# One Page Summary

## Case Study in Sentiment Analysis

### Problem Statement

In this project, I will try to recognize individuals' emotions/opinions on self-driving cars from tweets, a textual message users post on the social media website Twitter, using **sentiment analysis**. I will use Word2Vec and GloVe to generate word embeddings from text which will be inputted to a convolutional neural network (CNN) built using the Keras API for recognizing the emotion from sample tweets. I chose this statement because companies, researchers, etc. can learn more about individuals' emotional perceptions of self-driving cars using my technology.

### Overview of Technology:

Word2Vec and GloVe are two solutions that enable convenient generation of word embeddings, defined as mapping words to vectors, for corpuses of any size to mine meaningful information from textual data. The Keras API is a useful solution for building CNNs and tuning parameters to achieve optimal model performance. The combination of both of these solutions powers the creation of intuitive solutions to sentiment analysis problems (e.g., detecting Twitter users' emotions related to self-driving cars), which this project will explore.

### Description of Data:

A simple Twitter sentiment analysis dataset provided by Crowdflower where tweets are classified as very positive, slightly positive, neutral, slightly negative, or very negative. **Size of dataset**: 1.15 MB. **Format**: UTF-8 csv file.

### High Level Overview of Steps:

1. Install Anaconda, TensorFlow, Jupyter Notebook, Pandas, matplotlib, h5py, nltk, keras, genism, and scikit-learn
2. Download and preprocess the data
3. Generate Word2Vec embeddings
4. Input Word2Vec embeddings and GloVe embeddings into Convolutional Neural Network built with Keras API

### Description of Hardware

Windows 10 Home edition, Intel® Core™ i7-7700T CPU processor, 64-bit operating system, x86-based processor, 16GB of RAM. No graphics processing unit was used.

### Software:

The Anaconda platform/distribution (version 4.4.10), Jupyter Notebook, Keras, Word2Vec, GloVe, and TensorFlow Other packages include: pandas, matplotlib, h5py, nltk, genism, scikit-learn, The Natural Language Toolkit (NLTK).

### References:

"Step-by-Step Twitter Sentiment Analysis: Visualizing United Airlines' PR Crisis" (http://ipullrank.com/step-step-twitter-sentiment-analysis-visualizing-united-airlines-pr-crisis/)

"Using pre-trained word embeddings in a Keras model" (https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)

"How to Develop a Word Embedding Model for Predicting Movie Review Sentiment" (https://machinelearningmastery.com/develop-word-embedding-model-predicting-movie-review-sentiment/)

"Emotion Detection on Twitter Data using Knowledge Base Approach" (https://pdfs.semanticscholar.org/6698/5a996eab1e680ffdd88a4e92964ac4e7dd56.pdf)

### *Acknowledgements of Datasets*

- Twitter sentiment analysis: Self-driving cars dataset available at: https://data.world/crowdflower/sentiment-self-driving-cars.
- Pre-trained GloVe word vectors that GloVe creators provided, available at: https://nlp.stanford.edu/projects/glove/webpage.

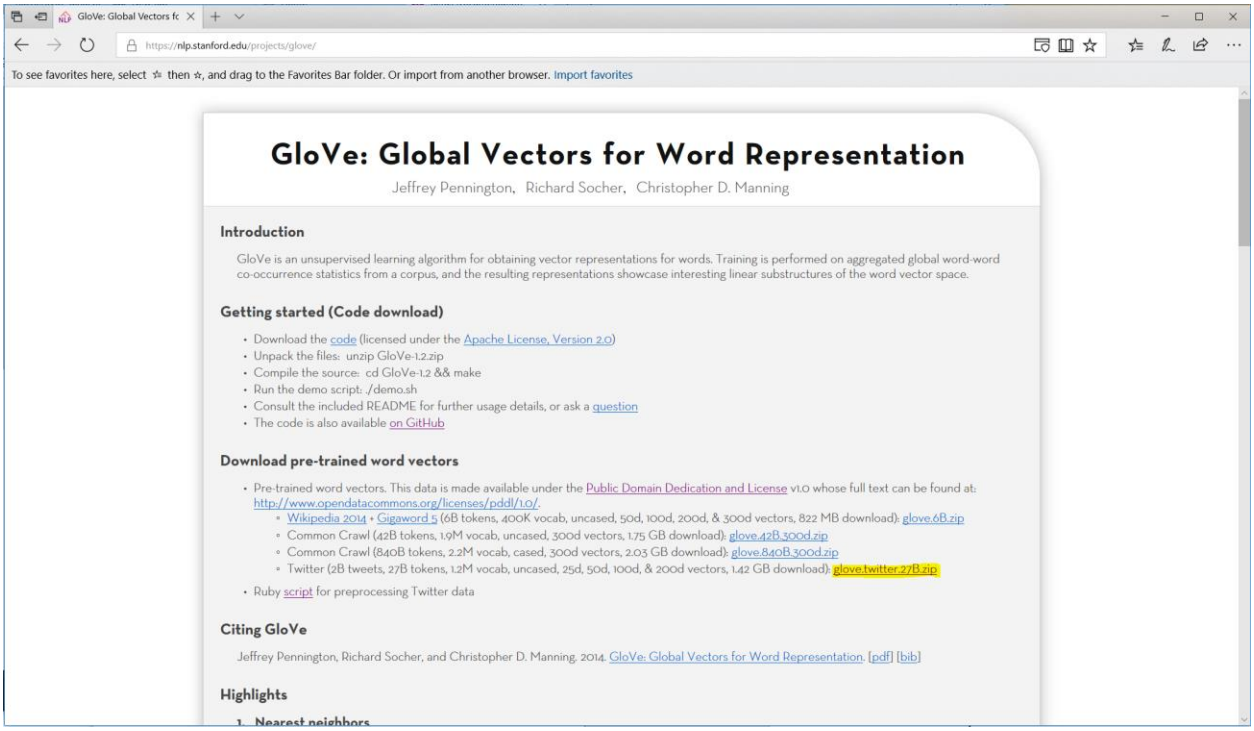
### Lessons Learned:

A larger data set would benefit the model performance. Using the Word2Vec embeddings trained on the same dataset the model was trained on led to better model performance compared to the GloVe embeddings trained by the creators of GloVe on a large Twitter Corpus. Pros: Ease of design and implementation using the Keras API, Word2Vec algorithm, and pre-trained GloVe embeddings. Tuning model parameters was convenient, and Keras' text-preprocessing functions were extremely helpful. Cons: Unbalanced dataset (more neutral tweets than very positive/negative tweets) and volatility of the predictions (the model's predictions vary from run to run).

## Description of Technology (Provide URLs of downloads)

The technology that is required for this problem includes:

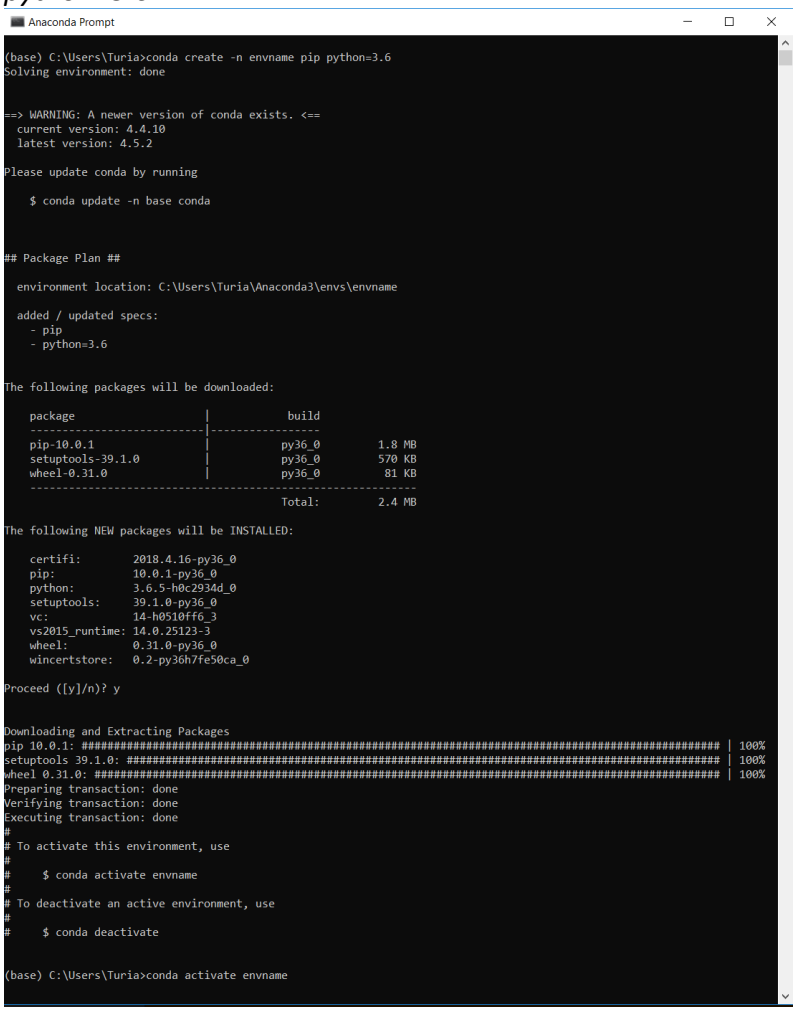
- The Anaconda platform/distribution that easily installs data science packages, such as pip, numpy, pandas, Jupyter Notebook, Matplotlib, etc. that will be used for this project. I used conda version 4.4.10, although the latest version could also be downloaded from this URL: https://www.anaconda.com/download/. The pandas, matplotlib, and h5py packages are also important for this project (detailed installation instructions are provided in the below section).
- Keras, a high-level neural networks API, that is used in this project to preprocess text and create the convolutional neural networks. Keras can be installed using the command *pip install keras* in the Anaconda Prompt (details on installation are below). The website https://keras.io/ provides more information on this API.
- Word2Vec is a group of neural network models used to generate word embeddings, or words mapped to vectors of real numbers. By using the command *pip install genism*, Word2Vec can be installed (details on installation are below). License: Apache License 2.0.
- The Natural Language Toolkit (NLTK) is a common platform used for text processing. In this project, it is used to remove stop-words (unnecessary/extraneous words) from the text. NLTK can be installed using the command *pip install nltk*.
- GloVe is an unsupervised learning algorithm for obtaining vector representations for words. In the field of sentiment analysis, the GloVe technology is most commonly used by training models on pre-trained word vectors. For this project, I am using pre-trained word vectors on the Twitter dataset, titled glove.twitter.27B.zip, available on the https://nlp.stanford.edu/projects/glove/ webpage.



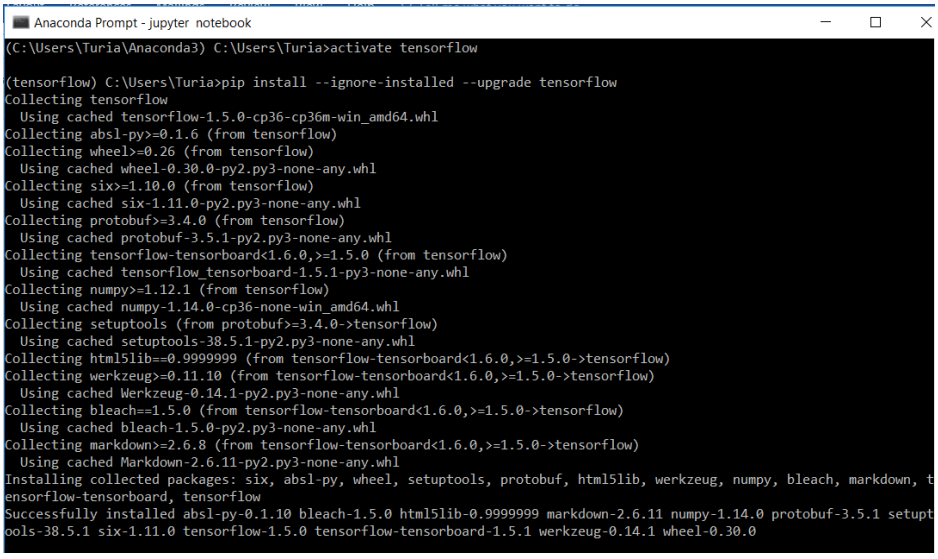
## Steps and Demonstration

### *Installation*

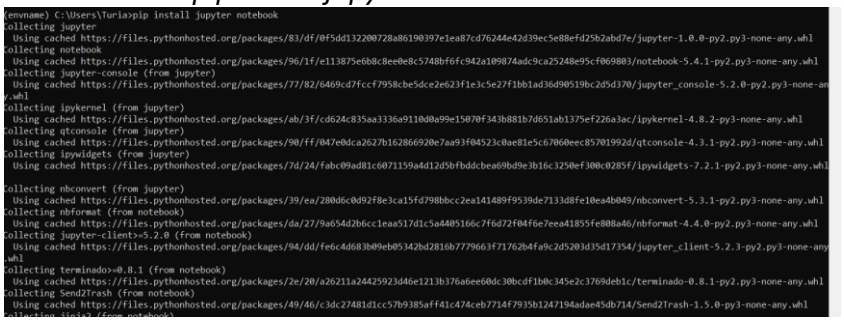
- After opening Anaconda Prompt (installed as a part of the Anaconda platform/distribution) please use the following commands. Please note, I have chosen to name my environment name tensorflow for convenience for other projects; feel free to replace the environment name tensorflow with another name. Commands to be entered in Anaconda Prompt are italicized.
    - Creating an environment for this project: *conda create -n envname pip python=3.6*



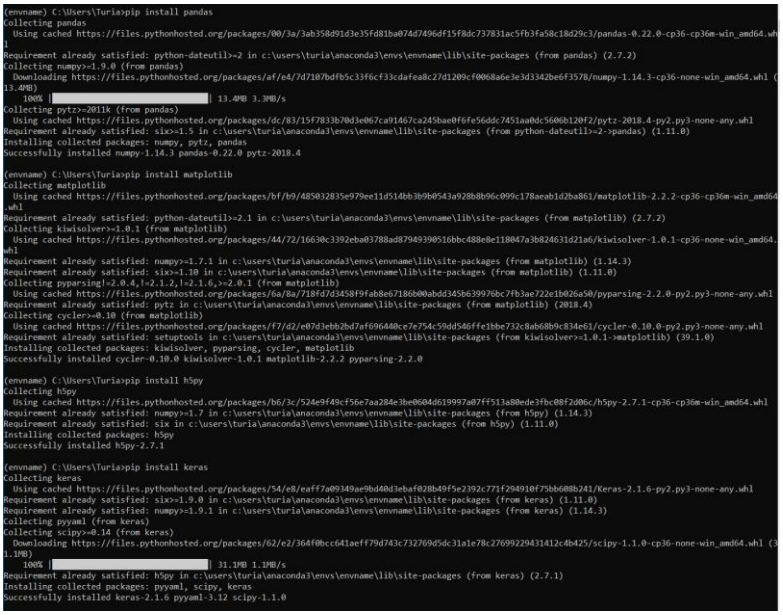

    - Activating the environment: *activate envname*
    - Installing TensorFlow: *pip install --ignore-installed --upgrade tensorflow*




- Next, install jupyter notebook within the environment that was created through the command *pip install jupyter notebook*.



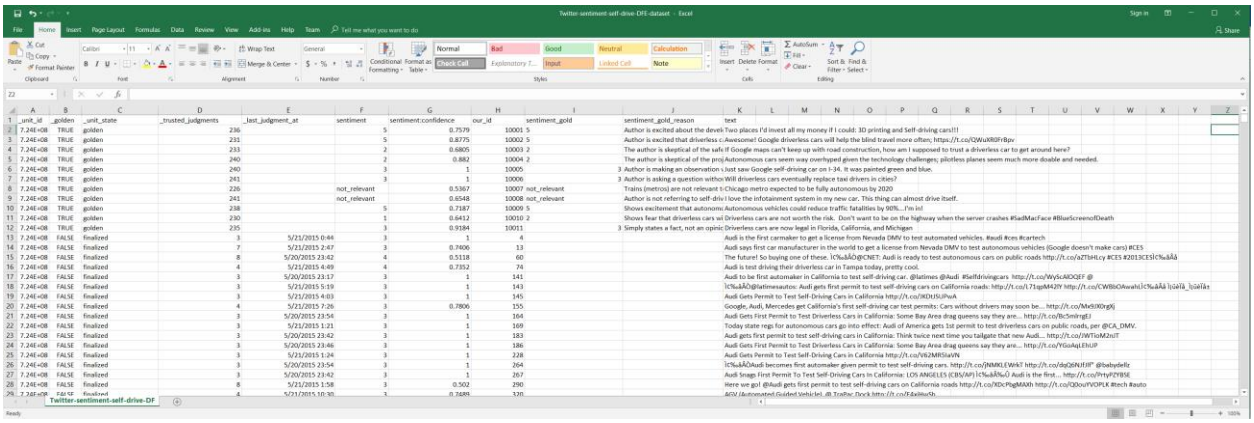

- Use *pip install pandas* command to install pandas within the environment
- Use *pip install matplotlib* command to install matplotlib within the environment.
- Use *pip install h5py* command to install hpy.
- Use *pip install nltk* command to install nltk.
- Use *pip install keras* command to install keras
- Use *pip install genism* command to install genism and Word.
- Use *conda install scikit-learn* command to install sklearn, used for generating PCA plots.




- Then, I typed in the command jupyter notebook, which opened up the jupyter notebook and provided me with a URL to click for first time setup.

## Describe data wrangling/pre-processing of data.

Here is a picture of the dataset file, which is about 1.15 MB. I converted the Microsoft Excel file into a UTF-8 csv file for compatibility with Jupyter Notebook. A picture of the data file in Microsoft Excel:



From now on, I will refer to this dataset as the Crowdflower dataset. This file contains 11 columns, although 9 of the columns are not relevant for solving the problem and are also not explained by the dataset provider:

- _unit_id
- _golden
- _unit_state
- _trusted_judgments
- _last_judgment_at
- sentiment
- sentiment:confidence
- our_id
- sentiment_gold

- sentiment_gold_reason
- text

For the purposes of solving the problem, I will focus on two columns of the dataset: *sentiment*, which is a column containing values one through 5 representing the various sentiments (explained below), and *text*, which is a column containing the tweets.

The *sentiment* column contains tweets written by individuals on the topic of self-driving cars that were classified by contributors into the following categories:

- Very positive – 5
  - Example tweet: "Two places I'd invest all my money if I could: 3D printing and Self-driving cars!!!"
- Slightly positive – 4
  - Example tweet: "Audi is test driving their driverless car in Tampa today, pretty cool."
- Neutral – 3
  - Example tweet: "Driverless cars are now legal in Florida, California, and Michigan"
- Slightly negative – 2
  - Example tweet: "If i need to constantly supervise the car it's not autonomous #vlabauto"
- Very negative – 1
  - Example tweet: "Driverless cars are not worth the risk. Don't want to be on the highway when the server crashes #SadMacFace #BlueScreenofDeath"

We can observe from the tweets that the five categories seem to nicely delineate the emotions of individuals regarding self-driving cars (e.g. the *very positive* category represents excitement and the *very negative* category represents anger and sarcasm).

First, I am going to import the necessary packages:

```
#Importing the necessary packages
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
%matplotlib inline
import urllib.request
import numpy as np
import pandas as pd
import sys
import os
import nltk
import gensim
from gensim.utils import tokenize
from gensim.models import Word2Vec
import re
from nltk.corpus import stopwords
import sklearn
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from numpy import asarray
from numpy import zeros
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.models import model_from_json
from keras import optimizers
from keras import regularizers
```

Next, I read the dataset into a pandas data frame using the following code, and consequently displayed the first 11 rows of the data frame:

```
#Read the dataset into a pandas data frame

senti = pd.read_csv("./Twitter-sentiment-self-drive-DFE-dataset.csv")
```

**Output: Dimensions of the Dataframe: (7156, 11)**

```
pd.options.display.max_colwidth = 2000

senti[:12]
```

| | _unit_id | _golden | _unit_state | _trusted_judgments | _last_judgment_at | sentiment | sentiment:confidence | our_id | sentiment_gold | sentiment_gold_reason | text |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 724227031 | True | golden | 236 | NaN | 5 | 0.7579 | 10001 | 5\n4 | Author is excited about the development of the technology. | Two places I'd invest all my money if I could: 3D printing and Self-driving cars!!! |
| 1 | 724227032 | True | golden | 231 | NaN | 5 | 0.8775 | 10002 | 5\n4 | Author is excited that driverless cars will benefit the disabled. | Awesome! Google driverless cars will help the blind travel more often; https://t.co/QWuXR0FrBpv |
| 2 | 724227033 | True | golden | 233 | NaN | 2 | 0.6805 | 10003 | 2\n1 | The author is skeptical of the safety and reliability of a driverless car. | If Google maps can't keep up with road construction, how am I supposed to trust a driverless car to get around here? |
| 3 | 724227034 | True | golden | 240 | NaN | 2 | 0.8820 | 10004 | 2\n1 | The author is skeptical of the project's value. | Autonomous cars seem way overhyped given the technology challenges; pilotless planes seem much more doable and needed. |
| 4 | 724227035 | True | golden | 240 | NaN | 3 | 1.0000 | 10005 | 3 | Author is making an observation without expressing an opinion. | Just saw Google self-driving car on I-34. It was painted green and blue. |
| 5 | 724227036 | True | golden | 241 | NaN | 3 | 1.0000 | 10006 | 3 | Author is asking a question without expressing an opinion. | Will driverless cars eventually replace taxi drivers in cities? |
| 6 | 724227037 | True | golden | 226 | NaN | not_relevant | 0.5367 | 10007 | not_relevant | Trains (metros) are not relevant to the focus of this project. Please re-read the instructions. | Chicago metro expected to be fully autonomous by 2020 |
| 7 | 724227038 | True | golden | 241 | NaN | not_relevant | 0.6548 | 10008 | not_relevant | Author is not referring to self-driving cars. Please re-read the instructions. | I love the infotainment system in my new car. This thing can almost drive itself. |
| 8 | 724227039 | True | golden | 238 | NaN | 5 | 0.7187 | 10009 | 5\n4 | Shows excitement that autonomous vehicles will improve safety | Autonomous vehicles could reduce traffic fatalities by 90%...I'm in! |
| 9 | 724227040 | True | golden | 230 | NaN | 1 | 0.6412 | 10010 | 2\n1 | Shows fear that driverless cars will not be safe or reliable | Driverless cars are not worth the risk. Don't want to be on the highway when the server crashes #SadMacFace #BlueScreenofDeath |
| 10 | 724227041 | True | golden | 235 | NaN | 3 | 0.9184 | 10011 | 3 | Simply states a fact, not an opinion | Driverless cars are now legal in Florida, California, and Michigan |
| 11 | 724227610 | False | finalized | 3 | 5/21/2015 0:44 | 3 | 1.0000 | 4 | NaN | NaN | Audi is the first carmaker to get a license from Nevada DMV to test automated vehicles. #audi #ces #cartech |

The many rows of the data frame make it challenging to view in one window. Consequently, I am only going to keep two columns of the dataset: *sentiment*, which is a column containing values one through 5

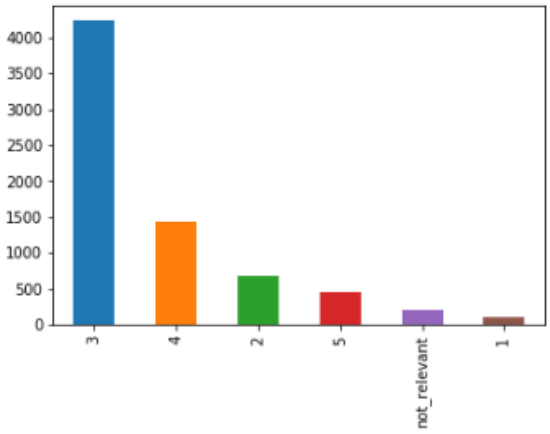
representing the various sentiments (explained below), and *text*, which is a column containing the tweets. I dropped the other columns using the code:

```
senti = senti.drop(['_unit_id', '_golden', '_unit_state', '_trusted_judgments',
'_last_judgment_at', 'sentiment:confidence', 'our_id', 'sentiment_gold',
'sentiment_gold_reason'], axis=1).
```

Now, let's visualize the frequency of the values in the sentiment column (ranging from 0 to 5).

```
In [4]: import matplotlib.pyplot as plt
        fig, ax = plt.subplots()
        senti['sentiment'].value_counts().plot(ax=ax, kind='bar')
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x1df12bbbdd8>
```




It appears that many values in the dataset are value *3* and few are value *1*, which means the dataset might not be representative of individuals with a *very negative* perception on self-driving cars expressed through their tweets. The *not_relevant* bar on the graph illustrates there are NaN values in the sentiment column, which I dropped using the following lines of code:

```
#Remove NaN values, which are called 'not_relevant' in this dataset
senti = senti[senti.sentiment != 'not_relevant']
#Resetting the index of the dataframe to reflect the modified dataset
senti = senti.reset_index(drop=True)
```

Now, the *senti* dataframe contains **6943 rows** and **2 columns**.

The paper "Emotion Detection on Twitter Data using Knowledge Base Approach" and the webpage "Step-by-Step Twitter Sentiment Analysis: Visualizing United Airlines' PR Crisis" both offer recommendations as to how to pre-process tweets for sentiment analysis, as the pre-processing tweet is often essential in the sentiment analysis field to achieve good model performance. First, let's look at a typical tweet.

```
The future! So buying one of these. Ì¢‰âÂÒ@CNET: Audi is ready to test autonomous cars on public roads http://t.co/aZTbHLcy #CES
#2013CESÌ¢‰âÂå
```

Special characters, @username links, #name links, and uppercase characters do not offer significant meaning in the phrase and can harm the model performance. Consequently, let's begin with the pre-processing step.

First, I made the characters of the tweet lowercase using *text_col = text_col.str.lower()* I encountered errors (e.g., AttributeError: 'Series' object has no attribute 'split') in terms when using *text_col = text_col.lower()* for every row in a dataframe in Python, which was solved by adding the *.str* for using string methods on series.

Next, using **regex** operations (a package that is already present in python), I removed URL links, @username links, #links, additional whitespaces, numbers/digits, special characters, and stop words with regular expressions. I took and **modified** the regex operations provided by the webpage "Step-by-Step Twitter Sentiment Analysis: Visualizing United Airlines' PR Crisis", as using the same code provided on the webpage led to errors. The primary modification I made was converting the use of re.sub() into text.replace(), which is a method that is dataframe-friendly.

```
#Making the characters lowercase
text_col = text_col.str.lower()
#Removing the URL Links with a regular expression.
text_col = text_col.replace(r'((www\.[^\s]+)|(https?://[^\s]+))', '', regex=True)
#Removing the @username Links with a regular expresion.
text_col = text_col.replace('@[^\s]+', '', regex=True)
#Some tweets involve the use of the at (@) symbol, whitespace, and then a word, such as: @ Trapack dock
#Here we are removing the @ username Links with a regular expression.
text_col = text_col.replace('@ [^\s]+', '', regex=True)
#Replacing the # Links with a regular expresion.
text_col = text_col.replace(r'#([^\s]+)', r'\1', regex=True)
#Removing additional whitespaces
text_col = text_col.replace('[\s]+', ' ', regex=True)
#Remove all digits
text_col = text_col.replace(r'\w*\d\w*', '', regex=True)
#Removing special Characters
text_col = text_col.replace('([^\s\w-]|_)+', "", regex=True)
text_col = text_col.replace('ì|¡|□|ü|è|ï|å|ã|_|ì|±|¢|‰|â|å|□|Ì|¢|‰|Â|Ò|ò|û|ª', " ", regex=True)
text_col = text_col.replace('-', " ", regex=True)
text_col
```

There are some important design choices that were made in the above code that were carefully thought out and are worth noting:

- In a word (e.g. "Ì¢‰âÂ□ÒAudi"), only the special characters are removed, and letters are kept as is. The goal is to keep words that could potentially contribute to the meaning/sentiment of the phrase.
- Furthermore, I did not remove the special character "-" from words, since the word "self-driving car" is prevalent in this dataset and removing the "-" would remove the spacing between "self" and "driving" and therefore impact the meaning of the phrase.
- For numbers/digits, the entire word containing the number(s) is completely deleted. If the digits are the only characters removed from a word, then words such as "3D printing" and "I-10" will become meaningless.

- **I purposefully did not use stemming** (e.g. reducing words "fishing" to the root word "fish"), as there is a bit of debate in the field regarding whether stemming can improve model performance or whether it **destroys** the sentiment of the word.

Stop words are removed using regex operations and the NLTK library's set of stopwords. An important design choice was made in this step: There is debate in the field of sentiment analysis regarding the removal of stop words, which might change the sentiment and meaning of the phrases. As an example:

**With stop-words**: Driverless cars are *not* worth the risk. I *won't* drive them.

**After pre-processing and removal of stop-words**: driverless cars worth risk drive

We can see after removal of stop words, the above sentence changes from a negative sentiment to a positive sentiment. Therefore, I have made the unique design decision to **not exclude** the stopwords "nor", "not", and "no" by subtracting these stop words from NLTK's set of stop words.

```
#Now, let's remove stopwords using NLTK
#Here, I am printing the list of stop words, so I know what stop words I should remove for the sentiment analysis problem
set(stopwords.words('english'))
#Here, I am removing the "not", "nor", and "no" stopwords from NLTK's stopwords set
stop_words = set(stopwords.words('english')) - {"nor", "not", "no"}
#Here, I am removing the stopwords from the text
stopwords_re = re.compile(r"(\s+)?\b({})\b(\s+)?".format("|".join(stop_words), re.IGNORECASE))
#Here, I am adjusting the whitespace accordingly
whitespace_re = re.compile(r"\s+")
text_col = text_col.replace(stopwords_re, " ").str.strip().str.replace(whitespace_re, " ")
```

Another novel design choice that I made is the expansion of contractions (e.g., transforming "I don't" into "I do not") in the text. There are two reasons why I made this choice: 1. When tokenizing, contractions are often split like "*I*", "*'m*", which has the potential to harm the performance of the model. 2. By expanding contractions, I do not need to manually subtract contractions like "don't" from the NLTK stop words set, since contractions might also contribute to the sentiment a sentence. The following code was taken from Yann Dubois's answer to a Stack Overflow question (available at https://stackoverflow.com/questions/43018030/replace-appostrophe-short-words-in-python/47091370#47091370) that covers the expansion of **most** contractions, which is good enough for this dataset. I modified the following code by converting the use of re.sub() into text.replace().

```
#Expanding contractions
#(code taken from https://stackoverflow.com/questions/43018030/replace-appostrophe-short-words-in-python/47091370#47091370)
text_col = text_col.replace(r"won't", "will not", regex=True)
text_col = text_col.replace(r"can\'t", "can not", regex=True)
text_col = text_col.replace(r"n\'t", " not", regex=True)
text_col = text_col.replace(r"\'re", " are", regex=True)
text_col = text_col.replace(r"\'s", " is", regex=True)
text_col = text_col.replace(r"\'d", " would", regex=True)
text_col = text_col.replace(r"\'ll", " will", regex=True)
text_col = text_col.replace(r"\'t", " not", regex=True)
text_col = text_col.replace(r"\'ve", " have", regex=True)
text_col = text_col.replace(r"\'m", " am", regex=True)
text_col
```

## Code

**Please Note: Running the code in the Jupyter notebook can result in different results than those described here, which I encountered. Consequently, the results in the .ipynb file provided along with this documentation are not the same as those described in the documentation.**

### *Generating Word2Vec embeddings*

Here is a picture of the dataset that is now pre-processed.

```
In [89]: #First, I will make sure that the dataset has been pre-processed correctly.
         senti.text = text_col
         senti[:18]
```

Out[89]:

| | sentiment | text |
|---|---|---|
| 0 | 5 | two places would invest money could printing self driving cars |
| 1 | 5 | awesome google driverless cars help blind travel often |
| 2 | 2 | google maps not keep road construction supposed trust driverless car get around |
| 3 | 2 | autonomous cars seem way overhyped given technology challenges pilotless planes seem much doable needed |
| 4 | 3 | saw google self driving car painted green blue |
| 5 | 3 | driverless cars eventually replace taxi drivers cities |
| 6 | 5 | autonomous vehicles could reduce traffic fatalities |
| 7 | 1 | driverless cars not worth risk not want highway server crashes sadmacface bluescreenofdeath |
| 8 | 3 | driverless cars legal florida california michigan |
| 9 | 3 | audi first carmaker get license nevada dmv test automated vehicles audi ces cartech |
| 10 | 3 | audi says first car manufacturer world get license nevada dmv test autonomous vehicles google not make cars ces |
| 11 | 4 | future buying one audi ready test autonomous cars public roads ces |
| 12 | 4 | audi test driving driverless car tampa today pretty cool |
| 13 | 3 | audi first automaker california test self driving car selfdrivingcars |
| 14 | 3 | audi gets first permit test self driving cars california roads |
| 15 | 3 | audi gets permit test self driving cars california |
| 16 | 3 | google audi mercedes get california first self driving car test permits cars without drivers may soon |
| 17 | 3 | audi gets first permit test driverless cars california bay area drag queens say |

Now that the data is pre-processed correctly, I can generate the Word2Vec embeddings. The first step is to **tokenize** the text. I am going to use the tokenize function provided by genism.utils to accomplish this task and append tokenized words to an array.

```
#Here, I am using gensim.util's tokenizer to tokenize the text, and I
am using a for loop to append the tokens to an array.
array = []
for i in range(0, len(senti.text)):
hello = list(tokenize(senti.text[i]))
array.append(hello)
```

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

```
[['two', 'places', 'would', 'invest', 'money', 'could', 'printing', 'self', 'driving', 'cars'], ['awesome', 'google', 'driverle
ss', 'cars', 'help', 'blind', 'travel', 'often'], ['google', 'maps', 'not', 'keep', 'road', 'construction', 'supposed', 'trus
t', 'driverless', 'car', 'get', 'around'], ['autonomous', 'cars', 'seem', 'way', 'overhyped', 'given', 'technology', 'challenge
s', 'pilotless', 'planes', 'seem', 'much', 'doable', 'needed'], ['saw', 'google', 'self', 'driving', 'car', 'painted', 'green',
'blue'], ['driverless', 'cars', 'eventually', 'replace', 'taxi', 'drivers', 'cities'], ['autonomous', 'vehicles', 'could', 'red
uce', 'traffic', 'fatalities'], ['driverless', 'cars', 'not', 'worth', 'risk', 'not', 'want', 'highway', 'server', 'crashes',
'sadmacface', 'bluescreenofdeath'], ['driverless', 'cars', 'legal', 'florida', 'california', 'michigan'], ['audi', 'first', 'ca
rmaker', 'get', 'license', 'nevada', 'dmv', 'test', 'automated', 'vehicles', 'audi', 'ces', 'cartech'], ['audi', 'says', 'firs
t', 'car', 'manufacturer', 'world', 'get', 'license', 'nevada', 'dmv', 'test', 'autonomous', 'vehicles', 'google', 'not', 'mak
e', 'cars', 'ces'], ['future', 'buying', 'one', 'audi', 'ready', 'test', 'autonomous', 'cars', 'public', 'roads', 'ces'], ['aud
i', 'test', 'driving', 'driverless', 'car', 'tampa', 'today', 'pretty', 'cool'], ['audi', 'first', 'automaker', 'california',
'test', 'self', 'driving', 'car', 'selfdrivingcars'], ['audi', 'gets', 'first', 'permit', 'test', 'self', 'driving', 'cars', 'c
alifornia', 'roads'], ['audi', 'gets', 'permit', 'test', 'self', 'driving', 'cars', 'california'], ['google', 'audi', 'mercede
s', 'get', 'california', 'first', 'self', 'driving', 'car', 'test', 'permits', 'cars', 'without', 'drivers', 'may', 'soon'],
['audi', 'gets', 'first', 'permit', 'test', 'driverless', 'cars', 'california', 'bay', 'area', 'drag', 'queens', 'say'], ['toda
y', 'state', 'regs', 'autonomous', 'cars', 'go', 'effect', 'audi', 'america', 'gets', 'permit', 'test', 'driverless', 'cars',
'public', 'roads', 'per'], ['audi', 'gets', 'first', 'permit', 'test', 'self', 'driving', 'cars', 'california', 'think', 'twic
e', 'next', 'time', 'tailgate', 'new', 'audi'], ['audi', 'gets', 'first', 'permit', 'test', 'driverless', 'cars', 'california',
'bay', 'area', 'drag', 'queens', 'say'], ['audi', 'gets', 'permit', 'test', 'self', 'driving', 'cars', 'california'], ['audi',
'becomes', 'first', 'automaker', 'given', 'permit', 'test', 'self', 'driving', 'cars'], ['audi', 'snags', 'first', 'permit', 't
est', 'self', 'driving', 'cars', 'california', 'los', 'angeles', 'cbsap', 'audi', 'first'], ['go', 'gets', 'first', 'permit',
'test', 'self', 'driving', 'cars', 'california', 'roads', 'tech', 'auto'], ['agv', 'automated', 'guided', 'vehicle', 'dock'],
```

Next, I inputted the array into the Word2Vec algorithm, set the size parameter to 100, the window parameter to 5, the workers parameter to 8, and the min_count parameter to 2. In this step, we are training the Word2Vec algorithm on **our** dataset (i.e. the dataset provided by Crowdflower). I am also saving the word2vec embeddings in a txt file for future use.

```
#Here, I am running the Word2Vec algorithm. The value of the min_count parameter was varied for
experimentation.
model = Word2Vec(array, size=100, window=5, workers=8, min_count=2)
print('The total training sentences: %d' % len(senti.text))
words = list(model.wv.vocab)
PCA_AR = model[model.wv.vocab]
print('The vocabulary size: %d' % len(words))
# save model in ASCII (word2vec) format
filename = 'embedding_word2vec.txt'
model.wv.save_word2vec_format(filename, binary=False)
```

**I experimented** with the value of the min_count parameter, which influences the vocabulary size and the cosine similarity scores of the model. Using *min_count = 1* was leading to a vocabulary size of 8813 and high similarity between words that are dissimilar:

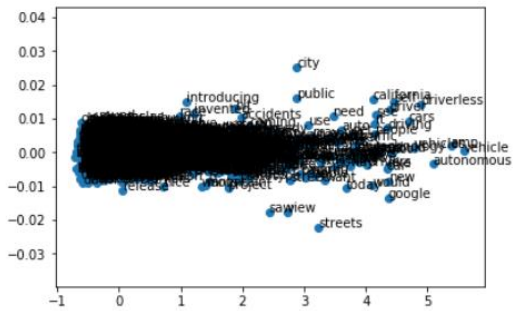
```
model.wv.similarity('wine', 'car')

0.7483841650467838

model.wv.similarity('utopia', 'departments')

0.9609718225837007
```

Using *min_count = 3* was leading to a vocabulary size of 2699 and lower (and more accurate) cosine similarity scores between words that were dissimilar. However, due to the low vocabulary size occurring when increasing the value of the *min_count* parameter, I used *min_count = 2*, which results in a vocabulary size of 3882.

When using *min_count=2*, we can see in the below image that we are receiving **low** cosine similarity scores when asking the model to provide words that are not similar to markets, which is better compared to using *min_count = 1*.

```
In [33]: model.wv.most_similar(negative=['markets'])

Out[33]: [('chart', 0.44268137216567993),
          ('predictions', 0.4416291415691376),
          ('diver', 0.38253650069236755),
          ('distraction', 0.013044580817222595),
          ('predator', 0.0037798210978507996),
          ('showing', -0.1869221180677414),
          ('backtothefuturefan', -0.2363729178905487),
          ('finest', -0.26595252752304077),
          ('autonomus', -0.27073991298675537),
          ('opt', -0.3551793098449707)]
```

I also generated a PCA plot of the model.wv.vocab




**Code for generating PCA Plot**

```
pca = PCA(n_components=10)
result = pca.fit_transform(X)
# create a scatter plot of the projection
plt.scatter(result[:, 0], result[:, 9])
words = list(model.wv.vocab)
for i, word in enumerate(words):
plt.annotate(word, xy=(result[i, 0], result[i, 9]))
plt.show()
```

I can see that the words car, cars, and people are grouped together; it also seems words such as introducing and inventing are grouped together.

### *Inputting the Word2Vec Embeddings into the Convolutional Neural Network*

Now, let's use the Word2Vec embeddings **trained on the CrowdFlower dataset**. There are a few steps we need to take before inputting the embeddings into the Convolutional Neural Network:

- Split the dataset into train and test datasets. The goal is to predict the sentiment value from the text:

```
#Now the training set and test set need to be defined.
#The goal is to predict the sentiment value from the text.
x_train, x_test, y_train, y_test = train_test_split(np.array(senti.text),
                                   np.array(senti.sentiment), test_size=0.2)
```

I tried printing values of x_train, y_train, x_test, and y_test to make sure the x_train/x_test contains text, and y_train/y_test contains the sentiment values.

```
In [43]: print(x_train)
['nissan says release driverless car good thing humans terrible driving'
 'gm announces new automated connected vehicle technologies cadillac models adillac model year'
 'driverless cars could reshape cities' ...
 'wonder accidents self driving car muni'
 'googlex folks talking driverless cars kvsummit perseverance'
 'please apple apple interested potential ways evolve car including autonomous driving']

In [44]: print(y_train)
['4' '3' '3' ... '3' '3' '3']

In [45]: print(x_test)
['saw google car going east bound broadway today'
 'google stock basically mutual fund holdings advertising robotics cean energy self driving cars'
 'sounds like need one self driving cars working haha' ...
 'california greenlights testing driverless cars'
 'loved drive self driving car mountain view freeway'
 'gave google car shocker drove iv']

In [46]: print(y_test)
['4' '3' '3' ... '3' '5' '3']
```

- I created and fit the Keras Tokenizer **on the training dataset**

```
# Create and Fit a Tokenizer
tokenizer = Tokenizer()
tokenizer.fit_on_texts(x_train)
```

Using the *print(tokenizer.word_counts)* displays the amount of times a word appears in the text and *print(tokenizer.word_index)* displays the word index.

```
In [48]: print(tokenizer.word_counts)
OrderedDict([('nissan', 46), ('says', 98), ('release', 10), ('driverless', 1529), ('car', 2870), ('good', 59), ('thing', 59), ('humans', 47), ('terrible', 8), ('driving', 3416), ('gm', 27), ('announces', 17), ('new', 182), ('automated', 41), ('connected', 29), ('vehicle', 157), ('technologies', 9), ('cadillac', 12), ('models', 7), ('adillac', 2), ('model', 16), ('year', 35), ('cars', 2630), ('could', 180), ('reshape', 24), ('cities', 32), ('driver', 165), ('vs', 16), ('autonomous', 299), ('wins', 2), ('twitterpoll', 1), ('article', 19), ('study', 20), ('self', 3248), ('would', 229), ('eliminate', 9), ('majority', 8), ('traffic', 78), ('deaths', 17), ('congestion', 7), ('washington', 11), ('ap', 25), ('ways', 10), ('not', 600), ('wait', 104), ('tesla', 62), ('great', 71), ('presentation', 9), ('afternoon', 4), ('sebastian', 6), ('thrun', 7), ('really', 88), ('fascinating', 7), ('stuff', 17), ('mt', 29), ('n', 5), ('seat', 18), ('googleme', 1), ('bigdocc', 1), ('time', 103), ('run', 23), ('google', 2280), ('make', 117), ('faces', 4), ('may', 56), ('honor', 2), ('roll', 11), ('child', 11), ('creat', 18), ('efficient', 7), ('less', 97), ('hey', 20), ('want', 141), ('quotable', 1), ('imagine', 27), ('negotiating', 1), ('paying', 2), ('fractional', 1), ('bitcoin', 8), ('neighboring', 1), ('exchange', 1), ('priority', 4), ('looking', 44), ('reward', 30), ('tonight', 12), ('boredom', 1), ('notoyota', 1), ('audi', 66), ('throwing', 3), ('hats', 2), ('ring', 2), ('voo', 61), ('lexus', 39), ('suv', 12), ('drove', 33), ('saw', 225), ('ca', 34), ('teslas', 6), ('six', 6), ('years', 92), ('away', 32), ('demo', 14), ('glitches', 2), ('hate', 13), ('wwdc', 9), ('go', 100), ('letsgetit', 1), ('race', 19), ('rt', 177), ('promises', 3), ('next', 101), ('live', 15), ('collaborate', 3), ('develop', 17), ('vehicles', 126), ('little', 20), ('hope', 41), ('house', 16), ('name', 4), ('program', 10), ('project', 37), ('jesus', 2), ('take', 101), ('wheel', 127), ('morality', 5), ('fwf', 1), ('futureworldfriday', 1), ('insurance', 54), ('implications', 8), ('raise', 2), ('gt', 23), ('indeed', 5), ('whoa', 11), ('complete', 13), ('miles', 46), ('without', 47), ('accident', 41), ('deemed', 5), ('ready', 79), ('commuting', 9), ('robot', 39), ('scifi', 3), ('future', 265), ('w', 35), ('pic', 36), ('everyone', 22), ('outlaw', 1), ('motorcycle

In [48]: print(tokenizer.word_index)
{'driving': 1, 'self': 2, 'car': 3, 'cars': 4, 'google': 5, 'driverless': 6, 'not': 7, 'autonomous': 8, 'future': 9, 'would': 10, 'saw': 11, 'no': 12, 'like': 13, 'one': 14, 'drive': 15, 'get': 16, 'new': 17, 'could': 18, 'rt': 19, 'via': 20, 'california': 21, 'driver': 22, 'uber': 23, 'vehicle': 24, 'see': 25, 'first': 26, 'need': 27, 'think': 28, 'want': 29, 'road': 30, 'wheel': 31, 'vehicles': 32, 'today': 33, 'amp': 34, 'technology': 35, 'people': 36, 'make': 37, 'going': 38, 'tech': 39, 'cool': 40, 'way': 41, 'drivers': 42, 'wait': 43, 'time': 44, 'next': 45, 'take': 46, 'go': 47, 'steering': 48, 'says': 49, 'less': 50, 'years': 51, 'test': 52, 'soon': 53, 'really': 54, 'us': 55, 'look': 56, 'city': 57, 'ride': 58, 'ready': 59, 'roads': 60, 'streets': 61, 'world': 62, 'traffic': 63, 'work': 64, 'day': 65, 'still': 66, 'around': 67, 'legal': 68, 'auto': 69, 'great': 70, 'testing': 71, 'come': 72, 'right': 73, 'coming': 74, 'know': 75, 'view': 76, 'audi': 77, 'street': 78, 'much': 79, 'already': 80, 'tesla': 81, 'video': 82, 'public': 83, 'good': 84, 'thing': 85, 'apple': 86, 'got': 87, 'spotted': 88, 'may': 89, 'use': 90, 'passed': 91, 'insurance': 92, 'gets': 93, 'prototype': 94, 'transportation': 95, 'real': 96, 'interesting': 97, 'awesome': 98, 'parking': 99, 'toyota': 100, 'idea': 101, 'seen': 102, 'love': 103, 'behind': 104, 'enough': 105, 'also': 106, 'buy': 107, 'humans': 108, 'without': 109, 'home': 110, 'nissan': 111, 'miles': 112, 'build': 113, 'human': 114, 'looking': 115, 'better': 116, 'even': 117, 'pretty': 118, 'glass': 119, 'two': 120, 'looks': 121, 'working': 122, 'unveils': 123, 'ford': 124, 'taxi': 125, 'automated': 126, 'hope': 127, 'accident': 128, 'stop': 129, 'hit': 130, 'big': 131, 'watch': 132, 'oh': 133, 'makes': 134, 'â': 135, 'lexus': 136, 'robot': 137, 'wonder': 138, 'long': 139, 'drones': 140, 'back': 141, 'maps': 142, 'trust': 143, 'become': 144, 'might': 145, 'last': 146, 'mountain': 147, 'sure': 148, 'invented': 149, 'yes': 150, 'project': 151, 'hands': 152, 'making': 153, 'let': 154, 'maybe': 155, 'electric': 156, 'pic': 157, 'innovation': 158, 'safer': 159, 'taxis': 160, 'drives': 161, 'many': 162, 'life': 163, 'accidents': 164, 'robots': 165, 'ever': 166, 'yea': 167, 'w': 168, 'lol': 169, 'problem': 170, 'amazing': 171, 'state': 172, 'getting': 173, 'ca': 174, 'never': 175, 'chang
```

The length of the word_index is 7692, as shown below.

```
In [50]: print(len(tokenizer.word_index))
7692
```

- Next, I wrote code to encode the sequences of the **training and test datasets**.

```
# integer encode sequences
encoded_train = tokenizer.texts_to_sequences(x_train)
encoded_test = tokenizer.texts_to_sequences(x_test)
```

```
In [52]: encoded_train
Out[52]: [[111, 49, 646, 6, 3, 84, 85, 108, 787, 1],
          [231, 387, 17, 126, 213, 24, 708, 533, 885, 2296, 413, 167],
          [6, 4, 18, 267, 185],
          [22, 414, 8, 3, 2297, 3374, 342],
          [321, 2, 1, 4, 10, 709, 788, 63, 388, 886, 585, 257, 647],
          [7, 43, 2, 1, 81],
          [70, 710, 1449, 1033, 887, 6, 4, 54, 888, 389],
          [214, 6, 4, 1193, 2, 1, 365, 3375, 3376],
          [44, 282, 5, 3, 37, 1450],
          [89, 2298, 586, 587, 366, 889, 22, 50, 3],
          [322, 5, 29, 2, 1, 3],
          [3377, 232, 2, 1, 3, 3378, 63, 2299, 3379, 789, 3380, 4, 3381, 1451],
          [7, 115, 201, 1, 534, 3382, 2, 1, 3],
          [3383, 77, 1774, 2300, 2301, 2, 1, 4, 82],
          [5, 2, 1, 136, 535, 182],
          [11, 5, 3, 174],
          [2, 1, 1034, 1035, 51, 186],
          [466, 2302, 495, 711, 2, 1, 3, 47],
          [3384, 343, 5, 19, 111, 1775, 2, 1, 4, 45, 435],
```

Now, I will define the max sentence length. In his post "How to Develop a Word Embedding Model for Predicting Movie Review Sentiment", Jason Brownlee notes that padding all reviews to the length of the longest review in the training dataset is helpful. I have taken the following line of code from his post.

```
# max sentence length
max_length = max([len(s.split()) for s in x_train])
```

Next, I wrote code to pad the sequences of the **training and test datasets**.

```
# pad sequences with 0 values
Xtrain = pad_sequences(encoded_train, maxlen=max_length, padding='post')
Xtest = pad_sequences(encoded_test, maxlen=max_length, padding='post')
```

- Finally, I will use the to_categorical function provided by keras.utils to one-hot encode the labels (y_train and y_test).

```
#one-hot-encode the training labels
from keras.utils import to_categorical
y_encoded_train = to_categorical(y_train)
print(y_encoded_train)
```

```
#one-hot-encode the training labels
from keras.utils import to_categorical
y_encoded_test = to_categorical(y_test)
print(y_encoded_test)
```

```
In [55]: from keras.utils import to_categorical
         y_encoded_train = to_categorical(y_train)
         print(y_encoded_train)

[[0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 ...
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]]
```

```
In [58]: from keras.utils import to_categorical
         y_encoded_test = to_categorical(y_test)
         print(y_encoded_test)

[[0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 ...
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0.]]
```

Now, I can use the Word2Vec embeddings as input into the convolutional neural network (CNN) built using the Keras API.

I will load the embeddings for computing an index mapping words to known embeddings. Francois Chollet provides code for loading the embeddings in a Keras tutorial, titled "Using pre-trained word embeddings in a Keras model". I used the same code he provided and I modified the variable names. However, the code provided in the Keras tutorial is for GloVe embeddings and Jason Brownlee suggests in his post "How to Develop a Word Embedding Model for Predicting Movie Review Sentiment" that for embeddings saved in the word2vec format, the first line needs to be skipped, which I have incorporated.

```
#Code taken from and modified from "Using pre-trained word embeddings in a Keras model"
(https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)
file = open('embedding_word2vec.txt','r')
lines = file.readlines()[1:] #Added the [1:] to skip the first line for Word2Vec files
file.close()
embeddings_index = dict()
for line in lines:
parts = line.split()
word = parts[0]
embeddings_index[word] = asarray(parts[1:], dtype='float32')
```

Next, I will create a **weight matrix** from the loaded embedding to input into the Convolutional Neural Network using the same code provided by Francois Chollet's Keras tutorial. I will be using the tokenizer.word_index and the embeddings_index generated in the previous code.

```
embedding_dimension = 100
word_index = tokenizer.word_index
#Code taken from and modified from "Using pre-trained word embeddings in a Keras model"
(https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)
# create a weight matrix for the Embedding layer
embedding_matrix = np.zeros((len(word_index) + 1, embedding_dimension))
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.
embedding_matrix[i] = embedding_vector
```

The shape of the embedding_matrix is (7703, 100), which is then inputted into an embedding layer for the CNN. As noted in Francois Chollet's Keras tutorial, *trainable=false* to keep the weights from being updated during the training process.

```
#Loading the embedding matrix into the Embeddings layer
embedding_layer = Embedding(embedding_matrix.shape[0], 100, weights=[embedding_matrix],
input_length=max_length, trainable=False)
```

Next, I wrote code to create the CNN, which incorporates 3 Conv1d layers, 3 Max Pooling layers, 1 Flatten layer, 1 Dropout layer, and 2 Dense layers (one of which has L2 regularization using the kernel_regularizer). I am using Conv1d, because sentences are linear lists of words, whereas conv2d is used for images which have height and width.

```
model2 = Sequential()
model2.add(embedding_layer)
model2.add(Conv1D(150, 3, activation='relu'))
model2.add(MaxPooling1D(pool_size=2))
model2.add(Conv1D(150, 3, activation='relu'))
model2.add(MaxPooling1D(pool_size=2))
model2.add(Conv1D(150, 3, activation='relu'))
model2.add(MaxPooling1D(pool_size=1))
model2.add(Flatten())
model2.add(Dropout(0.8)) #I added this
model2.add(Dense(150, kernel_regularizer=regularizers.l2(0.1), activation='relu'))
model2.add(Dense(6, activation='softmax'))
print(model2.summary())
```

The model summary:

```
WARNING:tensorflow:From c:\users\turia\anaconda3\envs\tensorflow\lib\site-packages\tensorflow\python\util\deprecation.py:497: c
alling conv1d (from tensorflow.python.ops.nn_ops) with data_format=NHWC is deprecated and will be removed in a future version.
Instructions for updating:
`NHWC` for data_format is deprecated, use `NWC` instead
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| embedding_1 (Embedding) | (None, 26, 100) | 770300 |
| conv1d_1 (Conv1D) | (None, 24, 150) | 45150 |
| max_pooling1d_1 (MaxPooling1 | (None, 12, 150) | 0 |
| conv1d_2 (Conv1D) | (None, 10, 150) | 67650 |
| max_pooling1d_2 (MaxPooling1 | (None, 5, 150) | 0 |
| conv1d_3 (Conv1D) | (None, 3, 150) | 67650 |
| max_pooling1d_3 (MaxPooling1 | (None, 3, 150) | 0 |
| flatten_1 (Flatten) | (None, 450) | 0 |
| dense_1 (Dense) | (None, 150) | 67650 |
| dense_2 (Dense) | (None, 6) | 906 |

```
Total params: 1,019,306
Trainable params: 249,006
Non-trainable params: 770,300
_________________________________________________________________
None
```

Next, I will take 2,000 values out of the training dataset and the training labels to create a validation dataset. Then, I will fit the model and evaluate it on the test dataset.

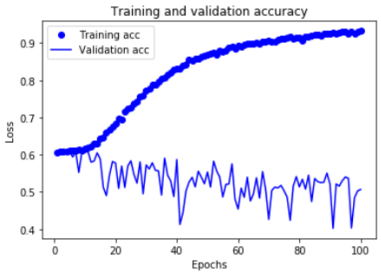
```
#Divide the training dataset into training and validation datasets
x_val_data = Xtrain[:2000]
partial_x_train = Xtrain[2000:]
y_val_data = y_encoded_train[:2000]
partial_y_train = y_encoded_train[2000:]
# Compile
model2.compile(loss='categorical_crossentropy', optimizer='Adagrad',
metrics=['accuracy'])
# Fit #epochs=100
history = model2.fit(partial_x_train, partial_y_train, epochs=100,
validation_data=(x_val_data, y_val_data), verbose=2)
# Evaluate
results = model2.evaluate(Xtest, y_encoded_test, verbose=2)
print(results)
print('Test Accuracy: %f' % (results[1]*100))
```

I performed extensive experimentation using epochs as 50, and observed the following: Including the Dropout layer = 0.8 and Kernel Regularizer = 0.1 was leading to consistent training accuracy after the 2nd epoch and approximately 58% test accuracy, whereas excluding the Dropout layer and the kernel regularizer was leading to changing training accuracy and lower test accuracy (approximately 54%).

Using 100 epochs:

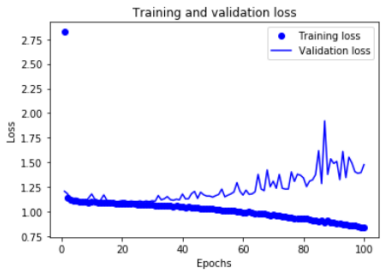
## Without Dropout and the Kernel Regularizer

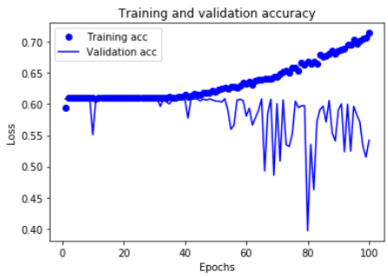
```
Epoch 83/100
 - 1s - loss: 0.2344 - acc: 0.9164 - val_loss: 3.4551 - val_acc: 0.5450
Epoch 84/100
 - 1s - loss: 0.2216 - acc: 0.9190 - val_loss: 3.4175 - val_acc: 0.4740
Epoch 85/100
 - 1s - loss: 0.2119 - acc: 0.9215 - val_loss: 3.5122 - val_acc: 0.5360
Epoch 86/100
 - 1s - loss: 0.2137 - acc: 0.9229 - val_loss: 3.4763 - val_acc: 0.5275
Epoch 87/100
 - 1s - loss: 0.2128 - acc: 0.9212 - val_loss: 3.4719 - val_acc: 0.5250
Epoch 88/100
 - 1s - loss: 0.2042 - acc: 0.9260 - val_loss: 3.5006 - val_acc: 0.5260
Epoch 89/100
 - 1s - loss: 0.2055 - acc: 0.9257 - val_loss: 3.6755 - val_acc: 0.5505
Epoch 90/100
 - 1s - loss: 0.2093 - acc: 0.9260 - val_loss: 3.5427 - val_acc: 0.5205
Epoch 91/100
 - 1s - loss: 0.2009 - acc: 0.9271 - val_loss: 3.8586 - val_acc: 0.4020
Epoch 92/100
 - 1s - loss: 0.2136 - acc: 0.9263 - val_loss: 3.5713 - val_acc: 0.5215
Epoch 93/100
 - 1s - loss: 0.2028 - acc: 0.9268 - val_loss: 3.5598 - val_acc: 0.5155
Epoch 94/100
 - 1s - loss: 0.1966 - acc: 0.9305 - val_loss: 3.6659 - val_acc: 0.5300
Epoch 95/100
 - 1s - loss: 0.1996 - acc: 0.9302 - val_loss: 3.6831 - val_acc: 0.5400
Epoch 96/100
 - 1s - loss: 0.1948 - acc: 0.9266 - val_loss: 3.7984 - val_acc: 0.5365

Epoch 97/100
 - 1s - loss: 0.1951 - acc: 0.9302 - val_loss: 3.9126 - val_acc: 0.4030
Epoch 98/100
 - 1s - loss: 0.2012 - acc: 0.9266 - val_loss: 3.6485 - val_acc: 0.4845
Epoch 99/100
 - 1s - loss: 0.1891 - acc: 0.9305 - val_loss: 3.6697 - val_acc: 0.5025
Epoch 100/100
 - 1s - loss: 0.1874 - acc: 0.9328 - val_loss: 3.6817 - val_acc: 0.5065
[3.2861913707807306, 0.5226781859597]
Test Accuracy: 52.267819
```


## With Dropout and the Kernel Regularizer

```
Epoch 83/100
 - 1s - loss: 0.9166 - acc: 0.6654 - val_loss: 1.3242 - val_acc: 0.5720
Epoch 84/100
 - 1s - loss: 0.9119 - acc: 0.6798 - val_loss: 1.3778 - val_acc: 0.5910
Epoch 85/100
 - 1s - loss: 0.9037 - acc: 0.6764 - val_loss: 1.6205 - val_acc: 0.5970
Epoch 86/100
 - 1s - loss: 0.9107 - acc: 0.6775 - val_loss: 1.2860 - val_acc: 0.5715
Epoch 87/100
 - 1s - loss: 0.8967 - acc: 0.6809 - val_loss: 1.9224 - val_acc: 0.6060
Epoch 88/100
 - 1s - loss: 0.9075 - acc: 0.6854 - val_loss: 1.3789 - val_acc: 0.5540
Epoch 89/100
 - 1s - loss: 0.8958 - acc: 0.6815 - val_loss: 1.5366 - val_acc: 0.5410
Epoch 90/100
 - 1s - loss: 0.8929 - acc: 0.6868 - val_loss: 1.4905 - val_acc: 0.5905
Epoch 91/100
 - 1s - loss: 0.8863 - acc: 0.6885 - val_loss: 1.5111 - val_acc: 0.6005
Epoch 92/100
 - 1s - loss: 0.8812 - acc: 0.6894 - val_loss: 1.3258 - val_acc: 0.5235
Epoch 93/100
 - 1s - loss: 0.8736 - acc: 0.6953 - val_loss: 1.6120 - val_acc: 0.6000
Epoch 94/100
 - 1s - loss: 0.8732 - acc: 0.6964 - val_loss: 1.3467 - val_acc: 0.5245
Epoch 95/100
 - 1s - loss: 0.8655 - acc: 0.7037 - val_loss: 1.5539 - val_acc: 0.5965
Epoch 96/100
 - 1s - loss: 0.8631 - acc: 0.6978 - val_loss: 1.4950 - val_acc: 0.5830

Epoch 97/100
 - 1s - loss: 0.8579 - acc: 0.7023 - val_loss: 1.4056 - val_acc: 0.5715
Epoch 98/100
 - 1s - loss: 0.8547 - acc: 0.7054 - val_loss: 1.3901 - val_acc: 0.5320
Epoch 99/100
 - 1s - loss: 0.8467 - acc: 0.7071 - val_loss: 1.3978 - val_acc: 0.5150
Epoch 100/100
 - 1s - loss: 0.8428 - acc: 0.7147 - val_loss: 1.4786 - val_acc: 0.5420
[1.3294213887510582, 0.5802735782853985]
Test Accuracy: 58.027358
```




```
#Plotting training and validation loss
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

I also experimented with the Adam, RMSProp, and Adagrad optimizers. Using Adam was leading to approximately 54% test accuracy compared to the 58% and sometimes higher accuracy achieved using Adagrad. In order to make sure model isn't overfitting (performing great on training and horrible on test) or underfitting (performing poorly on the training), let's make sure the accuracy is good for **test** and **train** datasets. Overfitting is defined as over-optimizing on the training data, and learning representations that are specific to the training data and do not generalize to data outside of the training set.

Next, I wrote code to serialize the model to JSON, and serialize the weights to HDF5:

```
# serialize model to JSON
model_json = model2.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model2.save_weights("model.h5")
print("Saved model to disk")
```

Now, that we have trained our model, I will use it to generate predictions.

```
# load json and create model
json_file = open('model2.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
#Defining labels to display the predicted label in a more human-friendly way
labels = [[''], ['Very negative'], ['Slightly negative'], ['Neutral'], ['Slightly positive'], ['Very positive']]
```

The string inputted in the exp_tests array was altered.

The model predicted a negative sentiment as slightly negative, which is good!

**Experimental tweet text:** 'Unnecessary waste of time. Creepy and freaky. Not worth it.'
**Predicted:** Slightly Negative

The model predicted a negative sentiment as neutral, which is incorrect. More data might be needed to improve the model's accuracy.

**Experimental tweet text:** 'Whoever did this completely wasted their time. #sadmacface'
**Predicted:** Neutral

**Experimental tweet text:** 'GENIUS! AWESOME! Who thought of this? They are amazing people.'
**Predicted:** Very Positive

**Experimental tweet text:** 'Self-driving cars are murderers that kill people. I HATE them, they suck, and I am NOT driving them.'
**Predicted:** Very Negative

The model outputs relatively accurate sentiments for the tweets given as input.

*Inputting pre-trained GloVe Embeddings into the Convolutional Neural Network*

We can use almost the same code (code for generating the weight matrix, the CNN layers, model fitting, generating the plots, and the experimental predictions) to input pre-trained GloVe embeddings into the CNN. As mentioned above, in the field of sentiment analysis, the GloVe technology is most commonly used by training models on pre-trained word vectors on extremely large corpuses. For this project, I am using pre-trained word vectors on an extremely large Twitter dataset that is **not** the dataset used to generate Word2Vec embeddings in this project. The extremely large Twitter dataset we will use is titled glove.twitter.27B.zip and is available on the https://nlp.stanford.edu/projects/glove/ webpage. I will be using the *glove.twitter.27B.100d.txt* file, which contains vectors with a dimension of 100 and is 997, 725

KB. The **only** modification that needs to be made is in the code for loading the embeddings, as we should **not skip** the first line of the GloVe embeddings text file:

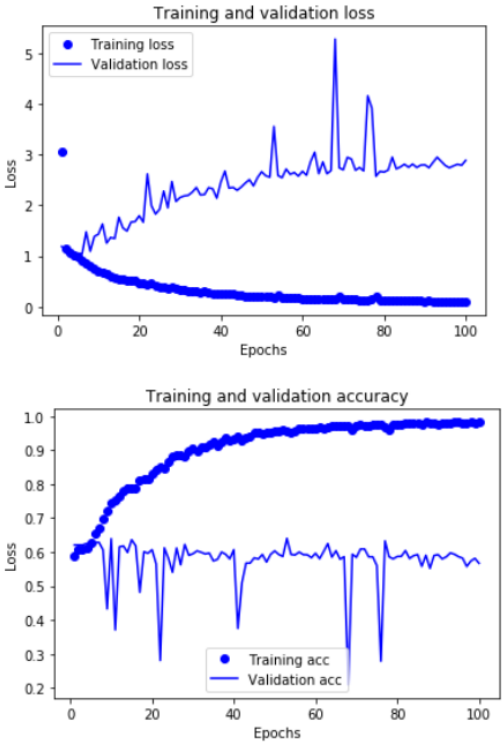
```
#Code taken from and modified from "Using pre-trained word embeddings in a Keras model" (https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)
file = open('./glove.twitter.27B.100d.txt','r', encoding="utf8")
#This GloVe dataset is made available under the Public Domain Dedication and License v1.0 whose full text
#can be found at: http://www.opendatacommons.org/licenses/pddl/1.0/
lines = file.readlines()
file.close()
embeddings_index = dict()
for line in lines:
parts = line.split()
word = parts[0]
embeddings_index[word] = asarray(parts[1:], dtype='float32')
```

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 26, 100)           762300
_________________________________________________________________
conv1d_16 (Conv1D)           (None, 24, 150)           45150
_________________________________________________________________
max_pooling1d_16 (MaxPooling (None, 12, 150)           0
_________________________________________________________________
conv1d_17 (Conv1D)           (None, 10, 150)           67650
_________________________________________________________________
max_pooling1d_17 (MaxPooling (None, 5, 150)            0
_________________________________________________________________
conv1d_18 (Conv1D)           (None, 3, 150)            67650
_________________________________________________________________
max_pooling1d_18 (MaxPooling (None, 3, 150)            0
_________________________________________________________________
flatten_6 (Flatten)          (None, 450)               0
_________________________________________________________________
dense_11 (Dense)             (None, 150)               67650
_________________________________________________________________
dense_12 (Dense)             (None, 6)                 906
=================================================================
Total params: 1,011,306
Trainable params: 249,006
Non-trainable params: 762,300
_________________________________________________________________
None
```

```
Train on 3554 samples, validate on 2000 samples
Epoch 1/100
 - 2s - loss: 2.8244 - acc: 0.5937 - val_loss: 1.2083 - val_acc: 0.6085
Epoch 2/100
 - 1s - loss: 1.1413 - acc: 0.6097 - val_loss: 1.1836 - val_acc: 0.6085
Epoch 3/100
 - 1s - loss: 1.1188 - acc: 0.6097 - val_loss: 1.1297 - val_acc: 0.6085
Epoch 4/100
 - 1s - loss: 1.1099 - acc: 0.6097 - val_loss: 1.1363 - val_acc: 0.6085
Epoch 5/100
 - 1s - loss: 1.1102 - acc: 0.6097 - val_loss: 1.1378 - val_acc: 0.6085
Epoch 6/100
 - 1s - loss: 1.1048 - acc: 0.6097 - val_loss: 1.1099 - val_acc: 0.6085
Epoch 7/100
 - 1s - loss: 1.1019 - acc: 0.6097 - val_loss: 1.1172 - val_acc: 0.6085
Epoch 8/100
 - 1s - loss: 1.1000 - acc: 0.6097 - val_loss: 1.1046 - val_acc: 0.6085
Epoch 9/100
 - 1s - loss: 1.0947 - acc: 0.6097 - val_loss: 1.1409 - val_acc: 0.6085
Epoch 10/100
 - 1s - loss: 1.0991 - acc: 0.6097 - val_loss: 1.1810 - val_acc: 0.5510
Epoch 11/100
 - 1s - loss: 1.0983 - acc: 0.6086 - val_loss: 1.1302 - val_acc: 0.6085
Epoch 12/100
 - 1s - loss: 1.0934 - acc: 0.6097 - val_loss: 1.1010 - val_acc: 0.6085
Epoch 13/100
 - 1s - loss: 1.0915 - acc: 0.6097 - val_loss: 1.1212 - val_acc: 0.6085
Epoch 14/100
 - 1s - loss: 1.0912 - acc: 0.6097 - val_loss: 1.1718 - val_acc: 0.6085
Epoch 15/100
 - 1s - loss: 1.0914 - acc: 0.6097 - val_loss: 1.1188 - val_acc: 0.6085
Epoch 16/100
 - 1s - loss: 1.0905 - acc: 0.6097 - val_loss: 1.1078 - val_acc: 0.6085
Epoch 17/100
 - 1s - loss: 1.0877 - acc: 0.6097 - val_loss: 1.0998 - val_acc: 0.6085
Epoch 18/100
 - 1s - loss: 1.0860 - acc: 0.6097 - val_loss: 1.1001 - val_acc: 0.6085
Epoch 19/100
 - 1s - loss: 1.0840 - acc: 0.6097 - val_loss: 1.1065 - val_acc: 0.6080
Epoch 20/100
 - 1s - loss: 1.0837 - acc: 0.6097 - val_loss: 1.1138 - val_acc: 0.6085
```

.....

```
Epoch 83/100
 - 1s - loss: 0.1242 - acc: 0.9795 - val_loss: 2.7155 - val_acc: 0.5810
Epoch 84/100
 - 1s - loss: 0.1170 - acc: 0.9806 - val_loss: 2.7539 - val_acc: 0.5895
Epoch 85/100
 - 1s - loss: 0.1246 - acc: 0.9781 - val_loss: 2.8073 - val_acc: 0.5925
Epoch 86/100
 - 1s - loss: 0.1170 - acc: 0.9772 - val_loss: 2.7469 - val_acc: 0.5580
Epoch 87/100
 - 1s - loss: 0.1123 - acc: 0.9820 - val_loss: 2.8107 - val_acc: 0.5905
Epoch 88/100
 - 1s - loss: 0.1148 - acc: 0.9800 - val_loss: 2.7433 - val_acc: 0.5520
Epoch 89/100
 - 1s - loss: 0.1157 - acc: 0.9797 - val_loss: 2.7899 - val_acc: 0.5900
Epoch 90/100
 - 1s - loss: 0.1098 - acc: 0.9772 - val_loss: 2.7956 - val_acc: 0.5925
Epoch 91/100
 - 1s - loss: 0.1138 - acc: 0.9809 - val_loss: 2.7391 - val_acc: 0.5800
Epoch 92/100
 - 1s - loss: 0.1046 - acc: 0.9809 - val_loss: 2.8433 - val_acc: 0.5860
Epoch 93/100
 - 1s - loss: 0.1092 - acc: 0.9795 - val_loss: 2.9459 - val_acc: 0.5975
Epoch 94/100
 - 1s - loss: 0.1050 - acc: 0.9828 - val_loss: 2.8629 - val_acc: 0.5935
Epoch 95/100
 - 1s - loss: 0.1065 - acc: 0.9820 - val_loss: 2.7827 - val_acc: 0.5865
Epoch 96/100
 - 1s - loss: 0.1055 - acc: 0.9811 - val_loss: 2.7373 - val_acc: 0.5820

Epoch 97/100
 - 1s - loss: 0.1046 - acc: 0.9817 - val_loss: 2.7743 - val_acc: 0.5575
Epoch 98/100
 - 1s - loss: 0.1002 - acc: 0.9823 - val_loss: 2.8067 - val_acc: 0.5735
Epoch 99/100
 - 1s - loss: 0.1006 - acc: 0.9817 - val_loss: 2.7863 - val_acc: 0.5815
Epoch 100/100
 - 1s - loss: 0.1005 - acc: 0.9828 - val_loss: 2.8821 - val_acc: 0.5670
[2.8469228559127377, 0.5593952484015855]
Test Accuracy: 55.939525
```



**Interpretation**: It appears that using the pre-trained GloVe embeddings on another dataset results in lower test accuracy (55% accuracy) compared to using the pre-trained Word2Vec embeddings on the Crowdflower dataset (58% accuracy). However, it is challenging to make to concrete conclusions on the accuracy, as the accuracy values change in relation to different runs. It appears that the gap between training and validation accuracy (defined as overfitting) is relatively larger when using the GloVe embeddings versus when using the Word2Vec embeddings.

```
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
#Defining labels to display the predicted label in a more human-friendly way
labels = [[''], ['Very negative'], ['Slightly negative'], ['Neutral'], ['Slightly positive'], ['Very positive']]
#Giving the model an experimental tweet.
exp_tests = ['Unnecessary waste of time. Creepy and freaky. Not worth it.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')
pred = model3.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

In [74]:
```
# load json and create model
json_file = open('model.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")

labels = [[''], ['Very negative'], ['Slightly negative'], ['Neutral'], ['Slightly positive'], ['Very positive']]

exp_tests = ['Unnecessary waste of time. Creepy and freaky. Not worth it.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model3.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
Loaded model from disk
[5]
['Very positive']
```

```
In [75]: exp_tests = ['Whoever did this completely wasted their time. #sadmacface']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model3.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
[4]
['Slightly positive']
```

```
In [76]: exp_tests = ['GENIUS! AWESOME! Who thought of this? They are amazing people.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model3.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
[4]
['Slightly positive']
```

```
In [77]: exp_tests = ['Self-driving cars are murderers that kill people. I HATE them, they suck, and I am NOT driving them.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model3.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
[2]
['Slightly negative']
```

It appears the model using the GloVe embeddings predicts negative sentiments as positive. Therefore, it seems that using Word2Vec models generated using the Crowdflower dataset, as opposed to GloVe embeddings generated on another large Twitter corpus, leads to better accuracy and more accurate predictions. **However, this is not true in all cases.** There are times when I found the Word2Vec model predicted these experimental tweets as Neutral, which varied with each run, so these predictions are **highly volatile** and solid interpretations cannot based on them. For example, see below (using the same model configuration used for the Word2Vec model):

```
In [67]: # load json and create model
json_file = open('model_new.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model_new.h5")
print("Loaded model from disk")
#Defining labels to display the predicted label in a more human-friendly way
labels = [[''], ['Very negative'], ['Slightly negative'], ['Neutral'], ['Slightly positive'], ['Very positive']]
#Giving the model an experimental tweet.
exp_tests = ['Unnecessary waste of time. Creepy and freaky. Not worth it.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')
pred = model2.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
Loaded model from disk
[3]
['Neutral']
```

```
In [ ]: #Giving the model an experimental tweet.
exp_tests = ['Whoever did this completely wasted their time. #sadmacface']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model2.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
In [68]: #Giving the model an experimental tweet.
exp_tests = ['GENIUS! AWESOME! Who thought of this? They are amazing people.']
encoded_exp = tokenizer.texts_to_sequences(exp_tests)
Xexp = pad_sequences(encoded_exp, maxlen=max_length, padding='post')

pred = model2.predict_classes(Xexp)
print(pred)
print(labels[pred[0]])
```

```
[3]
['Neutral']
```

## Lessons Learned:

Using word embeddings generated on the dataset the model has been trained on leads to better accuracy and more precise predictions rather than using word embeddings generated on datasets the model is not trained on. Certain factors may have influenced the accuracies obtained from the model, including differences in GloVe and Word2Vec, the size of the dataset, etc. **Issues:** The predictions are highly volatile and solid interpretations cannot based on them, as the vary based on the run. The steps used for pre-processing tweets for sentiment analysis are debated, and these issues need to be solved in order to establish guidelines that are proven to benefit model performance. For example, the exclusion/inclusion of stop words or the use of stemming is advertised as important by some authors and detrimental to model performance by others. Furthermore, I originally intended to use the DeepNl library to generate *sentiment-specific word embeddings*; however, I encountered numerous errors with the installation and the limited documentation and lack of support forums did not help. More research needs to be done in this field for the construction of sentiment analysis data science packages and guidelines. I also did not like the relatively small size of the dataset, which definitely influenced the model performance; I would have preferred a dataset of approximately 50,000 rows, as Word2Vec works best with large corpuses. **Benefits:** However, one benefit is that Word2Vec works can still provide meaningful information with small corpuses, such as the one used for this project. Furthermore, using the Keras API allowed easy inputting of the Word2Vec embeddings and GloVe embeddings into the CNN and I personally preferred Keras' text-preprocessing functions compared to NLTK's text-preprocessing functions. Next steps include more extensive tuning of the CNN to achieve higher model accuracy, training/testing the model on a larger dataset, and comparing the differences between Word2Vec and GloVe.

## References (for code and technical information):

"Step-by-Step Twitter Sentiment Analysis: Visualizing United Airlines' PR Crisis" (http://ipullrank.com/step-step-twitter-sentiment-analysis-visualizing-united-airlines-pr-crisis/)

"Using pre-trained word embeddings in a Keras model" (https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html)

"How to Develop a Word Embedding Model for Predicting Movie Review Sentiment" (https://machinelearningmastery.com/develop-word-embedding-model-predicting-movie-review-sentiment/)

"Emotion Detection on Twitter Data using Knowledge Base Approach" (https://pdfs.semanticscholar.org/6698/5a996eab1e680ffdd88a4e92964ac4e7dd56.pdf)

*Acknowledgements of Datasets*

**Twitter sentiment analysis: Self-driving cars dataset available at:**
https://data.world/crowdflower/sentiment-self-driving-cars.

**Pre-trained GloVe word vectors that GloVe creators provided, available at:**
https://nlp.stanford.edu/projects/glove/webpage.