0104_bag_of_words

December 15, 2018

1 104: Using free text for classification – 'Bag of Words'

There may be times in healthcare where we would like to classify patients based on free text data we have for them. Maybe, for example, we would like to predict likely outcome based on free text clinical notes.

Using free text requires methods known as 'Natural Language Processing'.

Here we start with one of the simplest techniques – 'bag of words'.

In a 'bag of words' free text is reduced to a vector (a series of numbers) that represent the number of times a word is used in the text we are given. It is also posible to look at series of two, three or more words in case use of two or more words together helps to classify a patient.

A classic 'toy problem' used to help teach or develop methos is to try to judge whether people rates a film as 'like' or 'did not like' based on the free text they entered into a widely used internet film review database (www.imdb.com).

Here will will use 50,000 records from IMDb to convert each review into a 'bag of words', which we will then use in a simple logistic regression machine learning model.

We can use raw word counts, but in this case we'll add an extra transformation called tf-idf (frequency–inverse document frequency) which adjusts values according to the number fo reviews that use the word. Words that occur across many reviews may be less discriminatory than words that occur more rarely, so tf-idf reduces the value of those words used frequently across reviews.

This code will take us through the following steps:

1) Load data from internet
2) Clean data – remove non-text, convert to lower case, reduce words to their 'stems' (see below for details), and reduce common 'stop-words' (such as 'as', 'the', 'of').
3) Split data into trainign and test dat sets
4) Convert cleaned reviews in word vectors ('bag of words'), and apply the tf-idf transform.
5) Train a logistic regression model on the tr-idf transformed word vectors.
6) Apply the logistic regression model to our previously unseen test cases, and calculate accuracy of our model.

1.1 Load data

```
In [1]: import pandas as pd

        # If you do not already have the data locally you may download (and save) by

        file_location = 'https://gitlab.com/michaelallen1966/00_python_snippets' +\
            '_and_recipes/raw/master/machine_learning/data/IMDb.csv'
        imdb = pd.read_csv(file_location)
        # save to current directory
        imdb.to_csv('imdb.csv', index=False)

        # If you already have the data locally then you may run the following

        # Load data example
        imdb = pd.read_csv('imdb.csv')

        # Truncate data for example if you want to speed up the example
        # imdb = imdb.head(5000)
```

1.2 Define Function to preprocess data

This function, as previously described, works on raw text strings, and:

1) changes to lower case
2) tokenizes (breaks down into words
3) removes punctuation and non-word text
4) finds word stems
5) removes stop words
6) rejoins meaningful stem words

```
In [2]: import nltk
        import pandas as pd
        import numpy as np
        from nltk.stem import PorterStemmer
        from nltk.corpus import stopwords

        # If not previously performed:
        # nltk.download('stopwords')

        stemming = PorterStemmer()
        stops = set(stopwords.words("english"))

        def apply_cleaning_function_to_list(X):
            cleaned_X = []
            for element in X:
                cleaned_X.append(clean_text(element))
            return cleaned_X
```

```
def clean_text(raw_text):
    """This function works on a raw text string, and:
        1) changes to lower case
        2) tokenizes (breaks down into words
        3) removes punctuation and non-word text
        4) finds word stems
        5) removes stop words
        6) rejoins meaningful stem words"""

    # Convert to lower case
    text = raw_text.lower()

    # Tokenize
    tokens = nltk.word_tokenize(text)

    # Keep only words (removes punctuation + numbers)
    # use .isalnum to keep also numbers
    token_words = [w for w in tokens if w.isalpha()]

    # Stemming
    stemmed_words = [stemming.stem(w) for w in token_words]

    # Remove stop words
    meaningful_words = [w for w in stemmed_words if not w in stops]

    # Rejoin meaningful stemmed words
    joined_words = ( " ".join(meaningful_words))

    # Return cleaned data
    return joined_words
```

Apply the data cleaning function (thi smay take a few minutes if you are using the full 50,000 reviews).

```
In [3]: # Get text to clean
        text_to_clean = list(imdb['review'])

        # Clean text
        cleaned_text = apply_cleaning_function_to_list(text_to_clean)

        # Add cleaned data back into DataFrame
        imdb['cleaned_review'] = cleaned_text

        # Remove temporary cleaned_text list (after transfer to DataFrame)
        del cleaned_text
```

1.3 Split data into training and test data sets

```
In [4]: from sklearn.model_selection import train_test_split
        X = list(imdb['cleaned_review'])
        y = list(imdb['sentiment'])
        X_train, X_test, y_train, y_test = train_test_split(
            X,y, test_size = 0.25)
```

1.4 Create 'bag of words'

The 'bag of words' is the word vector for each review. This may be a simple word count for each review where each position of the vector represnts a word (returned in the 'vocab' list) and the value of that position represents the number fo times that word is used in the review.

The function below also returns a tf-idf (frequency–inverse document frequency) which adjusts values according to the number fo reviews that use the word. Words that occur across many reviews may be less discriminatory than words that occur more rarely. The tf-idf transorm reduces the value of a given word in proportion to the number of documents that it appears in.

The function returns the following:

1) vectorizer - this may be applied to any new reviews to convert the revier into the same word vector as the training set.

2) vocab - the list of words that the word vectors refer to.

3) train_data_features - raw word count vectors for each review

4) tfidf_features - tf-idf transformed word vectors

5) tfidf - the tf-idf transformation that may be applied to new reviews to convert the raw word counts into the transformed word counts in the same way as the training data.

Our vectorizer has an argument called 'ngram_range'. A simple bag of words divides reviews into single words. If we have an ngram_range of (1,2) it means that the review is divided into single words and also pairs of consecutiev words. This may be useful if pairs of words are useful, such as 'very good'. The max_features argument limits the size of the word vector, in this case to a maximum of 10,000 words (or 10,000 ngrams of words if an ngram may be mor than one word).

```
In [5]: def create_bag_of_words(X):
            from sklearn.feature_extraction.text import CountVectorizer

            print ('Creating bag of words...')
            # Initialize the "CountVectorizer" object, which is scikit-learn's
            # bag of words tool.

            # In this example features may be single words or two consecutive words
            # (as shown by ngram_range = 1,2)
            vectorizer = CountVectorizer(analyzer = "word",    \
                                         tokenizer = None,     \
                                         preprocessor = None, \
                                         stop_words = None,    \
```

```
                                        ngram_range = (1,2), \
                                        max_features = 10000
                                       )

            # fit_transform() does two functions: First, it fits the model
            # and learns the vocabulary; second, it transforms our training data
            # into feature vectors. The input to fit_transform should be a list of
            # strings. The output is a sparse array
            train_data_features = vectorizer.fit_transform(X)

            # Convert to a NumPy array for easy of handling
            train_data_features = train_data_features.toarray()

            # tfidf transform
            from sklearn.feature_extraction.text import TfidfTransformer
            tfidf = TfidfTransformer()
            tfidf_features = tfidf.fit_transform(train_data_features).toarray()

            # Get words in the vocabulary
            vocab = vectorizer.get_feature_names()

            return vectorizer, vocab, train_data_features, tfidf_features, tfidf
```

Apply our bag of words function to our training set.

```
In [6]: vectorizer, vocab, train_data_features, tfidf_features, tfidf  = \
            create_bag_of_words(X_train)

Creating bag of words...
```

We can create a DataFrame of our words and counts, so that we may sort and view them. The count and tfidf_features exist for each X (each review in this case) - here we will look at just the first review (index 0).

Note that the tfidf_features differ from the count; that is because of the adjustment for how commonly they occur across reviews.

(Try changing the sort to sort by tfidf_features).

```
In [7]: bag_dictionary = pd.DataFrame()
        bag_dictionary['ngram'] = vocab
        bag_dictionary['count'] = train_data_features[0]
        bag_dictionary['tfidf_features'] = tfidf_features[0]

        # Sort by raw count
        bag_dictionary.sort_values(by=['count'], ascending=False, inplace=True)
        # Show top 10
        print(bag_dictionary.head(10))
```

```
         ngram  count  tfidf_features
9320        wa      4        0.139373
5528      movi      3        0.105926
9728     whole      2        0.160024
3473    german      2        0.249079
6327      part      2        0.140005
293   american      1        0.089644
9409   wa kind      1        0.160155
9576      wast      1        0.087894
7380       saw      1        0.078477
7599      sens      1        0.085879
```

1.5 Training a machine learning model on the bag of words

Now we have transformed our free text reviews in vectors of numbers (representing words) we can apply many different machine learning techniques. Here will will use a relatively simple one, logistic regression.

We'll set up a function to train a logistic regression model.

```
In [8]: def train_logistic_regression(features, label):
            print ("Training the logistic regression model...")
            from sklearn.linear_model import LogisticRegression
            ml_model = LogisticRegression(C = 100,random_state = 0)
            ml_model.fit(features, label)
            print ('Finished')
            return ml_model
```

Now we will use the tf-idf tranformed word vectors to train the model (we could use the plain word counts contained in 'train_data_features' (rather than using 'tfidf_features'). We pass both the features and the known label corresponding to the review (the sentiment, either 0 or 1 depending on whether a person likes the film or not.

```
In [9]: ml_model = train_logistic_regression(tfidf_features, y_train)
```

```
Training the logistic regression model...
Finished
```

1.6 Applying the bag of words model to test reviews

We will now apply the bag of words model to test reviews, and assess the accuracy.

We'll first apply our vectorizer to create a word vector for review in the test data set.

```
In [10]: test_data_features = vectorizer.transform(X_test)
         # Convert to numpy array
         test_data_features = test_data_features.toarray()
```

As we are using the tf-idf transform, we'll apply the tfid transformer so that word vectors are transformed in the same way as the training data set.

```
In [11]: test_data_tfidf_features = tfidf.fit_transform(test_data_features)
         # Convert to numpy array
         test_data_tfidf_features = test_data_tfidf_features.toarray()
```

Now the bit that we really want to do – we'll predict the sentiment of the all test reviews (and it's just a single line of code!). Did they like the film or not?

```
In [12]: predicted_y = ml_model.predict(test_data_tfidf_features)
         correctly_identified_y = predicted_y == y_test
         accuracy = np.mean(correctly_identified_y) * 100
         print ('Accuracy = %.0f%%' %accuracy)

Accuracy = 87%
```