Search Engine Prototype

Jaume Cloquell Capo

Resumen

This project contain all the documentation of the search engine prototype.

1. Introduction

This project will use scikit-learn (Naive Bayes, MultinomialNB, CountVectorizer, TfidfTransformer, Bagging, Boosting) on Numpy and Pandas for the data frame;

2. Preprocess

- converting all letters to lower or upper case **DONE**
- converting numbers into words or removing numbers **TO DO**
- removing punctuations, accent marks and other diacritics **DONE**
- removing white spaces **DONE**
- expanding abbreviations **TO DO**
- removing stop words, sparse terms, and particular words **DOING**
  - Stemming: Stemming is a process of reducing words to their word stem, base or root form (for example, books—book, looked—look) **DONE**
  - Lemmatization: he aim of lemmatization, like stemming, is to reduce inflectional forms to a common base form. As opposed to stemming, lemmatization does not simply chop off inflections. Instead it uses lexical knowledge bases to get the correct base forms of words.**TO DO**
  - POS: Part-of-speech tagging aims to assign parts of speech to each word of a given text (such as nouns, verbs, adjectives, and others) based on its definition and its context. **TO DO**

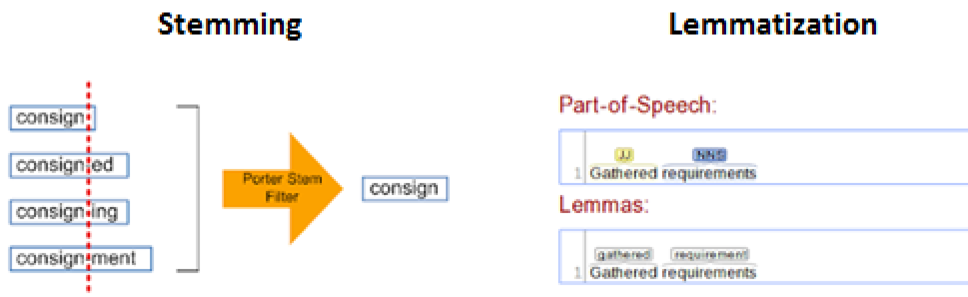

Figura 1: STEMMING VS Lemmatization

```
#Filter only lang == EN
data_en = data[data.locale == "en-US"]
# Remove empty docs
data_en= data_en[(data_en.medical_dictionary.notnull())]
data = data_en[['id_specialization','medical_dictionary']]
#data.medical_dictionary = data.medical_dictionary.apply(remove_html_tags)
dicttionary = {}
for index, row in data.iterrows():
    # TODO: Remove words lower 1
    # Converting to Lowercase
    document =row.medical_dictionary
    document = remove_html_tags(document)
    document = document.lower()
    # Lemmatization
    document = document.split()
    document = [stemmer.lemmatize(word) for word in document]
    document = ' '.join(document)
    dicttionary[row['id_specialization']] = [document]
```

3. Word2Vec and Doc2Ve

Google developed a method called Word2Vec that captures the context of words, while at the same time reducing the size of the data. Word2Vec is actually two different methods: Continuous Bag of Words (CBOW) and Skip-gram.

The above diagram is based on the CBOW model, but instead of using just nearby words to predict the word, we also added another feature vector, which is document-unique. So when training the word vectors W, the document vector D is trained as well, and in the end of training, it holds a numeric representation of the document.

The inputs consist of word vectors and document Id vectors. The word vector is a one-hot vector with a dimension 1xV. The document Id vector has a dimension of 1xC, where C is the number of total documents. The dimension of the weight matrix W of the hidden layer is VxN. The dimension

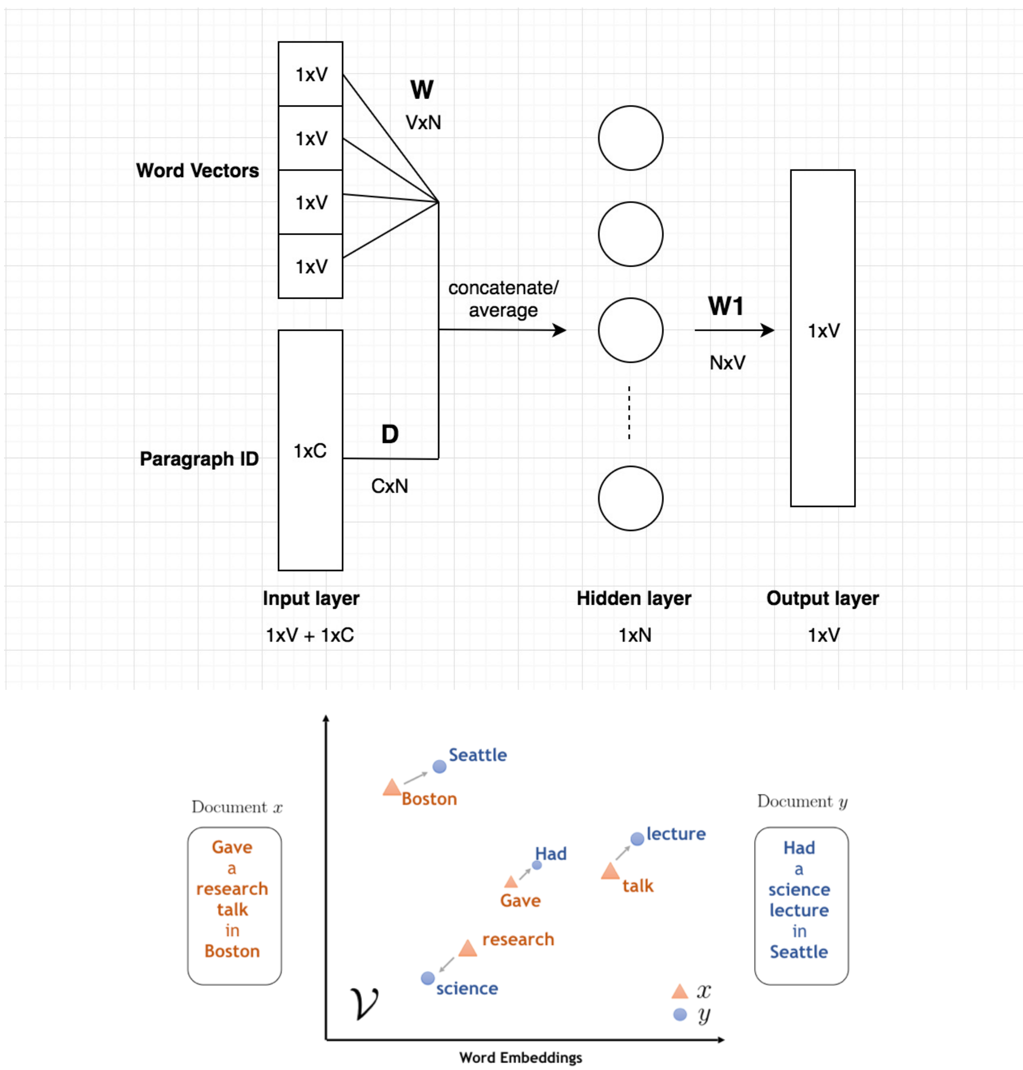

Figura 2: Doc2Vec

of the weight matrix D of the hidden layer is CxN.

```python
train_documents = [TaggedDocument(data_en['tokens'][ind], str(data_en['id_specialization'][ind]))
    for ind in data_en.index]

    cores = multiprocessing.cpu_count()
    epochs = 1000

    simple_models = [
        # PV-DM w/concatenation - window=5 (both sides) approximates paper's 10-word total window size
        Doc2Vec(dm=1, dm_concat=1, vector_size=100, hs=0, min_count=2, workers=cores,epochs=epochs),
        # PV-DBOW
        Doc2Vec(dm=0, vector_size=100, hs=0, min_count=2, workers=cores,epochs=epochs),
        # PV-DM w/average
        Doc2Vec(dm=1, dm_mean=1, vector_size=100, hs=0, min_count=2, workers=cores,epochs=epochs),
    ]

    model = simple_models[2]

    model.build_vocab([x for x in train_documents])

    model.train(train_documents, total_examples=model.corpus_count, epochs=model.epochs)
```

4. Models - Pros / Cons (Generalized)

- KNN
  - Pros
    - Is a non-parametric
    - Variety of distance criteria to be choose from
  - Cons
    - Define K
    - Slow algorith
    - Imbalanced data causes problems
    - Outlier sensitivity
- Naive Bayes
  - Pros
    - Computationally fast

        - Simple to implement
        - Works well with high dimensions

    - Cons
        - Relies on independence assumption and will perform badly if this assumption is not met

- Random Forest
    - Pros
        - reduced variance
    - Cons
        - Not as easy to visually interpret

- Bagged Trees : train multiple trees using bootstrapped data to reduce variance and prevent overfitting
    - Pros
        - Reduces variance in comparison to regular decision trees
    - Cons
        - Does not reduce variance if the features are correlated

- Boosted Trees : Similar to bagging, but learns sequentially and builds off previous trees
    - Pros
        - Somewhat more interpretable than bagged trees/random forest as the user can define the size of each tree resulting in a collection of stumps (1 level) which can be viewed as an additive model
    - Cons
        - Unlike bagging and random forests, can overfit if number of trees is too large

5. Code

$$\mathbf{tf}(t,d) = \frac{f_d(t)}{\max\limits_{w \in d} f_d(w)}$$

$$\mathbf{idf}(t,D) = \ln\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

$$\mathbf{tfidf}(t,d,D) = \mathbf{tf}(t,d) \cdot \mathbf{idf}(t,D)$$

$$\mathbf{tfidf}'(t,d,D) = \frac{\mathbf{idf}(t,D)}{|D|} + \mathbf{tfidf}(t,d,D)$$

$f_d(t) :=$ frequency of term t in document d

$D :=$ corpus of documents

Figura 3: Tf idf

```
trclass SearchEngine():
    def __init__(self, label_names, X_train, y_train):
        self.k = len(y_train) # K is the number of clases, in this case, specializations
        self.label_names = label_names
        self.X_train, self.y_train = X_train, y_train

    def fit(self):
        # min_df: This corresponds to the minimum number of documents that should contain this feature.
        # max_df: we should include only those words that occur in a maximum of 70% of all the documents
        self.vectorizer = CountVectorizer(ngram_range=(1, 1), max_features=1500, min_df=5, max_df=0.4, stop_words=stopwords.words('english'))

        X_train_vect = self.vectorizer.fit_transform(self.X_train)
        self.tfidf_transformer = TfidfTransformer()
        X_train_trans = self.tfidf_transformer.fit_transform(X_train_vect)

        #self.classifier = KNeighborsClassifier(n_neighbors=self.k)
        #self.classifier = RandomForestClassifier(n_estimators=500, max_features=0.25, criterion="entropy", class_weight="balanced")
        #self.classifier = BaggingClassifier(n_estimators =25, max_features=0.25)
        #self.classifier = GradientBoostingClassifier(n_estimators =100, learning_rate =0.1, max_depth=6, min_samples_leaf =1, max_features=1.0) clf.fit(X, training_set_y)
        self.classifier = MultinomialNB()

        self.classifier.fit(X_train_trans, self.y_train)

    def predict(self, X_test):
        X_test_vect = self.vectorizer.transform(X_test)
        X_test_trans = self.tfidf_transformer.transform(X_test_vect)
        y_pred = self.classifier.predict(X_test_trans)
        return y_pred
```

```
    def predict_single(self, doc):
        X_test_vect = self.vectorizer.transform([doc])
        X_test_trans = self.tfidf_transformer.transform(X_test_vect)
        y_pred = zip(self.classifier.classes_, self.classifier.predict_proba(X_test_trans)[0])
        y_pred = sorted([(self.label_names[ind], score) for ind, score in y_pred], key=lambda
x: -x[1])
        return y_pred

    def report(self, X_test, y_test, y_pred):
        print(classification_report(y_test, y_pred, target_names=self.label_names, digits=4))

        total = 0
        same = 0
        for i in range(len(y_test)):
            if y_test[i] == y_pred[i]:
                same += 1
            total += 1
        print(total, same)
```

6. What I would do

- MultiLanguage
  - MultiLanguage embeddings `https://github.com/facebookresearch/MUSE`
  - Diferent modals
- Add articles to the dataset
- Visit
  - ElasticSearch: `https://www.elastic.co/es/`
  - Spacy `https://spacy.io/`
  - Keras workshop `https://github.com/tensorflow/workshops/blob/master/extras/keras-bag-of-` `keras-bow-model.ipynb`
- POS
- TF-IDF(document) = TF-IDF(title) * alpha + TF-IDF(body) * (1-alpha)
  - Calculate TF-IDF for Body for all docs
  - Calculate TF-IDF for title for all docs
  - Multiply the Body TF-IDF with alpha