Assignment 4

Preparing the data sets

In [23]:

```
import pandas as pd
import string
import spacy
from math import log10
from collections import Counter
from nltk.corpus import stopwords
from iwnlp.iwnlp_wrapper import IWNLPWrapper
```

We use the Pandas library to work with the data. In the cell below, we populate `COLUMN_LABELS` with the relevant columns and then import the training and test data sets. Then we print the first five elements to visualize the data structure.

In [3]:

```
COLUMN_LABELS = ['Game Name', 'Class', 'Title', 'Review Text']
train = pd.read_csv('games-train.csv', sep='\t', names=COLUMN_LABELS)
test = pd.read_csv('games-test.csv', sep='\t', names=COLUMN_LABELS)
```

In [4]:

```
train.head()
```

Out[4]:

	Game Name	Class	Title	Review Text
0	Hay Day	gut	NaN	Spaß pur
1	Bike Race Free	gut	NaN	Top game mit sucht Potenzial
2	Subway Surfers	gut	Gut	Es lagt manchmal
3	Subway Surfers	gut	NaN	Es ist ein tolles Spiel aber manchmal bleibt e...
4	Hay Day	gut	NaN	Ccccccccoooooooooooooooo ooooooooooll

In [5]:

```
test.head()
```

Out[5]:

	Game Name	Class	Title	Review Text
0	Farmville 2	schlecht	NaN	Echt schlecht , immer wen ich versuche zu star...
1	Die Simpsons	gut	Buchi0202136	Suche noch freunde zum hinzufuegen
2	Die Simpsons	gut	Suchtgefähr :) !!	Ich find das Spiel gut,man muss nicht permanen...
3	Die Simpsons	gut	Dauerhafter Spaß...	... durch immer neue Events. Schon 1 1/2 Jahre...
4	Subway Surfers	gut	Great	I like the game but near the last update it st...

Creating the Model

Preprocessing

We use the SpaCy library for tokenization and a SpaCy extension class for German lemmatization. We then define a wrapper around the `lemmatizer`'s `lemmatize` function to fit our needs.

In [35]:

```
lemmatizer = IWNLPWrapper(lemmatizer_path='IWNLP.Lemmatizer_20181001.json')
nlp = spacy.load('de')
```

In [5]:

```
def lemmatize(token):
    """
    This function is a wrapper for the above lemmatizer. It modifies two
    behaviors:

        - when the lemmatizer cannot confidently predict a lemma, it returns
          None; this method returns the original token.
        - when the lemmatizer finds more than one possible lemma, it returns
          a list of the potential lemmmas; this method always chooses the first
          option.

    Args:
        token: a spacy.Token object representing a single token

    Returns:
        the first element in the lemma list or else the original token
    """
    lem = lemmatizer.lemmatize(str(token), pos_universal_google=token.pos_)
    if not lem:
        return token
    else:
        return lem[0]
```

In [30]:

```
def preprocess(doc):
    """
    Cleans a document by:
    - ensuring that it is indeed a string
    - converting it to lowercase
    - removing punctuation
    - removing German stopwords

    Args:
        doc: a given string

    Returns:
        an array containing cleaned and tokenized terms for the string
    """
    doc = str(doc).lower() # str() in case Pandas imported number as int type
    doc = doc.translate(str.maketrans('', '', string.punctuation)).strip()
    return [lemmatize(token) for token in nlp(doc)
            if token.text not in stopwords.words('german')]
```

In [8]:

```
def estimate_parameters(docs, collection_size): # docs = docs belonging to one c
lass
    """
    Estimates the parameters of a given class.

    Args:
        docs: a collection of lists of preprocessed terms
        collection_size: the size of the overall collection

    Returns:
        a tuple containing the following two variables:

            - p_y: the portion of the overall collection that these docs make up
            - count: a frequency distribution of terms in the docs
    """
    p_y = len(docs) / collection_size
    count = Counter()
    for doc in docs:
        count.update(preprocess(doc))

    return (p_y, count)
```

Now let's estimate the parameters for each class in the data sets. We use Python's dict comprehension to map every unique value in the training data's class column (in this case just 'gut', 'schlecht') to the result of running `estimate_parameters` on a data set containing only elements of that class. For added clarity, we enumerate the variables below.

- `params` = a dictionary of class to frequency distribution of terms in class
- `class_` = a string containing either "gut" or "schlecht"
- `train` = the training data as a `DataFrame`

In [42]:

```
params = {
    class_: estimate_parameters(
        train[train['Class'] == class_]['Review Text'], # Gets only the text of
the review
        len(train)
    ) for class_ in train['Class'].unique() # = ['gut', 'schlecht']
}
```

`p_y` for each class, or the distribution of each class.

In [43]:

```
print(params['gut'][0], params['schlecht'][0])
```

```
0.8230904656534169 0.1769095343465831
```

The most common words in the "*gut*" frequency distribution.

In [44]:

```
params['gut'][1].most_common(10)
```

Out[44]:

```
[('machen', 14616),
 ('besser', 11035),
 ('geil', 10941),
 ('Spiel', 10281),
 ('spielen', 8567),
 ('gut', 6532),
 ('echt', 6217),
 ('cool', 5736),
 ('finden', 5179),
 ('Spaß', 4662)]
```

The 10 most common words in the "*schlecht*" frequency distribution.

In [45]:

```
params['schlecht'][1].most_common(10)
```

Out[45]:

```
[('kommen', 3741),
 ('Update', 3158),
 ('beheben', 3073),
 ('gehen', 2851),
 ('spielen', 2685),
 ('Spiel', 2399),
 ('neu', 2387),
 ('machen', 2116),
 ('Stern', 1902),
 ('bekommen', 1766)]
```

Using the Model to Predict Class

In [7]:

```
def predict(test_doc, parameters):
    """Predicts the most probable class for a document."""
    test_doc = preprocess(test_doc)
    probs = []
    for class_, params in parameters.items():
        tokens_prob = 0
        p_y = params[0]
        counter = params[1]
        for token in test_doc:
            token_rel_freq = counter[token] / sum(counter.values())
            if token_rel_freq == 0:
                continue
            tokens_prob += log10(token_rel_freq)

        class_prob = log10(p_y) + tokens_prob
        probs.append((class_, abs(class_prob)))

    return min(probs, key=lambda x: x[1])[0]
```

Let's begin by testing the `predict` method on some easy examples.

In [47]:

```
predict('tolles Spiel', params)
```

Out[47]:

```
'gut'
```

In [48]:

```
predict('das Spiel stürtzt immer ab. bitte schnell beheben', params)
```

Out[48]:

```
'schlecht'
```

On both examples, it acted just as we would expect. Now let's move on to the test data set. Let's assign `result` to a `Series` equal to the the prediction of each row in the "Review Text" column of `test`, then print the first five results.

In [49]:

```
pred = test['Review Text'].apply(lambda x: predict(x, params))
```

In [50]:

```
pred.head()
```

Out[50]:

```
0    schlecht
1         gut
2    schlecht
3         gut
4         gut
Name: Review Text, dtype: object
```

To visualize what this looks like, we'll create a `DataFrame` of the two sequences of predicted and true values, then print the first five rows.

In [51]:

```
joined = pd.concat([test['Class'], pred], axis=1)
joined.columns = ['True', 'Predicted']
joined.head()
```

Out[51]:

	True	Predicted
0	schlecht	schlecht
1	gut	gut
2	gut	schlecht
3	gut	gut
4	gut	gut

Evaluation

In [6]:

```
def evaluate(target_class, true, predicted):
    """
    Evaluates the precision, recall, and f-score of the model on a specific
    class. It then returns a tuple containing those values as well as the
    intermediate values of instances in which the model guessed true and was
    correct, `tp`, instances in which it guessed true and was wrong, `fp`, and
    instances in which it guessed false and was wrong.

    Args:
        target_class: the target class
        true: a list-like object containing the gold standard
        predicted: a list-like object of the same shape containing the model's p
redictions

    Returns:
        a tuple containing true positives, false positives, false negatives,
        precision, recall, and f-score.
    """
    if len(true) != len(predicted):
        raise ValueError('Sequences are of different lengths.')
    evl = pd.DataFrame(list(zip(true, predicted)), columns=['True', 'Predicted']
)
    tp = len(evl[(evl['Predicted'] == target_class) & (evl['True'] == target_cla
ss)])
    fp = len(evl[(evl['Predicted'] == target_class) & (evl['True'] != target_cla
ss)])
    fn = len(evl[(evl['Predicted'] != target_class) & (evl['True'] == target_cla
ss)])
    prec = tp / (tp + fp)
    recall = tp / (tp + fn)
    fscore = (2 * prec * recall) / (prec + recall)
    return tp, fp, fn, prec, recall, fscore
```

In [53]:

```
print('gut:', evaluate('gut', test['Class'], pred))
print('schlecht:', evaluate('schlecht', test['Class'], pred))
```

```
gut: (33282, 3142, 3134, 0.913738194596969, 0.913938927943761, 0.913
838550247117)
schlecht: (4675, 3134, 3142, 0.5986682033551031, 0.5980555200204682,
0.5983617048508895)
```