Hacettepe University
Department Of Artificial Intelligence

AIN 442 Assignment 5 Report

Berke Bayraktar – 21992847

4.01.2021

1 Introduction

In this section I talk about problem of the assignment and explain the project goals.

In this assignment we are faced with a multinomial classification problem. More specifically, in this assignment we want to classify sentences into one of several topics. So a topic classification problem. Topic classification is an application of machine learning which is very relevant to daily life. A topic classification model could be used to determine what customers are talking about in their reviews, open-ended surveys and in social media. More generally, topic classification is also a type of text classification and considering the exponential growth of text data all around globe, topic classification is a very important task to study.

The specific problem we are faced in this assignments is the classification of Turkish sentences into one of 7 cateogories namely: siyaset, dünya, ekonomi, kültür, sağlık, spor teknoloji.

The main goal of this project is to perform the classification problem mentioned above by considering three main representations of text data, namely, LSA, word2vec and combination of these two representations. For all these 3 representations we also consider 3 classifier models, namely, logistic regression, gaussian naive bayes and random forest classifier. In this assignment we also consider the tuning of these models and I also tune the text representations mentioned earlier. This part will be explained more detailly in further sections.

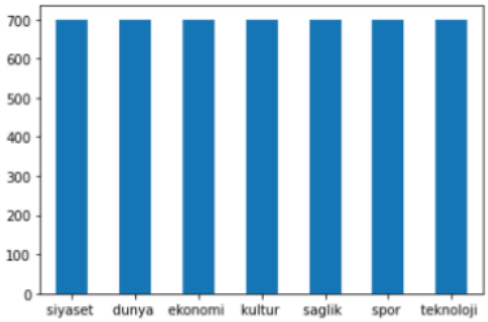
2 Data

In this section I briefly introduce the data

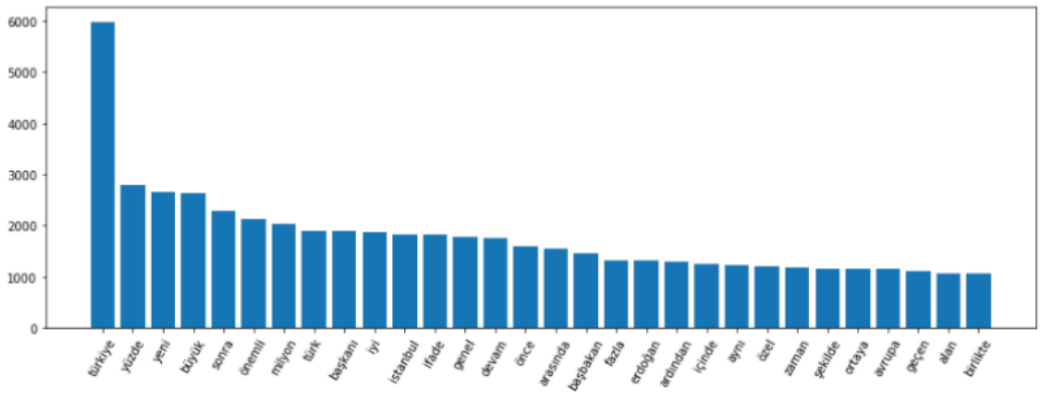
For this assignment we are provided a dataset of 4900 sentences where each sentence belongs to one of 7 categories mentioned above.

	category	text
0	siyaset	3 milyon ile ön seçim vaadi mhp nin 10 olağan...
1	siyaset	mesut_yılmaz yüce_divan da ceza alabilirdi pr...
2	siyaset	disko lar kaldırılıyor başbakan_yardımcısı ar...
3	siyaset	sarıgül anayasa_mahkemesi ne gidiyor mustafa_...
4	siyaset	erdoğan idamın bir haklılık sebebi var demek ...

We can see that data consist of only 2 columns one for the sentence (or the text) and other for the category that is assigned to that sentence. We can also see that some phrases contain underscores(_) that connect them together. Additionally we can see that suffixes are not split properly ("disko lar" example) and the suffix is considered as another word which we would not want so these and several other issues have to be dealt with in the preprocessing of the data.



And as for the category variable of the data we can see there are 7 categories as mentioned earlier and the data is balanced perfectly (each category contains 700 samples). So there won't be any need to handle data imbalance like in earlier assignments.

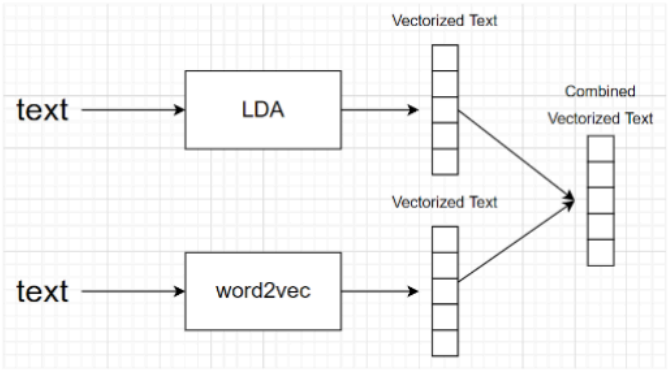


From this graph we can see the top-30 most occuring words along with their number of occurences. Not suprisingly the most occuring word, by far, is the word "türkiye" with 6000 occurunces. This word is followed by words like "yüzde, milyon" which are related to **ekonomi** and also words like "başkan, istanbul, başbakan, erdoğan, avrupa" which are related to **siyaset** and **dünya**.

3 Methodology

In this section I briefly explain the methodology that is used to solve the problem.

The methodology for this assignment consits of 2 main parts: text representation and classifier. The text representation is related to how we choose the represent the textual data. Textual data must be encoded somehow for computers and algorithms to understand and work with the data. In previous assignments we only considered representations that lacked semantic meaning like bag of words however in this assignment we consider approaches like LSA (latent semantic analysis), word2vec and combination of these which I call LSA2vec [1] for short.



This is for the text representation part. For the classifier part we consider 3 different classifiers: logistic regression, gaussian naive bayes and random forest. The reason I chose this models is beacause I am familiar with their theory. I could also use models like support vector machines and neural networks which are very easy to train and use thanks to libraries like with sci-kit

[1] The name LSA2vec is not used to refer to actual work we discussed in the class but just a combination of LSA and word2vec

learn however since I am not as familiar with those models as I am with 3 previously mentioned models. So I chose to use these 3 models. And the choice of gaussian naive bayes was because we work with continous data. Unlike bag of words or tf-idf, encodings for text data are continous with methods like LSA and word2vec. So contrary to previous assignments we use gaussian naive bayes instead of regular multinomial naive bayes as classifier.

For the tuning of the models. For each text representation (LSA, word2vec, LSA2vec) the 2 models are tuned (logistic regression, random forest classifier) the other model gaussian naive bayes only has one parameter to tune (var_smoothing) and I don't know what exactly it does so I decided not to tune this model.

In addition to tuning of these 2 classifiers I also tune the text representations more specifically the parameters of CountVectorizer and word2vec model from gensim is tuned for better results.

And finally, in addition to all these I try 2 different preprocessing techniques on the dataset. Both contain steps like removal of stopwords, punctuation, etc. however one of the techniques also apply normalization, stemming and lemmatization to the words. I run the code with one version of the dataset then apply additional preprocessing said earlier and run the code again. For this reason it is not possible to see both resutls at the same time on the ipynb file but I will include them in the report.

4 Development

4.1 Project Plan

In this section I explain the methodology in more detail by considering each step that has to be performed and explain why exactly these steps are selected considering requirements and limitations without going into implementation.

Step 1 Preprocessing: First step in this project is the preprocessing of the data which is done with and without normalization, stemming and lemmatization. I do this because these 3 operations take long time and I want to compare whether performing these operations actually provide any significant performance gains.

Step 2 LSA Vectorization: Second step is LSA vectorization of data. In this step TruncatedSVD model is used with *components=7*. This is LSA is a topic modelling method and encodes text data as a combination of topics and there are 7 different topics for this assignment so 7 is most natural value for this parameter. I was initally thinking giving different values to this components parameter to check whether if there would be any performance gains but due to time limitations I decided to skip it and just use the static value 7. However as a guess probably the value 7 would perform the best since the number of topics are 7.

The following three sub steps (2.1, 2.2 and 2.3) are related to tuning and testing classifiers where text data is static and is vectorized with LSA.

Step 2.1 Logistic Regression Tuning and Testing: 2 parameters are tuned *max_iter* and *penalty* initally I also wanted to tune *solver* parameter however some combinations of *solver* and *max_iter* caused convergence issues due to limited computation power so I skipped tuning of *solver* parameter. At the end of this step logistic regression model is tested on the actual test data rather than cross validation data. To do so the model that performed best on cross validation is selected.

Step 2.2 Gaussian Naive Bayes Testing: Unlike step 2.1 for gaussian naive bayes there is no tuning and the model is directly tested on test data to obtain a score. This is because there is only one parameter on we can tune on gaussian naive bayes and it is not worth the effort and I don't think it would actually provide any significant gain since parameter is just related to variable smoothing to avoid zero division cases when performing naive bayes.

Step 2.3 Random Forest Tuning and Testing: For random forest, 3 parameters are tuned namely, *criterion*, *max_depth* and *bootstrap*. Later the model that performed best on cross validation is selected and used for obtaining a score on test data.

Step 2.4 Vectorizer Tuning and Testing: After tuning and selecting best model out of 3 classifiers above, the vectorizer itself is also tuned. I use CountVectorizer for this assignment so the parameters that are tuned for the vectorizer are: *min_df, max_df, max_features*. Later best text representation is selected and used for obtaining a score on test data.

Step 3: Word2vec Vectorization: Similar to step 2 this step is related to text representation. In this step instead of LSA vectorization, word2vec vectorization is used. Word2vec is a model that encodes words and not sentences. There are 2 easy way to obtain sentence encodings from word encodings: 1) concatenating word vectors, 2) summing or averaging word vectors. One disadvantage of method 1is that it can't handle variable length sequences meaning that it would only work if the number of words on each sentence are the same, otherwise each sentence will have different vector sizes and in that case we can't pass the sentences into a classifier. As for the method 2 no matter how many words there are the size of the sentence vector will be the same as the individual word vectors since these vectors are summed or averaged element-wise. As for the distinction between summing vs averaging, averaging the vectors makes more sense to make sure that values are normalized.

Apart from these 2 approaches where sentence vectors are obtained using individual word vectors we could directly vectorize the sentences using approaches like doc2vec however I think that method is out of scope of this assignment so I decided not to try such vectorization.

Step 3.1, 3.2, 3.3 Tuning and Testing Classifiers: These steps are same with steps 2.1, 2.2, 2.3.

Step 3.4 Vectrorizer Tuning and Testing: Similar to step 2.4 the vectorizer which converts text to vectors are tuned. However in step 2.4 CountVectorizer is used to obtain a representation form LSA method whereas in step 3 word2vec vectorization is used and therefore in this step word2vec model is tuned and tested. Parameters that are tuned are *window* and *vector_size*.

Step 4: LSA2vec Vectorization: Similar to steps 1 and 2 this step is related to text representation. In this step vectorizations from previous steps (LSA and word2vec) are combined together to form a new representation and as explained earlier this is done by concatenation of the 2 vectors.

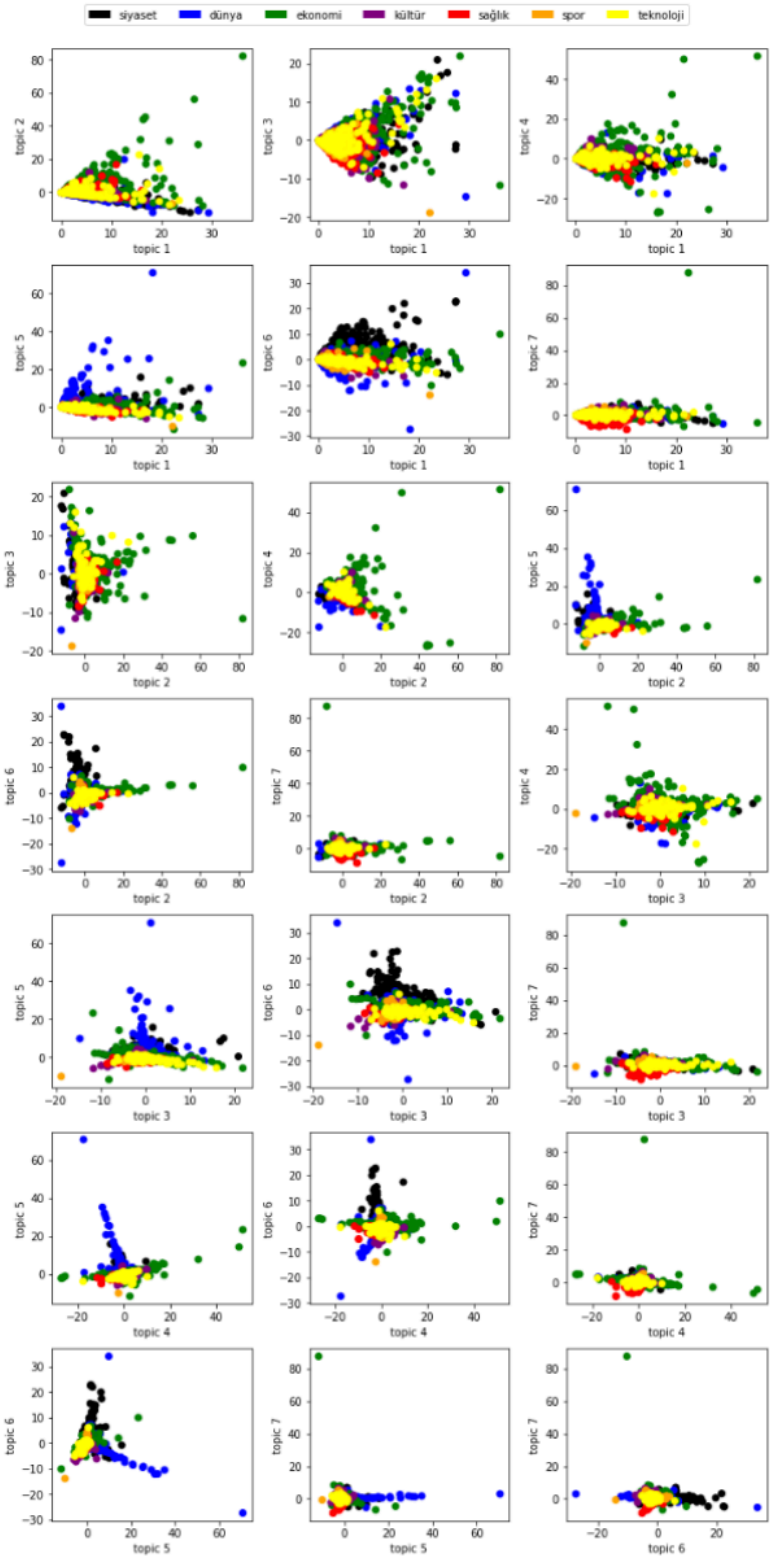
Step 4.1, 4.2, 4.3 Tuning and Testing Classifiers: These steps are same with steps 2.1, 2.2, 2.3.

Note: Step 4 doesn't have a step for tuning vectorizer this is because best vectorizers from 2.4 and 3.4 are already selected and concatenated so no need for further tuning of vectorizer in.

4.2 Analysis

In this section I explain the features that are available us for solving this problem after the text is vectorized. I also provide pairwise graphs for these featues and comment on them.

As said earlier for obtaining a TruncatedSVD with *components=7* is used and therefore there are 7 features per sentence. This parameter is selected because in total there are 7 different topics and LSA aims for representing text as a combination of topics therefore 7 was selected. Different values for this parameter could be tried however due to time limitations and already existing complexity of the code I skip doing so and believe 7 will work well.



In the figure above there are 21 graphs. This is because there are 7 features and I provide pairwise graphs so C(7, 2) = 21. As we can see on some of the graphs although points look cluttered it is possible top see that sentences from same topics extend on same directions. This can especially be seen on blue category on most of the graphs. And considering these are pairwise graphs it is natural for points to look cluttered and these features should be much more seperable on the original features space with 7 dimensions or else the classifiers would have performed very badly.

4.3 Design

In this section utility functions and how the code is structured in general why are methods are modularized.

In my code there are 2 different kinds of modularized functions first are related to text preprocessing and are methods that implement functionalities such as tokenization, normalization, stemming, lemmatization, etc. This piece of code is actually used once so there was no need to modularize it however doing so made the code more easy to read and also allows for use in future projects.

```
def tokenize_(sentence):
def lowercase_(sentence):
def remove_num_(sentence):
def remove_stopwords_(sentence):
def split_underscore_(sentence):
def remove_nonunicode_(sentence):
def remove_punc_(sentence):
def remove_shortwords_(sentence):
def normalize_(sentence):
def stemmize_(sentence):
def lemmatize_(sentence):
def join_(sentence):
```

The other type of modularized functions are functions related to tuning and testing of the model. In my assignment, 3 different vectorization methods are considered and for each of them 2 models are tuned and 3 models are tested so overall in my assignment 6 models are tuned and 9 models are tested. Since tuning and testing functionality is used a lot I have decided to modularize these.

```
def tune_model(model, x_train, y_train, params, scoring, cv)
def test_model(model, x_test, y_test):
```

Apart from utility functions defined which are used over and over the design also contains several other methods that modularize steps required the obtain vectorizations such as LSA, word2vec vectorizations as follows:

```
def get_lsa_model_data(topic_encoded_df):
def get_word2vec_vectorization(data, model, verbose=False):
```

And apart from function definitions the solution design follows a nicely structured path. As explained earlier there are 3 text vectorization methods and for each 2 models are trained and 3 models are tested and also for the first 2 text vectorizations (LSA and word2vec) the vectorizers themselves are also tuned so the overall design of the codebase looks like this:

1 LSA Vectorization
    1.1 LSA Logistic Regression Tuning & Testing
    1.2 LSA Gaussian Naive Bayes Testing
    1.3 LSA Random Forest Tuning & Testing
    1.4 LSA Vectorization Tuning & Testing
2 Word2vec Vectorization
    2.1 Word2vec Logistic Regression Tuning & Testing
    2.2 Word2vec Gaussian Naive Bayes Testing
    2.3 Word2vec Random Forest Tuning & Testing
    2.4 Word2vec Vectorization Tuning & Testing
3 LSA2vec Vectorization
    3.1 LSA2vec Logistic Regression Tuning & Testing
    3.2 LSA2vec Gaussian Naive Bayes Testing
    3.3 LSA2vec Random Forest Tuning & Testing

4.4 Implementation

In this section I will expand upın explanations from section 4.1 Project Plan. Before providing the whole code in programmer catalog section, I will provide important codeblocks alongside with their explanations in this section to make the code more easy to understand

Step 1 Preprocessing

```
# data["text"] = preprocess(data["text"], normalize=True, stemmize=True, lemmatize=True)
data["text"] = preprocess(data["text"])
```

In above piece of code only one of the lines are run and the other is commented out. Reason for this is explained at end of section 3.

Step 2 LSA Vectorization

```
vectorizer = CountVectorizer()
svd = TruncatedSVD(n_components=7)

lsa = get_lsa_vectorization(data, vectorizer, svd, verbose=True)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)
```

An initial vectorization of text is obtained to be used with classifiers without any tuning on count vectorizer. After classifiers are tuned, vectorizer will also be tuned at end.

Step 2.1 Logistic Regression Tuning and Testing

```
model = LogisticRegression(random_state=0)

params = {
    "max_iter":[500, 1000, 2000],
    "solver": ["newton-cg"],
    "penalty": ["l2", "none"]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)
```

```
cm, crep = test_model(best_model, x_test, y_test)
```

Logistic regression is tuned with 2 parameters: *max_iter* and *penalty* using any other solver other than newton-cg causes convergence issues so that parameter is fixed. `tune_model()` function returns a results dataframe that contains results for each of the parameter combinations and the model that achieved highest cross-validation score.

Step 2.2 Gaussian Naive Bayes Testing

```
model = GaussianNB().fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)
```

As explained earlier in gaussian naive bayes there is no tuning of the model.

Step 2.3 Random Forest Tuning and Testing

```
model = RandomForestClassifier(random_state=0)

params = {
    "criterion":["gini", "entropy"],
    "max_depth": [5, 10, 15, 20, None],
    "bootstrap": [True, False]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)
```

Random forest is tuned with 3 parameters: *criterion*, *max_depth*, *bootstrap*. Parameter *criterion* is the parameter that is used to compute splitting attribute on a given node, *max_depth* is the maximum possible depth for trees in the forest and *bootstrap* is a bool which decides whether to use the whole dataset or bootstrap the data for training of trees in the forest.

Step 2.4 Vectorizer Tuning and Testing

```
svd = TruncatedSVD(n_components=7, random_state=0)

min_dfs = [5, 20, 100]
max_dfs = [0.2, 0.6, 1.]
max_featuress = [1000, 20000, None]

for min_df in min_dfs:
    for max_df in max_dfs:
        for max_features in max_featuress:

            vectorizer = CountVectorizer(min_df=min_df, max_df=max_df,
max_features=max_features)
            svd = TruncatedSVD(n_components=7, random_state=0)

            lsa = get_lsa_vectorization(data, vectorizer, svd)
            topic_encoded_df = get_topic_encoded_df(data, lsa)

            x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

            model = RandomForestClassifier(criterion="gini", max_depth=20, bootstrap=True,
random_state=0)
            model.fit(x_train, y_train)
```

```
            score = model.score(x_test, y_test)
```

As we can see tuning of the vecorizer is done manually because GridSearchCV can't be used with Count Vectorizer class. However the dataset is large enough and I don't think not apply cross-validation will cause much problem.

```
vectorizer = CountVectorizer(min_df=20, max_df=0.2, max_features=20000)
svd = TruncatedSVD(n_components=7, random_state=0)

lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = RandomForestClassifier(criterion="gini", max_depth=20, bootstrap=True,
random_state=0)
model.fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)
```

As for testing, a new vectorization is performed with values that achieved best result on tuning.

Step 3 Word2vec Vectorization

```
model = gensim.models.Word2Vec(
    window=10,
    min_count=0,
    workers=4,
    vector_size=100
)

vectorized_texts = get_word2vec_vectorization(data, model, verbose=True)

x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)
```

Similar to step 2 LSA vectorization, an initial vectorization of text is obtained to be used with classifiers without any tuning on word2vec model. After classifiers are tuned, model will also be tuned at end.

Step 3.1, 3.2, 3.3 Tuning and Testing Classifiers: These steps are same with steps 2.1, 2.2, 2.3.

Step 3.4 Vectorizer Tuning and Testing

```
windows = [2, 5, 10, 20]
vector_sizes = [128, 256, 512]

for window in windows:
        for vector_size in vector_sizes:

            word2vec = gensim.models.Word2Vec(window=window,min_count=0, workers=4,
vector_size=vector_size)

            vectorized_texts = get_word2vec_vectorization(data, word2vec)
            x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts,
data)

            model = RandomForestClassifier(criterion="entropy", max_depth=15,
bootstrap=False, random_state=0)
            model.fit(x_train, y_train)
            score = model.score(x_test, y_test)
```

As we can see implementation is similar to step 2.4 however instead of LSA, word2vec model is tuned on parameters *window* and *vector_size*.

```
word2vec = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

text = data["text"].apply(tokenize_)

word2vec.build_vocab(text, progress_per=1000)
word2vec.train(text, total_examples=word2vec.corpus_count, epochs=word2vec.epochs)

model = RandomForestClassifier(criterion="entropy", max_depth=15, bootstrap=False,
random_state=0)
model.fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)
```

As for testing, a new vectorization is performed with values that achieved best result on tuning.

Step 4 LSA2vec Vectorization

```
vectorizer = CountVectorizer(min_df=5, max_df=0.2, max_features=None)
svd = TruncatedSVD(n_components=7)

lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train_lsa, x_test_lsa, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

vectorized_texts = get_word2vec_vectorization(data, model)
x_train_w2v, x_test_w2v, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)

num_training_sample = x_train_lsa.shape[0] # or = x_train_w2v.shape[0], they are the same
num_test_sample = x_test_lsa.shape[0] # or = x_test_w2v.shape[0], they are the same

new_feature_size = x_train_lsa.shape[1] + x_train_w2v.shape[1]

x_train_lsa2vec = np.empty((num_training_sample, new_feature_size))
x_test_lsa2vec = np.empty((num_test_sample, new_feature_size))

for i in range(num_training_sample):
    x_train_lsa2vec[i] = np.concatenate((x_train_lsa[i], x_train_w2v[i]), axis=0)

for i in range(num_test_sample):
    x_test_lsa2vec[i] = np.concatenate((x_test_lsa[i], x_test_w2v[i]), axis=0)

print("x_train_lsa2vec shape:", x_train_lsa2vec.shape)
print("x_test_lsa2vec shape:", x_test_lsa2vec.shape)

x_train, x_test = x_train_lsa2vec, x_test_lsa2vec
```

As we can see the steps for obtaining LSA2vec representation is, first obtaining each representation LSA and word2vec later concatenating these two into a single vector called lsa2vec. The operation is done seperately on train and test data due to structure of code.

Step 4.1, 4.2, 4.3 Tuning and Testing Classifiers: These steps are same with steps 2.1, 2.2, 2.3.

Note: There is no step 4.4 vectorizer tuning like in the case of step 2.4 and 3.4 this is beacuse lsa2vec model is already formed from best representations for both LSA and word2vec so there is no need to re-tune the parameters.

4.5 Programmer Catalog

In this section first I provide the whole code I've written, later explain the time spent for analysis design and implementation and finally how parts of this code can be reused by others.

```
### IMPORTS ###

# Generic Imports

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import unicodedata, re, itertools, sys
import nltk
from nltk.corpus import stopwords

# Zemberek Setup

import jpype
from jpype import JClass, JString
from examples import DATA_PATH

jpype.startJVM(classpath=["./zemberek-full.jar"])
Paths = JClass('java.nio.file.Paths')

TurkishMorphology = JClass('zemberek.morphology.TurkishMorphology')
TurkishTokenizer = JClass('zemberek.tokenization.TurkishTokenizer')
TurkishSentenceNormalizer = JClass('zemberek.normalization.TurkishSentenceNormalizer')
TurkishSpellChecker = JClass('zemberek.normalization.TurkishSpellChecker')
WordAnalysis = JClass('zemberek.morphology.analysis.WordAnalysis')

### DATA PREPROCESSING ###

# Preprocessing Functions

def tokenize_(sentence):
    return sentence.split()

def lowercase_(sentence):
    _new = []
    for word in sentence:
        _new.append(word.lower())
    return _new

def remove_num_(sentence):
    _new = []
    for word in sentence:
        try:
```

```
            fword = float(word)
        except:
            _new.append(word)
    return _new

with open("stopwords.txt", encoding="utf-8") as f:
    stopwords = f.readlines()

stopwords = [w.strip() for w in stopwords]

def remove_stopwords_(sentence):
    _new = []
    for word in sentence:
        if word not in stopwords:
            _new.append(word)
    return _new

def split_underscore_(sentence):
    _new = []
    for word in sentence:
        word = word.split("_")
        _new += word
    return _new

normalizer = TurkishSentenceNormalizer(
    TurkishMorphology.createWithDefaults(),
    Paths.get(str(DATA_PATH.joinpath('normalization'))),
    Paths.get(str(DATA_PATH.joinpath('lm', 'lm.2gram.slm')))
)

control_chars = ''.join(map(chr, itertools.chain(range(0x00,0x20), range(0x7f,0xa0))))
control_char_re = re.compile('[%s]' % re.escape(control_chars))

def remove_nonunicode_(sentence):
    _new = []
    for word in sentence:
        word = control_char_re.sub('', word)
        if word != "":
            _new.append(word)
    return _new

punc = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''

def remove_punc_(sentence):
    _new = []
    for word in sentence:
        if word not in punc:
            _new.append(word)
    return _new

pol_party = ["akp", "chp", "mhp", "hdp", "iyi", "tkp"]

def remove_shortwords_(sentence):
    _new = []
    for word in sentence:
        if len(word) > 3 or word in pol_party:
            _new.append(word)
    return _new

def normalize_(sentence):
    _new = []
```

```
    for word in sentence:
        if word != "":
            word = normalizer.normalize(JString(word))
        _new.append(word)
    return _new

morphology = TurkishMorphology.createWithDefaults()

def stemmize_(sentence):
    _new = []
    for word in sentence:
        if word != "":
            results = morphology.analyze(JString(word))
            results = [result.getLemmas() for result in results]
            if results != []:
                word = results[0]
        _new.append(word)
    return _new

morphology = TurkishMorphology.createWithDefaults()

def lemmatize_(sentence):
    _new = []
    for word in sentence:
        if word != "":
            results = morphology.analyze(JString(word))
            results = [result.getLemmas()[0] for result in results]
            if results != []:
                word = results[0]
        _new.append(word)
    return _new

def join_(sentence):
    return " ".join([str(word) for word in sentence])

def preprocess(text, tokenize=True, lowercase=True, split_underscore=True,
remove_punc=True,
               remove_num = True, remove_stopwords=True, remove_short_words=True,
remove_non_unicode=True,
               normalize=False, lemmatize=False, stemmize=False, join=True, log=False):

    if tokenize:
        text = text.apply(tokenize_)
        if log:
            print("tokenizing...")

    if lowercase:
        text = text.apply(lowercase_)
        if log:
            print("lowercasing...")

    if split_underscore:
        text = text.apply(split_underscore_)
        if log:
            print("splitting underscores...")

    if remove_punc:
        text = text.apply(remove_punc_)
        if log:
            print("removing punctuation...")
```

```
    if remove_num:
        text = text.apply(remove_num_)
        if log:
            print("removing numbers...")

    if remove_stopwords:
        text = text.apply(remove_stopwords_)
        if log:
            print("removing stopwords...")

    if remove_short_words:
        text = text.apply(remove_shortwords_)
        if log:
            print("removing shortwords...")

    if remove_non_unicode:
        text = text.apply(remove_nonunicode_)
        if log:
            print("removing non unicode characters...")

    if normalize:
        text = text.apply(normalize_)
        if log:
            print("normalizing...")

    if lemmatize:
        text = text.apply(lemmatize_)
        if log:
            print("lemmatizing")

    if stemmize:
        text = text.apply(stemmize_)
        if log:
            print("stemmizing")

    if join:
        text = text.apply(join_)
        if log:
            print("joining...")

    return text

# Data Reading

data = pd.read_csv("./turkish_dataset.csv", encoding="utf-8")

# Apply Preprocessing Functions

data["text"] = preprocess(data["text"], log=True)

### LSA VECTORIZATION ###

def get_lsa_vectorization(data, vectorizer, svd, verbose=False):

    # obtain bow

    if verbose:
        print("obtaining bow...")

    bow = vectorizer.fit_transform(data["text"])
```

```
    if verbose:

        # print bow information

        print("bow shape:", bow.shape)
        print(f"{bow.shape[0]} sentences and {bow.shape[1]} unique words", end="\n\n")

    # fit svd and obtain lsa

    if verbose:
        print("fitting svd and obtaining lsa...")

    lsa = svd.fit_transform(bow)

    if verbose:

        # print lsa information

        num_sentences, num_topics = lsa.shape
        print("lsa shape:", lsa.shape)
        print(f"{num_sentences} sentences encoded by {num_topics} different topics",
end="\n\n")

        # print svd information

        num_topics, vocab_size = svd.components_.shape
        print("svd components shape:", svd.components_.shape)
        print(f"{vocab_size} words encoded by {num_topics} different topics")

    return lsa

def get_topic_encoded_df(data, lsa):
    _, num_topics = lsa.shape
    topic_encoded_df = pd.DataFrame(lsa, columns=[f"topic_{i+1}" for i in range(0,
num_topics)])
    topic_encoded_df["sentence"] = data["text"]
    topic_encoded_df["category"] = data["category"]

    return topic_encoded_df

def get_encoding_matrix(svd, vectorizer):
    num_topics, _ = svd.components_.shape
    encoding_matrix = pd.DataFrame(svd.components_, index=[f"topic_{i+1}" for i in range(0,
num_topics)]).T

    dictionary = vectorizer.get_feature_names_out()
    encoding_matrix["terms"] = dictionary

    return encoding_matrix

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import TruncatedSVD

vectorizer = CountVectorizer()
svd = TruncatedSVD(n_components=7, random_state=0)

lsa = get_lsa_vectorization(data, vectorizer, svd, verbose=True)
topic_encoded_df = get_topic_encoded_df(data, lsa)

# Model Tuning & Testing Utility
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

def get_lsa_model_data(topic_encoded_df):

    x = topic_encoded_df.iloc[:, :-2]
    y = topic_encoded_df["category"]

    le = LabelEncoder()
    y = le.fit_transform(y)

    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.33,
random_state=0)
    return x_train.values, x_test.values, y_train, y_test

from sklearn.model_selection import GridSearchCV

def tune_model(model, x_train, y_train, params, scoring, cv):
    gs = GridSearchCV(model, param_grid=params, scoring=scoring, cv=cv, verbose=4)
    gs.fit(x_train, y_train)

    results_df = pd.DataFrame(gs.cv_results_)

    cols = [f"param_{param}" for param in params] + ["mean_test_score"]
    results_df = results_df[cols]

    results_df.sort_values("mean_test_score", ascending=False, inplace=True)

    return results_df, gs.best_estimator_

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

def test_model(model, x_test, y_test):
    pred = model.predict(x_test)
    return confusion_matrix(y_test, pred), classification_report(y_test, pred)

x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

# LSA Logistic Regression Tuning & Testing

from sklearn.linear_model import LogisticRegression

model = LogisticRegression(random_state=0)

params = {
    "max_iter":[500, 1000, 2000],
    "solver": ["newton-cg"],
    "penalty": ["l2", "none"]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)

# LSA Gaussian Naive Bayes Testing

from sklearn.naive_bayes import GaussianNB

model = GaussianNB().fit(x_train, y_train)
```

```
cm, crep = test_model(model, x_test, y_test)

# LSA Random Forest Tuning & Testing

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(random_state=0)

params = {
    "criterion":["gini", "entropy"],
    "max_depth": [5, 10, 15, 20, None],
    "bootstrap": [True, False]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)

# LSA Vectorizer Tuning & Testing

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

svd = TruncatedSVD(n_components=7, random_state=0)

min_dfs = [5, 20, 100]
max_dfs = [0.2, 0.6, 1.]
max_featuress = [1000, 20000, None]

total_fits = len(min_dfs) * len(max_dfs) * len(max_featuress)

df_dict = {"param_min_df":[], "param_max_df":[], "param_max_features":[], "accuracy":[]}

current_fit = 0
for min_df in min_dfs:
    for max_df in max_dfs:
        for max_features in max_featuress:

            current_fit += 1
            print(f"fitting with min_df:{min_df}, max_df:{max_df},
max_features:{max_features} ({current_fit}/{total_fits})")

            vectorizer = CountVectorizer(min_df=min_df, max_df=max_df,
max_features=max_features)
            svd = TruncatedSVD(n_components=7, random_state=0)

            lsa = get_lsa_vectorization(data, vectorizer, svd)
            topic_encoded_df = get_topic_encoded_df(data, lsa)

            x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

            model = RandomForestClassifier(criterion="gini", max_depth=20, bootstrap=True,
random_state=0)
            model.fit(x_train, y_train)
            score = model.score(x_test, y_test)

            df_dict["param_min_df"].append(min_df)
```

```
            df_dict["param_max_df"].append(max_df)
            df_dict["param_max_features"].append(max_features)
            df_dict["accuracy"].append(score)

results_df = pd.DataFrame(df_dict).sort_values("accuracy", ascending=False)


vectorizer = CountVectorizer(min_df=20, max_df=0.2, max_features=20000)
svd = TruncatedSVD(n_components=7, random_state=0)

lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = RandomForestClassifier(criterion="gini", max_depth=20, bootstrap=True,
random_state=0)
model.fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)

### WORD2VEC VECTORIZATION ###

import gensim

def get_word2vec_vectorization(data, model, verbose=False):

    if verbose:
        print("tokenizing data...")

    text = data["text"].apply(tokenize_)

    if verbose:
        print("building vocabulary...")

    model.build_vocab(text, progress_per=1000)

    if verbose:
        print("training model...")

    model.train(text, total_examples=model.corpus_count, epochs=model.epochs)

    if verbose:
        print("testing model...\n")
        print("encoding word 'türkiye'")
        vec = model.wv["türkiye"]
        print("shape:", vec.shape)
        print(vec, end="\n\n")

    if verbose:
        print("encoding sentences...")

    encodings = data["text"].apply(encode, model=model)

    num_sentence, = encodings.shape
    emb_size = model.vector_size
    result = np.empty((num_sentence, emb_size), dtype=np.float32)

    for i, encoding in enumerate(encodings):
        result[i] = encoding

    return result
```

```
def encode(sentence, model):
    words = []
    sentence = sentence.split()
    for word in sentence:
        encoding = model.wv[word]
        words.append(encoding)
    return np.mean(words, axis=0)

def get_word2vec_model_data(vectorized_texts, data):
    x = vectorized_texts
    y = data["category"]

    le = LabelEncoder()
    y = le.fit_transform(y)

    return train_test_split(x, y, test_size=0.33, random_state=0)

model = gensim.models.Word2Vec(
    window=10,
    min_count=0,
    workers=4,
    vector_size=100
)

vectorized_texts = get_word2vec_vectorization(data, model, verbose=True)
x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)

# Word2vec Logistic Regression Tuning & Testing

from sklearn.linear_model import LogisticRegression

model = LogisticRegression(random_state=0)

params = {
    "max_iter":[500, 1000, 2000],
    "solver": ["sag", "saga"],
    "penalty": ["l2"]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)

# Word2vec Gaussian Naive Bayes Tuning & Testing

from sklearn.naive_bayes import GaussianNB

model = GaussianNB().fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)

# Word2vec Random Forest Tuning & Testing

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(random_state=0)

params = {
    "criterion":["gini", "entropy"],
    "max_depth": [5, 10, 15, 20, None],
```

```
    "bootstrap": [True, False]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)

# Word2vec Vectorizer Tuning & Testing

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

windows = [2, 5, 10, 20]
vector_sizes = [128, 256, 512]

total_fits = len(windows) * len(vector_sizes)

df_dict = {"param_window":[], "param_vector_size":[], "accuracy":[]}

current_fit = 0
for window in windows:
        for vector_size in vector_sizes:

            current_fit += 1
            print(f"fitting with window:{window}, vector_size:{vector_size}
({current_fit}/{total_fits})")

            word2vec = gensim.models.Word2Vec(
                window=window,
                min_count=0,
                workers=4,
                vector_size=vector_size)

            vectorized_texts = get_word2vec_vectorization(data, word2vec)
            x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts,
data)

            model = RandomForestClassifier(criterion="entropy", max_depth=15,
bootstrap=False, random_state=0)
            model.fit(x_train, y_train)
            score = model.score(x_test, y_test)

            df_dict["param_window"].append(window)
            df_dict["param_vector_size"].append(vector_size)
            df_dict["accuracy"].append(score)

results_df = pd.DataFrame(df_dict).sort_values("accuracy", ascending=False)

word2vec = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

text = data["text"].apply(tokenize_)

word2vec.build_vocab(text, progress_per=1000)
```

```
word2vec.train(text, total_examples=word2vec.corpus_count, epochs=word2vec.epochs)

model = RandomForestClassifier(criterion="entropy", max_depth=15, bootstrap=False,
random_state=0)
model.fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)

### LSA2VEC VECTORIZATION ###

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import TruncatedSVD

vectorizer = CountVectorizer(min_df=20, max_df=0.2, max_features=20000)
svd = TruncatedSVD(n_components=7, random_state=0)

lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train_lsa, x_test_lsa, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

vectorized_texts = get_word2vec_vectorization(data, model)
x_train_w2v, x_test_w2v, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)

num_training_sample = x_train_lsa.shape[0] # or = x_train_w2v.shape[0], they are the same
num_test_sample = x_test_lsa.shape[0] # or = x_test_w2v.shape[0], they are the same

new_feature_size = x_train_lsa.shape[1] + x_train_w2v.shape[1]

x_train_lsa2vec = np.empty((num_training_sample, new_feature_size))
x_test_lsa2vec = np.empty((num_test_sample, new_feature_size))

for i in range(num_training_sample):
    x_train_lsa2vec[i] = np.concatenate((x_train_lsa[i], x_train_w2v[i]), axis=0)

for i in range(num_test_sample):
    x_test_lsa2vec[i] = np.concatenate((x_test_lsa[i], x_test_w2v[i]), axis=0)

print("x_train_lsa2vec shape:", x_train_lsa2vec.shape)
print("x_test_lsa2vec shape:", x_test_lsa2vec.shape)

x_train, x_test = x_train_lsa2vec, x_test_lsa2vec

# LSA2Vec Logistic Regression Tuning & Testing

from sklearn.linear_model import LogisticRegression

model = LogisticRegression(random_state=0)

params = {
    "max_iter":[500, 1000, 2000],
    "solver": ["newton-cg"],
    "penalty": ["l2"]
}

results_df, best_model = tune_model(model, x_train, y_train,
```

```
                                    params=params, scoring="accuracy", cv=10)

cm, crep = test_model(best_model, x_test, y_test)

# LSA2Vec Gaussian Naive Bayes Testing

from sklearn.naive_bayes import GaussianNB

model = GaussianNB().fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)

# LSA2Vec Random Forest Tuning & Testing

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(random_state=0)

params = {
    "criterion":["gini", "entropy"],
    "max_depth": [5, 10, 15, 20, None],
    "bootstrap": [True, False]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)
cm, crep = test_model(best_model, x_test, y_test)
```

Time Spent On This Assignment

Time spent for analysis	Analysis section includes reading and handling turkish_dataset.csv and exploring and plotting various features of this dataset. The time spent here mostly went for setting up the Zemberek python library and executions of preprocessing steps related to zemberek such as normalization, lemmatization and stemming took long time. However as for exploration of features of the dataset, it didn't took much time since I always consider 7 features for LSA vectorization statically. And for word2vec vectorization features are already determined by the gensim library itself. Overall I can say spent about 5 hours to this section.
Time spent for design & implementation	The implementation part took relatively long time. This is beacuase there are many parts to my implementation. For each of the 3 vectorization methods (LSA, word2vec, LSA2vec) I tune and test 2 models (logistic regression, random forest) and also test 1 model without tuning (gaussian naive bayes). In addition I tune the vectorizations for both LSA and word2vec. So, implementation took long time and as you can see from code at beginning of this section, there are lot of code. Overall I can say this part took 10-20 hours.
Time spent for testing and reporting	The final testing part just includes obtaining a text representation using LSA and word2vec vectorizers with best hyperparameters (found earlier in tuning sections). And also obtaining the best classifier with best hyperparameters (again as found on earlier tuning sections). Finally the classifier is trained and tested. So the testing part did not take much maybe about 1 hour. As for the time spent reporting, in my report I tried to put everything I did and everything I couldn't do within assignment due to limitations like time and computation. I also explain the reasoning behind everything I do to make sure readers understand. So I put a lot of effort for the report and It took more than 20 hours to write.

How and when to use this code for other purposes ?

I've written my code in highly modularized way therefore it can be used by others for different purposes as well.

The first part that is highly modularized are the text processing utility functions discussed in section 4.2 design. The utility functions are almost always same for any turkish text preprocessing application which means they can be used as-is on different codebases without need for any modification.

Second part is other utility functions such as model tuning and model testing functions again discussed in sections 4.2 design. These functions may also be used for other applications however depending on dataframes of other applications slight modification may be required.

4.6 User Catalog

Previously I discussed how programmers might use part of this code in their application. In this section I discuss how other actual users may use the code for their purposes and limitations.

As stated in the programmer catalog section first part that is relatively easy to use is the text preprocessing functions.

```
def preprocess(text, tokenize=True,
                     lowercase=True,
                     split_underscore=True,
                     remove_punc=True,
                     remove_num = True,
                     remove_stopwords=True,
                     remove_short_words=True,
                     remove_non_unicode=True,
                     normalize=False,
                     lemmatize=False,
                     stemmize=False,
                     join=True,
                     log=False):
```

Users that want to apply preprocessing to their data can use the function above with the following usage: `data["text"] = preprocess(data["text"])`. However it should be noted that steps normalize, lemmatize and stemmize are done using the Zemberek library meaning that they only work for turkish words. But rest of the parameters can be used for english as well.

Moreover, if the users want slight changes on how these operations are done they can easily modify the function related to that step thanks to highly modularized code I've written. The implementation for the function above looks like:

```
if tokenize:
        text = text.apply(tokenize_)
        if log:
            print("tokenizing...")

if lowercase:
        text = text.apply(lowercase_)
        if log:
            print("lowercasing...")
```

and the functions applied like `tokenize_` and `lowercase_` have their own function definitions elsewhere in the code in a modularized way like so:

```
def tokenize_(sentence):
    return sentence.split()

def lowercase_(sentence):
    _new = []
    for word in sentence:
        _new.append(word.lower())
    return _new
```

And these implementations can be easily modified to suit the needs of an another application.

The other part that is reusable is functions `tune_model` and `test_model`.

```
from sklearn.model_selection import GridSearchCV

def tune_model(model, x_train, y_train, params, scoring, cv):
    gs = GridSearchCV(model, param_grid=params, scoring=scoring, cv=cv, verbose=4)
    gs.fit(x_train, y_train)

    results_df = pd.DataFrame(gs.cv_results_)

    cols = [f"param_{param}" for param in params] + ["mean_test_score"]
    results_df = results_df[cols]

    results_df.sort_values("mean_test_score", ascending=False, inplace=True)

    return results_df, gs.best_estimator_
```

As we can see the `tune_model` function expects a model and training data alongside with set of parameters to tune the model on. This function can be used in other applications as well without need for any major modifications. This function also returns a dataframe where we can see the result for every hyperparameter combination and their performed scores. And as a second argument function returns the best estimator (model that performed best on cross-validation).

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

def test_model(model, x_test, y_test):
    pred = model.predict(x_test)
    return confusion_matrix(y_test, pred), classification_report(y_test, pred)
```

As for the `test_model` function, it expects the trained model and testing data to returns a confusion matrix and a classification report for detailed overiew of performance on test data.

5 Results, Discussions and Conclusion

In this section I will first explain the choice of performance metric, show the results for different models and text representations with respect to this metric. Later I move onto discussions about the project and I will explain how this project could be made better. Finally in the conclusion section I will briefly summarize work done on this project and comment on findings of the project.

5.1 Results

Before we look at results from models and text representations let us first discuss about which metric to use. In previous assignments I considered precision and recall on class base using the classification report. In this assignment I also provide confusion matrices and classification reports with each model testing however in this assignment the data is perfectly balanced as discussed in section 3 therefore we focus more on accuracy and fine tuning of the models is also done with respect to accuracy metric (model with best accuracy is considered best model).

1 LSA Vectorization

```
vectorizer = CountVectorizer()
svd = TruncatedSVD(n_components=7, random_state=0)
lsa = get_lsa_vectorization(data, vectorizer, svd, verbose=True)
topic_encoded_df = get_topic_encoded_df(data, lsa)
x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)
```

To use with the models an initial LSA vectorization is obtained. The tuning for the vectorizer will be done later. This initial LSA vectorization considers a count vectorizer with default parameters.

1.1 LSA Logistic Regression Tuning and Testing

```
model = LogisticRegression(random_state=0)

params = {
    "max_iter":[500, 1000, 2000],
    "solver": ["newton-cg"],
    "penalty": ["l2", "none"]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)
```

After tuning the model following set of scores are obtained from cross validation:

	param_max_iter	param_solver	param_penalty	mean_test_score
0	500	newton-cg	l2	0.568073
2	1000	newton-cg	l2	0.568073
4	2000	newton-cg	l2	0.568073
1	500	newton-cg	none	0.567772
3	1000	newton-cg	none	0.567772
5	2000	newton-cg	none	0.567772

We can see that best score is obtained with *max_iter=500* and *penalty="l2"* with an accuracy of 0.568.

As for the testing of model:

```
cm, crep = test_model(best_model, x_test, y_test)
```

Following results are obtained as confusion matrix and classification report:

accuracy	0.57
confusion matrix	[[157 8 2 0 32 16 4] [27 126 11 5 15 15 37] [2 17 81 6 4 104 16] [8 3 3 188 1 9 19] [43 9 0 5 169 8 3] [10 12 49 1 3 135 24] [18 17 31 16 3 85 60]]
classification report	precision recall f1-score support 0 0.59 0.72 0.65 219 1 0.66 0.53 0.59 236 2 0.46 0.35 0.40 230 3 0.85 0.81 0.83 231 4 0.74 0.71 0.73 237 5 0.36 0.58 0.45 234 6 0.37 0.26 0.31 230 accuracy 0.57 1617 macro avg 0.58 0.57 0.56 1617 weighted avg 0.58 0.57 0.56 1617

1.2 LSA Gaussian Naive Bayes Testing

```
model = GaussianNB().fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)
```

There is no tuning for gaussian naive bayes as discussed earlier so test results are:

accuracy	0.37
confusion matrix	[[41 9 4 2 17 123 23] [0 65 17 2 9 75 68] [2 15 31 5 7 158 12] [0 9 4 127 1 78 12] [18 7 9 3 106 82 12] [3 2 19 2 2 197 9] [0 5 10 4 5 177 29]]
classification report	precision recall f1-score support 0 0.64 0.19 0.29 219 1 0.58 0.28 0.37 236 2 0.33 0.13 0.19 230 3 0.88 0.55 0.68 231 4 0.72 0.45 0.55 237 5 0.22 0.84 0.35 234 6 0.18 0.13 0.15 230 accuracy 0.37 1617 macro avg 0.51 0.37 0.37 1617 weighted avg 0.51 0.37 0.37 1617

1.3 LSA Random Forest Tuning and Testing

```
params = {
    "criterion":["gini", "entropy"],
    "max_depth": [5, 10, 15, 20, None],
    "bootstrap": [True, False]
}

results_df, best_model = tune_model(model, x_train, y_train,
                                    params=params, scoring="accuracy", cv=10)
```

After tuning the model following set of scores are obtained from cross validation:

	param_criterion	param_max_depth	param_bootstrap	mean_test_score
3	gini	20	True	0.664932
9	entropy	None	True	0.664314
7	entropy	15	True	0.663100
8	entropy	20	True	0.662786
4	gini	None	True	0.662192
2	gini	15	True	0.660063
17	entropy	15	False	0.659739
14	gini	None	False	0.655176
12	gini	15	False	0.653052
6	entropy	10	True	0.652137
19	entropy	None	False	0.651522
18	entropy	20	False	0.650913
13	gini	20	False	0.650303
16	entropy	10	False	0.646349
1	gini	10	True	0.643308
11	gini	10	False	0.638421
5	entropy	5	True	0.575388
15	entropy	5	False	0.574167
0	gini	5	True	0.567780
10	gini	5	False	0.563816

We can see that best score is obtained with *criterion="gini", max_depth="20"* and *bootstrap=True* with an accuracy of 0.664.

As for the testing of model:

```
cm, crep = test_model(best_model, x_test, y_test)
```

Following results are obtained as confusion matrix and classification report:

accuracy	0.66
confusion matrix	[[149 9 2 0 40 5 14] [12 153 13 8 19 9 22] [1 18 140 2 2 47 20] [5 9 3 185 4 7 18] [21 12 1 5 190 4 4] [7 9 50 2 7 134 25] [16 26 28 7 5 32 116]]
classification report	precision recall f1-score support 0 0.71 0.68 0.69 219 1 0.65 0.65 0.65 236 2 0.59 0.61 0.60 230 3 0.89 0.80 0.84 231 4 0.71 0.80 0.75 237 5 0.56 0.57 0.57 234 6 0.53 0.50 0.52 230 accuracy 0.66 1617 macro avg 0.66 0.66 0.66 1617 weighted avg 0.66 0.66 0.66 1617

Out of three models with their best results, we get a table as follows:

model	accuracy
Logistic Regression	0.57
Gaussian Naive Bayes	0.37
Random Forest	0.66

So in the next step where we tune the vectorizer, we will consider the random forest model along with its best hyperparameters since it performed best.

1.4 LSA Vectorizer Tuning and Testing

```
svd = TruncatedSVD(n_components=7, random_state=0)

min_dfs = [5, 20, 100]
max_dfs = [0.2, 0.6, 1.]
max_featuress = [1000, 20000, None]

for min_df in min_dfs:
    for max_df in max_dfs:
        for max_features in max_featuress:

            vectorizer = CountVectorizer(min_df=min_df, max_df=max_df,
max_features=max_features)
            svd = TruncatedSVD(n_components=7, random_state=0)

            lsa = get_lsa_vectorization(data, vectorizer, svd)
            topic_encoded_df = get_topic_encoded_df(data, lsa)

            x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

            model = RandomForestClassifier(criterion="gini", max_depth=20, bootstrap=True,
random_state=0)
```

```
model.fit(x_train, y_train)
score = model.score(x_test, y_test)
```

After tuning the model following set of scores are obtained from cross validation:

	param_min_df	param_max_df	param_max_features	accuracy
10	20	0.2	20000.0	0.701917
11	20	0.2	NaN	0.701917
1	5	0.2	20000.0	0.696351
2	5	0.2	NaN	0.694496
5	5	0.6	NaN	0.669140
8	5	1.0	NaN	0.669140
9	20	0.2	1000.0	0.667285
4	5	0.6	20000.0	0.664811
7	5	1.0	20000.0	0.664811
13	20	0.6	20000.0	0.662956
17	20	1.0	NaN	0.662956
16	20	1.0	20000.0	0.662956
14	20	0.6	NaN	0.662956
0	5	0.2	1000.0	0.661719
18	100	0.2	1000.0	0.658627
19	100	0.2	20000.0	0.658627
20	100	0.2	NaN	0.658627
12	20	0.6	1000.0	0.639456
15	20	1.0	1000.0	0.639456
6	5	1.0	1000.0	0.639456
3	5	0.6	1000.0	0.639456
21	100	0.6	1000.0	0.633271
22	100	0.6	20000.0	0.633271
23	100	0.6	NaN	0.633271
24	100	1.0	1000.0	0.633271
25	100	1.0	20000.0	0.633271
26	100	1.0	NaN	0.633271

We can see that best score is obtained with *min_df=5*, *max_depth=0.2* and *max_features*=20000 with an accuracy of 0.701.

As for the testing of the vectorization:

```
vectorizer = CountVectorizer(min_df=5, max_df=0.2, max_features=None)
svd = TruncatedSVD(n_components=7)
```

```
lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train, x_test, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = RandomForestClassifier(criterion="entropy", max_depth=15, bootstrap=True,
random_state=0)
model.fit(x_train, y_train)
score = model.score(x_test, y_test)
```

Following results are obtained as confusion matrix and classification report:

accuracy	0.70
confusion matrix	[[149 14 5 0 30 3 18] [13 155 18 4 18 5 23] [0 14 148 1 7 35 25] [3 10 1 193 3 4 17] [21 10 3 4 187 4 8] [7 8 43 1 2 156 17] [13 20 21 10 8 11 147]]
classification report	precision recall f1-score support 0 0.72 0.68 0.70 219 1 0.67 0.66 0.66 236 2 0.62 0.64 0.63 230 3 0.91 0.84 0.87 231 4 0.73 0.79 0.76 237 5 0.72 0.67 0.69 234 6 0.58 0.64 0.61 230 accuracy 0.70 1617 macro avg 0.71 0.70 0.70 1617 weighted avg 0.71 0.70 0.70 1617

2 Word2vec Vectorization

```
model = gensim.models.Word2Vec(
    window=10,
    min_count=0,
    workers=4,
    vector_size=100
)

vectorized_texts = get_word2vec_vectorization(data, model, verbose=True)
x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)
```

To use with the models an initial word2vec vectorization is obtained. The tuning for the vectorizer will be done later. The current model has arbitrary parameters above.

2.1 Word2vec Logistic Regression Tuning and Testing

The code and process is same with 1.1 tuning for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.59
confusion matrix	[[108 13 24 11 37 13 13] [16 126 13 19 14 16 32] [4 14 134 15 7 39 17] [6 17 11 168 2 7 20] [25 6 8 9 177 9 3] [10 4 56 25 17 112 10] [9 15 26 40 4 2 134]]
classification report	precision recall f1-score support 0 0.61 0.49 0.54 219 1 0.65 0.53 0.58 236 2 0.49 0.58 0.53 230 3 0.59 0.73 0.65 231 4 0.69 0.75 0.72 237 5 0.57 0.48 0.52 234 6 0.59 0.58 0.58 230 accuracy 0.59 1617 macro avg 0.60 0.59 0.59 1617 weighted avg 0.60 0.59 0.59 1617

2.2 Word2vec Gaussian Naive Bayes Testing

The code and process is same with 1.2 testing for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.41
confusion matrix	[[81 7 42 28 21 22 18] [24 70 22 50 6 29 35] [4 12 94 39 7 66 8] [6 6 17 173 0 16 13] [33 0 21 18 124 41 0] [14 7 56 62 14 72 9] [9 16 34 104 1 11 55]]
classification report	precision recall f1-score support 0 0.47 0.37 0.42 219 1 0.59 0.30 0.40 236 2 0.33 0.41 0.36 230 3 0.36 0.75 0.49 231 4 0.72 0.52 0.60 237 5 0.28 0.31 0.29 234 6 0.40 0.24 0.30 230 accuracy 0.41 1617 macro avg 0.45 0.41 0.41 1617 weighted avg 0.45 0.41 0.41 1617

2.3 Word2vec Random Forest Tuning and Testing

The code and process is same with 1.3 tuning for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.64
confusion matrix	[[129 10 13 3 37 18 9] [12 143 6 11 16 19 29] [11 7 139 6 5 44 18] [8 19 9 159 3 7 26] [31 6 6 6 177 9 2] [13 4 30 3 15 155 14] [22 22 20 24 6 5 131]]
classification report	precision recall f1-score support 0 0.57 0.59 0.58 219 1 0.68 0.61 0.64 236 2 0.62 0.60 0.61 230 3 0.75 0.69 0.72 231 4 0.68 0.75 0.71 237 5 0.60 0.66 0.63 234 6 0.57 0.57 0.57 230 accuracy 0.64 1617 macro avg 0.64 0.64 0.64 1617 weighted avg 0.64 0.64 0.64 1617

On 3 models with their results we get the following results:

model	accuracy
Logistic Regression	0.59
Gaussian Naive Bayes	0.41
Random Forest	0.64

So in the next step where we tune the vectorizer, we will consider the random forest model along with its best hyperparameters since it performed best.

2.4 Word2Vec Vectorizer Tuning and Testing

```
windows = [2, 5, 10, 20]
vector_sizes = [128, 256, 512]

current_fit = 0
for window in windows:
        for vector_size in vector_sizes:
            word2vec = gensim.models.Word2Vec(
                window=window,
                min_count=0,
                workers=4,
                vector_size=vector_size)

            vectorized_texts = get_word2vec_vectorization(data, word2vec)
            x_train, x_test, y_train, y_test = get_word2vec_model_data(vectorized_texts,
data)

            model = RandomForestClassifier(criterion="entropy", max_depth=15,
bootstrap=False, random_state=0)
            model.fit(x_train, y_train)
            score = model.score(x_test, y_test)
```

After tuning the model following set of scores are obtained from cross validation:

	param_window	param_vector_size	accuracy
9	20	128	0.678417
10	20	256	0.678417
11	20	512	0.662338
6	10	128	0.630179
0	2	128	0.623995
1	2	256	0.623377
7	10	256	0.620284
8	10	512	0.611626
3	5	128	0.601732
4	5	256	0.601732
2	2	512	0.599258
5	5	512	0.578850

We can see that best score is obtained with *window=20* and *vector_size=128* with an accuracy of 0.678.

As for testing of the vectorization:

```
word2vec = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

text = data["text"].apply(tokenize_)

word2vec.build_vocab(text, progress_per=1000)
word2vec.train(text, total_examples=word2vec.corpus_count, epochs=word2vec.epochs)

model = RandomForestClassifier(criterion="entropy", max_depth=15, bootstrap=False,
random_state=0)
model.fit(x_train, y_train)
cm, crep = test_model(model, x_test, y_test)
```

accuracy	0.66
confusion matrix	[[131 15 10 6 35 15 7] [22 143 10 12 11 16 22] [8 9 153 5 8 33 14] [8 19 12 166 3 5 18] [25 6 5 6 182 11 2] [9 7 31 3 13 162 9] [18 24 23 20 9 2 134]]

		precision	recall	f1-score	support
classification report	0	0.59	0.60	0.60	219
	1	0.64	0.61	0.62	236
	2	0.63	0.67	0.65	230
	3	0.76	0.72	0.74	231
	4	0.70	0.77	0.73	237
	5	0.66	0.69	0.68	234
	6	0.65	0.58	0.61	230
	accuracy			0.66	1617
	macro avg	0.66	0.66	0.66	1617
	weighted avg	0.66	0.66	0.66	1617

After tuning both of the vectorization and 3 models for each of them we get the following results:

	LSA	Word2vec	LSA (tuned)	Word2vec (tuned)
Logistic Regression	0.57	0.59	-	-
Gaussian Naive Bayes	0.37	0.41	-	-
Random Forest	0.66	0.64	0.70	0.66

Notice that vectorizer are tuned only for random forest since it is the best performing model.

And finally in next section we combine both vectorizations to obtain LSA2vec vectorization.

3 LSA2vec Vectorization

```
vectorizer = CountVectorizer(min_df=20, max_df=0.2, max_features=20000)
svd = TruncatedSVD(n_components=7, random_state=0)

lsa = get_lsa_vectorization(data, vectorizer, svd)
topic_encoded_df = get_topic_encoded_df(data, lsa)

x_train_lsa, x_test_lsa, y_train, y_test = get_lsa_model_data(topic_encoded_df)

model = gensim.models.Word2Vec(
window=20,
min_count=0,
workers=4,
vector_size=128)

vectorized_texts = get_word2vec_vectorization(data, model)
x_train_w2v, x_test_w2v, y_train, y_test = get_word2vec_model_data(vectorized_texts, data)

num_training_sample = x_train_lsa.shape[0] # or = x_train_w2v.shape[0], they are the same
num_test_sample = x_test_lsa.shape[0] # or = x_test_w2v.shape[0], they are the same

new_feature_size = x_train_lsa.shape[1] + x_train_w2v.shape[1]

x_train_lsa2vec = np.empty((num_training_sample, new_feature_size))
x_test_lsa2vec = np.empty((num_test_sample, new_feature_size))

for i in range(num_training_sample):
    x_train_lsa2vec[i] = np.concatenate((x_train_lsa[i], x_train_w2v[i]), axis=0)

for i in range(num_test_sample):
    x_test_lsa2vec[i] = np.concatenate((x_test_lsa[i], x_test_w2v[i]), axis=0)
```

```
x_train, x_test = x_train_lsa2vec, x_test_lsa2vec
```

Notice that LSA2vec is concatenation of the LSA and word2vec vectorizations with best hyperparameters. These parameters were obtained from sections 1.4 and 2.4 therefore there is no need for further tuning of the vectorizer in this section.

After this LSA2vec vectorization is obtained the 3 models are tuned again with respect to this new vectorization.

3.1 LSA2vec Logistic Regression Tuning and Testing

The code and process is same with 1.1 tuning for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.72
confusion matrix	[[145 11 12 0 38 7 6] [21 140 9 5 14 10 37] [4 14 146 5 6 30 25] [9 1 6 191 3 3 18] [21 9 4 4 194 2 3] [3 4 31 2 8 172 14] [14 13 17 8 4 1 173]]
classification report	precision recall f1-score support 0 0.67 0.66 0.67 219 1 0.73 0.59 0.65 236 2 0.65 0.63 0.64 230 3 0.89 0.83 0.86 231 4 0.73 0.82 0.77 237 5 0.76 0.74 0.75 234 6 0.63 0.75 0.68 230 accuracy 0.72 1617 macro avg 0.72 0.72 0.72 1617 weighted avg 0.72 0.72 0.72 1617

3.2 LSA2vec Gaussian Naive Bayes Testing

The code and process is same with 1.2 tuning for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.45
confusion matrix	[[85 8 24 13 24 39 26] [30 73 21 31 10 26 45] [5 15 74 28 9 79 20] [7 7 14 174 0 13 16] [37 0 13 7 140 37 3] [12 7 43 43 12 93 24] [7 17 31 81 0 7 87]]

		precision	recall	f1-score	support
classification report	0	0.46	0.39	0.42	219
	1	0.57	0.31	0.40	236
	2	0.34	0.32	0.33	230
	3	0.46	0.75	0.57	231
	4	0.72	0.59	0.65	237
	5	0.32	0.40	0.35	234
	6	0.39	0.38	0.39	230
	accuracy			0.45	1617
	macro avg	0.47	0.45	0.44	1617
	weighted avg	0.47	0.45	0.45	1617

3.2 LSA2vec Random Forest Tuning and Testing

The code and process is same with 1.3 tuning for LSA vectorization however the results are of course different since this time a different vectorization is used. Results are:

accuracy	0.71						
confusion matrix	[[143	13	5	1	36	14	7]
	[18	151	11	4	13	13	26]
	[5	15	157	3	4	35	11]
	[2	13	8	186	4	1	17]
	[28	4	2	7	193	2	1]
	[7	6	27	0	11	169	14]
	[16	24	21	11	6	7	145]]

		precision	recall	f1-score	support
classification report	0	0.65	0.65	0.65	219
	1	0.67	0.64	0.65	236
	2	0.68	0.68	0.68	230
	3	0.88	0.81	0.84	231
	4	0.72	0.81	0.77	237
	5	0.70	0.72	0.71	234
	6	0.66	0.63	0.64	230
	accuracy			0.71	1617
	macro avg	0.71	0.71	0.71	1617
	weighted avg	0.71	0.71	0.71	1617

On 3 models with their results we get the following results:

model	accuracy
Logistic Regression	0.72
Gaussian Naive Bayes	0.45
Random Forest	0.71

So, finally we have the scores for 3 different text vectorizations (LSA, word2vec, LSA2vec) and 3 different classifiers (Logistic Regression, Gaussian Naive Bayes, Random Forest) and the overall results are as follows:

After tuning both of the vectorization and 3 models for each of them we get the following results:

	LSA	Word2vec	LSA2vec
Logistic Regression	0.57	0.59	0.72
Gaussian Naive Bayes	0.37	0.41	0.45
Random Forest	0.70 (tuned)	0.66 (tuned)	0.71

We can see that the combined LSA2vec representation increases scores of the classifiers, meaning that it does a better job vectorizing the text data. The performance increase is quite big for Logistic Regression with increase from 0.57-0.59 to 0.72. And in the case of naive bayes there is also significant increase from 0.37-0.41 to 0.45. As for the random forest there is 5% increase from word2vec model however only 1% increrase from LSA model.

The result for the small increrase of 1% maybe beacuse of 0.7 is a natural threshold for the problem at hand and we can't get better results without using more advanced techniques like deep neural networks. This makes sense since logistic regression also gets stuck at 0.72 however this is just a thought and may not be true.

Dataset Tuning

As discussed earlier the operations above are done on a dataset that doesn't apply normalization stemming and lemmatization. This is because I wanted to run 2 seperate experimentations to see the results of these operations on the final scores. However as discussed earlier the way this is done is, same notebook is run twice with changing the comment on following piece of code:

```
# data["text"] = preprocess(data["text"], normalize=True, stemmize=True, lemmatize=True)
data["text"] = preprocess(data["text"])
```

on previous experiments code is run like this and in the second run I change code to:

```
data["text"] = preprocess(data["text"], normalize=True, stemmize=True, lemmatize=True)
# data["text"] = preprocess(data["text"])
```

And results of tuning the dataset like this are as follows:

	LSA	Word2vec	LSA2vec
Logistic Regression	0.73	0.81	0.84
Gaussian Naive Bayes	0.53	0.76	0.77
Random Forest	0.81 (tuned)	0.84 (tuned)	0.85

As we can see applying normalization, stemming and lemmatization provides significant performance gain. However our previous finding still holds which is the fact that combined LSA2vec vectorization provides better performance compared to individiual vectorizations.

5.1 Discussion

In this section I will discuss what could have been done more to make this project better and what the project lacks.

Firstly, in this assignment I try 3 different vectorization approaches: LSA, word2vec and LSA2vec. The LSA2vec vectorization is obtained by concatenating the LSA and word2vec representations however concatenation is one of the simplest way to combine and other techniques could also have been used but due to time limitations.

As for the classifiers I consider 3 different classifiers: logistic regression, gaussian naive bayes and random forest. Although these are very easy and understandable classifiers they have limitations. More complicated methods like support vector machines or deep neural networks could have been used to increase the performance on the dataset however since I don't know the theory behind SVMs and ANNs as much as I know for the 3 mentioned classifiers I didn't want to consider these classifiers. Time was again also a limitation.

A problematic part with implementation is that 2 versions of data preprocessing with and without [normalization, stemming, lemmatization] can not be run and investigated at the same time. I could write the code in such a way that allows for this however it would require additional time which I sadly don't have. For this reason, the 2 version of preprocessing have to be checked by running the notebook twice with one of the preprocessings commented out as shown in an earlier section.

5.1 Conclusion

Summary of work done

In this section I summarize work done in this assignment and report and reiterate the findings of this assignment.

In section 1 I discuss the problem and briefly go over the solution without going much into detail to give the reader an idea of how the report will be structured for rest of the sections. And the problem is classification of turkish sentences into one of 7 topics.

In section 2 I take a closer look at the data, address the problems in the text data and briefly introduce the solutions to these problems which are applied in further sections which are mostly related to preprocessing of the text data such as removal of underscores.

In section 3 I explain the methodology for the assignment from a high level. The methodology was to consider 3 different text vectorizations (LSA, word2vec, LSA2vec) alongside with 3 classifier models for each (logistic regression, gaussian naive bayes and random forest).

In section 4 I discuss the solution development for the problem with respect to methodology explained in previous section (section 3). Firstly I explain the steps that have to be performed one-by-one in project plan section, later in analysis section I perform a more in depth analysis of the data that considers features to be used in the solution of the problem. Later in the design section I explain the overall design of the code and explain utility functions that are constantly used throughout the code in the next implementation section. In the implementation section I expand upon the steps described in project plan section and provide simplified code blocks along with their explanations. Finally in programmer catalog and user catalog sections I provide the whole code that was used in the assignment, explain how these code might be used for other purposes by other programmers or users and finally explain the time spent on the assignment.

Lastly in section 5 I provide the results for the steps described in section 4 along with both the piece of code that yields to that results and the result it self (accuracy, confusion matrix, classification report) In this section I also provide comparitive results to see which of the text vectorizations or the models have performed the best on the test data. Later in discussion section I discuss the shortcomings of my implementation and what more could have been done to make the project better.

Findings of the project

There are two main findings on this project which both are explained in section 5.1 results. First finding is that we see a text vectorization that is a combination of multiple vectorizations such as LSA and word2vec contribute significant performance gains to topic classification task as can be see on the tables below.

And the other imporant finding is that applying steps like normalization, stemming, lemmatization to the text data may significantly increase the model performance, again this can be observed from tables below. In the first table these steps are not applied to data and in second they are.

	LSA	Word2vec	LSA2vec
Logistic Regression	0.57	0.59	0.72
Gaussian Naive Bayes	0.37	0.41	0.45
Random Forest	0.70 (tuned)	0.66 (tuned)	0.71

	LSA	Word2vec	LSA2vec
Logistic Regression	0.73	0.81	0.84
Gaussian Naive Bayes	0.53	0.76	0.77
Random Forest	0.81 (tuned)	0.84 (tuned)	0.85

References

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
https://en.wikipedia.org/wiki/Topic_model
https://towardsdatascience.com/topic-modeling-and-latent-dirichlet-allocation-in-python-9bf156893c24
https://monkeylearn.com/topic-analysis/
https://en.wikipedia.org/wiki/Word2vec
https://en.wikipedia.org/wiki/Latent_semantic_analysis
https://www.youtube.com/watch?v=hB51kkus-Rc&t=2s
https://paperswithcode.com/method/lda2vec