Problem statement

The objective of this project is to apply sentiment analysis, the most common text classification tool, to analyse an incoming message and tell whether the underlying sentiment is positive, negative or neutral. The dataset contains tweets on US Airline of February 2015 classified in positive, negative and neutral tweets. The negative tweets are also classified by the negative reason.

The first challenge is that the dataset is very unstructured. Before we do data visualization and model building, we need to do some data pre-processing or wrangling to remove unnecessary characters, symbols and tokens. The definition of unnecessary information actually depends on which machine learning techniques to use. In this project, we will apply both traditional and advanced techniques to build sentiment analysis models. For traditional ones, we will try to clean the unnecessary information as much as possible since we convert the documents into individual tokens without considering the context. By contrast, we need to keep import contextual connection between words if word2vec or deep learning model is built.

Once the model is put into implementation, the business owner is able to predict whether the customers' feedback is positive, negative or neutral. This can help identify potential issues in the process of corporate operations and improve the customers' satisfaction.

Data pre-processing

There can be multiple ways of cleaning and pre-processing textual data. In this project, we will take some of the most popular approaches which are used heavily in Natural Language Processing (NLP) pipelines.

1. Tweet preprocessor:

Tweet preprocessor is a preprocessing library for tweet data written in Python. When building machine learning systems based on tweet data, a preprocessing is required. This library makes it easy to clean, parse or tokenize the tweets. It supports cleaning, tokenizing and parsing: URLs, Hashtags, Mentions, Reserved words (RT,FAV),Emojis and Smileys.Reserved words (RT, FAV). For example, the original content of the first tweet is **@VirginAmerica What @dhepburn said.** After this step, it was converted into "What said".

2. Expanding contractions:

In the English language, contractions are basically shortened versions of words or syllables. These shortened versions of existing words or phrases are created by removing specific letters and sounds. Converting each contraction to its expanded, original form often helps with text standardization. For example, the content of third row is “`@VirginAmerica I didn't today... Must mean I need to take another trip!`. The step converted “didn’t “ into “did not”.

3. Removing special characters:

Special characters and symbols which are usually non alphanumeric characters often add to the extra noise in unstructured text. More than often, simple regular expressions (regexes) can be used to achieve this. In this project, we use the regular expression '[^a-zA-z0-9\s]'.

4. Stemming and lemmatization:

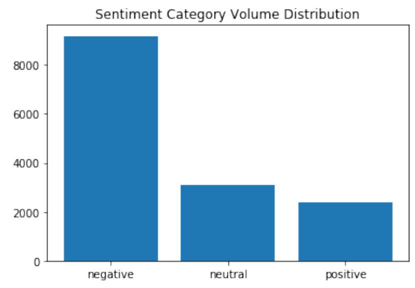
Word stems are usually the base form of possible words that can be created by attaching *affixes* like *prefixes* and *suffixes* to the stem to create new words. This is known as inflection. The reverse process of obtaining the base form of a word is known as *stemming*. For example, the words “said” have been converted into “say”.

5. Removing stopwords:

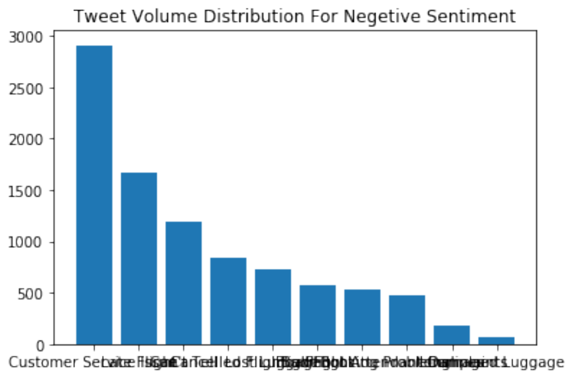
Words which have little or no significance especially when constructing meaningful features from text are known as stopwords or stop words. These are usually words that end up having the maximum frequency if we do a simple term or word frequency in a corpus. We use package NLTK to create a stopwords list, which contains 17 words.

Exploratory Data Analysis

The first talk for exploratory data analysis is to look at how the volume of different target variables is distributed. We use function .value_counts() to achieve this.

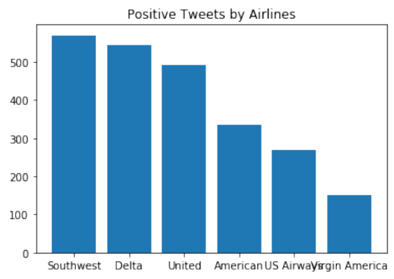


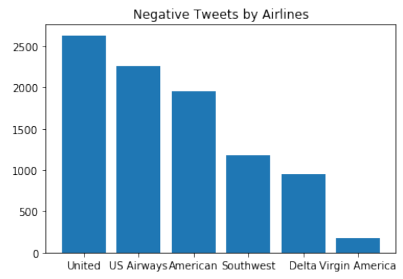
From this chart, we can see this dataset doesn't have an issue of 'imbalance". Another topic we are interested to see is how the negative reasons are distributed for "negative" sentiment group.

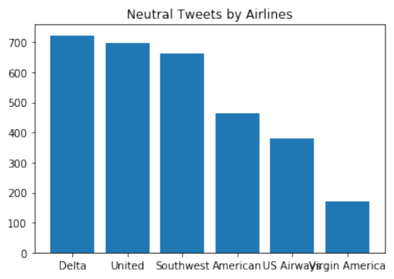


We noticed that the top 3 reasons are Customer Service, Late Flight and Can't tell.

This dataset contains the tweets provided by six airlines: Delta, United, Southwest, American, US Airways and Virgin America. To make a comparison between these operators, we will look at how the number of tweets distributed under different sentiment categories for all these six airlines. The chart below indicates that the Southwest had the best performance in terms of positive tweets; United had the worst performance measured by negative tweets.







Unlike other machine learning tasks, Natural Language Processing mainly focuses on textual data. This means we are not going to take the traditional approaches to visualize the data. The package NLTK can help us calculate and view the frequency of each word in the corpus. The size of vocabulary is 20,127. The 10 most frequent -ly occurred words are `'@', '.', 'to', 'I', 'the', '!', '?', 'a', ',', 'for'.`
They are either stop words or special words, which will be removed in the stage of text preprocessing. The package WordCloud enables us to visualize the occurrence of each word.

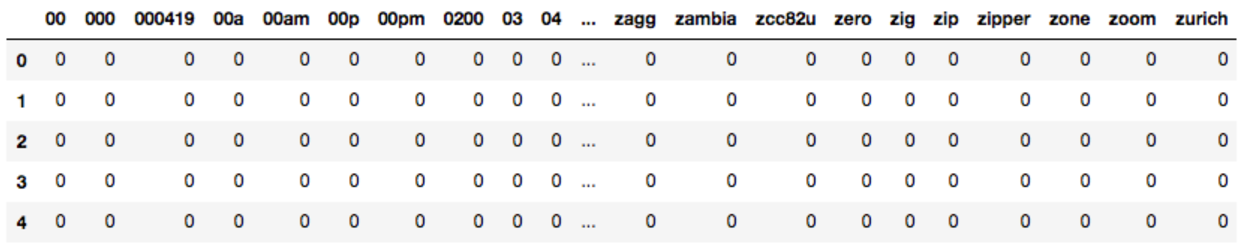
Text data encoding

Text data requires special preparation before using it for predictive modeling. The text must be parsed to remove words, called tokenization. Then the words need to be encoded as integers or floating point values for use as input to a machine learning algorithm. A simple and effective model for thinking about text documents in machine learning is called the Bag-of-Words Model, or BoW. The model is simple in that it throws away all of the order information in the words and focuses on the occurrence of words in a document.

The scikit-learn library provides two different schemes that we can use:

Word Counts with CountVectorizer

The CountVectorizer provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary. For this project, we created a data frame with 6875 columns.

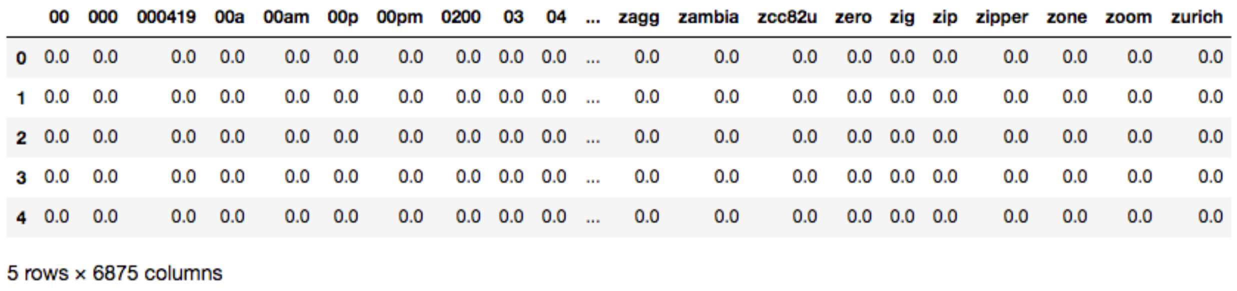


	00	000	000419	00a	00am	00p	00pm	0200	03	04	...	zagg	zambia	zcc82u	zero	zig	zip	zipper	zone	zoom	zurich
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 6875 columns

Word Frequencies with TfidfVectorizer

One issue with simple counts is that some words will appear many times and their large counts will not be very meaningful in the encoded vectors. An alternative is to calculate word frequencies, and by far the most popular method is called TF-IDF. This stands for "*Term Frequency – Inverse Document*" Frequency which are the components of the resulting scores assigned to each word.



	00	000	000419	00a	00am	00p	00pm	0200	03	04	...	zagg	zambia	zcc82u	zero	zig	zip	zipper	zone	zoom	zurich
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 6875 columns

Traditional Machine Learning Model Building

So far, we have completed the data visualization, data preprocessing and text data encoding. As a result, we created two data frames for CountVectorization and TF-IDF respectively. Both of these data frames contain numerical values only. We can use traditional classification machine learning techniques to build predictive models: Naïve Bayes multinomial classification and Support Vector classification techniques.

- CountVectorization

Naïve Bayes multinomial classification	Support Vector classification
Accuracy: `0.7491`	Accuracy: `0.7692`

- TF-IDF

Naïve Bayes multinomial classification	Support Vector classification
Accuracy: `0.6841`	Accuracy: `0.7651`

TD-IDF has two important hyper-parameters: max_df and ngram_range.
When building the vocabulary, the hyper-parameter max_df ignores terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). If float in range [0.0, 1.0], the parameter represents a proportion of documents, integer absolute counts. The parameter ngram_range defines the lower and upper boundary of the range of n-values for different n-grams to be extracted. SVC has also an important regularization parameter C.
The strength of the regularization is inversely proportional to C. We created a new pipeline by combining TF-IDF and SVC.

Grid search is the process of performing hyper-parameter tuning in order to determine the optimal values for a given model. This is significant as the performance of the entire model is based on the hyper-parameter values specified. We applied the method GridSearchCV on the built pipeline and found the optimal values of hyper-parameter as below:

```
{'model__C': 10, 'tfidf__max_df': 0.5, 'tfidf__ngram_range': (1, 1)}
```

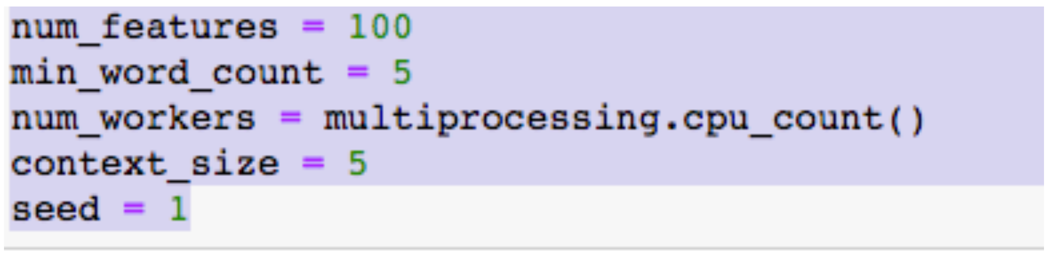
The accuracy score we achieved is 0.7719.

Using word embeddings

Word embedding is one of the most important techniques in natural language processing(NLP), where words are mapped to vectors of real numbers. Word embedding is capable of capturing the meaning of a word in a document, semantic and syntactic similarity, relation with other words.

Word2vec is one of the most popular technique to learn word embeddings using a two-layer neural network. Its input is a text corpus and its output is a set of vectors. Word embedding via word2vec can make natural language computer-readable, then further implementation of mathematical operations on words can be used to detect their similarities.

Gensim is an open source python library for natural language processing. Gensim library will enable us to develop word embeddings by training our own word2vec models on a custom corpus either with CBOW of skip-grams algorithms. To achieve better performance using Gensim word2vec, we will specify some key hyper-parameters as below:



```
num_features = 100
min_word_count = 5
num_workers = multiprocessing.cpu_count()
context_size = 5
seed = 1
```

After the Gensim word2vec model is trained, we will obtain a vocabulary with length 2631, which means only 2,631 important words are kept. We then calculate the average vector values for each document in the corpus. This value will be treated as a predictor for traditional machine learning model building. Similar to Bag-of-Words model, we will apply support vector classification and logistic regression to the data frame that contains average vector values.

Logistic Regression	Support Vector classification
Accuracy: 0.7485	Accuracy: 0.7593

Gensim also enables us to use pre-trained word2vec model. Firstly, we will import pre-trained Google News model with 300 dimensions for word vector. Then we will calculate the average vector value for each document in the corpus. Lastly, we built the applied traditional machine learning models to this data set and achieved the accuracy score as below:

SVC	decision tree	logistic regression
0.7823	0.6226	0.7659

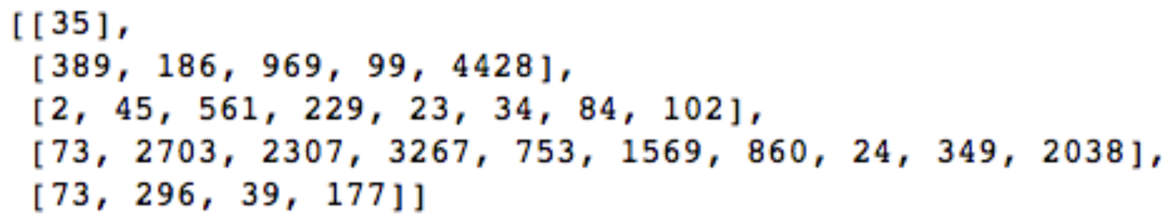
Deep learning on Keras

- Learning word embeddings with the embedding layer

In real world, there is no "ideal" word embedding space that would perfectly map human language and could be used for any natural language processing task. Also, there isn't such a thing as "human language", there are many different languages and they are not isomorphic, as a language is the reflection of a specific culture and a specific context. What makes a good word embedding space depends heavily on the specific task we perform. It is thus reasonable to learn a new embedding space with every new task. Thankfully, backpropagation makes this really easy, and Keras makes it even easier. It's just about learning the weights of a layer: the Embedding layer.

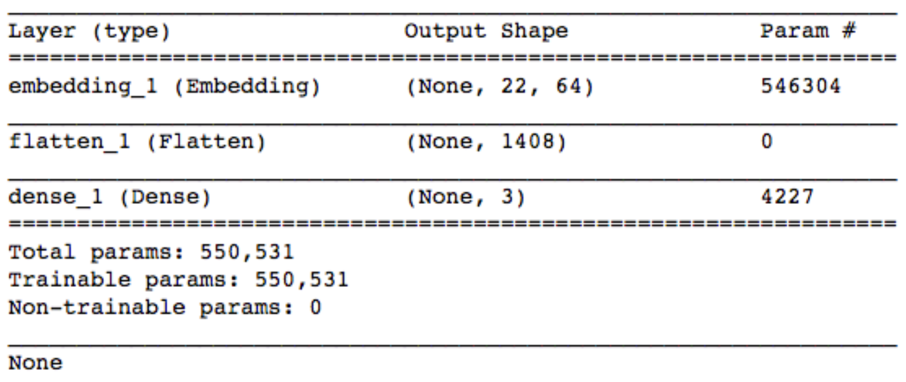
First of all, we converted a class vector (integers) to binary class matrix using keras.utils.to.categorical.

The Embedding layer is best understood as a dictionary mapping integer indices (which stand for specific words) to dense vectors. It takes as input integers, it looks up these integers into an internal dictionary, and it returns the associated vectors. The Embedding layer takes as input a 2D tensor of integers, of shape (samples, sequence_length), where each entry is a sequence of integers. All sequences in a batch must have the same length, so sequences that are shorter than others should be padded with zeros, and sequences that are longer should be truncated. The image below shows the sequences of word index for the first five documents in the corpus.



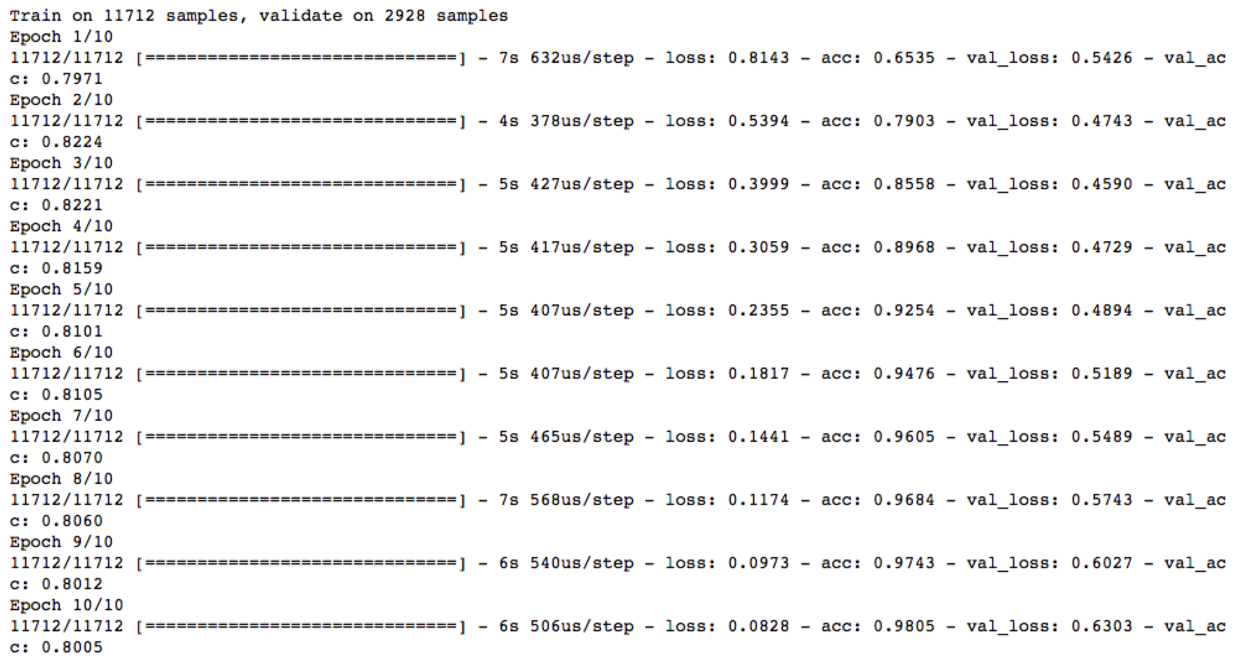
```
[[35],
 [389, 186, 969, 99, 4428],
 [2, 45, 561, 229, 23, 34, 84, 102],
 [73, 2703, 2307, 3267, 753, 1569, 860, 24, 349, 2038],
 [73, 296, 39, 177]]
```

We then identify the maximum length is 22. Since this data set is relatively small, we want to keep all these word indices. All the sequences will be padded with zeros to the length of 22. Once we have known the total number of unique tokens, calculated using len(word_index), which is 8,535, we are able to build the neural network model as follows:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 22, 64)            546304    
_________________________________________________________________
flatten_1 (Flatten)          (None, 1408)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 4227      
=================================================================
Total params: 550,531
Trainable params: 550,531
Non-trainable params: 0
_________________________________________________________________
None
```

We then trained this model and achieved the accuracy score of 0.8224 when the training process hit the second epoch.



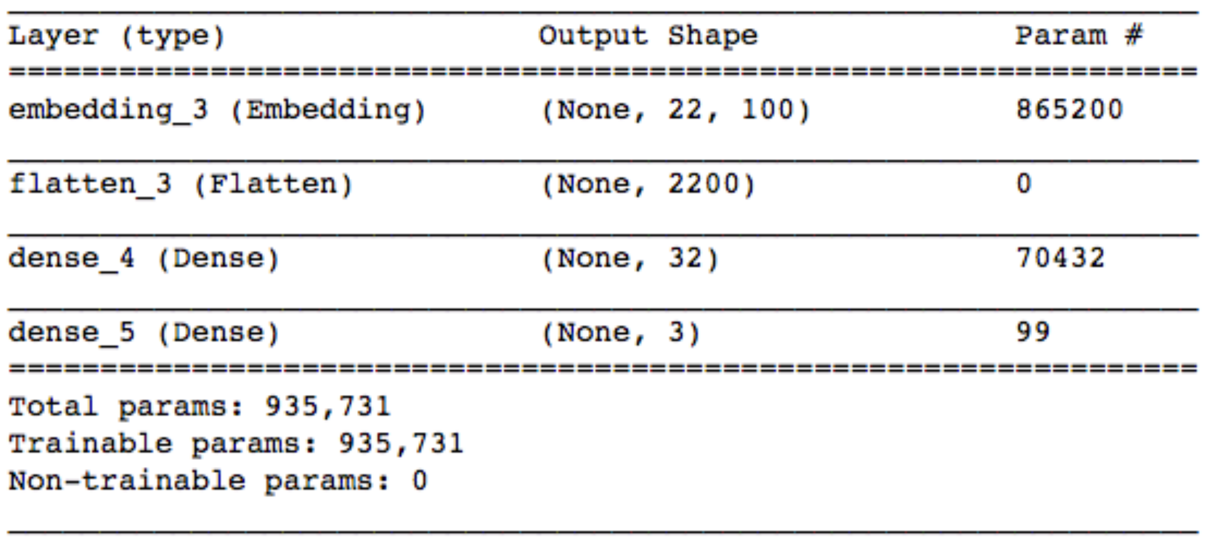
```
Train on 11712 samples, validate on 2928 samples
Epoch 1/10
11712/11712 [==============================] - 7s 632us/step - loss: 0.8143 - acc: 0.6535 - val_loss: 0.5426 - val_ac
c: 0.7971
Epoch 2/10
11712/11712 [==============================] - 4s 378us/step - loss: 0.5394 - acc: 0.7903 - val_loss: 0.4743 - val_ac
c: 0.8224
Epoch 3/10
11712/11712 [==============================] - 5s 427us/step - loss: 0.3999 - acc: 0.8558 - val_loss: 0.4590 - val_ac
c: 0.8221
Epoch 4/10
11712/11712 [==============================] - 5s 417us/step - loss: 0.3059 - acc: 0.8968 - val_loss: 0.4729 - val_ac
c: 0.8159
Epoch 5/10
11712/11712 [==============================] - 5s 407us/step - loss: 0.2355 - acc: 0.9254 - val_loss: 0.4894 - val_ac
c: 0.8101
Epoch 6/10
11712/11712 [==============================] - 5s 407us/step - loss: 0.1817 - acc: 0.9476 - val_loss: 0.5189 - val_ac
c: 0.8105
Epoch 7/10
11712/11712 [==============================] - 5s 465us/step - loss: 0.1441 - acc: 0.9605 - val_loss: 0.5489 - val_ac
c: 0.8070
Epoch 8/10
11712/11712 [==============================] - 7s 568us/step - loss: 0.1174 - acc: 0.9684 - val_loss: 0.5743 - val_ac
c: 0.8060
Epoch 9/10
11712/11712 [==============================] - 6s 540us/step - loss: 0.0973 - acc: 0.9743 - val_loss: 0.6027 - val_ac
c: 0.8012
Epoch 10/10
11712/11712 [==============================] - 6s 506us/step - loss: 0.0828 - acc: 0.9805 - val_loss: 0.6303 - val_ac
c: 0.8005
```

- Using pre-trained word embeddings

Sometimes we have little training data available that could never use the data alone to learn an appropriate task-specific embedding of the vocabulary. Instead of learning word embeddings jointly with the problem to be solved, we can load embedding vectors from a pre-computed embedding space known to be highly structured and to exhibit useful properties -- that captures generic aspects of language structure.

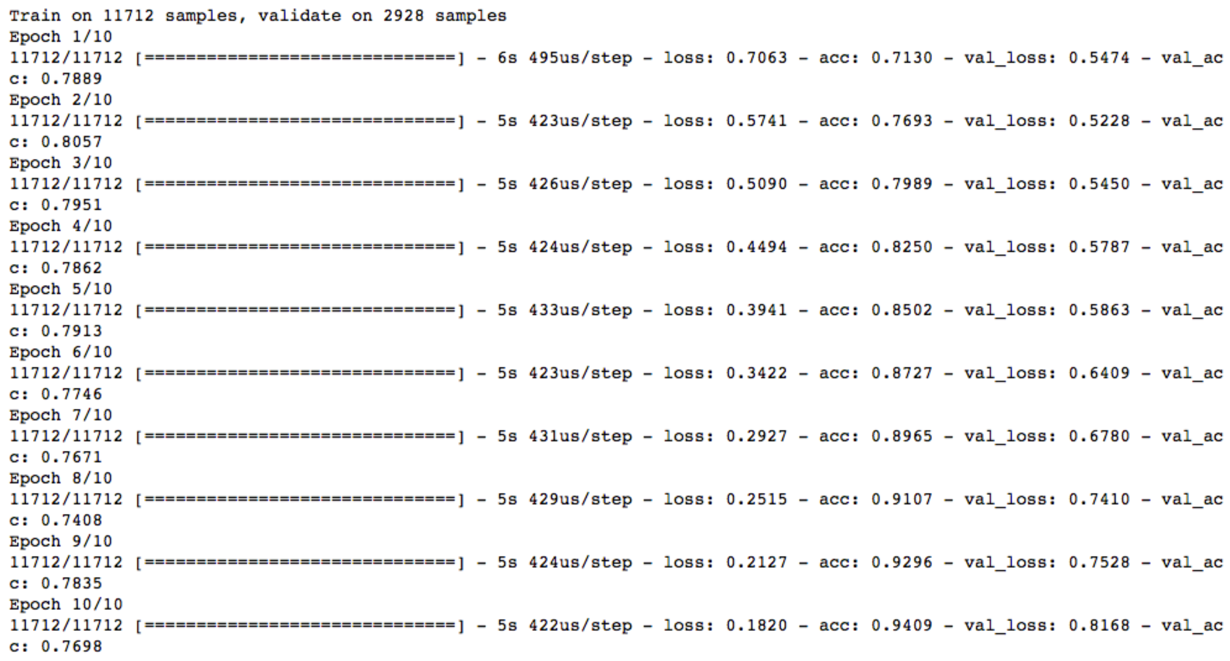
There are various pre-computed databases of word embeddings that can download and start using in a Keras Embedding layer. Word2Vec is one of them. Another popular one is called "GloVe", developed by Stanford researchers in 2014. It stands for "Global Vectors for Word Representation", and it is an embedding technique based on factorizing a matrix of word co-occurrence statistics.

First of all, after downloading and importing 100-dimensional “Glove” database, we created the dictionary of embedding index. Then we need to build an embedding matrix that we will be able to load into an Embedding layer. It must be a matrix of shape (max_words, embedding_dim), where each entry i contains the embedding_dim-dimensional vector for the word of index i in our reference. We will be using the same model architecture as before:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 22, 100)           865200
_________________________________________________________________
flatten_3 (Flatten)          (None, 2200)              0
_________________________________________________________________
dense_4 (Dense)              (None, 32)                70432
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 99
=================================================================
Total params: 935,731
Trainable params: 935,731
Non-trainable params: 0
_________________________________________________________________
```

Then we trained this model and achieved the accuracy score of 0.8057 at the second epoch.



```
Train on 11712 samples, validate on 2928 samples
Epoch 1/10
11712/11712 [==============================] - 6s 495us/step - loss: 0.7063 - acc: 0.7130 - val_loss: 0.5474 - val_ac
c: 0.7889
Epoch 2/10
11712/11712 [==============================] - 5s 423us/step - loss: 0.5741 - acc: 0.7693 - val_loss: 0.5228 - val_ac
c: 0.8057
Epoch 3/10
11712/11712 [==============================] - 5s 426us/step - loss: 0.5090 - acc: 0.7989 - val_loss: 0.5450 - val_ac
c: 0.7951
Epoch 4/10
11712/11712 [==============================] - 5s 424us/step - loss: 0.4494 - acc: 0.8250 - val_loss: 0.5787 - val_ac
c: 0.7862
Epoch 5/10
11712/11712 [==============================] - 5s 433us/step - loss: 0.3941 - acc: 0.8502 - val_loss: 0.5863 - val_ac
c: 0.7913
Epoch 6/10
11712/11712 [==============================] - 5s 423us/step - loss: 0.3422 - acc: 0.8727 - val_loss: 0.6409 - val_ac
c: 0.7746
Epoch 7/10
11712/11712 [==============================] - 5s 431us/step - loss: 0.2927 - acc: 0.8965 - val_loss: 0.6780 - val_ac
c: 0.7671
Epoch 8/10
11712/11712 [==============================] - 5s 429us/step - loss: 0.2515 - acc: 0.9107 - val_loss: 0.7410 - val_ac
c: 0.7408
Epoch 9/10
11712/11712 [==============================] - 5s 424us/step - loss: 0.2127 - acc: 0.9296 - val_loss: 0.7528 - val_ac
c: 0.7835
Epoch 10/10
11712/11712 [==============================] - 5s 422us/step - loss: 0.1820 - acc: 0.9409 - val_loss: 0.8168 - val_ac
c: 0.7698
```

RNN & LSTM

Keras provides two different layers, SimpleRNN & LSTM. We can use them together with embedding layer to construct neural network model. We trained them on our dataset, and achieved the performance as below:

```
Train on 11712 samples, validate on 2928 samples
Epoch 1/10
11712/11712 [==============================] - 6s 518us/step - loss: 0.7643 - acc: 0.6867 - val_loss: 0.5623 - val_ac
c: 0.7964
Epoch 2/10
11712/11712 [==============================] - 5s 394us/step - loss: 0.5662 - acc: 0.7836 - val_loss: 0.5132 - val_ac
c: 0.8012
Epoch 3/10
11712/11712 [==============================] - 5s 389us/step - loss: 0.4733 - acc: 0.8227 - val_loss: 0.5505 - val_ac
c: 0.7862
Epoch 4/10
11712/11712 [==============================] - 5s 400us/step - loss: 0.4126 - acc: 0.8487 - val_loss: 0.6630 - val_ac
c: 0.7377
Epoch 5/10
11712/11712 [==============================] - 5s 390us/step - loss: 0.3578 - acc: 0.8725 - val_loss: 0.5906 - val_ac
c: 0.7848
Epoch 6/10
11712/11712 [==============================] - 5s 393us/step - loss: 0.3133 - acc: 0.8881 - val_loss: 0.5705 - val_ac
c: 0.7900
Epoch 7/10
11712/11712 [==============================] - 5s 399us/step - loss: 0.2772 - acc: 0.9016 - val_loss: 0.5892 - val_ac
c: 0.7937
Epoch 8/10
11712/11712 [==============================] - 5s 407us/step - loss: 0.2402 - acc: 0.9152 - val_loss: 0.7106 - val_ac
c: 0.7490
Epoch 9/10
11712/11712 [==============================] - 5s 422us/step - loss: 0.2084 - acc: 0.9263 - val_loss: 0.6624 - val_ac
c: 0.7702
Epoch 10/10
11712/11712 [==============================] - 5s 424us/step - loss: 0.1825 - acc: 0.9365 - val_loss: 0.6946 - val_ac
c: 0.7811
```

```
Train on 11712 samples, validate on 2928 samples
Epoch 1/10
11712/11712 [==============================] - 15s 1ms/step - loss: 0.8327 - acc: 0.6361 - val_loss: 0.5610 - val_ac
c: 0.7688
Epoch 2/10
11712/11712 [==============================] - 18s 2ms/step - loss: 0.6253 - acc: 0.7229 - val_loss: 0.5160 - val_ac
c: 0.7971
Epoch 3/10
11712/11712 [==============================] - 20s 2ms/step - loss: 0.5508 - acc: 0.7717 - val_loss: 0.5224 - val_ac
c: 0.7900
Epoch 4/10
11712/11712 [==============================] - 31s 3ms/step - loss: 0.5086 - acc: 0.7978 - val_loss: 0.5100 - val_ac
c: 0.7917
Epoch 5/10
11712/11712 [==============================] - 26s 2ms/step - loss: 0.4782 - acc: 0.8147 - val_loss: 0.5026 - val_ac
c: 0.8033
Epoch 6/10
11712/11712 [==============================] - 23s 2ms/step - loss: 0.4554 - acc: 0.8262 - val_loss: 0.5226 - val_ac
c: 0.8053
Epoch 7/10
11712/11712 [==============================] - 24s 2ms/step - loss: 0.4338 - acc: 0.8336 - val_loss: 0.5225 - val_ac
c: 0.7848
Epoch 8/10
11712/11712 [==============================] - 24s 2ms/step - loss: 0.4145 - acc: 0.8459 - val_loss: 0.5235 - val_ac
c: 0.8023
Epoch 9/10
11712/11712 [==============================] - 28s 2ms/step - loss: 0.3908 - acc: 0.8595 - val_loss: 0.5188 - val_ac
c: 0.8016
Epoch 10/10
11712/11712 [==============================] - 27s 2ms/step - loss: 0.3643 - acc: 0.8711 - val_loss: 0.5492 - val_ac
c: 0.7934
```