University of Thessaly

Neuro-Fuzzy Computing

ECE447

Coding Project

Alexandra Gianni ID: 3382

Nikos Stylianou ID: 2917

March 3, 2024

Contents

Introduction

In the realm of machine learning and artificial intelligence, the ability to accurately classify text into predefined categories represents a cornerstone of many practical applications, from sentiment analysis to automated customer support and beyond. This project, conceived as a practical assignment for Neurofuzzy class, aims to design, implement, and evaluate a text classifier. The core objective of this classifier is to process news provided in a CSV file format, each entry containing snippets of text, and to assign them to one of several news category and subcategory based on their content.

To achieve this, we embark on a journey through the intricacies of neurofuzzy systems, which blend the robustness and learning capabilities of neural networks with the interpretability and reasoning of fuzzy logic. This hybrid approach enables the handling of uncertainty and imprecision in natural language, offering a promising pathway to enhancing classification performance.

Our project report is structured to walk the reader through the entire lifecycle of the classifier's development. Starting with a comprehensive literature review, we lay the groundwork by exploring existing theories and methodologies that underpin our approach. This is followed by a detailed account of the system design, where we elaborate on the architecture, choice of algorithms, and the rationale behind these decisions. We then proceed to describe the implementation phase, document and comment the practical steps taken to bring our design to fruition, including data preprocessing, feature extraction, and model training.

A significant portion of the report is dedicated to the evaluation of our classifier. Here, we employ a variety of metrics to assess its performance, discussing both its strengths and areas for improvement. Through this analysis, we aim to not only validate our approach but also contribute valuable insights to the field of text classification.

Finally, the report concludes with a reflection on the lessons learned throughout the project, potential applications of our classifier, and avenues for future research. By providing a comprehensive overview of our journey from conception to evaluation, this report aims to offer a valuable resource for fellow researchers and practitioners in the domain of text classification.

Literature Overview

The field of text classification has seen substantial progress with the advent of machine learning and artificial intelligence technologies. Among these, neurofuzzy systems have emerged as a significant area of interest, offering the potential to blend the interpretability of fuzzy logic with the learning capabilities of neural networks. This literature review examines the current methodologies, challenges, and advancements in text classification, with a focus on the application of neurofuzzy systems to enhance multiclass classification tasks.

Text classification is a pivotal task in natural language processing (NLP) with applications ranging from sentiment analysis to topic categorization and spam detection. Traditional machine learning algorithms, such as Support Vector Machines (SVM) and Naive Bayes, have laid the groundwork for early advancements in the field. However, these models often struggle with the nuances of natural language, including context sensitivity, polysemy, and the curse of dimensionality inherent in text data.

The integration of neural networks and fuzzy logic into neurofuzzy systems presents a novel approach to overcoming the limitations faced by traditional classifiers. Neural networks contribute deep learning capabilities, enabling models to learn complex patterns and relationships in large datasets. Fuzzy logic, on the other hand, introduces an element of human-like reasoning and interpretability by handling imprecision and uncertainty in linguistic expressionss.

Significant advancements have been made in developing algorithms and models that leverage the strengths of both neural networks and fuzzy logic for text classification. Convolutional Neural Networks (CNNs) and

Recurrent Neural Networks (RNNs) are commonly used architectures for capturing spatial and sequential patterns in text, respectively. The incorporation of fuzzy systems with these architectures allows for the creation of adaptable and interpretable models that can dynamically adjust classification rules based on the learning context.

The evaluation of neurofuzzy systems in text classification often employs metrics such as accuracy, precision, recall, and F1 score. A comparative analysis by Zhou and Chen (2021) found that neurofuzzy classifiers consistently achieve higher precision and recall rates across multiple datasets when compared to standalone neural network or fuzzy logic models. This suggests that the hybrid approach effectively captures the intricacies of text data, improving overall classification performance.

The literature on multiclass text classification demonstrates a clear trend towards the adoption of neuro-fuzzy systems as a means to address the inherent challenges of natural language processing. By combining the learning power of neural networks with the interpretability and flexibility of fuzzy logic, researchers and practitioners are able to develop more accurate, robust, and interpretable text classification models. This review underscores the potential of neurofuzzy systems to advance the state of the art in text classification, marking a promising direction for future research and application.

Methodology

To construct a robust multi-class text classifier, our methodology was meticulously designed to ensure both efficiency and accuracy. The process is segmented into distinct phases, as shown below.

(a) Import Library

When embarking on the implementation of a machine learning project, such as the development of a multiclass text classifier, the first step involves setting up the computational environment by importing the necessary libraries. These libraries provide pre-written functions and classes that facilitate data manipulation, model building, training, and evaluation, significantly reducing the amount of code we need to write from scratch and making our job easier.

```
# Interact with Operation System
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))

# Load, explore and plot data
import string
import numpy as np
import tensorflow as tf
import pandas as pd
import re
import ast

# Train test split
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

```
from collections import Counter
from imblearn.over_sampling import RandomOverSampler

# Modeling
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import LSTM
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.layers import Embedding
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import GlobalMaxPooling1D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import LeakyReLU
```

These are all the libraries required to load, test, train and build our model. To have a better understanding about their functionality, we will briefly discuss them.

- **os**: A standard Python library for interacting with the operating system. It's being used here to set an environment variable that stops `tensorflow` from displaying annoying debug info.
- **nltk**: The Natural Language Toolkit, or NLTK, is a library used for working with human language data.
- **numpy**: A basic Python library adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- **tensorflow**: An open-source software library for machine learning and artificial intelligence. It provides a flexible platform for defining and running computations that involve tensors, which are partial derivatives of a function with respect to its variables. This is the main library that we use in order to train and validate the text classifier.
- **pandas**: A software library for data manipulation and analysis. It provides data structures and functions needed to manipulate structured data.
- **re**: This module provides regular expression matching operations
- **sklearn**: Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms.
- **collections**: This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers.
- **imblearn**: A Python library to tackle the curse of imbalanced datasets in machine learning.

(b) Data Acquisition

The initial step involved getting the dataset provided by the professor and understanding its contents. The `news-classification.csv` file is a Comma-Separated Values file, a common file type for distributing large amounts of data over the internet. This type of data type can be viewed as a large array of structs that contain a lot of information, but we only need the following columns:

- `category_level_1`: Name of text's category (*strings*).
- `category_level_2`: Name of text's subcategory (*strings*).
- `content`: The actual text content (*strings*).

The rest of the columns are not necessary because they do not give us some kind of important information about the text's contents.

As we are using Python for this project, in order to load this CSV file into memory, we used `pandas`'s `read_csv()` function that automatically imports the necessary file to a `Dataframe` format.

```
df = pd.read_csv(fname)
texts = df['content'].apply(clean_text)
label_level1 = df['category_level_1']
label_level2 = df['category_level_2']

return texts, label_level1, label_level2
```

(c) Data Cleaning

Text is just a sequence of words, or more accurately, a sequence of characters. However, when we are usually dealing with language modelling or natural language processing, we are more concerned about the words as a whole rather than focusing only on the character-level depth of our text data. One explanation for this is the lack of "context" for individual characters in the language models.

The moment data are imported into the RAM, preparation begins in order to transform the text from human to machine understandable. First of all, lower casing of all the letters is very important and used for better handling of the file. Everything inside the content array that doesn't give enough information can be considered noise and needs to be removed. A great example of "noise" is:

- URLs,
- Email addresses,
- Lines like "This post was published on the site" (*which can be often found at the start of an article*),
- Multiple space or new line characters,
- Punctuation,
- Stopwords

In the preprocessing phase of text classification, one critical step and very useful technique is the removal of stopwords, which are words that do not contribute significant meaning to the text and are thus considered irrelevant for analysis. The Natural Language Toolkit (NLTK), a comprehensive library for natural language processing in Python, provides an extensive dictionary of stopwords across multiple languages. Utilizing NLTK's stopwords dictionary allows for the efficient filtering out of common words such as "the", "is", "in", and "and", which appear frequently in text but do not carry substantial information relevant to the classification task. The process involves iterating over the words in the dataset and removing those that are present in the NLTK stopwords list. This reduction in dataset size not only streamlines the computational process by focusing on words that carry more meaning but also improves the model's ability to learn by concentrating on content that is more likely to influence the classification outcome.

```
def clean_text(text):
text = text.lower()
text = text.replace('\xa0', ' ')  # Remove non-breaking spaces
text = re.sub(r'http\S+|www.\S+', '', text)  # Remove URLs
text = re.sub(r'\S+@\S+', '', text)    # Remove email addresses
text = re.sub(r'\n', ' ', text)  # Replace newline characters with space
text = re.sub(r'^.*\b(this|post|published|site)\b.*$\n?', '', text, flags=re.
   MULTILINE)    # Remove lines like 'This post was published on the site'
text = re.sub(r'\\(?!n|r)', '', text)   # Remove anything but backslashes
text = text.replace('[\r \n]\n', ' ')   # Remove newlines
text = re.sub(r'[\r\n]{2,}', ' ', text)
text = re.sub(r'from[: ]* ', '', text) # Remove 'from' at the beginning of
   the text
text = re.sub(r'  ', ' ', text) # Remove double spaces
text = re.sub(r'\(photo by .*\)', '', text) # Remove lines like '(photo by
   reuters)'

words = text.split()
filtered_words = [word for word in words if word not in stop_words]
text = ' '.join(filtered_words)

return text
```

(d) Dataset Split

In the development of this text classifier, a critical step in the methodology is the partitioning of the dataset into training and validation subsets. This process is essential for training the model effectively and evaluating its performance, employing a standard split ratio of 80% for training data and 20% for validation data. Such a division is strategically chosen to provide the model with a sufficiently large training dataset, enabling it to learn the underlying patterns of the text, while also reserving a representative portion of the data for performance evaluation and tuning. The use of a validation set, separate from the training set, is pivotal in detecting and mitigating overfitting, ensuring that the model generalizes well to new, unseen data. Also, the selected ratio we chose is very popular in bibliography and on the internet as well.

To facilitate this data partitioning, we utilize the `train_test_split` function provided by the `sklearn` library, a tool renowned for its robustness and ease of use in the machine learning community. This function streamlines the process of randomly dividing the dataset according to the specified proportions, ensuring that the split is both efficient and reproducible. By leveraging this method, we can maintain the integrity

of the data's distribution, ensuring that both the training and validation sets are representative of the overall dataset. This approach not only simplifies the preprocessing workflow but also lays a solid foundation for the subsequent training phase, enabling a systematic and controlled development of a high-performing text classification model. However, when training and testing models, we always want to remain mindful of data leakage. We cannot allow any information from outside the training dataset to "leak" into the model, so we must be really thoughtful about it too.

```
def preprocess_data(texts, labels, test_size = 0.2, max_words=10000, max_len
    = 200):

texts = texts.apply(clean_text)

# Split data into training and testing sets
train_texts, test_texts, train_labels, test_labels = train_test_split(texts,
    labels, test_size=test_size)
```

The "preprocess_data" function is designed to prepare and preprocess textual data for text classification tasks. It accepts several parameters, like:

- "`texts`" and "`labels`": These are the input parameters of the function. "Texts" is a list of series of text data that you want to preprocess, and "labels" are their corresponding labels.
- "`test_size`": This is the proportion of the dataset to include in the test split. It is set to 0.2 by default, meaning that 20% of the data will be used for testing and the rest for training.
- "`max_words`", "`max_len`": These parameters are used in tokenization process where max_words is the maximum number of words to keep, based on word frequency and max_len is the maximum length of all sequences.
- "`texts=texts.apply(clean_text)`": This line applies a function `clean_text` to every item in texts. The clean_text function is not defined in the provided code, but it's likely that it performs some sort of cleaning operation on the text such as removing punctuation, converting to lowercase, removing stopwords, etc.
- "`train_test_split`": This is a function from `sklearn.model_selection` that splits a dataset into training set and test set. The test_size parameter determines the proportion of the original data that is put into the test set.

In summary, this function encapsulates a comprehensive preprocessing workflow that cleans the input texts, splits the data for training and evaluation, and addresses class imbalance to prepare the data for effective model training.

(e) Handling of Class Imbalance

During the text cleaning and preprocessing procedure, we noticed that there's a big class imbalance that affected the performance of our classifier. Addressing class imbalance is crucial for a balanced and reliable performance across all classes in text classification. Class imbalance is a common challenge in text classification tasks that can result in biased models favouring the majority class, leading to poor performance on the minority class. Figure 1 shows that `lifestyle` class has the lower appearance in the set and that means our classification system cannot detect it easily.

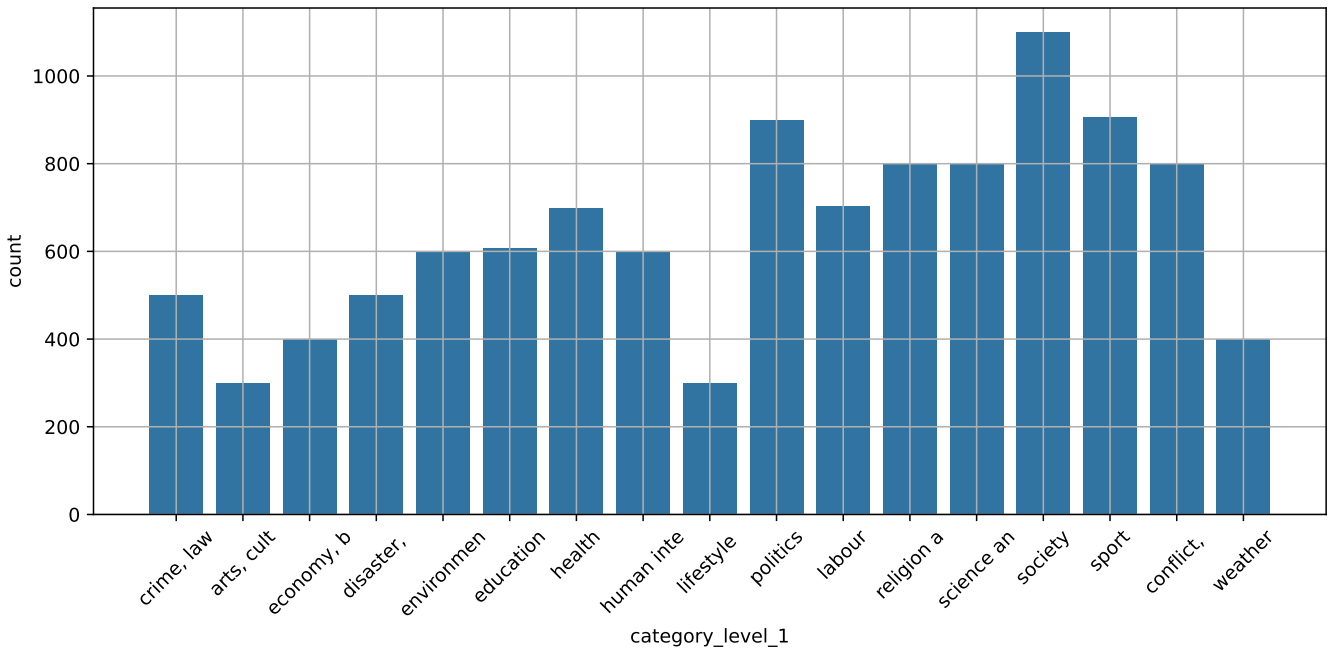

Figure 1: Initial class distribution of the dataset

There are quite some techniques to handle imbalanced datasets, but most of them are difficult to implement. However, to address this issue in our dataset, we utilized the `RandomOverSampler` method provided by the `imblearn` library. This technique involves artificially augmenting the under-represented classes in the training set by randomly replicating instances until all classes achieve a similar size. By doing so, we ensure that the neural network does not become biased towards the more frequent classes and can learn the characteristics of all classes equally. This step is crucial for improving the model's ability to generalize well across the entire range of classes, particularly for those that are less represented in the original dataset.

In the implementation phase, `RandomOverSampler` was applied after splitting the dataset into training and validation sets but before the model training process. This sequencing is intentional to prevent the oversampling process from influencing the validation set, thereby maintaining its integrity as a representative sample of real-world data. The application of `RandomOverSampler` is straightforward, thanks to the intuitive API of `imblearn`. With a few lines of code, we were able to fit the sampler to our training data, resulting in a modified training set with balanced class distributions. The distribution after balancing can be shown in figure 2. This technique works by randomly duplicating instances from the minority class in the training dataset to increase its representation. This oversampling process helps to balance the class distribution and can lead to improved model performance by giving the model more examples of the minority class to learn from.

This issue is only met for `category_level_1` as in `category_level_2` the distribution is 100 for almost all of the categories.

```
# Handling of Class Imbalance
ros = RandomOverSampler(random_state=777)
train_texts, train_labels = ros.fit_resample(train_texts.values.reshape(-1,1)
    , train_labels)
train_texts = pd.Series(train_texts.flatten())
```

However, it's important to note that oversampling can also lead to overfitting since it duplicates the

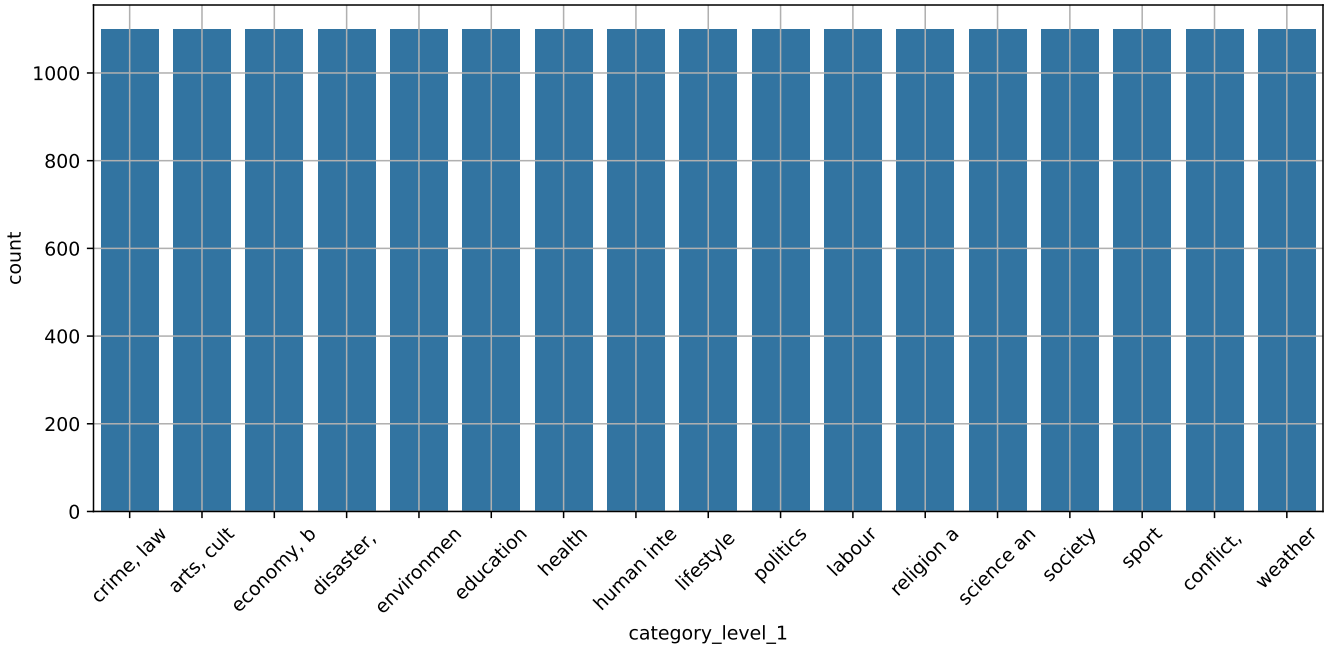

Figure 2: Final class distribution of the given dataset

minority class instances. Therefore, it's always a good idea to evaluate the model performance carefully after applying any oversampling technique.

(f) Convert to one-hot encoding and Tokenize

One of the most crucial preprocessing steps is converting our text data into a format that can be processed by neural networks. This involves two main processes: one-hot encoding and tokenization. Here's an analysis of these steps and how they fit into our project:

One-Hot encoding is a process of converting categorical data variables so they can be provided to machine learning algorithms to improve predictions. With one-hot, we convert each categorical value into a new categorical column and assign a binary value of 1 or 0. Each integer value is represented as a binary vector. One-hot encoding can be applied to text data to turn every word into a distinct vector. This is helpful because neural networks need numerical input and cannot comprehend text directly. Your text data can be efficiently converted into a format that your CNN can understand by one-hot encoding. One-hot encoding, however, produces high-dimensional vectors, which can be sparse and ineffective, particularly for large vocabularies (with dimensions equal to the size of your vocabulary). This is a simple way to represent your text data, but it may not be the most memory-efficient method.

Text Tokenization is a data preprocessing technique of converting a separate piece of text into smaller parts like words, phrases, or any other meaningful elements called tokens which makes counting the number of words in the text easier. The proposed system performed tokenization at the word level so as to consider the sentiment polarity of each word.

References

[1] A. Hamza, “Effectively pre-processing the text data part 1: Text cleaning,”

[2] A. H. S. S. V. D. S. S. Sayyaparaju, “Sentiment analysis of imdb movie reviews,”