Lesson 6 - Supervised learning

March 21, 2018

1 Lesson 6 - Supervised Learning* ## 1. Machine learning (ML)

The sentiment analysis program we wrote earlier (in session 1) adopts a non-machine learning algorithm. That is, it tries to define what good and bad sentiments are and assumes all the necessary words of good and bad sentiments exist in the word_sentiment.csv file.

Machine Learning (ML) is a class of algorithms, which are data-driven, i.e. unlike "normal" algorithms, it is the data that "tells" what the "good answer" is. A machine learning algorithm would not have such coded definition of what a good and bad sentiment is, but would "learn-by-examples". That is, you will show several words which have been labeled as good sentiment and bad sentiment and a good ML algorithm will eventually learn and be able to predict whether or not an unseen word has a good or bad sentiment. This particular example of sentiment analysis is "supervised", which means that your example words must be labeled, or explicitly say which words are good and which are bad.

On the other hand, in the case of unsupervised learning, the word examples are not labeled. Of course, in such a case the algorithm itself cannot "invent" what a good sentiment is, but it can try to cluster the data into different groups, e.g. it can figure out that words that are close to certain other words are different from words closer to some other words (eg. words close to the word "mother" are most likely good). There are "intermediate" forms of supervision, i.e. semi-supervised and active learning. Technically, these are supervised methods in which there is some "smart" way to avoid a large number of labeled examples.

- In active learning, the algorithm itself decides which thing you should label (e.g. it can be pretty sure about a sentence that has the word fantastic, but it might ask you to confirm if the sentence may have a negative like “not”).
- In semi-supervised learning, there are two different algorithms, which start with the labeled examples, and then "tell" each other the way they think about some large number of unlabeled data. From this "discussion" they learn.

Figure : Supervised learning approach

```
<img src="ML_supervised.gif"  width="500" title="Supervised learning">
```

*REF: http://www.nltk.org/book/ch06.html

1.0.1 1.1 Feature extraction

Define what features of a word that you want to use in order to classify the data set. We will select two features the first and the last letter of the word.

```
In [ ]: def feature_extractor(word):
            """Extract the features for a given word and return a dictonary of the features"""
            start_letter = word[0]
            last_letter = word[-1]
            return {'start_letter' : start_letter,'last_letter' : last_letter}

        def main():
            print(feature_extractor('poonacha'))

        main()
```

1.0.2 1.2 Training the ML algorithm

NLTK module is built for working with language data. NLTK supports classification, tokenization, stemming, tagging, parsing, and semantic reasoning functionalities. We will use the NLTK module and employ the naive Bayes method to classify words as being either positive or negative sentiment. You can also use other modules specifically meant for ML eg. sklearn module.

Step 1: Create the feature set We will use the corpus of sentiments from the word_sentiment.csv file to create a feature dataset which we will use to train and test the ML model.

```
In [ ]: import csv

        def feature_extractor(word):
            """Extract the features for a given word and return a dictonary of the features"""
            start_letter = word[0]
            last_letter = word[-1]
            return {'start_letter' : start_letter,'last_letter' : last_letter}

        def ML_train(sentiment_corpus):
            """Create feature set from the corpus given to to it."""
            feature_set = []
            with open(sentiment_corpus,'rt',encoding = 'utf-8') as sentobj:
                sentiment_handle = csv.reader(sentobj)

                for sentiment in sentiment_handle:
                    new_row = []
                    new_row.append(feature_extractor(sentiment[0])) #get the dictionary of fea
                    if int(sentiment[1]) >= 0: # Club the sentiment values (-5 to + 5) to just
                        new_row.append('positive')
                    else:
                        new_row.append('negative')
                    feature_set.append(new_row)
                print(feature_set)

        def main():
            sentiment_csv = "C:/Users/kmpoo/Dropbox/HEC/Teaching/Python for PhD Mar 2018/pytho
            ML_train(sentiment_csv)
```

```
main()
```

Step 2: Split the feature set into training and testing sets We will split the feature data set into training and test data sets. The training set is used to train our ML model and then the testing set can be used to check how good the model is. It is normal to use 20% of the data set for testing purposes. In our case we will retain 1500 words for training and the rest for testing.

```
In [ ]: import csv
        import random

        def feature_extractor(word):
            """Extract the features for a given word and return a dictonary of the features"""
            start_letter = word[0]
            last_letter = word[-1]
            return {'start_letter' : start_letter,'last_letter' : last_letter}

        def ML_train(sentiment_corpus):
            """Create feature set from the corpus given to to it. Split the  feature set into
            feature_set = []
            with open(sentiment_corpus,'rt',encoding = 'utf-8') as sentobj:
                sentiment_handle = csv.reader(sentobj)

                for sentiment in sentiment_handle:
                    new_row = []
                    new_row.append(feature_extractor(sentiment[0])) #get the dictionary of fea
                    if int(sentiment[1]) >= 0: # Club the sentiment values (-5 to + 5) to just
                        new_row.append('positive')
                    else:
                        new_row.append('negative')
                    feature_set.append(new_row)

                random.shuffle(feature_set)
                # We need to shuffle the features since the word_sentiment.csv had words arran
                train_set = feature_set[:1500] #the first 1500 words becomes our training set
                test_set = feature_set[1500:]
                print(len(test_set))

        def main():
            sentiment_csv = "C:/Users/kmpoo/Dropbox/HEC/Teaching/Python for PhD Mar 2018/pytho
            ML_train(sentiment_csv)

        main()
```

Step 3: Use ML method (naive Bayes) to create the classifier model The NLTK module gives us several ML methods to create a classifier model using our training set and based on our selected features.

```
In [ ]: import csv
        import random
        import nltk

        def feature_extractor(word):
            """Extract the features for a given word and return a dictonary of the features"""
            start_letter = word[0]
            last_letter = word[-1]
            return {'start_letter' : start_letter,'last_letter' : last_letter}

        def ML_train(sentiment_corpus):
            """Create feature set from the corpus given to to it. Split the  feature set into
            Train the classifier using the naive Bayes model and return the classifier. """
            feature_set = []
            with open(sentiment_corpus,'rt',encoding = 'utf-8') as sentobj:
                sentiment_handle = csv.reader(sentobj)

                for sentiment in sentiment_handle:
                    new_row = []
                    new_row.append(feature_extractor(sentiment[0])) #get the dictionary of fea
                    if int(sentiment[1]) >= 0: # Club the sentiment values (-5 to + 5) to just
                        new_row.append('positive')
                    else:
                        new_row.append('negative')
                    feature_set.append(new_row)

                random.shuffle(feature_set)
                # We need to shuffle the features since the word_sentiment.csv had words arran
                train_set = feature_set[:1500] #the first 1500 words becomes our training set
                test_set = feature_set[1500:]
                classifier = nltk.NaiveBayesClassifier.train(train_set)
                # Note: to create the classifier we need to provide a dictonary of features an
                return classifier

        def main():
            sentiment_csv = "C:/Users/kmpoo/Dropbox/HEC/Teaching/Python for PhD Mar 2018/pytho
            classifier = ML_train(sentiment_csv)
            input_word = input('Enter a word ').lower()
            sentiment = classifier.classify(feature_extractor(input_word))
            print('Sentiment of word "', input_word,'" is : ',sentiment)

        main()
```

Step 4: Testing the model Find how good the model is in identifying the labels. Ensure that the test set is distinct from the training corpus. If we simply re-used the training set as the test set, then a model that simply memorized its input, without learning how to generalize to new examples, would receive misleadingly high scores. The function nltk.classify.accuracy() will calculate the

accuracy of a classifier model on a given test set.

```
In [ ]: import csv
        import random
        import nltk

        def feature_extractor(word):
            """Extract the features for a given word and return a dictonary of the features"""
            start_letter = word[0]
            last_letter = word[-1]
            return {'start_letter' : start_letter,'last_letter' : last_letter}

        def ML_train(sentiment_corpus):
            """Create feature set from the corpus given to to it. Split the  feature set into
            Train the classifier using the naive Bayes model and return the classifier. """
            feature_set = []
            with open(sentiment_corpus,'rt',encoding = 'utf-8') as sentobj:
                sentiment_handle = csv.reader(sentobj)

                for sentiment in sentiment_handle:
                    new_row = []
                    new_row.append(feature_extractor(sentiment[0])) #get the dictionary of fea
                    if int(sentiment[1]) >= 0: # Club the sentiment values (-5 to + 5) to just
                        new_row.append('positive')
                    else:
                        new_row.append('negative')
                    feature_set.append(new_row)

                random.shuffle(feature_set)
                # We need to shuffle the features since the word_sentiment.csv had words arran
                train_set = feature_set[:1500] #the first 1500 words becomes our training set
                test_set = feature_set[1500:]
                classifier = nltk.NaiveBayesClassifier.train(train_set)
                # Note: to create the classifier we need to provide a dictonary of features an
                print('Test accuracy of the classifier = ',nltk.classify.accuracy(classifier,
                print(classifier.show_most_informative_features())
                return classifier

        def main():
            sentiment_csv = "C:/Users/kmpoo/Dropbox/HEC/Teaching/Python for PhD Mar 2018/pytho
            classifier = ML_train(sentiment_csv)
            input_word = input('Enter a word ').lower()
            sentiment = classifier.classify(feature_extractor(input_word))
            print('Sentiment of word "', input_word,'" is : ',sentiment)

        main()
```

Excercise *Improve the feature extractor (by adding new features) so that the test accuracy can go up to atleast 70%.*

```
In [ ]: #Enter code here
```

```
        #
```

Step 5: Development testing and error analysis Using a seperate dev-test set, we can generate a list of the errors that the classifier makes when predicting the sentiment. We can then examine individual error cases where the model predicted the wrong label, and try to determine what additional pieces of information would allow it to make the right decision (or which existing pieces of information are tricking it into making the wrong decision). The feature set can then be adjusted accordingly.