Outcome of the project:

Your code (a Colab notebook or a python file)

A report of about 3-pages: latex and doc templates are available on Moodle

The report must contain: Dataset and task explanation, Methods used, Results

We will consider Kaggle challenges for our projects • https://www.kaggle.com/ • If you don’t have an account, register to the platform • On Kaggle you can download the dataset, create a notebook (or you can work on Colab), see other people’s code, and submit your predictions on the test set • If you don’t know how to create a file with your predictions, you can use 10% of the training as test set and report results on that • The project can be done in groups of 2 people

Disaster Tweets:

The ubiquitousness of smartphones enables people to announce an emergency they’re observing in realtime

More agencies are interested in programatically monitoring Twitter (i.e. disaster relief organizations and news agencies).

Binary Classification Task: 1 disaster tweet, 0 not a disaster tweet

* Input data (~8500 examples):
* The text of a tweet
* A keyword from that tweet (although this may be blank)
* The location the tweet was sent from (may also be blank)
* <https://www.kaggle.com/c/nlp-gettingstarted/overview>

Text Representation

Bag of words:

Table

Description automatically generated with medium confidence

Tokenization

Graphical user interface, text, application, email

Description automatically generated

Normalization

A screenshot of a computer

Description automatically generated with medium confidence

Diagram

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

Term-Frequency Based Representation

Table

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text Preprocesing: <https://en.wikipedia.org/wiki/Tf%E2%80%93idf>

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, table, Excel

Description automatically generated

NLP in Python

Regular expressions:

* Strings with special syntax
* Pattern: series of letters or symbols which can map to actual text
* Import re
* Re.match(‘abc’, ‘abcdef’) returns a match object
* Common regex patterns: ‘\w+’ matches a word, ‘\d’ for digits, ‘\s’ space, ‘.\*’ as a wildcard that matches anything
* + or \* for a greedy match just as ‘aaaaa’
* ‘\S’ for ‘not a space’
* [a-z] lowercase group
* Split, findall, search, match
* Split a string on regex, find all patterns in a string, search for a pattern, match an entire string or substring based on a pattern
* These methods take pattern first, string second
* Text

  Description automatically generated
* Text

  Description automatically generated

Tokenization

* One step in preparing a text
* Some examples: break out words or sentences, separate punctuation or all hashtags in a tweet
* Nltk: from nltk.tokenize import word\_tokenize

Word\_tokenize(‘Hi there!’)

* Can help with matching common words, removing unwanted words
* sent\_tokenize: tokenize a document into sentences
* regexp\_tokenize: tokenize based on a regular expression pattern
* TweetTokenizer: special class just for tweet tokenization allowing me to separate hashtags, mentions and lots of exclamation points
* Re.search() and re.match(): the difference is:
  + When we search for a patter later in the string, the methods will lead to different outputs
  + For a pattern that isn’t in the beginning of the string, use search
* # Search for the first occurrence of "coconuts" in scene\_one: match
* match = re.search("coconuts", scene\_one)
* # Print the start and end indexes of match
* print(match.start(), match.end())
* regex groups using |
  + we can define a group using ()
  + or explicit character ranges using []
  + match\_digits\_and\_words = (‘(\d+|\w+)’)
  + [A-Za-\]+ matches upper and lowercase English alphabet
  + [0-9] matches nums from 0 to 9
  + [A-Za-z\-\.]+ matches upper, lowercase alphabet, - and .
  + (a-z) matches a, - and z
  + (\s+|,) matches spaces or a comma
* *Unlike the syntax for the regex library, with nltk\_tokenize() you pass the pattern as the****second****argument.*
* # Import the necessary modules
* from nltk.tokenize import regexp\_tokenize
* from nltk.tokenize import TweetTokenizer
* # Define a regex pattern to find hashtags: pattern1
* pattern1 = r"#\w+"
* # Use the pattern on the first tweet in the tweets list
* hashtags = regexp\_tokenize(tweets[0], pattern1)
* print(hashtags)
* # Use the TweetTokenizer to tokenize all tweets into one list
* tknzr = TweetTokenizer()
* all\_tokens = [tknzr.tokenize(t) for t in tweets]
* print(all\_tokens)
* # Tokenize and print all words in german\_text
* all\_words = word\_tokenize(german\_text)
* print(all\_words)
* # Tokenize and print only capital words
* capital\_words = r"[A-Z\Ü]\w+"
* print(regexp\_tokenize(german\_text, capital\_words))
* # Tokenize and print only emoji
* emoji = "['\U0001F300-\U0001F5FF'|'\U0001F600-\U0001F64F'|'\U0001F680-\U0001F6FF'|'\u2600-\u26FF\u2700-\u27BF']"
* print(regexp\_tokenize(german\_text, emoji))

Charting word length

* from matplotlib import pyplot as plt
* words = word\_tokenize(‘This is a pretty cool tool!’)
* word\_lengths = [len(w) for w in words]
* plt.hist(word\_lengths)
* # Split the script into lines: lines
* lines = holy\_grail.split('\n')
* # Replace all script lines for speaker
* pattern = "[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
* lines = [re.sub(pattern, '', l) for l in lines]
* # Tokenize each line: tokenized\_lines
* tokenized\_lines = [regexp\_tokenize(s, r"\w+") for s in lines]
* # Make a frequency list of lengths: line\_num\_words
* line\_num\_words = [len(t\_line) for t\_line in tokenized\_lines]
* # Plot a histogram of the line lengths
* plt.hist(line\_num\_words)
* # Show the plot
* plt.show()

Bag of Words

# Import Counter

from collections import Counter

# Tokenize the article: tokens

tokens = word\_tokenize(article)

# Convert the tokens into lowercase: lower\_tokens

lower\_tokens = [w.lower() for w in tokens]

# Create a Counter with the lowercase tokens: bow\_simple

bow\_simple = Counter(lower\_tokens)

# Print the 10 most common tokens

print(bow\_simple.most\_common(10))

Text preprocessing:

* e.g. lowercasing words helps with tokenization to create a bag of words
* lemmatization/stemming
* word.isalpha() returns only alphabetical strings, so removes punctuation
* stopwords.words(‘english’) to remove fillers like the and is
* # Import WordNetLemmatizer
* from nltk.stem import WordNetLemmatizer
* # Retain alphabetic words: alpha\_only
* alpha\_only = [t for t in lower\_tokens if t.isalpha()]
* # Remove all stop words: no\_stops
* no\_stops = [t for t in alpha\_only if t not in english\_stops]
* # Instantiate the WordNetLemmatizer
* wordnet\_lemmatizer = WordNetLemmatizer()
* # Lemmatize all tokens into a new list: lemmatized
* lemmatized = [wordnet\_lemmatizer.lemmatize(t) for t in no\_stops]
* # Create the bag-of-words: bow
* bow = Counter(lemmatized)
* # Print the 10 most common tokens
* print(bow.most\_common(10))

Gensim

* a library for nlp using models
* <https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation>
* A corpus is a set of text used to help perform nlp tasks
* After preprocessing, we do: dictionary = Dictionary(tokenized\_docs) to begin building our corpus
* From genism.corpora.dictionary import Dictionary
* A normal corpus is usually a collection of words, but we can also build a special corpus using: corpus = [dictionary.doc2bow(doc) for doc in tokenized\_docs] transforming each document into a bag of words using token ids
* The output is a list of tuples, in which the first element is the token id and the second one is its frequency in the document
* # Import Dictionary
* from gensim.corpora.dictionary import Dictionary
* # Create a Dictionary from the articles: dictionary
* dictionary = Dictionary(articles)
* # Select the id for "computer": computer\_id
* computer\_id = dictionary.token2id.get("computer")
* # Use computer\_id with the dictionary to print the word
* print(dictionary.get(computer\_id))
* # Create a MmCorpus: corpus
* corpus = [dictionary.doc2bow(article) for article in articles]
* # Print the first 10 word ids with their frequency counts from the fifth document
* print(corpus[4][:10])
* # Save the fifth document: doc
* doc = corpus[4]
* # Sort the doc for frequency: bow\_doc
* bow\_doc = sorted(doc, key=lambda w: w[1], reverse=True)
* # Print the top 5 words of the document alongside the count
* for word\_id, word\_count in bow\_doc[:5]:
* print(dictionary.get(word\_id), word\_count)
* # Create the defaultdict: total\_word\_count
* total\_word\_count = defaultdict(int)
* for word\_id, word\_count in itertools.chain.from\_iterable(corpus):
* total\_word\_count[word\_id] += word\_count
* # Create a sorted list from the defaultdict: sorted\_word\_count
* sorted\_word\_count = sorted(total\_word\_count.items(), key=lambda w: w[1], reverse=True)
* # Print the top 5 words across all documents alongside the count
* for word\_id, word\_count in sorted\_word\_count[:5]:
* print(dictionary.get(word\_id), word\_count)

TF\_IDF

* Determine the most important words in the corpus
* Stopwords should be down-weighted in importance
* Ensures most common words don’t show up as key words
* Words that are contained in almost all documents, they will be downweighed
* From genism.models.tfidfmodel import TfidfModel
* The output weight can help us determine the importance of the words
* # Create a new TfidfModel using the corpus: tfidf
* tfidf = TfidfModel(corpus)
* # Calculate the tfidf weights of doc: tfidf\_weights
* tfidf\_weights = tfidf[doc]
* # Print the first five weights
* print(tfidf\_weights[:5])
* # Sort the weights from highest to lowest: sorted\_tfidf\_weights
* sorted\_tfidf\_weights = sorted(tfidf\_weights, key=lambda w: w[1], reverse=True)
* # Print the top 5 weighted words
* for term\_id, weight in sorted\_tfidf\_weights[:5]:
* print(dictionary.get(term\_id), weight)

Named Entity Recognition

* Used to recognize important names: places, people, organization, dates, states, works of art
* Can be used alongside topic identification
* The Stanford CoreNLP library – integrated in nltk / java based
* Preprocess via tokenization
* Tag for parts of speech: nltk.pos\_tag(sokenized\_sent)
* Nltk.ne\_chunk(tagged\_sent) – returns token with a label using a trained parser
* # Tokenize the article into sentences: sentences
* sentences = sent\_tokenize(article)
* # Tokenize each sentence into words: token\_sentences
* token\_sentences = [word\_tokenize(sent) for sent in sentences]
* # Tag each tokenized sentence into parts of speech: pos\_sentences
* pos\_sentences = [nltk.pos\_tag(sent) for sent in token\_sentences]
* # Create the named entity chunks: chunked\_sentences
* chunked\_sentences = nltk.ne\_chunk\_sents(pos\_sentences, binary = True)
* # Test for stems of the tree with 'NE' tags
* for sent in chunked\_sentences:
* for chunk in sent:
* if hasattr(chunk, "label") and chunk.label() == "NE":
* print(chunk)

# Create the defaultdict: ner\_categories

ner\_categories = defaultdict(int)

# Create the nested for loop

for sent in chunked\_sentences:

    for chunk in sent:

        if hasattr(chunk, 'label'):

            ner\_categories[chunk.label()] += 1

# Create a list from the dictionary keys for the chart labels: labels

labels = list(ner\_categories.keys())

# Create a list of the values: values

values = [ner\_categories.get(v) for v in labels]

# Create the pie chart

plt.pie(values, labels=labels, autopct='%1.1f%%', startangle=140)

# Display the chart

plt.show()

SpaCy

* Focus on creating nlp pipelines
* Open source
* Similar to gensim
* Displacy entity recognition visualizer
* Import spacy
* Pre-trained vectors are useful
* Nlp = spacy.load(‘en’)
* Nlp.entity is an entity recognizer object used to find entities in text
* Doc.ents 🡪 returns the identified entitities
* Doc.ents[0], doc.ents[0].label\_ 🡪 get the entity
* Spacy comes with informal language corpora
* # Import spacy
* import spacy
* # Instantiate the English model: nlp
* nlp = spacy.load('en', tagger = False, parser = False, matcher = False)
* # Create a new document: doc
* doc = nlp(article)
* # Print all of the found entities and their labels
* for ent in doc.ents:
* print(ent.label\_, ent.text)

Polyglot

* NLP library uses word vectors
* Supports many languages
* Used for tasks like transliteration – translate text by swapping characters from one language to another
* Also for NER in other languages
* It has a language detection model

Classifying fake news with NLP

* Scikit-learn
* Using bag-of-words model or tf-idf
* From sklearn.feature\_extraction.text import CountVectorizer
  + Count\_vectoriezer = CountVectrizer(stop\_words = ‘english’)
  + Count\_train = count\_vectorizer.fit\_transform(X\_train.values)
  + Same for the test data
* # Import the necessary modules
* from sklearn.feature\_extraction.text import CountVectorizer
* from sklearn.model\_selection import train\_test\_split
* # Print the head of df
* print(df.head())
* # Create a series to store the labels: y
* y = df.label
* # Create training and test sets
* X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['text'], y, test\_size = 0.33, random\_state = 53)
* # Initialize a CountVectorizer object: count\_vectorizer
* count\_vectorizer = CountVectorizer(stop\_words = 'english')
* # Transform the training data using only the 'text' column values: count\_train
* count\_train = count\_vectorizer.fit\_transform(X\_train)
* # Transform the test data using only the 'text' column values: count\_test
* count\_test = count\_vectorizer.transform(X\_test)
* # Print the first 10 features of the count\_vectorizer
* print(count\_vectorizer.get\_feature\_names()[:10])
* # Import TfidfVectorizer
* from sklearn.feature\_extraction.text import TfidfVectorizer
* # Initialize a TfidfVectorizer object: tfidf\_vectorizer
* tfidf\_vectorizer = TfidfVectorizer(stop\_words = 'english', max\_df = 0.7)
* # Transform the training data: tfidf\_train
* tfidf\_train = tfidf\_vectorizer.fit\_transform(X\_train)
* # Transform the test data: tfidf\_test
* tfidf\_test = tfidf\_vectorizer.transform(X\_test)
* # Print the first 10 features
* print(tfidf\_vectorizer.get\_feature\_names()[:10])
* # Print the first 5 vectors of the tfidf training data
* print(tfidf\_train.A[:5])
* # Create the CountVectorizer DataFrame: count\_df
* count\_df = pd.DataFrame(count\_train.A, columns = count\_vectorizer.get\_feature\_names())
* # Create the TfidfVectorizer DataFrame: tfidf\_df
* tfidf\_df = pd.DataFrame(tfidf\_train.A, columns = tfidf\_vectorizer.get\_feature\_names())
* # Print the head of count\_df
* print(count\_df.head())
* # Print the head of tfidf\_df
* print(tfidf\_df.head())
* # Calculate the difference in columns: difference
* difference = set(count\_df.columns) - set(tfidf\_df.columns)
* print(difference)
* # Check whether the DataFrames are equal
* print(count\_df.equals(tfidf\_df))

Train and test a classification model

* naïve bayes model – given a piece of data, how likely is a particular outcome
* given ‘space’ and ‘alien’, how likely is the movie sci-fi
* each word from CountVectorizer is a feature
* from sklearn.naive\_bayes import MultinomialNB
* from sklearn import metrics
* nb\_classifier = MultinomialNB()
* nb\_classifier.fit(count\_train, y\_train)
* pred = nb\_classifier.predict(count\_test)
* metrics.accuracy\_score(y\_test, pred)
* Confusion matrix
  + Metrics.confusion\_matrix(y\_test, pred, labels = [0,1])
* # Import the necessary modules
* from sklearn import metrics
* from sklearn.naive\_bayes import MultinomialNB
* # Instantiate a Multinomial Naive Bayes classifier: nb\_classifier
* nb\_classifier = MultinomialNB()
* # Fit the classifier to the training data
* nb\_classifier.fit(count\_train, y\_train)
* # Create the predicted tags: pred
* pred = nb\_classifier.predict(count\_test)
* # Calculate the accuracy score: score
* score = metrics.accuracy\_score(y\_test, pred)
* print(score)
* # Calculate the confusion matrix: cm
* cm = metrics.confusion\_matrix(y\_test, pred, labels = ['FAKE', 'REAL'])
* print(cm)

# Create the list of alphas: alphas

alphas = np.arange(0,1,0.1)

# Define train\_and\_predict()

def train\_and\_predict(alpha):

    # Instantiate the classifier: nb\_classifier

    nb\_classifier = MultinomialNB(alpha = alpha)

    # Fit to the training data

    nb\_classifier.fit(tfidf\_train, y\_train)

    # Predict the labels: pred

    pred = nb\_classifier.predict(tfidf\_test)

    # Compute accuracy: score

    score = metrics.accuracy\_score(y\_test, pred)

    return score

# Iterate over the alphas and print the corresponding score

for alpha in alphas:

    print('Alpha: ', alpha)

    print('Score: ', train\_and\_predict(alpha))

    print()