the 'build_analyzer' method to include the lemmatizer for each list in the raw text matrix.

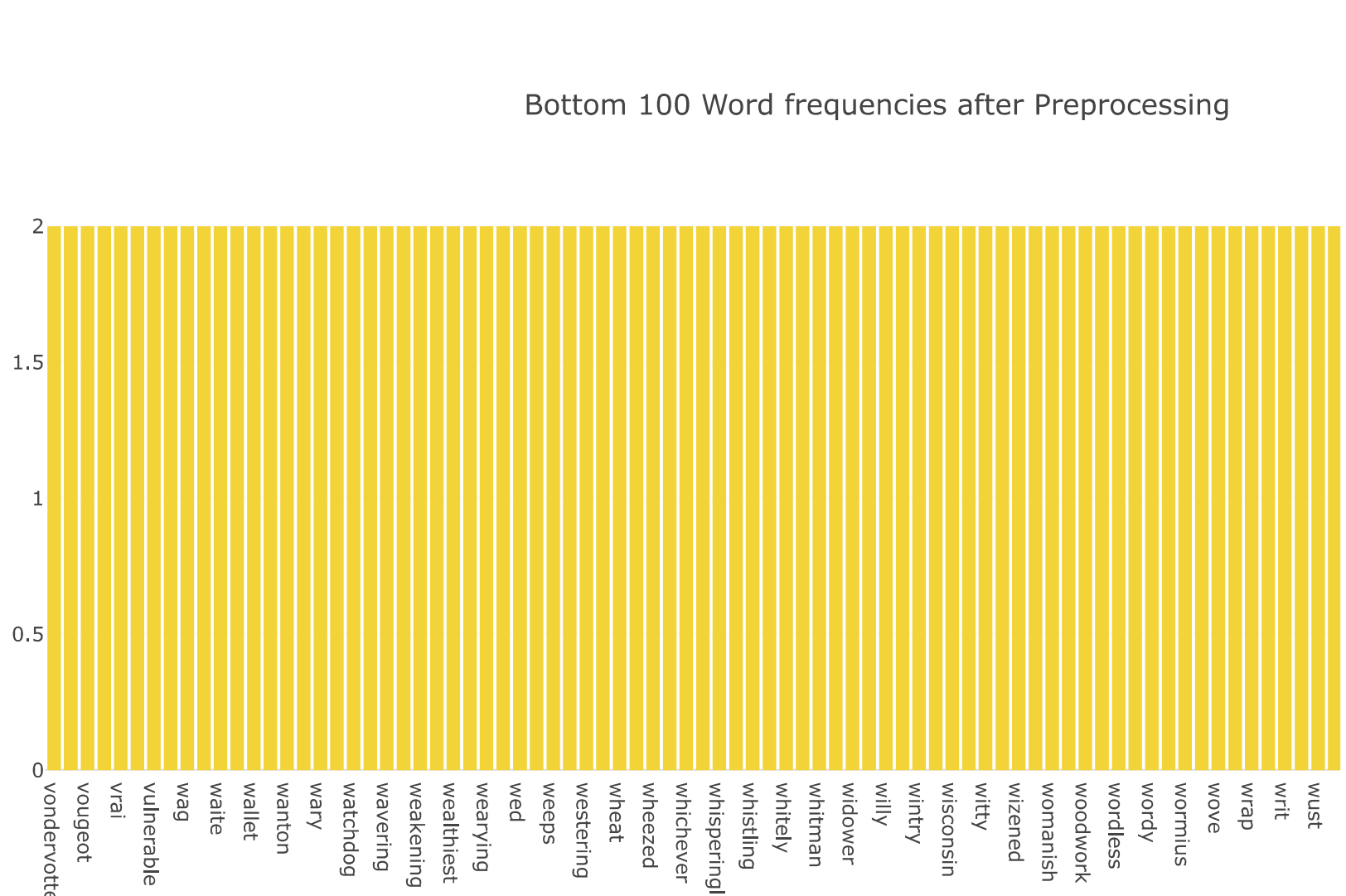
In [37]:
```
# Storing the entire training text in a list
text = list(train.text.values)
# Calling our overwritten Count vectorizer
tf_vectorizer = LemmaCountVectorizer(max_df=0.95,
                                     min_df=2,
                                     stop_words='english',
                                     decode_error='ignore')
tf = tf_vectorizer.fit_transform(text)
```

Revisiting our Term frequencies

After implementing our lemmatized count vectorizer, let's revisit the plots displaying the term frequencies of the top 50 words (by frequency). As evident from the plot, our previous preprocessing efforts have proven worthwhile. The removal of stopwords has rendered the remaining words much more meaningful, distinctly contrasting with the presence of stopwords in the earlier term frequency plot.

In [38]:
```
feature_names = tf_vectorizer.get_feature_names()
count_vec = np.asarray(tf.sum(axis=0)).ravel()
zipped = list(zip(feature_names, count_vec))
x, y = (list(x) for x in zip(*sorted(zipped, key=lambda x: x[1], reverse=True)))
# Now I want to extract out on the top 15 and bottom 15 words
Y = np.concatenate([y[0:15], y[-16:-1]])
X = np.concatenate([x[0:15], x[-16:-1]])

# Plotting the Plot.ly plot for the Top 50 word frequencies
data = [go.Bar(
            x = x[0:50],
            y = y[0:50],
            marker= dict(colorscale='Jet',
                         color = y[0:50]
                        ),
            text='Word counts'
    )]

layout = go.Layout(
    title='Top 50 Word frequencies after Preprocessing'
)

fig = go.Figure(data=data, layout=layout)

py.iplot(fig, filename='basic-bar')

# Plotting the Plot.ly plot for the Top 50 word frequencies
data = [go.Bar(
            x = x[-100:],
            y = y[-100:],
            marker= dict(colorscale='Portland',
                         color = y[-100:]
                        ),
            text='Word counts'
    )]

layout = go.Layout(
    title='Bottom 100 Word frequencies after Preprocessing'
)

fig = go.Figure(data=data, layout=layout)

py.iplot(fig, filename='basic-bar')
```



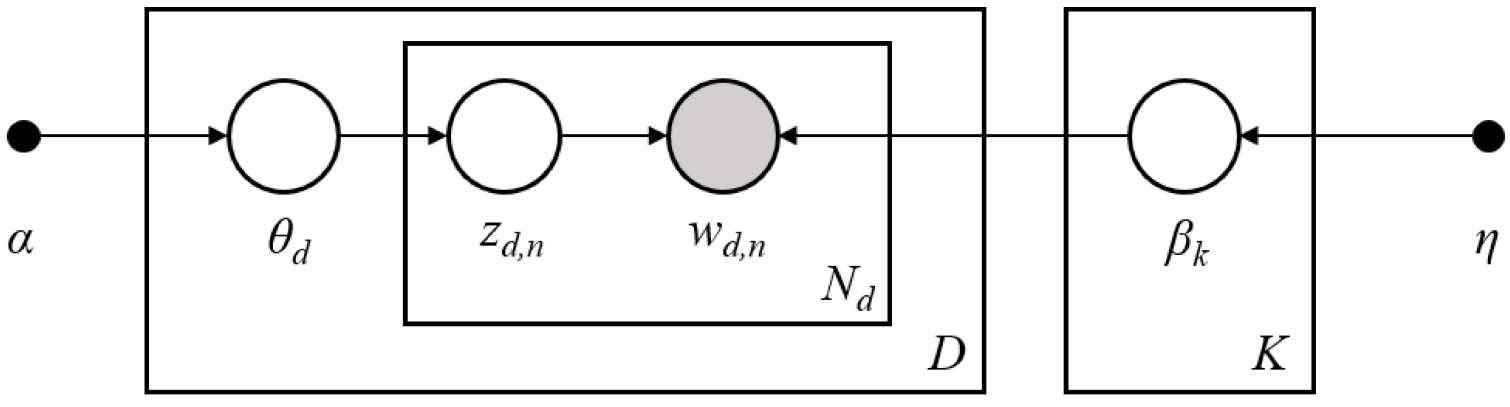


Latent Dirichlet Allocation

Finally, we delve into the realm of topic modeling and the application of a couple of unsupervised learning algorithms. The first method I'll discuss is Latent Dirichlet Allocation (LDA). Various implementations of the LDA algorithm exist, but in this notebook, I'll be utilizing Sklearn's implementation. It's worth exploring Radim Rehurek's gensim, another well-known LDA implementation, as it offers a different perspective.

Corpus - Document - Word : Topic Generation

In LDA, the modelling process revolves around three things: the text corpus, its collection of documents, D and the words W in the documents. Therefore the algorithm attempts to uncover K topics from this corpus via the following way (illustrated by the diagram)



The LDA algorithm first models documents via a mixture model of topics. From these topics, words are then assigned weights based on the probability distribution of these topics. It is this probabilistic assignment over words that allow a user of LDA to say how likely a particular word falls into a topic. Subsequently from the collection of words assigned to a particular topic, are we thus able to gain an insight as to what that topic may actually represent from a lexical point of view.

From a standard LDA model, there are really a few key parameters that we have to keep in mind and consider programmatically tuning before we invoke the model:

1. n_components: The number of topics that you specify to the model
2. α parameter: This is the dirichlet parameter that can be linked to the document topic prior
3. β parameter: This is the dirichlet parameter linked to the topic word prior

To invoke the algorithm, we simply create an LDA instance through the Sklearn's *LatentDirichletAllocation* function. The various parameters would ideally have been obtained through some sort of validation scheme. In this instance, the optimal value of n_components (or topic number) was found by conducting a KMeans + Latent Semantic Analysis Scheme (as shown in this paper here) whereby the number of Kmeans clusters and number of LSA dimensions were iterated through and the best silhouette mean score.

In [39]:
```
lda = LatentDirichletAllocation(n_components=11, max_iter=5,
                                learning_method = 'online',
                                learning_offset = 50.,
                                random_state = 0)
```

In [40]:
```
##
lda.fit(tf)
```

Out[40]:
```
LatentDirichletAllocation(batch_size=128, doc_topic_prior=None,
             evaluate_every=-1, learning_decay=0.7,
learning_method='online', learning_offset=50.0,
max_doc_update_iter=100, max_iter=5, mean_change_tol=0.001,
n_components=11, n_jobs=None, n_topics=None, perp_tol=0.1,
random_state=0, topic_word_prior=None,
total_samples=1000000.0, verbose=0)
```

Topics generated by LDA

We will utilise our helper function we defined earlier "print_top_words" to return the top 10 words attributed to each of the LDA generated topics. To select the number of topics, this is handled through the parameter n_components in the function.

In [41]:
```
n_top_words = 40
print("\nTopics in LDA model: ")
tf_feature_names = tf_vectorizer.get_feature_names()
print_top_words(lda, tf_feature_names, n_top_words)
```

```
Topics in LDA model:

Topic #0:mean night fact young return great human looking wonder countenance difficulty greater wife finally set posses
sed regard struck perceived act society law health key fearful mr exceedingly evidence carried home write lady various
recall accident force poet neck conduct investigation
================================================================================

Topic #1:death love raymond hope heart word child went time good man ground evil long misery replied filled passion bed
till happiness memory heavy region year escape spirit grief visit doe story beauty die plague making influence thou let
ter appeared power
================================================================================

Topic #2:left let hand said took say little length body air secret gave right having great arm thousand character minut
e foot true self gentleman pleasure box clock discovered point sought pain nearly case best mere course manner balloon
fear head going
================================================================================

Topic #3:called sense table suddenly sympathy machine sens unusual labour thrown mist solution suppose specie movement
whispered urged frequent wine hour appears ring turk place stage noon justine ceased obscure chair completely exist sit
ting supply weird bottle seated drink material bell
================================================================================

Topic #4:house man old soon city room sight did believe mr light entered sir cloud order ill way dr apparently clear ce
rtain forgotten day quite door considered need great fine began journey search walked disposition view long concerning
walk drawn saw
================================================================================

Topic #5:thing thought eye mind said men night like face life head dream knew saw form world away deep stone told matte
r morning perdita dead general man strange seen terrible sleep tell object tear know account better black say remained
little
================================================================================

Topic #6:father moon stood longer attention end sure leave remember time excited period trace dream given star place ab
le grew subject set cut visited captain consequence marie taking forward started descent atmosphere impulse departure d
og men truly abyss appear magnificent quarter
================================================================================

Topic #7:day did heard life time friend new far horror nature come look tree year present soul passed known people hear
t felt degree scene idea hand feeling world came country adrian moment make word affection sun gone reached idris youth
seen
================================================================================

Topic #8:came earth street near like sound wall window just open lay fell wind looked saw moment water eye dark spirit
beneath mountain old did light foot long town space floor low happy held half voice living direction ear small end
================================================================================

Topic #9:shall place sea time think long fear know mother day person say brought expression land change question night
result ye week mad month feel god rest got manner course horrible large resolved kind passage far discovery word answer
eye ago
================================================================================

Topic #10:door turned close away design view doubt ordinary tried oh madness room enemy le lower exertion chamber openi
ng candle legend occupation abode lofty author compartment breath flame accursed machinery horse iron proceeded curse v
e louder desired entering appeared lock oil
================================================================================
```

In [42]:
```
first_topic = lda.components_[0]
second_topic = lda.components_[1]
third_topic = lda.components_[2]
fourth_topic = lda.components_[3]
```

In [43]:
```
first_topic.shape
```

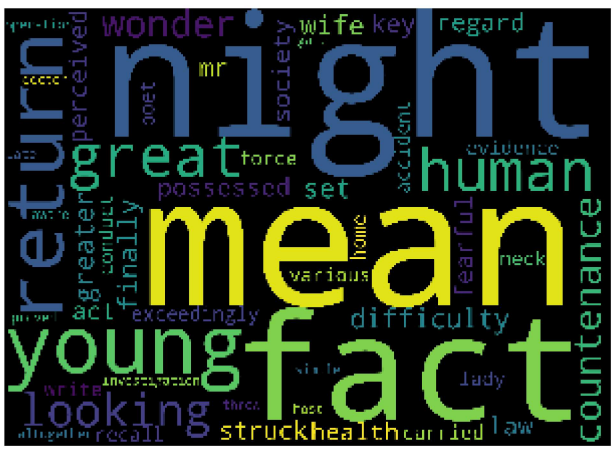
Out[43]:
```
(13781,)
```

Word Cloud visualizations of the topics

In [44]:
```
first_topic_words = [tf_feature_names[i] for i in first_topic.argsort()[:-50 - 1 :-1]]
second_topic_words = [tf_feature_names[i] for i in second_topic.argsort()[:-50 - 1 :-1]]
third_topic_words = [tf_feature_names[i] for i in third_topic.argsort()[:-50 - 1 :-1]]
fourth_topic_words = [tf_feature_names[i] for i in fourth_topic.argsort()[:-50 - 1 :-1]]
```

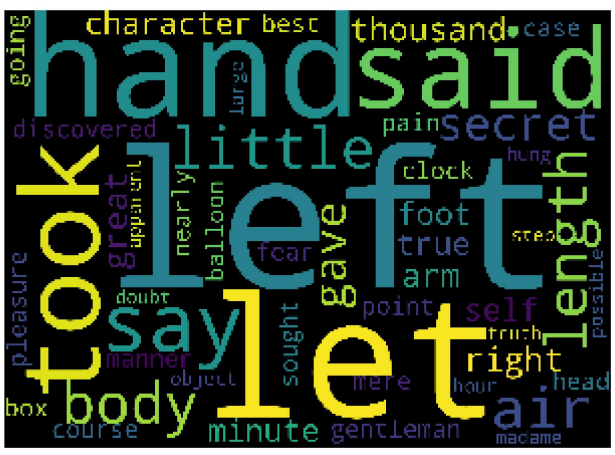
Word cloud of First Topic

In [45]:
```
# Generating the wordcloud with the values under the category dataframe
firstcloud = WordCloud(
                          stopwords=STOPWORDS,
                          background_color='black',
                          width=2500,
                          height=1800
                         ).generate(" ".join(first_topic_words))
plt.imshow(firstcloud)
plt.axis('off')
plt.show()
```

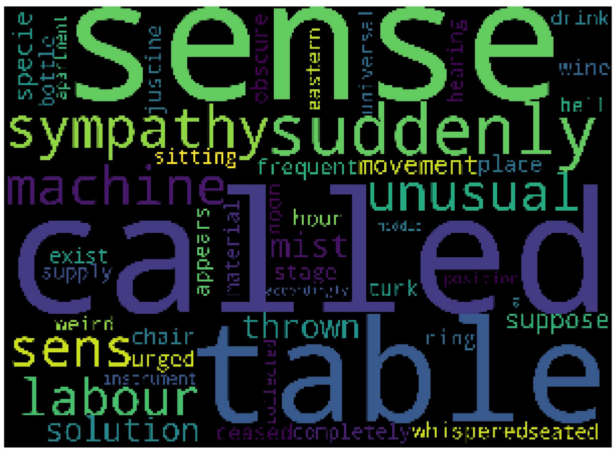


In [46]:
```
# Generating the wordcloud with the values under the category dataframe
cloud = WordCloud(
                            stopwords=STOPWORDS,
                            background_color='black',
                            width=2500,
                            height=1800
                           ).generate(" ".join(second_topic_words))
plt.imshow(cloud)
plt.axis('off')
plt.show()
```


In [47]:
```
# Generating the wordcloud with the values under the category dataframe
cloud = WordCloud(
                            stopwords=STOPWORDS,
                            background_color='black',
                            width=2500,
                            height=1800
                           ).generate(" ".join(third_topic_words))
plt.imshow(cloud)
plt.axis('off')
plt.show()
```



In [48]:
```
# Generating the wordcloud with the values under the category dataframe
cloud = WordCloud(
                            stopwords=STOPWORDS,
                            background_color='black',
                            width=2500,
                            height=1800
                           ).generate(" ".join(fourth_topic_words))
plt.imshow(cloud)
plt.axis('off')
plt.show()
```


sense
called
table
sympathy
suddenly
machine
unusual
sens
labour
solution
thrown
suppose
mist
specie
bottle
juscine
obscure
eastern
universal
hearing
drink
wine
he'll
sitting
frequent
movement
place
hour
exist
supply
appears
material
noon
stage
turk
position
weird
chair
urged
ring
instrument
ceased
completely
whispered
seated