Next word prediction - Personalized Autocomplete

Introduction

The original goal of my project is to create a keylogger that would track everything I write for a given period of time and then to use that information to make a next word predictor model that could I could use as a personalized autocomplete model.

However, several different version of the keylogger proved to be ineffective as they would record any and all characters written but they had no awareness of the window switches nor in what window I would be writing. Furthermore, due to always assuming that the cursor pointer is at the end of the text (which obviously is not always the case), the recorded text was mixed and was completely incomprehensible. Later on, I developed a more advanced version of the keylogger that would run on startup and was being able to record text depending on the window focus. However, due to only running for a several days it did not record nearly enough text to be able to make a model that would be able to predict the next word with any enviable success. Thus, I decided to strengthen the data that I had collected by merging it with previous papers that I had written and with the Harry Potter book franchise which was available on project Gutenberg. (Later on, due to overfitting that I observed, I tried to expand the data by adding some copyright free classics that I found on project Gutenberg such as Dostoyevsky's works, Picture of Dorian Grey, Great Gatsby, Ana Karenina, Little Money, War of the Worlds, Count of Monte Cristo, Don Quixote, etc. However, I did not observe any significant improvement in the model's performance. However, due to lack of memory I had to resort back to Harry Potter books). Of course, by doing this, the autocomplete is no longer predicted based on my personal writing style but rather on some combination of the writing styles of the authors of the books that I have used. However, this is not necessarily a bad thing, since I don't think anyone would really mind having a writing style similar to Dostoyevsky or some other famous author.

Mathematical explanation of our goal

Our goal is to, given an input of sequence of words, predict what the most likely next word is.

In our case, a single word will be predicted, although predict more words is certainly possible. Single word is useful since it is more close to the autocomplete feature and accuracy is higher given a smaller dataset. Since we want to generate sequences longer than just 1 word, we will predict the next word and then use that predicted next word and append it to the previous input sequence. Using that newly updated input sequence we will be able to generate the next word in the sequence. This process can be repeated to generate as many sequence as we would like.

To demonstrate, let's say that the number of features/words we predict on is 3 and our initial input sequence is ["today", "was", "a"] and our goal is to predict the next 2 words. Let us assume that our model predicted the next word in the sequence to be "good" giving us the total sequence to be ["today", "was", "a", "good"]. We can now move the sliding window of size 3 to use the last 3 words of this (we added "good" to the end of the input sequence and dropped "today"). We now use this new input sequence ["was", "a", "good"] to predic the next word which would ideally be ["day"] fulfilling Ice Cube's lyrics. However, it is entirelly possible that some other word could be predicted such as ["boy"]. This is entirely dependent on the distribution learned from the training data. "was a good boy" makes sense, however, "today was a good boy" does not, which is why we will use much larger sequences of words (around 50) to predict the next word.

Essentially, the goal of the model is to learn and mimic from the training data as best as possible the probability distribution of the next word, given the previous words in the sequence - this is done through Bayes' theorem

We can represent this as:

$$P(w_{t+1}|w_1, w_2, \dots, w_t)$$

where w_{t+1} is the target word we want to predict, and w_1, w_2, ..., w_t are the previous words in the sequence. w_{t+1} can be any of the words in the vocabulary V. Vocabulary is the set of all possible words that can be predicted.

Our goal is to find the word w_{t+1} that maximizes the probability of the next word given the previous words. To do this, we find need to find the probability distribution for each word in the vocabulary.

We can express this probability through the chain rule of probability by multiplying the probabilities of each word in the sequence:

- Probability of the first word in the sequence being w_1 = $P(w_1)$
- Probability of the second word in the sequence being w_2 given that the first word in the sequence is w_1 = $P(w_2|w_1)$
- Probability of the third word in the sequence being w_3 given that the first two words in the sequence are w_1 and w_2 = $P(w_3|w_1, w_2)$

In general, we have the probability that the word w_i is word i, given that first i-1 words in the sequence are $w_1, w_2, \dots, w_{i-1}$

$$P(w(1), w(2), \dots, w(i)) = P(w(1)) \times P(w(2)|w(1)) \times P(w(3)|w(1), w(2)) \times \dots$$
$$\times P(w(i)|w(1), w(2), \dots, w(i-1)) = \Pi_{t=1}^{T} P(w_t|w_t, \dots, w_1)$$

Pipeline

Below is the general overview of the project pipeline. Individual choices and decisions are explained in the corresponding sections.

1. Get data
   A. Keylogs
      a. Use keylogger to record keystrokes
      b. Merge keylogs into one file
   B. Books and articles from Project Gutenberg
      a. Download books and articles
      b. Merge books and articles into one file

2. Data processing 1. Lowercase 2. Single whitespace 3. Remove punctuation 4. Remove numbers 5. Widen contractions 6. (keylogs) Fix typos 7. (keylogs) Remove emojis 8. [optional] (keylogs) Remove non-English 9. Remove stopwords 9. Split into tokens 10. Lemmatize or Stem (Lemmatize preferred)

3. Tokenization

4. Create features and target sequences (N-grams)

5. Split into train, validation, and test sets 1. Train: 80% 2. Validation: 10% 3. Test: 10%

6. Vectorization - Creating an embedding matrix from GloVe

7. Creating an LSTM sequential model

8. Evaluation 1. Perplexity 2. Word similarity

9. Prediction - Generating sequences

Necessary installs:

1. pip instal TextBlob
2. pip install tensorflow (or tensorflow-gpu) and keras
3. pip install nltk
4. pip install git+https://github.com/MCFreddie777/language-check.git
5. pip install contractions
6. pip install pycontractions
7. pip install numpy
8. pip install scikit-learn
9. pip install pandas

10. pip install matplotlib
11. pip install regex
12. pip install pynput
13. pip install win32gui
14. pip install fuzzywuzzy

Keylogger

Below I have written the code for the keylogger. This is only to demonstrate the code and not to be run (clicking f12 will terminate the script). This script can be found withing the './keylogger/' folder. It was run on startup (the code to create it to a bat file and to run it on startup can be found in the './keylogger/' folder as well).

The final optimized version of the keylogger found below fully record everything that the user inputs into the keyboard and mimics functions such as ctrl + backspace as well (which I frequently use). One drawback is that it always assume that the cursor pointer is at the end of the text, however, given that this does not happen most of the time which it statistically won't, given enough data, it will not be a problem and can be dealt with in the data processing stage. The keylogger keeps a dictionary for every active window the user writes and then applies the text changes done in that window. When the script terminates, all the values of the dictionary are added sequentially thus preventing the previous problem of words being from mixed contexts. Finally, for every different run the keylogger generates a different txt file.

For iOS users, use use pyautogui.getActiveWindowTitle() to get the active window instead of win32gui.

In [1]:
```
def run_keylogger():
    import logging
    from pynput import keyboard
    import os
    import pickle as pkl
    import re
    from win32gui import GetWindowText, GetForegroundWindow

    # Log the keystrokes to the console with the current date and time
    logging.basicConfig(level=logging.INFO, format='%(asctime)s: %(message)s

    # List of non-alphanumeric characters that are allowed to be typed
    SPECIAL_CHARS = ('.', ';', ',', '/', '\\', '"', "'", '(', ')',
                     '[', ']', '{', '}', '<', '>', '!', '?', ':', '-', '_',
    # List of keys to ignore
    NONALLOWED_KEYS = [keyboard.Key.esc, keyboard.Key.caps_lock, keyboard.Ke
                       keyboard.Key.alt, keyboard.Key.alt_l, keyboard.Key.al
                       keyboard.Key.cmd_r, keyboard.Key.up, keyboard.Key.dow
                       keyboard.Key.f1, keyboard.Key.f2, keyboard.Key.f3, ke
                       keyboard.Key.f7, keyboard.Key.f8, keyboard.Key.f9, ke

    # deletes the existing pickle file - uncomment only if you want to start
    #open("lastfile.pkl", "w").close()

    try:  # try to load the pickle file
        with open('lastfile.pkl', 'rb') as fp:
            lastfile = pkl.load(fp)  # load the pickle file
            logging.info(
                f"Pickle file found. Last log file number: {lastfile}. Initi
    except (FileNotFoundError, EOFError):  # error catch if the file is not
        logging.info("Pickle file not found. Initializing log files from 1."
        lastfile = 0

    # GLOBAL VARIABLES
    # Create a dictionary to store the words written in the currently active
    window_words = {}  # key: window title, value: list of words
    ctrl_l_pressed = False  # check if ctrl_l is pressed

    def mimic_ctrl_backspace(input_string):
        """
        Mimics the ctrl + backspace functionality in the terminal.
        Takes the input string, assumes the cursor is at the end of the stri

        Parameters
        ----------
        input_string : str
            The string to mimic the ctrl + backspace functionality on.

        Returns
        -------
        str
            The modified string after the ctrl + backspace functionality is
        """
        if len(input_string) <= 1:  # if the input string is empty or has on
            return ''
        else:  # len(input_string) > 1
            # if the last two characters are spaces
            if input_string[-1] == ' ' and input_string[-2] == ' ':
                # remove all the whitespace at the end of the input_string
                input_string = re.sub(r"\s+$", "", input_string)
                return input_string
            # if the last character is a space and the penultimate character
            elif input_string[-1] == ' ' and input_string[-2].isalnum():
                # first remove all the whitespace at the end of the input_st
                input_string = input_string.rstrip()
                while len(input_string) > 0 and input_string[-1] not in SPEC
                    # keep removing last character until a space or a specia
                    input_string = input_string[:-1]
                return input_string
            else:
                if input_string[-1].isalnum():  # if the last character is a
                    while len(input_string) > 0 and input_string[-1] not in
                        # keep removing last character until a space or a sp
                        input_string = input_string[:-1]
                # if the last character is not alphanumeric (i.e. a special
                else:
                    while len(input_string) > 0 and not input_string[-1].isa
                        # keep removing last character until an alphanumeric
                        input_string = input_string[:-1]
                return input_string

    def save_keystrokes():
        """ Saves the keystrokes to a file.
        The file is saved in the logs folder and is named Log <lastfile+1>.t

        Returns
        -------
        None
        """

        global lastfile
        lastfile += 1  # increment the lastfile number

        try:
            # create the filename
            file_name = f"Log {lastfile}.txt"
            # get the current path
            path = os.path.dirname(os.path.realpath(__file__))
            # open the file in write mode (creates or overwrites the existir
            with open(f'{path}/logs/{file_name}', "w") as file:
                for text in window_words.values():  # iterate through the wo
                    # append the words sequentially so there are mixed sente
                    file.write(text)
                    file.write(" ")
                logging.info("Saved keystrokes to file: %s", file_name)

        except Exception as e:
            logging.error("Could not save keystrokes to file: %s", str(e))

    def on_press(key):
        """Appends the pressed key to the keystroke string for the currently

        Parameters
        ----------
        key : keyboard.Key
            The key that was pressed.

        Returns
        -------
        None
        """

        global ctrl_l_pressed

        try:
            # get the title of the currently active window
            # Works only on Windows. For iOS, use pyautogui.getActiveWindowT
            active_window = GetWindowText(GetForegroundWindow())
            if active_window not in window_words:  # if the active window ha
                window_words[active_window] = ""

            # get the keystroke string for the currently active window
            keystroke_log = window_words[active_window]
```

```
            if key in NONALLOWED_KEYS:  # if the pressed key is not allowed
                pass
            elif key == keyboard.Key.f12:  # if the pressed key is f12, stop
                listener.stop()
                save_keystrokes()
                logging.info("Stopped the keylogger.")
                pass
            # if the pressed key is space or enter, add a space to the keyst
            elif key == keyboard.Key.space or key == keyboard.Key.enter:
                keystroke_log += " "
            # checking if ctrl_l and backspace are pressed at the same time
            elif key == keyboard.Key.ctrl_l:  # if the pressed key is ctrl_l
                ctrl_l_pressed = True
            elif key == keyboard.Key.backspace:  # if the pressed key is bac
                # if ctrl_l is pressed and the keystroke_log is not empty
                if ctrl_l_pressed and len(keystroke_log) > 0:
                    # mimic the ctrl + backspace functionality
                    keystroke_log = mimic_ctrl_backspace(keystroke_log)
                else:  # if ctrl_l is not pressed or the keystroke_log is em
                    # remove the last character from the keystroke_log
                    keystroke_log = keystroke_log[:-1]
            # if the pressed key is alphanumeric or a special character
            elif key.char.isalnum() or key.char in SPECIAL_CHARS:
                # append the pressed key to the keystroke_log
                keystroke_log += str(key).strip("'")

            # log the keystroke_log for the currently active window
            logging.info(f"{active_window}: {keystroke_log}")
            # update the keystroke_log for the currently active window
            window_words[active_window] = keystroke_log

        except:
            logging.error(
                f"Error while appending keystroke {key}. Unallowed character

    def on_release(key):
        """Checks if the released key is ctrl_l and sets ctrl_l_pressed to F

        Parameters
        ----------
        key : keyboard.Key
            The key that was released.

        Returns
        -------
        None
        """

        global ctrl_l_pressed

        if key == keyboard.Key.ctrl_l:  # if the released key is ctrl_l, set
            ctrl_l_pressed = False

    try:
        # Start the keyboard listener and the timer to save the keystrokes
        with keyboard.Listener(on_press=on_press, on_release=on_release) as
```

```
            listener.join()
    finally:  # save the lastfile number to a file
        with open('lastfile.pkl', 'wb') as fp:
            pkl.dump(lastfile, fp)
            print("Saved new last file number: ", lastfile)
            # wait for the user to press enter before closing the terminal
            input("\nPress Enter to close the terminal...")
```

Reading the data

Merging text files / keylogger logs

Given that the keylogger has been running for a few days or that there are several books, the txt files have to be merged into a single txt file.

In [2]:
```
def merge_txt_files(input_dir=".\keylogger\logs", output_dir="database/merge
    """Merges all .txt files in the input directory into a single .txt file

    Parameters
    ----------
    input_dir : str
        The path to the directory containing the .txt files to merge. Defaul
    output_dir : str
        The path to the directory where the merged 'master.txt' file will be

    Returns
    -------
    None
    """

    import os

    # Create the output directory if it does not exist
    if not os.path.exists(output_dir):
        print(
            "Creating a folder '{output_dir}' to store the merged text file.
        os.makedirs(output_dir)

    # Merge the contents of all .txt files in the input directory into a si
    merged_text = ""
    for filename in os.listdir(input_dir):
        if filename.endswith(".txt"):
            with open(os.path.join(input_dir, filename), "r") as f:
                merged_text += f.read()
    print(
        f"Merged all .txt files from the {input_dir} folder into a single va

    # Write the merged text to a new file in the output directory
    output_filename = os.path.join(output_dir, name)
    with open(output_filename, "w") as f:
        f.write(merged_text)
    print(f"Saved the merged text to ./{output_filename}")
```

Reading the master text file

Below is the function used to read a text file and return a string containing the entire dataset which we use in our preprocessing.

In [3]:
```
def read_txt_file(file_name, folder_path="./database/processed/"):
    """Reads a text file.

    Parameters
    ----------
    file_name : str
        The name of the text file.
    folder_path : str (optional)
        The path to the folder containing the text file. Defaults to "./data

    Returns
    -------
    text : str
        The text read from the file.
    """

    import os  # Import the os module to work with file paths

    text = open(os.path.join(folder_path, file_name), 'r').read()
    print(
        f"\nRead {file_name}. It contains {len(text)} characters and {len(te
    return text
```

Preprocessing

Preprocessing is the fundamental process of preparing the data for the model. It is a very important step as it can have a significant impact on the model's performance (can reduce complexity and number of features, increase performance by finidng only the key features, etc.)

Two libraries will be primarily used for preprocessing: regex and nltk. Regex is used for simpler text cleaning such as finding and removing punctuation, numbers, etc. NLTK is a very standard library used for more complex tasks such as tokenization, lemmatization, stemming, which can be easily done with its built-in functions.

In [4]:
```
import re  # regular expressions
import nltk  # natural language toolkit
# nltk.download() # uncomment to download the nltk data
```

Converting to lowercase and stripping multiple whitespaces.

Converting all text to lowercase makes the vocabulary size smaller and prevents the model from learning different embeddings for the same word in different cases. For example, without lowercasing, the model would learn different embeddings for "Apple" and "apple," even though they refer to the same thing. If we want to do capitalization, it is better to do train the model with all lowercase so the model produces better results and then apply capitalization in the post-processing stage by applying some model that capitalizes.

The texts used contain a lot of whitespaces and although they will not necessarily increase the vocabulary size (a tokenizer will easily get rid of them), looking at the text, they are not necessary and take away from readability.

In [5]:
```
def lowercase_and_strip_whitespaces(text):
    """Converts a given text to lowercase and strips multiple whitespaces to

    Parameters
    ----------
    text : str
        The text to convert.

    Returns
    -------
    str
        The converted text with multiple whitespaces stripped to a single wh
    """
    # strip multiple whitespaces
    text = ' '.join(text.split())

    # convert to lower case
    text = text.lower()

    return text
```

Remove punctuation

Analogous to above, punctuation marks and numbers typically do not carry much meaning in natural language and can be safely removed without significant loss of information. Removing punctuation and numbers can also help to reduce the noise in the data, since these characters can appear in many different contexts and make it more difficult for models to learn the correct relationships between words.

In the case of next word prediction, removing punctuation and numbers can also help to ensure that the model is able to focus on the words themselves rather than being distracted by other characters in the input. This can help to improve the accuracy and effectiveness of the model in predicting the most likely next word given a sequence of words.

In [6]:
```
def remove_punctuation_and_numbers(text):
    """Removes punctuation and numbers from a given text.

    Parameters
    ----------
    text : str
        The text to remove punctuation and numbers from.

    Returns
    -------
    str
        The text with punctuation and numbers removed.
    """

    text = ''.join([char for char in text if char.isalpha() or char == ' '])
    return text
```

Widen contractions

Another thing we can do is widen the contractions we have in the text. This is useful because it helps to standardize the text and reduce the dimensionality of the data. E.g. we want "I'm" and "I am" to be deemed the same by the model and we want to reduce the number of unique words that the model needs to learn in order to make accurate predictions.

Widening contractions involves expanding commonly used contractions (e.g., "can't" to "cannot", "won't", etc.) into their full forms. By expanding contractions, we are converting a single word (the contraction) into multiple words (the full form), which can help reduce the dimensionality of the data and ensure that the model learns the relationships between the full words rather than just the contractions. This can help improve the model's accuracy and make it more robust to different variations of contractions.

Although I have placed this step after the removal of punctuation since it is slightly more complex, we do it before otherwise the punctuation from the contractions will be removed as well making it impossible to widen the contractions.

In the code below, I made it possible to use two models to expand contractions: 'contractions' and 'pycontractions'. The first one is a simple model that expands contractions by looking at a dictionary of contractions and their full forms.

The 'contractions' library uses a simple dictionary-based approach for expanding contractions. It contains a dictionary of common English contractions and their expanded forms, which is used to replace the contractions in the text. For example, the contraction "won't" is expanded to "will not". The library is very lightweight and easy to use, but it can't handle some less common or informal contractions that can be expanded to multiple different words (such as "ain't"), and it may not work well on text with spelling or grammatical errors. It also does not take context into account reducing its accuracy.

The 'pycontractions' library, on the other hand, uses a more sophisticated approach by leveraging a pre-trained language model to identify and expand contractions in text. The library uses spaCy, which is a popular open-source NLP library, to parse the input text and identify contractions. Then, it uses a pre-trained machine learning model that you have to load (such as Glove-twitter-100). The transformer model is fine-tuned on a large corpus of English text to learn the mapping between contractions and their expanded forms which leads to better accuracy and awareness of context - which leads to much better results.

The advantage of this approach is that it can handle a wide range of contractions, including those that are less common or informal, and it can also handle variations in spelling or grammar. However, its higher accuracy comes at the price of computational cost and speed. This can be slow for large texts or in real-time applications.

Despite making both accessible below, I prioritized using the simple 'contractions' as it is sufficient for this project.

In [7]:
```
def expand_contractions(text, model='contractions'):
    """
    Expands contractions in a given text.

    Parameters
    ----------
    text : str
        The text to expand contractions in.
    model : str
        The model to use to expand contractions. Defaults to 'contractions'.
        Options:
            'contractions' - uses the contractions library
            'pycontractions' - uses the pycontractions library. Greater accu

    Returns
    -------
    str
        The text with contractions expanded.
    """

    try:
        if model == 'contractions':
            import contractions
            text = contractions.fix(text)
            return text

        elif model == 'pycontractions':
            import pycontractions

            # Load the contraction model
            cont = pycontractions.Contractions(api_key="glove-twitter-100")
            cont.load_models()

            # Expand contractions in the text
            expanded_text = list(cont.expand_texts([text], precise=True))[0]
            return expanded_text
        else:
            raise Exception(
                f"Model '{model}' is not supported. Please choose either 'co
    except Exception as e:
        print(f"Error expanding contractions: {e}")
```

Remove stopwords

Removing stopwords such as "the", "a", "an", "and", "in", "on", "at", etc. is a common step in text preprocessing. They are usually removed from text data during preprocessing because they do not add much meaning to the text and are unlikely to be useful for NLP tasks. By removing this, we can sometimes significantly reduce the number of features and improve the performance of the model since only the most important words are kept and used for decision making and the feature space is reduced.

Removing stopwords also reduces noise since stopwords are occur very frequently in the corpus of text thereby adding noise to text data. By removing them, we can reduce the number of meaningless words in the text and focus on more meaningful words.

Of course, sometimes stopwords can be useful for certain tasks such as matching query items. Thus, treating this as a hyperparameter and comparing the results with and without is always a good idea. In my case, the results were better without stopwords.

In [8]:
```
def eliminate_stopwords(tokens):
    """Removes stopwords from a given list of tokens.

    Parameters
    ----------
    tokens : list
        A list of tokens to remove stopwords from.

    Returns
    -------
    filtered_words_tokens: list
        A list of tokens with stopwords removed.

    """

    stopwords = nltk.corpus.stopwords.words("english")

    from string import punctuation
    # since our text doesn't have punctuation we also remove punctuation fro
    stopwords = [word for word in stopwords if word not in punctuation]

    # remove stop words from tokens
    filtered_words = [word for word in tokens if word not in stopwords]

    return filtered_words
```

Remove emojis

There were a couple of emojis in my text so I decided to include a function to remove them. This is oftentimes not necessary but I wanted to include it for completeness. When creating the keylogger, I made sure they are not included and that I do as little preprocessing as possible.

In [9]:
```
def remove_emojis(text):
    """Removes emojis from a given text.
    Parameters
    ----------
    text : str
        The text to remove emojis from.

    Returns
    -------
    text : str
        The text with emojis removed.
    """

    # remove emojis from the text
    emoji = re.compile("["
                       u"\U0001F600-\U0001FFFF"  # emoticons
                       u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                       u"\U0001F680-\U0001F6FF"  # transport & map symbols
                       u"\U0001F1E0-\U0001F1FF"  # flags (iOS)
                       u"\U00002702-\U000027B0"
                       u"\U000024C2-\U0001F251"
                       "]+", flags=re.UNICODE)
    text = emoji.sub(r'', text)
    return text
```

Tokenization

Tokenization is a fundamental step in NLP. It takes a string of text (such as "today is a good day") and splits it into a list of tokens / word (["today", "is", "a", "good", "day"]).

Doing this allows us to analyze text by breaking it down into smaller, more manageable units. It is a critical step in creating a bag-of-words model, which is a simple but effective way to represent text data numerically for use in machine learning algorithms. Generally, it makes it much easier to vectorize text data, to perform other tasks such as counting the number of occurrences of each word in the text, cleaning tasks, such as removing stopwords, stemming or lemmatization, and other preprocessing tasks.

Although Tokenization can also help in visualizing the text data. For instance, word clouds can be generated, which can help in quickly identifying the most commonly occurring words in the text. I chose not do this as it did not give any useful insights.

NLTK's word_tokenize function is a popular tokenization method that works well for most use cases. It uses regular expressions to split the text into individual words. The function also has built-in features for handling contractions, hyphenated words, and other special cases.

In [10]:
```
def tokenize(text):
    """Tokenizes a given text into a list of words.

    Parameters
    ----------
    text : str
```

```
        The text to tokenize. E.g. "This is a sentence"

    Returns
    -------
    list
        A list of words. E.g. ["This", "is", "a", "sentence"]
    """

    # split text into tokens (words) - also gets rid of multiple whitespaces
    tokens = nltk.word_tokenize(text)
    return tokens
```

Correct typos

Although this step is not particularly useful for books since they are usually written by professionals and do not have any typos, it is a very useful step for text data that is collected from the internet or social media - or in my case, my personal writting.

I have included two libraries: TextBlob and FuzzyWuzzy. The foundation of all of these libraries is the Levenshtein distance which is a string metric for measuring the difference between two sequences. In other words. it is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. The lower the Levenshtein distance between two words, the more similar they are. Thus, intuitively, we change the mispelled word with the word from the dictionary with the smallest Levenshtein distance. However, sometimes there are more of them so using different models can help.

TextBlob

TextBlob's correct() method has a dictionary-based approach where it compares the misspelled (every word actually) word to its dictionary of known words (it has around 30k English words) and finding the closest match based on its edit distance metric which is a slightly more complicated version of Levensthein distance.

The method relies on Bayes' theorem as it uses a probabilistic language model to estimate the likelihood of each possible correction based on the frequency of occurrence of the corrected word in a large corpus of text. This approach helps to correct typos that are not in the dictionary, such as those caused by keyboard errors or phonetic confusion. Its model also takes into account the context of the word and the surrounding words when making corrections. It accomplishes this using a Markov chain model to estimate the probability of each possible correction based on the context of the word and the surrounding words in the text.

In a first-order Markov chain model, the probability of a word is dependent only on the previous word in the sequence, and not on any other words that came before it. To estimate the probability of each possible correction, TextBlob's correct() method uses the Bayes' rule formula:

$$P(A|B) = P(B|A) * P(A)/P(B)$$

Bayes' rule is a formula that describes how to calculate the probability of an event based on prior knowledge or evidence. In the context of spelling correction, Bayes' rule can be used to calculate the probability of a correction given the context of the word and the surrounding words in the text. Thus, A represents the possible correction for a misspelled word and B represents the context of the word and the surrounding words in the text.

This approach helps to correct words that are spelled correctly but are used in the wrong context, such as homophones (e.g., "there" vs. "their") or words with multiple meanings (e.g., "bank" as a financial institution or the side of a river). However, it also sometimes does not correct certain words or wrongly corrects the wrong ones. Finally, it is also statistically possible that some words are wrongly corrected (e.g. 'I love to teech' gets converted to 'I love to teeth').

In [11]:
```
def correct_typos_textblob(text):
    """Corrects typoes in a given text using the TextBlob library.

    Parameters
    ----------
    text : str
        The text to correct typos in.

    Returns
    -------
    str
        The text with typos corrected.
    """

    from textblob import TextBlob
    blob = TextBlob(text)
    corrected_text = blob.correct()
    text = corrected_text.string
    return text
```

Fuzzy matching - removing typos and non-english words

The FuzzyWuzzy library is a Python library that uses Levenshtein distance and other techniques to calculate the differences between sequences and to find matches between strings. The reason I included this is because we can write an algorithm that can potentially not only correct typos but eliminate non-english words as well (which my keylogs had).

The 'fuzz.ratio()' scorer calculates a score between 0 and 100 that represents the similarity between two strings based on the number of matching characters they share.

In the function below, each word in the input text is compared to a corpus of words using the 'process.extractOne()' function from 'fuzzywuzzy'. This function returns the closest match to the input word in the corpus, along with a score indicating the similarity between the two strings.

The fuzz.ratio() scorer used by process.extractOne() calculates the similarity score based on the number of characters that the two strings have in common, as a percentage of the total number of characters in the longer string. For example, if the input word is "apple" and the closest match in the corpus is "apples", the similarity score would be 83, because "apple" and "apples" share 5 out of 6 characters, or 83% similarity. If the score is above the threshold, the closest match is considered a valid correction for the input word and is added to the list of corrected words. If the score is below the threshold, no correction is made for that word.

This approach is not perfect and can sometimes return the wrong correction. However, assuming that non-english words are going to have a lower similarity score to the English dicitionary, we can use this to eliminate non-english words. In my case, I set the threshold to 80 and it worked well. The problem with this approach is that it can take a very long time to run and of course, there are certain Serbian words that are quite similar to English words or are short enough to be considered as typos.

In [12]:
```python
def correct_typos_fuzzy_match(text):
    """Fuzzy matches a given text to a corpus of words. Returns the closest

    Parameters
    ----------
    text : str
        The text to fuzzy match.
    corpus : list
        A list of words to use as a reference for fuzzy matching the text.

    Returns
    -------
    str
        The fuzzy matched text.
    """
    from fuzzywuzzy import process, fuzz
    from nltk.corpus import words

    corpus = words.words()

    # Split the text into a list of words
    words = text.split()

    # Correct each word in the text
    fuzzy_matched_words = []
    for word in words:
        # Find the closest match to the word in the corpus
        closest_match = process.extractOne(word, corpus, scorer=fuzz.ratio)
        # If the closest match is a perfect match, use it
        if closest_match[1] > 85:
            fuzzy_matched_words.append(closest_match[0])

    # Join the corrected words into a string
    fuzzy_matched_text = " ".join(fuzzy_matched_words)
    return fuzzy_matched_text
```

Lemmatization

Lemmatization and stemming are techniques used to reduce words to their base/root/lemma form. The difference between them is that stemming simply removes the suffix of a word to derive the root form, while lemmatization maps a word to its base or dictionary form, called a lemma.

Stemming is a simpler and more computationally efficient process than lemmatization, but it is also less accurate. Oftentimes this will be sufficient. E.g. 'studies' to 'studi'; 'studying' to 'study'. Other times, for example, a stemming algorithm might convert the word "running" to "run", but it might also convert the word "runner" to "run", even though these words have a completely different meaning (one is a verb one is a noun) the meaning of the word is lost. More general drawback of stemming is that for a word such as 'saw', it would most likely always return 'saw' whereas lemmatization is more sophisticated and could potentially return 'see'.

Lemmatization is more complex and computationally expensive than stemming and is thus slower for large datasets than stemming. It is generally more accurate because it takes into account the context of the word and its part of speech. The process of lemmatization typically involves identifying the part of speech of a word in a sentence, such as whether it is a noun, verb, adjective, or adverb, and then mapping the word to its corresponding lemma in a predefined vocabulary, such as WordNet. Here, the WordNetLemmatizer uses a set of rules and a pre-defined vocabulary called WordNet to determine the base form of a word based on its part of speech. Lemmatized form of 'mice' would be 'mouse', 'geese' would be 'goose' and 'saw' in a sentence like "I saw a hourse" would be 'see', rather than 'saw'.

In this case, since the dataset is considered small, lemmatization is much preferred to retain as much information as possible.

A lemmatization algorithm, due to its complexity, is much more difficult to create for other languages compared to stemming since it necessitates a deep understanding of the language and its grammatical structural form.

In [13]:
```
def lemmatize_words(tokens):
    """Lemmatizes a list of words.

    Parameters
    ----------
    tokens : list
        A list of words to lemmatize.

    Returns
    -------
    lemmatized_words : list
        A list of lemmatized words.
    """

    from nltk.stem import WordNetLemmatizer
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(word) for word in tokens]
    return lemmatized_words
```

Preprocessing function

We can now combine all the preprocessing steps into a single function that can be applied to the entire dataset and easily reused. I also added optional parameters so that certain preprocessing steps can be skipped if not necessary and that verbose is made optional.

In [14]:
```
def data_preprocessing(text, save_file=False, directory='./database/processe
    """
    Preprocesses the text data by: converting to lowercase, removing punctua
                                   splitting into tokens, removing stop w

    Parameters
    ----------
    text : str
        The text to preprocess.
    remove_stopwords : bool, optional
        Whether to remove stopwords from the text, by default True
    lemmatize : bool, optional
        Whether to lemmatize the text, by default True
    remove_emojis : bool, optional
        Whether to remove emojis from the text, by default False
    correct_typos : var, optional
        Whether to correct typos in the text, by default False. Extremelly s
        You can pass 'textblob' or 'fuzzy_match' to correct typos using the
    expand_contractions_model : var, optional
        Model used to expand contractions, by default 'contractions'.
        You can pass 'contractions' or 'pycontractions' to expand contractio
    save_file : string or False, optional
        Variable to save the preprocessed text to a file, by default False.
        If you want to save the file, pass the file name as a string.


    Returns
    -------
    list
        A list of preprocessed tokens.
    """
    if verbose:
        print('\nPreprocessing text...')

    # convert to lowercase and strip multiple whitespaces
    text = lowercase_and_strip_whitespaces(text)
    if verbose:
        print('\tConverted to lowercase and stripped multiple whitespaces.')

    # remove punctuation and numbers
    text = remove_punctuation_and_numbers(text)
    if verbose:
        print('\tRemoved punctuation and numbers.')

    # remove emojis
    if to_remove_emojis:
        text = remove_emojis(text)
        if verbose:
            print('\tRemoved emojis.')

    # correct typos
    if to_correct_typos is not False:
        if verbose:
            print('\tCorrecting typos... This may take a while...')
        if to_correct_typos == 'textblob':
            text = correct_typos_textblob(text)
        elif to_correct_typos == 'fuzzy_match':
            text = correct_typos_fuzzy_match(text)
        if verbose:
            print(f'\tTypos corrected using {to_correct_typos}.')

    # expand contractions
    if verbose:
        print(
            f'\tExpanding contractions using {expand_contractions_model} moc
    text = expand_contractions(text, model=expand_contractions_model)

    # tokenize
    tokens = tokenize(text)
    if verbose:
        print('\tSplit the text into tokens.')

    if remove_stopwords:
        tokens = eliminate_stopwords(tokens)
        if verbose:
            print('\tRemoved stopwords.')

    if lemmatize:
        tokens = lemmatize_words(tokens)
        if verbose:
            print('\tLemmatized words.')

    # save preprocessed text to file
    if save_file is not False:
        with open(directory + save_file, 'w') as f:
            f.write(' '.join(tokens))
            if verbose:
                print(f'\tPreprocessed text saved to {directory + save_file}
    if verbose:
        print(f'Preprocessing finished. There are now {len(tokens)} tokens.\

    return tokens
```

Tokenization: word to index mappings

Similar to tokenization explained above. The function 'get_word_to_index_mappings()' takes a list of tokens and creates word-to-index and index-to-word mappings using the Keras Tokenizer class. The tokenizer first fits on the text data, meaning it learns the vocabulary (vocabulary is the list of all unique words found in the corpus of text) of the text and assigns a unique integer index to each word. The word-to-index dictionary contains a mapping from each word in the vocabulary to its unique corresponding integer index. The index-to-word dictionary contains the reverse mapping, from each integer index to its corresponding word.

The function also uses the tokenizer to convert the original text to a list of tokens, where each token is represented by its corresponding integer index. This is useful for vectorization and feeding the data to our model later on.

In [15]:
```python
def get_word_to_index_mappings(tokens):
    """Creates word-to-index and index-to-word mappings.

    Parameters
    ----------
    tokens : list
        A list of tokens. E.g. ['the', 'cat', 'sat', 'on', 'the', 'mat']

    Returns
    -------
    word_index : dict
        A dictionary with word-to-index mappings. E.g. {'the': 1, 'cat': 2,
    index_word : dict
        A dictionary with index-to-word mappings. E.g. {1: 'the', 2: 'cat',
    text_as_tokens : list
        A list of tokens. [1, 2, 3, 4, 1, 5]
    """

    from keras.preprocessing.text import Tokenizer
    from pickle import dump

    # create a tokenizer object
    # lower=True converts all text to lowercase, oov_token='<OOV>' replaces
    tokenizer = Tokenizer(lower=True, oov_token='<OOV>')

    # fit the tokenizer on the text data
    tokenizer.fit_on_texts(tokens)

    # get the word-to-index mappings
    word_index = tokenizer.word_index

    print(
        f'Created word-to-index dictionary. Total number of unique tokens: {

    # get the index-to-word mappings
    index_word = {v: k for k, v in word_index.items()}

    # convert the text to a list of tokens
    # the output is a list of lists, so we take the first element
    text_tokenized = tokenizer.texts_to_sequences([tokens])[0]

    # save the tokenizer object
    dump(tokenizer, open('tokenizer.pkl', 'wb'))

    return word_index, index_word, text_tokenized
```

Split into features and targets: N-grams

We can now split the dataset into features and targets by appling a rolling window that will extract a sequence of size n. The features are the sequences of words [:-1] and the targets are the final words in the sequences. The target is the word that we want to predict given the sequence of words that precede it. This makes the process a supervised learning problem.

These continous overlapping sequences of n words or characters are called N-grams. They can be of varying length but in this case, after trying several different variants I defaulted to sequences of length 50, where the features are the first 49 and the target is the 50th number.

We create a window of size N which we move along the sequence of words (our corpus). The window is the sequence of words that we use to predict the next word. The target is the next word in the sequence. Every time we move the window we get a new datapoint.

Below we have an example of a Bigram (N=2).

Ngrams example

Mathematical explanation of the model

Since, in this case, we use N-1 tokens as the features and only 1 as the target, we want to represent that target in a format that is suitable for our Machine Learning model.

The target is a single word, but our model needs to be able to predict a probability distribution over the entire vocabulary. In other word and as explained in the introduction of the paper, each of the words (w_1, w_V) will have a conditional probability of being the next word given the input sequence. Thus, since we want our target to be written in the shape of a probability distribution. The target should be a vector of size V, where V is the size of the vocabulary.

This can most easily be obtained through one-hot encoding. One-hot encoding is a representation of categorical variables as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1.

If our indexed vocabulary is: {0: 'the', 1: 'cat', 2: 'sat', 3: 'on', 4: 'mat'}

and our training example is: ['the', 'cat', 'sat', 'on', 'the', 'mat'] = [0, 1, 2, 3, 0, 4]

Then, the features would be: ['the', 'cat', 'sat', 'on', 'the'] = [0, 1, 2, 3, 0] the target would be: ['mat'] = [4]

Then the one-hot encoded vector for the word 'mat' would be: [0, 0, 0, 0, 1]

As we can see, the dimensionality is equal to the size of the vocabulary and everything besides the index of the target word is 0, with the index of the target word being 1.

The language model can then use this one-hot encoded target vectors to compute a probability distribution over the entire vocabulary. This is typically done using a softmax activation function on the output layer of the neural network which maps the output of the neural network to a probability distribution over the vocabulary by exponentiating each output and normalizing the sum of the exponentiated values to 1. This gives the probability distribution over the vocabulary and allows us to choose the word with the highest probability as the predicted next word. In training, this probability distribution is also used to compute the loss and update the model parameters during training.

In [16]:
```
def get_features_targets(text_tokenized, total_unique_tokens, seq_len=30):
    """Creates features and targets from a list of tokens.

    Parameters
    ----------
    text_tokenized : list
        The list of tokens - the entire vocabulary tokenized. E.g. [1, 2, 3,
    total_unique_tokens : int
        The total number of unique tokens in the vocabulary. E.g. 5
    seq_len : int, optional
        The length of the sequences, by default 5.
        If seq_len=5, the testing sequence will be made of 4 tokens and the

    Returns
    -------
    features : matrix
        A list of sequences. # E.g. [[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 5
    targets : matrix
        A list of targets. E.g. [[0,0,0,1,0], [0,0,0,0,1], [0,1,0,0,0]]
        One-hot encoded. 1 only for the target word in the vocabulary, every
    """
    from keras.utils import to_categorical  # one-hot encoding

    features = []
    targets = []

    for i in range(seq_len, len(text_tokenized)):
        seq = text_tokenized[i-seq_len:i]
        target = text_tokenized[i]
        features.append(seq)
        target_one_hot = to_categorical(
            target, num_classes=total_unique_tokens)
        targets.append(target_one_hot)

    if len(features) != len(targets):
        raise ValueError(
            f'Number of feature examples ({len(features)}) is different from

    print(
        f'Created feature ({len(features[0])} words) and target (1 word) pai
    return features, targets
```

Split into training testing and validation

We split the features and targets we obtained above into training, testing and validation sets. The training set is used to train the model. The validation set is used to tune the hyperparameters of the model by trying models with different hyperparameters and then choosing the model that gave the best performance on the validation set. The testing set is used to evaluate the final model on unseen data.

In [17]:
```
def split_training_testing_validation(features, targets, test_size=0.1, val_
    """
    Splits the text into training, testing, and validation sets.

    Parameters
    ----------
    text : str
        The text to split.
    test_size : float, optional
        The size of the testing set, by default 0.1
    val_size : float, optional
        The size of the validation set, by default 0.1

    Returns
    -------
    X_train : list
        A list of training features.
    X_test : list
        A list of testing features.
    X_val : list
        A list of validation features.
    y_train : list
        A list of training targets.
    y_test : list
        A list of testing targets.
    y_val : list
        A list of validation targets.
    """

    from sklearn.model_selection import train_test_split

    X_train, X_test, y_train, y_test = train_test_split(
        features, targets, test_size=test_size+val_size, random_state=56, sh
    X_test, X_val, y_test, y_val = train_test_split(
        X_test, y_test, test_size=val_size/(test_size+val_size), random_stat

    print(
        f"Split dataset into training ({(1-test_size-val_size)*100}%), valic

    return X_train, X_test, X_val, y_train, y_test, y_val
```

Vectorization

Before we feed the sequences to our model we want to ensure our numerical representation contains as much useful information as possible. As humansn, we subconsciously recognize whether words are nouns, verbs, adjectives, etc. We understand grammar, context, synonyms, similar words and many other linguistic nuances. Ideally, we want our model to be able to do the same which is not a trivial task considering computers are only able to understand numbers.

There are many ways to extract text features such as bag-of-words, TF-IDF, etc. In this case, I will focus on one of the most effective and widely used methods, word embeddings.

Most famous word embedding models are Word2Vec, GloVe and FastText. Due to space constraints I will only focus on GloVe which is an extension to Word2Vec.

Word embedding is a way of numerically representing words as vectors so that the words with similar meaning are close to each other in the multidimensional vector space. The goal of this is to capture the semantic meaning of the words. For example, words that appear in a similar context most often have similar meanings and are thus in proximity to one another in the vector space.

Glove (Global Vectors for Word Representation)'s main idea is to use a co-occurrence matrix to capture the relation between words. In a co-occurence matrix the rows and columns are the words in the vocabulary and the values are the number of times the words co-occur *in the same context* such as the same sentence or paragraph.

GloVe then learns to factorize this co-occurrence matrix into two low-rank matrices, representing the words and their contexts, respectively. The resulting word vectors are the rows of the word matrix, while the column vectors of the context matrix represent the context of the corresponding words.

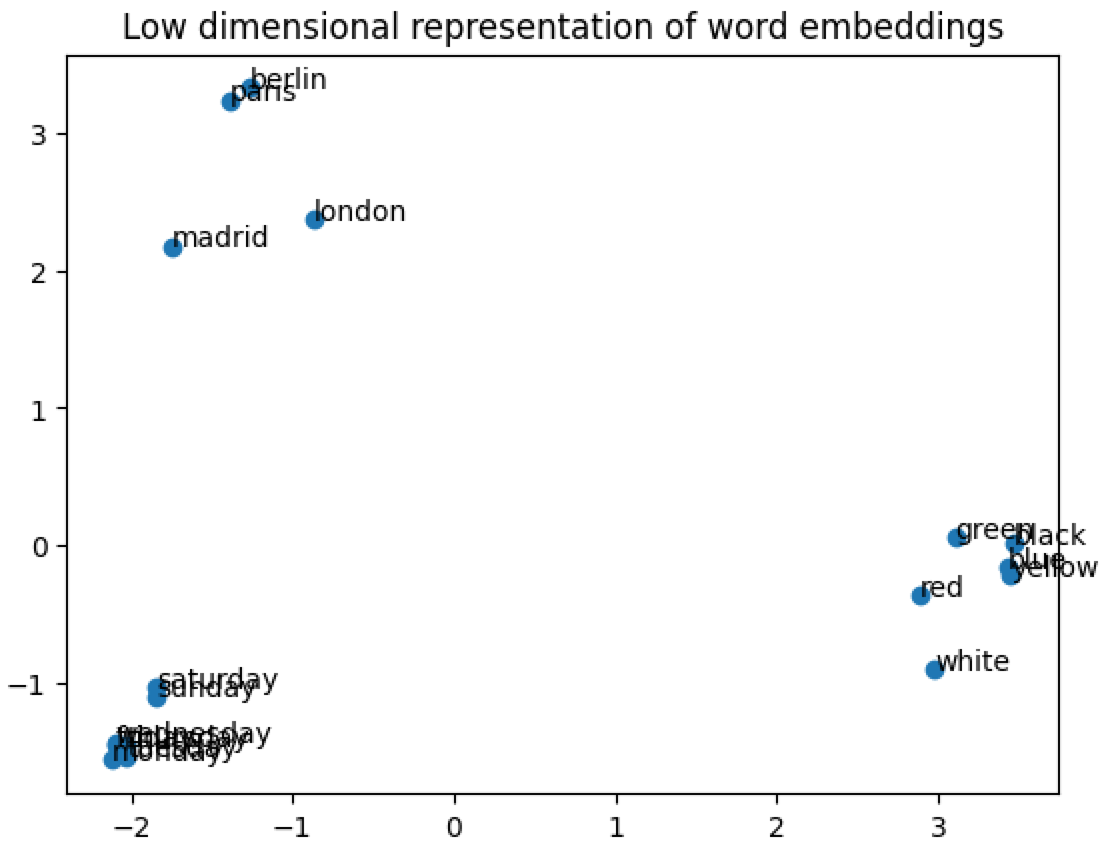
The training objective of the GloVe model is to minimize the difference between the dot product of two word vectors and the log of their co-occurrence count, across all pairs of words in the corpus. This allows the model to learn word vectors that capture the semantic relationships between words and learn things such as analogies or synonyms.

One of the most brilliant GloVe features is that by representing words as vectors, you can do mathematical operations on them such as adding, subtracting, etc. This allows us to do things such as finding the closest words to a given word, finding the odd one out in a list of words, etc.

To demonstrate, I have loaded GloVe and used PCA to reduce the dimensionality of the vectors from 50 to 2 so that we can visualize them and see the clusters of words that have common traits ('colors', 'capitals', 'days of the week')

In [41]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import os

glove_dir = 'glove/glove.6B/'
glove_file = 'glove.6B.50d.txt'
# Load GloVe embeddings
```

```
word_embeddings = {}
with open(os.path.join(glove_dir, glove_file), "rb") as f:
    for line in f:
        values = line.split()
        word = values[0].decode('utf-8')
        coefs = np.asarray(values[1:], dtype='float32')
        word_embeddings[word] = coefs

# Define words to compare
words = ['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday',
         'paris', 'london',  'berlin', 'madrid', 'red', 'blue', 'green', 'ye

# Get their corresponding embeddings
word_vectors = [word_embeddings[w] for w in words]

# Reduce dimensionality to 2D using PCA
pca = PCA(n_components=2)
pca_vectors = pca.fit_transform(word_vectors)

# Plot the 2D embeddings
x = pca_vectors[:, 0]
y = pca_vectors[:, 1]
fig, ax = plt.subplots()
ax.scatter(x, y)

# Add labels to the scatter plot
for i, word in enumerate(words):
    ax.annotate(word, (x[i], y[i]))
plt.title('Low dimensional representation of word embeddings')
plt.show()
```



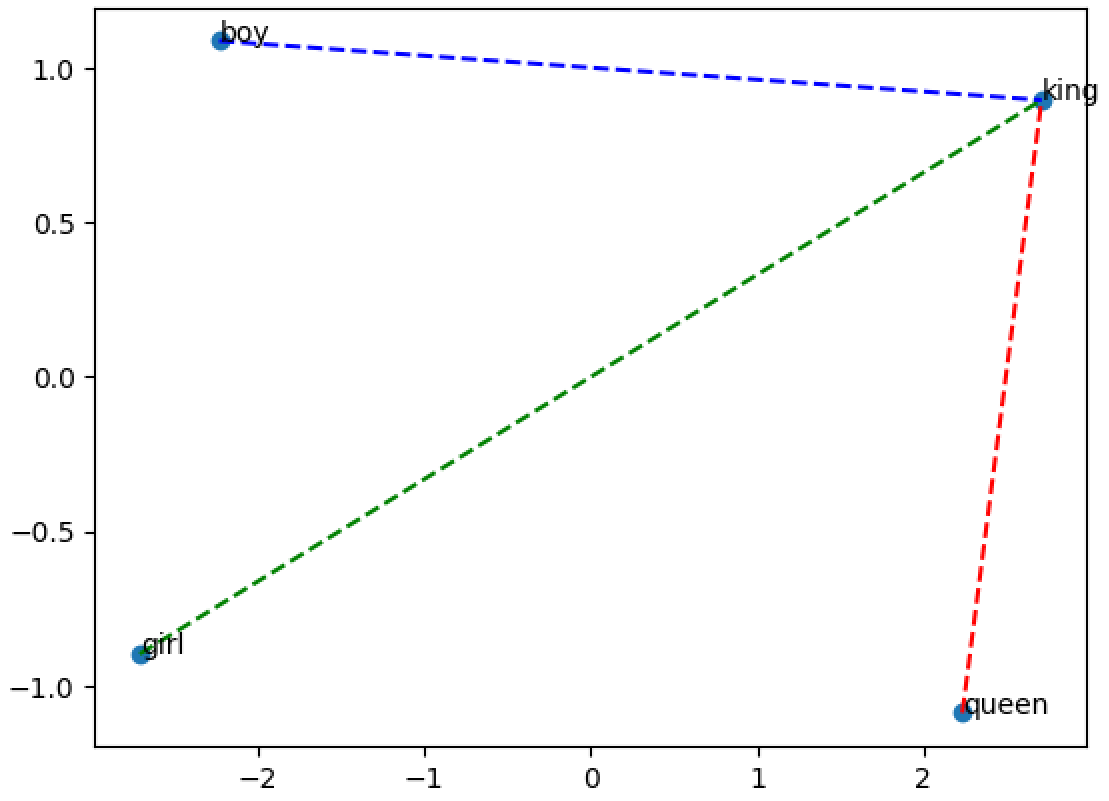
Another interesting thing we can do with word embeddings is perform mathematical operations on the vectors whilst still preserving the meaning of the words. For example, assuming that these high dimensional vectors preserve meaning well, the distance between a *'boy'* and a *'girl'* should be similar to the distance between a *'king'* and a *'queen'*.

Moreover, the vector obtained with *'queen'* - *'girl'* + *'boy'* should be similar to the vector of *'king'*. Lets check if these statements are true below.

In [43]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

glove_dir = 'glove/glove.6B/'
glove_file = 'glove.6B.50d.txt'
# Load GloVe embeddings
word_embeddings = {}
with open(os.path.join(glove_dir, glove_file), "rb") as f:
    for line in f:
        values = line.split()
        word = values[0].decode('utf-8')
        coefs = np.asarray(values[1:], dtype='float32')
        word_embeddings[word] = coefs

# Define the words to perform the operation on
# Define the words to perform the operation on
queen_vec = word_embeddings['queen']
girl_vec = word_embeddings['girl']
boy_vec = word_embeddings['boy']
king_vec = queen_vec - girl_vec + boy_vec

# Calculate cosine similarity between the king vector and the "queen - girl
cosine_sim = np.dot(king_vec, (queen_vec - girl_vec + boy_vec)) / \
    (np.linalg.norm(king_vec) * np.linalg.norm(queen_vec - girl_vec + boy_ve

print("Cosine similarity between the king vector and the queen-girl+boy vect

# Get the corresponding embeddings
word_vectors = [queen_vec, girl_vec, boy_vec, king_vec]

# Reduce dimensionality to 2D using PCA
pca = PCA(n_components=2)
pca_vectors = pca.fit_transform(word_vectors)

# Plot the 2D embeddings
x = pca_vectors[:, 0]
y = pca_vectors[:, 1]
fig, ax = plt.subplots()
ax.scatter(x, y)

# Add lines connecting the "queen", "girl", "boy", and "king" points
ax.plot([x[0], x[3]], [y[0], y[3]], 'r--')
ax.plot([x[1], x[3]], [y[1], y[3]], 'g--')
ax.plot([x[2], x[3]], [y[2], y[3]], 'b--')
```

```
# Add labels to the scatter plot
words = ['queen', 'girl', 'boy', 'king']
for i, word in enumerate(words):
    ax.annotate(word, (x[i], y[i]))

plt.show()
```

```
Cosine similarity between the king vector and the queen-girl+boy vector:
1.0000001
```



As we can see, the value of *cosine similarity* $= (A.B)/(||A||||B||) = 1$ which indicates that the two vectors are identical. -1 would indicate oposite directions whilst 0 would indicate orthogonal vectors.

Finally, we can use the pre-trained GloVe word vectors are as a form of pretrained word embeddings in our model. Below I have written code that loads the GloVe vector and creates an embeddings matrix which we will use as the weights of our embedding layer in our model.

The embeddings matrix is of shape (V, GD) where V is the size of the vocabulary and GD is the dimensionality of the GloVe vectors. Thus, the embedding layer will have an input dimension of V and an output dimension of GD. The weights of the embedding layer will be initialized with the embeddings matrix. This means that the embedding layer will not be trained and will only be used to map the input tokens to their corresponding GloVe vectors. In the embedding matrix, ith row corresponds to the ith vector in our word index dictionary.

In [18]:
```
def load_glove(word_index, glove_dir='./glove/glove.6B', embedding_dim=100):
    """Loads the GloVe embedding matrix.

    Parameters
    ----------
    word_index : dict
        A dictionary with the word index. E.g. {'the': 1, 'cat': 2, 'sat': 3
    glove_dir : str, optional
        The directory where the GloVe embeddings are stored, by default './g
        Can be downloaded from https://nlp.stanford.edu/projects/glove/.
    embedding_dim : int, optional
        The dimension of the GloVe embeddings, by default 100

    Returns
    -------
    embedding_matrix : np.array
        A numpy array with the embedding matrix. Dimensions: (num_words, emb
        Used to initialize the embedding layer in the model. ith row corresp
    """

    import os
    import numpy as np

    try:
        embeddings_index = {}
        f = open(os.path.join(
            glove_dir, f'glove.6B.{embedding_dim}d.txt'), encoding='utf-8')
        for line in f:  # Each line is a word and its embedding vector
            values = line.split()
            word = values[0]
            # The embedding vector
            coefs = np.asarray(values[1:], dtype='float32')
            # Add the word and its embedding vector to the dictionary
            embeddings_index[word] = coefs
        f.close()

        print(f'\nLoaded glove. Found {len(embeddings_index)} word vectors.'

        # prepare embedding matrix
        # The number of words in the word_index + 1 (for the 0th index)
        num_words = len(word_index) + 1
        embedding_matrix = np.zeros((num_words, embedding_dim))
        for word, i in word_index.items():
            embedding_vector = embeddings_index.get(word)
            if embedding_vector is not None:
                # words not found in embedding index will be all-zeros.
                # ith row corresponds to the ith vector in the word_index
                embedding_matrix[i] = embedding_vector
        print(
            f'Created embedding matrix. Dimensions: {embedding_matrix.shape}
        return embedding_matrix
    except Exception as e:
        print('Error occured:', e)
```

Model

I have decided to use an LSTM (Long Short-Term Memory) model architecture which is a type of recurrent neural network (RNN) architecture designed to model sequential data. It has been developed and mitigate the problem of vanishing gradients in vanilla RNNs.

RNNs

To understand LSTMs we must firstly have some understanding of what RNNs are. They are neural networks architecture designed to process sequential data such as text, time-series data, speech, etc. We call them recurrent because they operate as sort of a feedback loop allowing information from one tep in the sequence to be passed and used in the next step, which makes them great for sequential data and data of variable length.

Below we have a visualization of a RNN courtesy of Towards AI.

RNN

They are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.

Now, the vanishing gradient problem happens when the gradients become too small and close to 0 to be of any use for updating the weights of the network. As such, the network becomes unable to learn any long-term dependencies in the data since the information from the very early time steps is lost as the information gets propagated forward through the network. Consequently, RNNs may perform on tasks that require long-term memory, which we frequently need when analyzing time series or text.

Mathematically, the vanishing gradients happen due to backpropagation. Specifically, we take the gradient of the loss function with respect to the weights in the network and multiply it by the gradient of the activation function with respect to the output of the previous layer. This is repeated for every layer in the network. Sometimes these derivatives are small (for example - when we have a sigmoid function as the activation function) and they become even smaller as we multiply them by the gradients of the previous layers. This means that the gradients get smaller and smaller and as the result the weights of the network are not updated properly and the network is unable to learn long-term dependencies.

LSTM

To mitigate this problem, Hochreiter, S., & Schmidhuber, Jurgen. (1997) introduced Long Short-Term memory.

LSTM Visualization by Saul Dobilas.

In the LSTM, every time step of the input sequence is processed by a cell consisting of several important components:

1. *Input gate* - controls what information from the input sequence is passed to the cell state.

$i_t = \sigma(W_{xi}x_t + U_{hi} \times h_{t-1} + b_i)$ 2. *Forget gate* - controls what information from the cell state is passed to the next time step.
$f_t = \sigma(W_{xf}x_t + U_{hf} \times h_{t-1} + b_f)$ 3. *Output gate* - controls what information from the cell state gets passed to the output sequence.
$o_t = \sigma(W_{xo}x_t + U_{ho} \times h_{t-1} + b_o)$ 4. *Memory cell* - stores the information about the input sequence. Can be thought of as a conveyor belt that carries information from one time step to the next whilst selectively removing or adding information to the cell state.
$\tilde{C}_t = f_t \times c_{t_1} + i_t \times \tanh(W_cx \times x_t + U_ch \times h_{t-1} + b_c)$ 5. *Hidden state* - the output of the LSTM cell and is passed to the next time step.
$h_t = o_t \times \tanh(\tilde{C}_t)$

x_t is the input to the cell at time step t, h_{t-1} is the hidden state from the previous time step, i_t is the input gate at t, f_t is the forget gate at t, o_t is the output gate at t, c_{t-1} is the cell state from the previous time step, $\tilde{C}_t$ is the cell state at t, h_t is the hidden state at t.

$W_{xi}, W_{xf}, W_{xo}, W_{xc}$ are the weights of the input gate, $U_{hi}, U_{hf}, U_{ho}, U_{hc}$ are the weights of the hidden state, b_i, b_f, b_o, b_c are the biases of the gates.

The input gate i_t controls how much of the new input x_t should be added to the cell state c_{t-1}. The forget gate f_t controls how much of the previous cell state c_{t-1} should be retained and passed on to the nexts time step, and how much should be discardedd. The output gate o_t controls how much of the cell state c_t should be output to the next layer or the final output of the network.

The memory cell c_t is updated based on the input x_t, the previous cell state c_{t-1}, and the values of the input and forget gates. The tanh function introduces nonlinearity and allows the model to capture more complex patterns in the data.

The hidden state h_t is the final output of the LSTM cell, and is calculated by applying the output gate to the current cell state c_t and passing the result through the tanh function. The hidden state can be used as the input to the next LSTM cell in the sequence or as the final output of the network.

Going back to the vanishing gradient problem, LSTMs mitigate it through the memory cell explained above which controlls the input, forget and the output gate and allows the model to retain information for a long time. Moreover, since the gates use sigmoid and tanh activation function, the values are always between 0 and 1 allowing control of information through the memory cell without them being too small or too large.

Training the LSTM

To train them, we use regular backpropagation with the goal of minimizing the loss that measures the difference between the the predictions and true output for the given input.

Important aspect of sequential dat is that both the input and the output have a temporal relationship which must not be broken. Typically, we use a metric such as a mean squared error or, in this case, categorical cross entropy loss, since our goal is to predict the probability distribution over the vocabulary (categorical variable) given the previous words in the sentence.

The loss function is calculated as follows:

$$H(p, q) = -\sum_i p(i) \log q(i)$$

where p is the true probability distribution over the categories and q is the predicted probability distribution over the categories. In our case, p is the one-hot encoded vector of the true word (1 for the target word and 0 for all others) and q is the probability distribution over the vocabulary.

The negative sign in the equation ensures that the loss is minimized by increasing the predicted probability of the true word and decreasing the predicted probabilities of all other words.

By minimizing the categorical cross-entropy loss during training, the model learns to predict the correct probability distribution over the vocabulary given the previous words in the sentence allowing the model to generate more accurate predictions.

This model is fully implemented under the function *train_lstm* below.

Evaluation

For evaluation of our model we can use varius metrics such as *accuracy*, *perplexity*, *cosine similarity*,*BLEU*, *ROUGE*, etc.

In this case, although I have defined perplexity below since it is just an exponentiated categorical crossentropy or the average negative log-likelihood of a sequence of words, I will just use accuracy and leave other metrics for future versions. Although they are better, they are more time consuming so I had to satisfy myself with accuracy for now.

Accuracy is the number of correct predictions divided by the total number of predictions. In our case, it is the number of correct words we predicted divided by the total number of the predictions we made.

In [19]:
```
# defining custom metric - perplexity
def perplexity(y_true, y_pred):
    """
    Calculates the perplexity of the model.

    Parameters
    ----------
    y_true : tensor
        The true targets.
    y_pred : tensor
        The predicted targets.
    """
    import keras.backend as K

    cross_entropy = K.categorical_crossentropy(y_true, y_pred)
    perplexity = K.exp(cross_entropy)
    return perplexity
```

The function *train_lst* below is used to train the model. The model has the following layers. Stacking two LSTM layers next to one another greatly improved my performance. The final layers is a softmax layer since it gives us the probability distribution over the vocabulary.

The layer hyperparameters were chosen based on the tuning I did through the validation set. The following hyperparameters yielded the greatest validation accuracy.

The following code includes several additional features. I have added both early stopping and model checkpointing. Early stopping is used to prevent overfitting by stopping the training process when the validation loss stops improving. Model checkpointing is used to save the model with the best validation accuracy.

I have also added dropout layers to prevent overfitting. This is a regularization technique that randomly drops out units (along with their connections) from the neural network during training. This prevents units from co-adapting too much.

Besides dropout layers I have also added an l2.regularizer. This is a regularization technique that adds a penalty to the loss function for large weights. This helps prevent overfitting by forcing the model to learn smaller weights.

For the optimizer, I used Adam with the learning rate of 0.001 which proved the best in the validation set. Adam is preferred here to a static learning rate since it has an adaptive learning rate that changes for each parameter. This allows the model to converge faster and more efficiently. It also incorporates momentum by updating the moving average of the gradient and the moving average of the squared gradient once again leading to faster convergence and also helping in leaving local optima.

In [44]:
```
def train_lstm(word_index, embedding_matrix, X_train, X_test, X_val, y_trair
    """
    Trains the LSTM model. It also saves the model and the history as well a

    Parameters
    ----------
    word_index : dict
        A dictionary with the word index. E.g. {'the': 1, 'cat': 2, 'sat': 3
    embedding_matrix : np.array
        A numpy array with the embedding matrix. Dimensions: (num_words, emb
    epochs : int, optional
        The number of epochs to train the model, by default 100
    X_train : list
        A list of training features.
    X_test : list
        A list of testing features.
    X_val : list
        A list of validation features.
    y_train : list
        A list of training targets.
    y_test : list
        A list of testing targets.
    y_val : list
        A list of validation targets.
    batch_size : int, optional
        The batch size, by default 256
    lr : float, optional
        The learning rate, by default 0.001
    embedding_dim : int, optional
        The dimension of the GloVe embeddings, by default 100
    seq_len : int, optional
        The length of the sequences, by default 30
    dropout_rate : float, optional
        The dropout rate, by default 0.2
    weight_decay : float, optional
        The weight decay, by default 1e-3

    Returns
    -------
    model : keras model
        The trained model.
    """

    from keras.models import Sequential
    from keras.layers import Embedding, LSTM, Dense, Dropout
    from keras.optimizers import Adam
    from keras.callbacks import ModelCheckpoint, EarlyStopping
    from keras import regularizers
    from pickle import dump
    from keras.utils.vis_utils import plot_model

    num_words = len(word_index) + 1  # +1 because of the 0 index

    print("\nStarting LSTM model training...")

    model = Sequential()
    model.add(Embedding(num_words, embedding_dim, weights=[
        embedding_matrix], input_length=seq_len, trainable=False))
    model.add(LSTM(units=50, return_sequences=True,
              kernel_regularizer=regularizers.l2(weight_decay)))  # used to
    model.add(Dropout(dropout_rate))  # used to prevent overfitting
    model.add(LSTM(50, kernel_regularizer=regularizers.l2(weight_decay)))
    model.add(Dropout(dropout_rate))  # used to prevent overfitting
    model.add(Dense(50, activation='relu',
              kernel_regularizer=regularizers.l2(weight_decay)))
    model.add(Dense(num_words, activation='softmax'))
    plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer
    model.summary()

    # compile model
    my_opt = Adam(learning_rate=lr)
    model.compile(loss='categorical_crossentropy',
                  optimizer=my_opt,
                  metrics=['accuracy'])

    checkpoint_path = "training_1"

    # Create a callback that saves the model's weights
    cp_callback = ModelCheckpoint(filepath=checkpoint_path,
                                  save_weights_only=True,
                                  verbose=0,
                                  save_freq='epoch')
    # patience is the number of epochs to wait before stopping, if the model
    earlystopping = EarlyStopping(
        monitor='val_accuracy', verbose=0, patience=3, restore_best_weights=

    import numpy as np
    X_train = np.array(X_train)
    y_train = np.array(y_train)
    X_val = np.array(X_val)
    y_val = np.array(y_val)
    X_test = np.array(X_test)
    y_test = np.array(y_test)

    # fit model
    history = model.fit(X_train, y_train, validation_data=(
        X_val, y_val), batch_size=batch_size, epochs=epochs, shuffle=True, c

    # save the model to file
    model_name = f"saved_models/lstm_{embedding_dim}d_{seq_len}seq_{epochs}e
    model.save(model_name)
    history_name = f"saved_models/lstm_{embedding_dim}d_{seq_len}seq_{epochs
    dump(history.history, open(history_name, 'wb'))

    # evaluate model
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
    print()
    print('Test Loss: %f' % (loss))
    print('Test Accuracy: %f' % (accuracy))

    return model, history
```

Plot the results

Below is the code for plotting the result of our training model. The plot shows the loss and accuracy of the model on the training and validation sets.

In [21]:
```
def plot_results(history):
    """
    Plots the training and validation loss and accuracy.

    Parameters
    ----------
    history : keras history
        The history of the training.
    """
    import matplotlib.pyplot as plt

    # plot training and validation loss
    plt.plot(history['loss'], label='train')
    plt.plot(history['val_loss'], label='validation')
    plt.title('Training and validation loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()

    # plot training and validation loss
    plt.plot(history['accuracy'], label='train')
    plt.plot(history['val_accuracy'], label='validation')
    plt.title('Training and validation accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()
    plt.show()
```

Generate sequences

The following code is used to generate new text using the trained model based on the given seed_text. It follows the exact structure explained in the introduction of this paper. If we want to generate 5 words, we do it 1 by one and every time we generate the next word we add it to the seed_text.

Based on the input sequence we use the trained model to make a prediction of the next word which gives us a probability distribution or in other words, the probability for every word in the vocabulary that it is the next word. Then, we just find the word with the highest probability, find its index and see what word that index represents. Done. We add it to the input and repeat the process until we have generated the desired number of words.

In [22]:
```
def generate_seq(model, tokenizer, seq_length, seed_text, n_words):
    """
    Generates a sequence of n_words using the trained model from the seed_te

    Parameters
    ----------
    model : keras model
        The trained model.
    tokenizer : keras tokenizer
        The tokenizer used to tokenize the text.
    seq_length : int
        The length of the sequences.
    seed_text : str
        The seed text.
    n_words : int
        The number of words to generate.
    """

    from keras.utils import pad_sequences
    import numpy as np
    result = list()
    in_text = seed_text
    # generate a fixed number of words
    in_text = data_preprocessing(in_text, save_file=False, verbose=False)
    # remove words that are not in the vocabulary
    in_text = [word for word in in_text if word in tokenizer.word_index]
    in_text = " ".join(in_text)

    # generate a fixed number of words
    for _ in range(n_words):
        # encode the text as integer
        # [0] because it returns a list of lists
        encoded = tokenizer.texts_to_sequences([in_text])[0]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pr
        # predict probabilities for each word
        yhat = model.predict(encoded, verbose=0)
        # get the index of the highest probability
        yhat_max = np.argmax(yhat[0])

        # map predicted word index to word
        predicted_word = ''

        for word, index in tokenizer.word_index.items():
            if index == yhat_max:
                # get the word represented by the index with the highest pro
                predicted_word = word
        # append to input
        in_text += ' ' + predicted_word
        result.append(predicted_word)

    return ' '.join(result)
```

Tying it all together

We tie all of the above mentioned functions together in the following code which follows the pipeline in the introduction. This allows for reproducibility and easy testing of different models and hyperparameters.

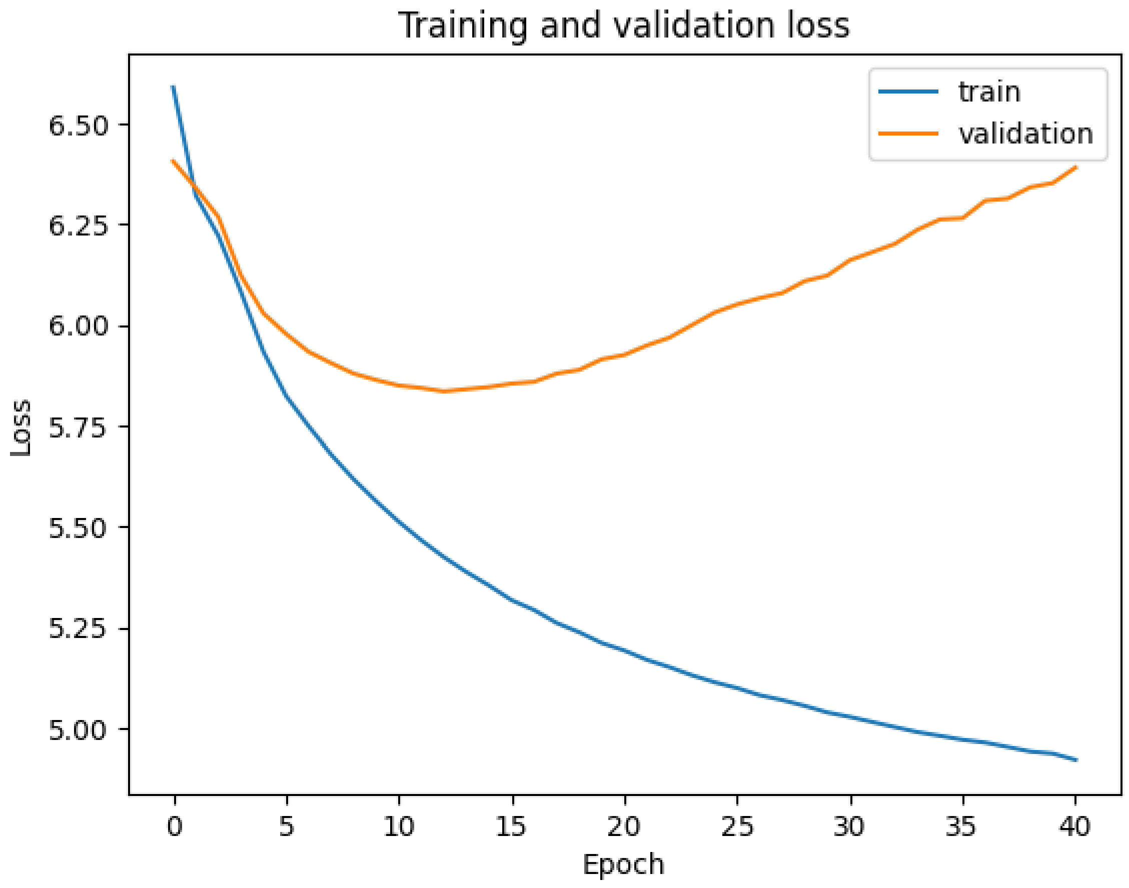
In [48]:
```
def run_everything():
    from pickle import load
    from keras.models import load_model
    import os

    SEQUENCE_LENGTH = 30
    EPOCHS = 50
    BATCH_SIZE = 128
    EMBEDDING_DIM = 50
    LR = 0.001

    # # merge all the data
    # merge_txt_files(input_dir='./database/hp_unprocessed',
    #                 output_dir='database/merged/', name='hp_unprocessed_me
    FILE_NAME = 'Book1.txt'
    # read the data
    text = read_txt_file(file_name=FILE_NAME,
                         folder_path='./database/hp_unprocessed/')

    # preprocess the data
    preprocessed_tokens = data_preprocessing(text, save_file=f'{FILE_NAME}_p
                                             lemmatize=True,  to_remove_emoj

    del text

    # # create the word to index mappings
    word_index, index_word, text_tokenized = get_word_to_index_mappings(
        preprocessed_tokens)

    # # # get the features and targets
    features, targets = get_features_targets(
        text_tokenized, total_unique_tokens=len(word_index)+1, seq_len=SEQUE

    # save features, targets in df file
    import pandas as pd
    df = pd.DataFrame({'features': features, 'targets': targets})
    df.to_csv('features_targets.csv', index=False)
    del df

    # # # split the data into training, testing and validation sets
    X_train, X_test, X_val, y_train, y_test, y_val = split_training_testing_
        features, targets, test_size=0.05, val_size=0.15)

    embedding_matrix = load_glove(word_index, embedding_dim=EMBEDDING_DIM)

    print(
        f"\nSEQUENCE_LENGTH: {SEQUENCE_LENGTH}, EPOCHS: {EPOCHS}, BATCH_SIZE

    model, history = train_lstm(word_index, embedding_matrix, epochs=EPOCHS,
                                seq_len=SEQUENCE_LENGTH, X_train=X_train, X_

    # loading the saved history to make a plot
    saved_model_dir = 'saved_models'
    history_loaded = load(open(os.path.join(
        saved_model_dir, f'lstm_{EMBEDDING_DIM}d_{SEQUENCE_LENGTH}seq_{EPOCH

    # plot the results
    try:
        plot_results(history_loaded)
    except Exception as e:
        print(f"\nError while plotting results: {e}")


if __name__ == '__main__':
    run_everything()
```

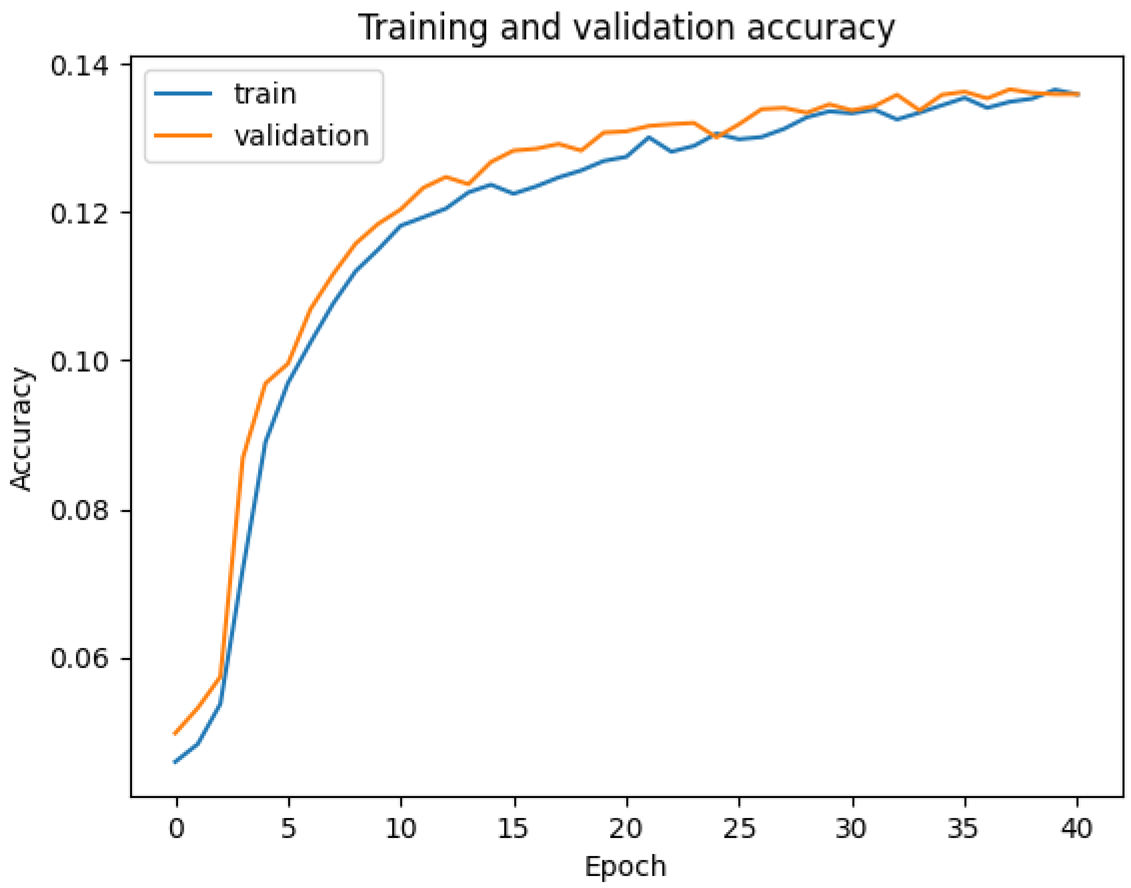
```
Read Book1.txt. It contains 474402 characters and 83183 words.

Preprocessing text...
        Converted to lowercase and stripped multiple whitespaces.
        Removed punctuation and numbers.
        Expanding contractions using contractions model...
        Split the text into tokens.
        Lemmatized words.
        Preprocessed text saved to ./database/processed/Book1.txt_processe
d.
Preprocessing finished. There are now 82488 tokens.

Created word-to-index dictionary. Total number of unique tokens: 5363.
Created feature (30 words) and target (1 word) pairs. Total number of datap
oints: 82458.
Split dataset into training (80.0%), validation (15.0%), testing(5.0%). Siz
es: X_train: 65966, X_test: 4123, X_val: 12369

Loaded glove. Found 400000 word vectors.
Created embedding matrix. Dimensions: (5364, 50).

SEQUENCE_LENGTH: 30, EPOCHS: 50, BATCH_SIZE: 128, EMBEDDING_DIM: 50, LR: 0.
001

Starting LSTM model training...
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 30, 50)            268200

 lstm_2 (LSTM)               (None, 30, 50)            20200

 dropout_2 (Dropout)         (None, 30, 50)            0

 lstm_3 (LSTM)               (None, 50)                20200

 dropout_3 (Dropout)         (None, 50)                0

 dense_2 (Dense)             (None, 50)                2550

 dense_3 (Dense)             (None, 5364)              273564

=================================================================
Total params: 584,714
Trainable params: 316,514
Non-trainable params: 268,200
_________________________________________________________________
Epoch 1/50
516/516 [==============================] - 64s 119ms/step - loss: 6.5902 -
accuracy: 0.0459 - val_loss: 6.4065 - val_accuracy: 0.0498
Epoch 2/50
516/516 [==============================] - 63s 122ms/step - loss: 6.3216 -
accuracy: 0.0483 - val_loss: 6.3407 - val_accuracy: 0.0532
Epoch 3/50
516/516 [==============================] - 63s 123ms/step - loss: 6.2216 -
```

```
accuracy: 0.0537 - val_loss: 6.2686 - val_accuracy: 0.0574
Epoch 4/50
516/516 [==============================] - 64s 123ms/step - loss: 6.0829 -
accuracy: 0.0719 - val_loss: 6.1237 - val_accuracy: 0.0869
Epoch 5/50
516/516 [==============================] - 61s 117ms/step - loss: 5.9336 -
accuracy: 0.0890 - val_loss: 6.0291 - val_accuracy: 0.0969
Epoch 6/50
516/516 [==============================] - 60s 116ms/step - loss: 5.8240 -
accuracy: 0.0969 - val_loss: 5.9783 - val_accuracy: 0.0995
Epoch 7/50
516/516 [==============================] - 60s 115ms/step - loss: 5.7490 -
accuracy: 0.1024 - val_loss: 5.9337 - val_accuracy: 0.1069
Epoch 8/50
516/516 [==============================] - 60s 116ms/step - loss: 5.6794 -
accuracy: 0.1076 - val_loss: 5.9055 - val_accuracy: 0.1116
Epoch 9/50
516/516 [==============================] - 60s 116ms/step - loss: 5.6180 -
accuracy: 0.1120 - val_loss: 5.8794 - val_accuracy: 0.1157
Epoch 10/50
516/516 [==============================] - 65s 126ms/step - loss: 5.5633 -
accuracy: 0.1149 - val_loss: 5.8635 - val_accuracy: 0.1184
Epoch 11/50
516/516 [==============================] - 66s 128ms/step - loss: 5.5125 -
accuracy: 0.1181 - val_loss: 5.8497 - val_accuracy: 0.1203
Epoch 12/50
516/516 [==============================] - 61s 119ms/step - loss: 5.4664 -
accuracy: 0.1192 - val_loss: 5.8437 - val_accuracy: 0.1232
Epoch 13/50
516/516 [==============================] - 61s 118ms/step - loss: 5.4246 -
accuracy: 0.1204 - val_loss: 5.8352 - val_accuracy: 0.1247
Epoch 14/50
516/516 [==============================] - 62s 119ms/step - loss: 5.3876 -
accuracy: 0.1226 - val_loss: 5.8407 - val_accuracy: 0.1237
Epoch 15/50
516/516 [==============================] - 62s 121ms/step - loss: 5.3542 -
accuracy: 0.1236 - val_loss: 5.8461 - val_accuracy: 0.1267
Epoch 16/50
516/516 [==============================] - 60s 117ms/step - loss: 5.3178 -
accuracy: 0.1224 - val_loss: 5.8545 - val_accuracy: 0.1282
Epoch 17/50
516/516 [==============================] - 60s 117ms/step - loss: 5.2932 -
accuracy: 0.1234 - val_loss: 5.8589 - val_accuracy: 0.1285
Epoch 18/50
516/516 [==============================] - 60s 117ms/step - loss: 5.2610 -
accuracy: 0.1246 - val_loss: 5.8794 - val_accuracy: 0.1291
Epoch 19/50
516/516 [==============================] - 61s 117ms/step - loss: 5.2380 -
accuracy: 0.1255 - val_loss: 5.8886 - val_accuracy: 0.1282
Epoch 20/50
516/516 [==============================] - 61s 118ms/step - loss: 5.2113 -
accuracy: 0.1268 - val_loss: 5.9150 - val_accuracy: 0.1306
Epoch 21/50
516/516 [==============================] - 61s 118ms/step - loss: 5.1926 -
accuracy: 0.1274 - val_loss: 5.9257 - val_accuracy: 0.1308
Epoch 22/50
```

```
516/516 [==============================] - 61s 119ms/step - loss: 5.1694 -
accuracy: 0.1300 - val_loss: 5.9494 - val_accuracy: 0.1315
Epoch 23/50
516/516 [==============================] - 61s 119ms/step - loss: 5.1514 -
accuracy: 0.1281 - val_loss: 5.9687 - val_accuracy: 0.1318
Epoch 24/50
516/516 [==============================] - 63s 122ms/step - loss: 5.1311 -
accuracy: 0.1289 - val_loss: 6.0000 - val_accuracy: 0.1319
Epoch 25/50
516/516 [==============================] - 61s 119ms/step - loss: 5.1139 -
accuracy: 0.1305 - val_loss: 6.0310 - val_accuracy: 0.1300
Epoch 26/50
516/516 [==============================] - 62s 120ms/step - loss: 5.0994 -
accuracy: 0.1297 - val_loss: 6.0512 - val_accuracy: 0.1318
Epoch 27/50
516/516 [==============================] - 62s 120ms/step - loss: 5.0816 -
accuracy: 0.1301 - val_loss: 6.0668 - val_accuracy: 0.1338
Epoch 28/50
516/516 [==============================] - 63s 122ms/step - loss: 5.0696 -
accuracy: 0.1312 - val_loss: 6.0794 - val_accuracy: 0.1340
Epoch 29/50
516/516 [==============================] - 64s 124ms/step - loss: 5.0551 -
accuracy: 0.1327 - val_loss: 6.1092 - val_accuracy: 0.1333
Epoch 30/50
516/516 [==============================] - 62s 121ms/step - loss: 5.0388 -
accuracy: 0.1335 - val_loss: 6.1229 - val_accuracy: 0.1344
Epoch 31/50
516/516 [==============================] - 63s 123ms/step - loss: 5.0277 -
accuracy: 0.1333 - val_loss: 6.1611 - val_accuracy: 0.1336
Epoch 32/50
516/516 [==============================] - 66s 128ms/step - loss: 5.0151 -
accuracy: 0.1337 - val_loss: 6.1812 - val_accuracy: 0.1342
Epoch 33/50
516/516 [==============================] - 63s 121ms/step - loss: 5.0024 -
accuracy: 0.1324 - val_loss: 6.2017 - val_accuracy: 0.1357
Epoch 34/50
516/516 [==============================] - 68s 132ms/step - loss: 4.9900 -
accuracy: 0.1333 - val_loss: 6.2369 - val_accuracy: 0.1336
Epoch 35/50
516/516 [==============================] - 67s 129ms/step - loss: 4.9807 -
accuracy: 0.1343 - val_loss: 6.2624 - val_accuracy: 0.1357
Epoch 36/50
516/516 [==============================] - 66s 128ms/step - loss: 4.9711 -
accuracy: 0.1353 - val_loss: 6.2653 - val_accuracy: 0.1361
Epoch 37/50
516/516 [==============================] - 64s 124ms/step - loss: 4.9641 -
accuracy: 0.1340 - val_loss: 6.3088 - val_accuracy: 0.1353
Epoch 38/50
516/516 [==============================] - 62s 121ms/step - loss: 4.9530 -
accuracy: 0.1348 - val_loss: 6.3142 - val_accuracy: 0.1365
Epoch 39/50
516/516 [==============================] - 64s 125ms/step - loss: 4.9418 -
accuracy: 0.1352 - val_loss: 6.3424 - val_accuracy: 0.1360
Epoch 40/50
516/516 [==============================] - 62s 119ms/step - loss: 4.9366 -
accuracy: 0.1364 - val_loss: 6.3527 - val_accuracy: 0.1358
```

```
Epoch 41/50
516/516 [==============================] - 61s 119ms/step - loss: 4.9210 -
accuracy: 0.1358 - val_loss: 6.3911 - val_accuracy: 0.1358

Test Loss: 6.372669
Test Accuracy: 0.135581
```





In [47]:
```
# using the saved model to generate text

from pickle import load
from keras.models import load_model

SEQUENCE_LENGTH = 30
EPOCHS = 50
BATCH_SIZE = 128
EMBEDDING_DIM = 50
LR = 0.001

num_of_words_to_generate = 20
input_text = "What do you think? Harry potter is an amazing wizard. He fough

tokenizer = load(open('tokenizer.pkl', 'rb'))
model_name = f'lstm_{EMBEDDING_DIM}d_{SEQUENCE_LENGTH}seq_{EPOCHS}epochs_{LR
saved_model_dir = 'saved_models'
model_loaded = load_model(os.path.join(
    saved_model_dir, model_name))
history_loaded = load(open(os.path.join(
    saved_model_dir, f'lstm_{EMBEDDING_DIM}d_{SEQUENCE_LENGTH}seq_{EPOCHS}ep

# generate text
generated_seq = generate_seq(
    model_loaded, tokenizer, SEQUENCE_LENGTH, seed_text=input_text, n_words=
print(
    f'\nGenerating text...\nInput: {input_text}. \nGenerated text: {generate
```

```
Generating text...
Input: What do you think? Harry potter is an amazing wizard. He fought agai
nst many but we might need to adjust the proposition to.
Generated text: you to be a bit of the clock i am not got to get to get to
get a bit
```

Discussion

Although the final result was modestly successful (accuracy of around 14% but more importantly the text generated does follow the style of Rowling), I spent days testing different models and hyperparameters on the validation set to get to this point. I tried so many different things. The primary problem was that the model would overfit heavily even after the first epoch. I actually tried using perplexity but it would not cooperate and would skyrocket in every epoch by nearly 2 orders of magnitude sometimes. To mitigate overfitting, I modified many things. I tried reducing the batch size so it is a bit more sensitive to change. I changed other hyperparameters and tried them as well such as different optimizers and different learning rates, tried different activation functions, different loss functions, different regularization techniques as well as different number of layers and different number of units in each layer. I tried the changes in structure since I though that it might be possible is to complex and that I didn't have enough data so it was able to learn the peculiarities and patterns in the data too well (bias-variance tradeoff).

It was also taking around 20 minutes per epoch at some times so I had to reduce the number of parameters and the dimensionality.

I added more books increased the dataset but my local machine could not handle that so I actually resorted to using only a single one. This is the final structure and hyperparameters that gave the best results.

Another problem is that it sometimes gets stuck and just generated one sequence of words over and over again, which I believe is caused by the small dataset.

In the future versions, I will try bigger datasets and will have to use a cloud service to train the model. I could also try different metrics and understanding why validation perplexity tends to jump rapidly. I will also hopefully have more of my own personal data to use. More importantly, trying out different models such as some transformer models or GNN models and comparing them to this result would be a great next step.

Ethical considerations

Since I did not end up fully utilizing the my personal keylogged data, there were no ethical considerations to make in regards to the privacy of the data. Had I used that, it would have been seriously compromising to my own privacy and safety (since it logs everything from passwords to personal conversations and could potentially infringe on the privacy of others if I am not careful)

However, perhaps there should be a discussion the ethics of mimicking the writing styles of others however and the implications of that. These models, especially LLMs require a lot of data and very frequently that is copyrighted material. In this case, all the books that I had downloaded were copyright free and distributed under the license of Project Gutenberg which allows for free distribution of the books. However, this is not always the case. For example, I really wanted to use Tolkien's works but they are not public domain - which is perhaps for the better.