ANKARA UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

COM 4061 PROJECT REPORT

Contextual Diet Chatbot Using Recommendation System

Mehmet Bayram Alpay **Çağrı Zengin**

20290310 **19290344**

Begüm Mutlu Bilge

January, 2024

ABSTRACT

The purpose of this graduation project report is to gather information about the current technologies used by researchers and developers who utilize artificial intelligence to create diet applications globally. The report aims to provide an overview of natural language processing technology and deep learning techniques, and to use them to develop a diet chatbot that can be utilized by anyone. The primary aim of this report is to advance our knowledge of machine learning, deep learning, and NLP technologies and design a recommendation system. The project will implement the following tools: Google Colab, Jupyter Notebook, Anaconda, Hugging Face, as well as the following libraries: NumPy, Pandas, PyTorch, NLTK, Scikit-learn, Matplotlib, and Collections. The report includes a comprehensive literature review, explaining the current research on recommendation systems and contextual chatbots. The report also includes a detailed explanation of the terms that are often used in the wrong contexts and create confusion in people's minds. During the project, advanced applications developed in the healthcare field were analyzed, the recommendation system process was detailed, and chatbot examples were created using natural language processing techniques. As a result of this project, with the help of my advisor and teammate, we have taken the traditional programming approach to the next level. We now have an application that can compete with other studies in this field or guide future studies. This project has also encouraged us to realize the project we planned to undertake in the research techniques 2 course and has boosted our self-confidence. At the end of the project, all content, articles, codes, and reports related to the subject have been filed by us and can be shared publicly in the future.

TABLE OF CONTENTS

1. INTRODUCTION

1. 1. Background and Subjects Declarations

Throughout this project, we had the opportunity to research and learn the underlying logic of recommendation systems, what to consider when creating a contextual chatbot, how to create a dataset, and how to implement various developed applications. This opportunity allowed us to increase our knowledge of machine learning and deep learning and learn new technologies. In short, our project includes the processes of controlling, directing, and creating meaningful output of data, which makes the recommendation system part important.

• **Recommendation systems** are software systems designed to suggest products, content, or services tailored to the interests of users or customers. These systems typically analyze users' past preferences, behaviors, or preferences of similar users to provide personalized suggestions, enhancing the user experience and increasing engagement.

NLP researchers and staff use data collected from healthcare organizations, patients, and other sources. It combines various skills, including programming languages, NLP techniques (bag of words, tf-idf, etc.), computer science and current tools to analyze data and produce logical output accordingly. Every stage, from creating a dataset to making meaningful inferences and applying the methods, can sometimes become so difficult and inextricable that an attempt is made to solve this problem by dividing it into as few parts as possible. In this way, such projects become one of the most important tools that have the potential to use and transform information to find solutions to people's problems, to convey the right information as simply as possible, and to improve the user experience. For example,

• **Natural Language Processing (NLP) techniques**, such as Bag-of-Words, TF-IDF, and NLTK, are pivotal in text analysis. Bag-of-Words represents text by word frequency, TF measures word frequency in a document, and TF-IDF combines term importance across the corpus. NLTK, a Python library, aids in tasks like tokenization and part-of-speech tagging, making it essential for NLP applications.

• **Python and Machine Learning Techniques**, Python is incredibly useful for building machine learning models. especially when creating a diet chatbot. K-Means clusters

user data, aiding in personalized recommendations, while RandomForest's ensemble learning provides robust predictions for food suggestions. Both models, seamlessly integrated with Python, elevate the chatbot's ability to offer tailored dietary guidance, showing their impact in the field of machine learning applications.

2. CONTEXTUAL CHATBOT AND RECOMMENDATION SYSTEM

2.1. Definitions and Declarations about Contextual Chatbot

Chatbots have become one of the important parts of our lives with chatgpt and are at the forefront of the future of artificial intelligence. It is of great importance to understand what chatbots are and what value they can add to our lives. In the past, studies were carried out on similar applications written in fuzzy logic structure or traditional programming, and these were used especially as mobile applications. However, today's data environment is often unstructured or semi-structured. A similar structure is observed in the health sector, but there is also an increase in structured data. Traditional approaches are not suitable for the input problem, large datasets and text-based datasets.

This is exactly why we need more advanced analytical tools and algorithms, and NLP plays a very important role in this transformation. NLP techniques help us understand prompts, intent files created for the chatbot, the context of human language, and even nuances. For example, NLP can use created files or datasets to detect disease information, focus on needs, distinguish positive and negative characteristics, and classify and categorize different types of days and foods according to similar content.

As a result of the need for NLP technology and the rapid development of these technologies, chatbots have become structures that can produce solutions to many of our current problems using NLP technology. However, this technology cannot currently perform well in all areas and it appears that it has serious shortcomings, especially in the field of healthcare. At this point, it may be necessary to conduct deeper studies specific to each field and problem-based special chatbots can be developed. Creating a diet chatbot is also one of the best examples of this.

2.2. Tools and Platforms for Chatbots

2.2.1. Data Collection Tools

Different datasets may be needed for chatbots and recommendation systems. High-quality structured data is especially important for the recommendation system. For the chatbot part, enriching the intent file as much as possible will increase the number of classes and examples, which will contribute to our model giving better results in the training and test set.

• Dialogflow, Rasa, Luis.ai, Wit.ai for intents.json examples

• Kaggle, UCI Machine Learning Repository, Amazon Product Reviews

2.2.2. Data Analysis Tools

Being able to extract meaning from textual data and label it is very important for NLP and Chatbots. Text analysis tools help NLP practitioners make informed decisions by allowing them to derive meaning and obtain information, and contribute significantly to the fields we work in, such as chatbots.

• Natural Language Processing libraries like spaCy, NLTK

• Machine learning frameworks for NLP, including TensorFlow and PyTorch

2.2.3. Data Storage and Processing

The data available for this project was quite limited, but in order to progress such projects, a lot of data is needed and when this process is reached, storing and processing the data becomes important. NLP generally involves working with unstructured or semi-structured textual data, which requires appropriate tools. Some tools that are likely to be used in future processes are listed below.

• Distributed data processing frameworks like Apache Spark for large-scale text data processing

• NoSQL databases for storing and retrieving unstructured text data

• Cloud-based services such as AWS Comprehend or Azure Text Analytics for language processing

2.2.4. Visualization Tools for Dataset and Results

Visualizing data, train and test results helps identify patterns, trends and important insights. Important problems such as overfitting can be detected and models can be updated for precautions. Although matplotlib is generally preferred, there are visualization tools specific to NLP problems.

• Word cloud generators (e.g. WordCloud or Tableau) to visualize word frequency

• Tools such as Plotly or Power BI

• SpaCy's dependency parsing visualization tools like dependency visualizer

2.2.5. Chatbot, NLP Community and Resources

Chatbot and Natural Language Processing (NLP) developers leverage a variety of tools, each offering unique features and functionality. Among these, Rasa stands out as a prominent open-source framework that provides developers with powerful capabilities to create context-aware and intelligent chatbots. Hugging Face serves as a central hub for the NLP community, offering NLP practitioners a wealth of resources, pre-trained models, and collaboration spaces. Its comprehensive range of tools makes it an invaluable platform for a variety of tasks, from text creation to sentiment analysis. Data published by Kaggle, or world health centers are an important source of information. Systems such as the Food Standard Agency can provide important information about the systems to be implemented in this field. Collectively, these tools form the backbone of the chatbot and NLP environment, providing developers with the resources needed to create cutting-edge conversational experiences.

2.3. How Are Chatbots Transforming Our Lives?

Chatbots, including advanced models like ChatGPT, are revolutionizing the way we interact with technology, transcending traditional communication boundaries and impacting various aspects of our daily lives. These intelligent conversational agents serve as virtual assistants, providing seamless and responsive experiences in a multitude of domains.

In customer service, chatbots have become valuable assets, offering instant assistance, efficiently resolving queries, and enhancing overall satisfaction.

Beyond customer service, chatbots are reshaping education by acting as personalized learning companions. They adapt to individual learning styles, provide tailored explanations, and democratize access to education by making learning resources widely available.

The healthcare sector has witnessed a transformative impact with the integration of chatbots. Health-focused chatbots contribute to improved accessibility and efficiency. They assist users in monitoring health metrics, provide timely reminders for medication, and offer preliminary medical advice. This proactive approach enhances healthcare accessibility, especially for individuals in remote or underserved areas.

2.3.1 How Chatbots are Conducted ?

Developing chatbots involves a dynamic and iterative process. Starting with a clear conception, the purpose and target audience are identified, laying the foundation for design. The design phase focuses on creating a natural conversation flow and a positive user experience. Implementation follows, integrating natural language processing (NLP) capabilities to understand and respond to user inputs. Training the chatbot involves exposing it to diverse datasets and refining its responses over time. Evaluation and user feedback drive continuous refinement, ensuring the chatbot evolves adaptively. This iterative approach allows chatbots to continually enhance their capabilities as language understanding and user context improve or new data becomes available.

2.3.2. Chatbots and NLP: Solving Crucial Problems

With technology all around us, chatbots and natural language processing (NLP) are now a great pair to deal with different kinds of problems we face today. Text data such as social media continues to increase. Chatbots with NLP capabilities can help process and use such large amounts of text. Chatbots can improve the user experience and make communication more efficient.

Healthcare Chat is a great example of how robotics and his NLP can be used to improve accessibility and efficiency. Chatbots with NLP capabilities can help patients monitor their health metrics, provide medication reminders, and make healthy eating

recommendations. Individuals residing in remote areas or those with limited access to healthcare can greatly benefit from these tools. The utilization of such technologies can encourage a proactive healthcare approach and facilitate improved healthcare accessibility.

2.4. Chatbot Intent Files, Tokenization, Stemming, Lowercase, Stopwords

2.4.1. Tokenization, Stemming, Lowercase and Stopwords

Natural Language Processing (NLP) is a field of artificial intelligence that revolves around the interaction between computers and human language. NLP plays a crucial role in enabling machines to understand and process human language. Concepts such as tokenization, stemming and stop words are important NLP techniques. Tokenization is the process of splitting a text into words or subwords. This process helps the computer analyze and understand the language better.

Figure 2.1. Tokenization Example

Here are some of the reasons why tokenization is so important:

Tokenization makes it easier to work by dividing a continuous text stream into subunits.

Each symbol represents a word, and by parsing symbols like a word, meaning and context can be captured more easily.

It makes it possible to count the occurrences of certain words or expressions that require tokenization before using structures such as bag of words. This makes it possible to convert words into numerical values and provide input to deep learning models. In this way, the machine can understand our own language.

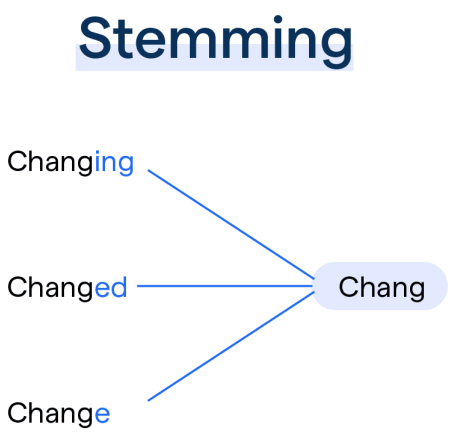


Figure 2.2. Stemming Example

Stemming is another language processing technique. This method tries to find word roots or put similar words in the same form. Although there are different versions, PorterStemmer was used for our application and these structures may work differently from language to language. As in the image above, the words Changing, Changed and Change can be reduced to the Chang form for the English language. This process provides better matching between similar words and makes it easier to extract meaning in text analysis. This technique may be preferred as a pre-processing step.

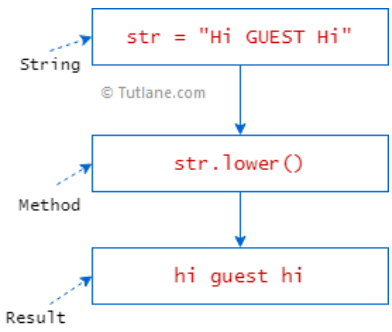


Figure 2.3. Lower Case Example

Figure 2.4. Commonly used Stopwords

Lowercase and Stopwords techniques are the last of the techniques we will talk about. Lowercase operation is a frequently used function in Python that converts all letters to lowercase, regardless of the software. For example, it transforms an example such as "HeLLo mY NAME is" into the form "hello my name is", and this helps us to ignore both spelling errors and simple spelling differences.

Stopwords, on the other hand, are words that are commonly used in the language and do not carry any significant meaning. In addition to the common words such as "and," "the," and symbols such as “!, ., ?" they may also include other frequently used words. These words are often ignored or filtered out. Since they are found very frequently, they can sometimes distract us from the main purpose we want to focus on. Therefore, removing stopwords is an important step.

2.4.2. Intents File and Points to Consider

Intent files refer to the file type used in chatbot design. These files contain users' intentions, patterns related to these intentions, and possible responses that can be given if the conditions are met. The intents file is of great importance for contextual chatbots because determining the intent is a critical point in responding to user requests more effectively and meaningfully.

Intentions are the main issues the user is trying to express. Questions, requests, and topical requests may correspond to different intentions. Patterns are user expressions paired with each intent. Although it is important for the bot to determine the correct intention, a response must be given by the bot after the intention is determined. At this point, an action is taken from the chatbot to the user using the response section.

In general, creating datasets or such files is extremely laborious, and such files need to be created specific to each study, and this can be extremely costly. However, such files are indispensable for contextual chatbots and are important to improve user experience and communicate more effectively.

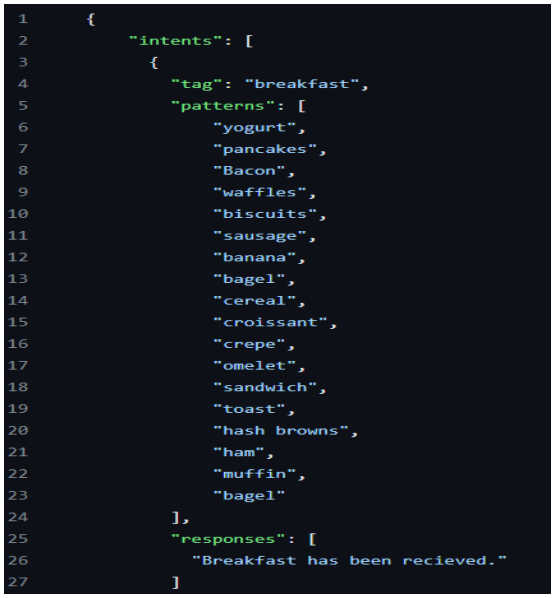
Figure 2.5 shows a section of the intent file we use in our application. The possible answers for the "greeting" and "goodbye" tags perceived by the user are clearly visible. Increasing such structures for all possible scenarios that may occur in a conversation is decisive in terms of the quality of the work done. Shaping this file according to the

field we work in may make our job easier. For example, if we are designing a chatbot related to the health sector, it is mandatory to create tags related to diseases.

```
{
  "intents": [
    {
      "tag": "greeting",
      "patterns": [
        "Hi",
        "Hey",
        "How are you",
        "Is anyone there?",
        "Hello",
        "Good day"
      ],
      "responses": [
        "Hey :-)",
        "Hello, thanks for visiting!",
        "Hi there, what can I do for you?",
        "Hi there, how can I help?"
      ]
    },
    {
      "tag": "goodbye",
      "patterns": ["Bye", "See you later", "Goodbye"],
      "responses": [
        "See you later, thanks for visiting!",
        "Have a nice day",
        "Bye! Come back again soon."
      ]
    },
    {
```

Figure 2.5. Intents.json file example

The "patterns" part can be perceived as the words searched in the text to determine each tag. For example, in order to detect the "greeting" tag, examples such as "Hi, Hey, How are you?, Hello, Is anyone there, Good day" must be caught in the text. When such examples are encountered in the text, our chatbot selects a random answer from the responses section and sends it to the user.



```
{
    "intents": [
      {
        "tag": "breakfast",
        "patterns": [
            "yogurt",
            "pancakes",
            "Bacon",
            "waffles",
            "biscuits",
            "sausage",
            "banana",
            "bagel",
            "cereal",
            "croissant",
            "crepe",
            "omelet",
            "sandwich",
            "toast",
            "hash browns",
            "ham",
            "muffin",
            "bagel"
        ],
        "responses": [
          "Breakfast has been recieved."
        ]
```

Figure 2.6. Meal based Intent file

In the example of using the intent file given in Figure 2.6, it can be understood which part of the day the foods mentioned by the user are referring to and appropriate suggestions can be designed in the background. For example, patterns such as "yogurt, toast, pancakes, croissant" and other patterns can be created for the "breakfast" tag. The concept is detected by the chatbot based on user input. Of course, language and eating habits must also be taken into account when designing these chatbots. But in general, intent files are very important for contextual chatbots and serious attention should be paid to their design.

2.5. Neural Network, Bag of Words, TF-IDF, TF Creation

After the pre-processing stages such as tokenization, stemming, lowercase and stopwords that we mentioned in the previous sections, input must be provided for the neural network using structures such as tf, tf-idf, bag of words. Afterwards, the neural network is trained. This section includes the design of these functions.

TF stands for term frequency and holds values that represent how often any word occurs in a document.

TF-IDF stands for Term Frequency-Inverse Document Frequency and It tries to find out how important a word is by dividing the frequency of occurrence of a particular word in the text by the total number of words.

BoW means Bag of Words and expresses the frequency of words in the text with a vector. Each of the vectors tells you how many times the word appears in the text.

```
def bag_of_words(tokenized_sentence, words):
    """
    return bag of words array:
    1 for each known word that exists in the sentence, 0 otherwise
    example:
    sentence = ["hello", "how", "are", "you"]
    words = ["hi", "hello", "I", "you", "bye", "thank", "cool"]
    if sentence[i] is available in words[i], then put one for bog.
    bog   = [  0 ,    1 ,    0 ,   1 ,    0 ,    0 ,      0]
    """
    # stem each word
    sentence_words = [stem(word) for word in tokenized_sentence]
    # initialize bag with 0 for each word
    bag = np.zeros(len(words), dtype=np.float32)
    for idx, w in enumerate(words):
        if w in sentence_words:
            bag[idx] = 1

    return bag
```

Figure 2.7. Bag of Words function

The function in Figure 2.7 defines a function that implements the bag of words technique. The purpose of this technique is to represent a text by creating a vector of vocabulary. The function takes as input the tokenized sentence and with it a list of all words. It then searches for each word in the sentence in the list of all words, if the word is present it sets the corresponding element to 1. The resulting vector expresses a binary representation where 1 indicates the presence of a word and 0 indicates its absence.

```
def tfidf(tokenized_sentences, all_words):
    sentence_words = [stem(word) for word in tokenized_sentence]

    # TF-IDF Vectorizer
    tfidf_vectorizer = TfidfVectorizer(vocabulary=all_words, lowercase=True)
    tfidf_vector = tfidf_vectorizer.fit_transform([' '.join(sentence_words)])

    # Convert sparse matrix to dense array
    tfidf_array = np.array(tfidf_vector.todense())[0]

    return tfidf_array
```

Figure 2.8. TF-IDF function

The TF-IDF function in Figure 2.8 takes tokenized_sentence and words parameters, just like the previous one. While tokenized_sentence means the input sentence that is tokenized, words is the list of all words. After the stem and lowercase operations are performed, it converts the sentence into a vector using the TfidfVectorizer under sklearn. It converts the resulting sparse matrix into a NumPy array and what we get is the tokenized_sentence converted into a TF-IDF vector.

```
def tf(tokenized_sentence, all_words):
    sentence_words = [stem(word) for word in tokenized_sentence]
    word_counts = Counter(sentence_words)

    # TF hesaplama
    tf_array = [word_counts[word] if word in word_counts else 0 for word in all_words]

    return np.array(tf_array)
```

Figure 2.9. TF function

The TF function in Figure 2.9 performs the function of calculating the Term Frequency value using the tokenized_sentence and words parameters. After the stemming process is completed, the Counter method under the collections library is applied, this method ensures that each root is counted and keeps the values. It then calculates the Term Frequency for each word in the specified dictionary. The TF of each word is obtained by dividing the number of that word in the sentence by the total number of words in the sentence. Finally, it returns it as a numpy array.

```python
import torch
import torch.nn as nn

#feed forward neural net with 2 hidden layers
#bag of woods is input
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes, dropout_rate=0.5):
        super(NeuralNet, self).__init__()
        self.l1 = nn.Linear(input_size, hidden_size)
        self.dropout1 = nn.Dropout(p=dropout_rate)
        self.l2 = nn.Linear(hidden_size, hidden_size)
        self.dropout2 = nn.Dropout(p=dropout_rate)
        self.l3 = nn.Linear(hidden_size, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.dropout1(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.dropout2(out)
        out = self.l3(out)
        # we don't want activation and no softmax at the end because later on there will be cross entropy loss
        return out
```

Figure 2.10. Feed Forward Neural Network

The function is generally a feed-forward neural network model created using the PyTorch library. It is a code created by inheritance, but it generally has a simple structure. It is derived from Pytorch's nn.Module class and first layers, dropouts and activation functions are defined in the __init__ section because these are the basic things we need to use in our neural network. The dropout section is set to protect against overfitting problems and its default value is 0.5. This value can be changed by the user in the run section if desired. input_size, hidden_size and output_size mean input size, hidden layer size and output size and these values are used. The forward part allows the model to progress from entry to exit. Starting from the input, linear layers, ReLU activation functions and dropout layers are applied respectively. Finally, the output value is returned. As a note, the activation function and softmax were not used because a loss function such as cross entropy loss will be used later.

2.6. Hands-on calling Libraries, Functions and Training Our Model for Contextual Chatbot Part

1. Import Required Libraries

```python
import numpy as np
import random
import json
from collections import Counter
import matplotlib.pyplot as plt
import matplotlib as mpl
import torch
import torch.nn as nn
import torchmetrics
from torch.utils.data import Dataset, DataLoader
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from torcheval.metrics.functional import multiclass_f1_score
from nltk_utils import bag_of_words, tokenize, stem
from model import NeuralNet
import pdb
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from imblearn.over_sampling import RandomOverSampler
```

Figure 2.11. Import Libraries

In the Import libraries section, we imported the libraries required for our code. NumPy is a very important library because it is useful for storing arrays and vector operations. We imported matplotlib.pyplot because we do approximately 1000 epochs for each training and observing the training process is very important to analyze. We imported Pytorch and Pytorchmetrics because these modules will be helpful for both training the model and adjusting metrics such as accuracy, recall, precision, f1-score. We generally use the sklearn library for pre-processing operations. The last parts are our own functions and model that we mentioned in the previous stages.

2. Reading the Dataset and Preprocessing Steps

```
with open('intents.json', 'r') as f:
    intents = json.load(f)

all_words = []
tags = []
xy = []
# loop through each sentence in our intents patterns
for intent in intents['intents']:
    tag = intent['tag']
    # add to tag list
    tags.append(tag)
    for pattern in intent['patterns']:
        # tokenize each word in the sentence
        w = tokenize(pattern)
        # add to our words list
        all_words.extend(w)
        # add to xy pair
        xy.append((w, tag))

# stem and lower each word
ignore_words = ['?', '.', '!']
all_words = [stem(w) for w in all_words if w not in ignore_words]

# remove duplicates and sort
all_words = sorted(set(all_words))
tags = sorted(set(tags))
```

Figure 2.12. Calling the intent File and Preprocess Step

We transfer the intent file that we explained in the previous stages to our code here, and then we get our patterns for each intent with the for loop and tokenize them. After this process, the patterns corresponding to each intent become a dictionary. After all tags and patterns are scanned, we remove stopwords from our list of words and apply the stemming process. Next,

there is the set operation. This ignores the repetition of the same words and saves us from doing too much processing.

```
# create training data
X_train = []
y_train = []
#tf için for döngüsü
for (pattern_sentence, tag) in xy:
    # X: term frequency for each pattern_sentence, which is already tokenized
    tf_vector = tf(pattern_sentence, all_words)
    X_train.append(tf_vector)
    # y: PyTorch CrossEntropyLoss needs only class labels(#'s for all labels), not one-hot
    label = tags.index(tag)
    y_train.append(label)
```

Figure 2.13. Usage of TF Function and Train Lists

X_train holds the values of the TF vector and Y_train holds the labels. Here we get the values created for each tag using the xy list, and both tags and values are taken together. The TF vector of each pattern_sentence across all words is calculated and added to the X_train list. Labels are obtained and added to Y_train using index information.

```
X_train = np.array(X_train)
y_train = np.array(y_train)
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.2, random_state=42, shuffle=True)
# Hyper-parameters- TEST THESE WITH DIFFERENT #'S
num_epochs = 1000
batch_size = 9
learning_rate = 0.001
input_size = len(X_train[0])
# 8 is based on tags so will be different for charlie
hidden_size = 9
output_size = len(tags)
```

Figure 2.14. Split the Dataset and Adjust Hyper-parameters

This part includes preliminary preparation procedures. Using the X_train and y_train lists we created, we split our dataset into 20% test set and 80% train set. While doing this, we shuffle the dataset and then throw random_state because in this way, the same distribution is always maintained and we can obtain the same results. Defining variables in the next section can be considered as setting hyperparameters because we will use these values when training our neural network.

3. DataLoader Creation, Loss and Metrics before Training

In Figure 2.15, the ChatDataset function is created to produce a special dataset for the DataLoader section. It is derived using Dataset from the Pytorch library. In the __init__ section, there is a part to get the size of the data and get the X and y values. __getitem__ returns the instance bound to a given index. The __len__ method returns the total number of

samples of the dataset. PyTorch's DataLoader class is used to load the dataset and pass it to the model in minibatches. The train_dataset and test_dataset instances derived from the ChatDataset class are set up for use with the DataLoader. The batch_size parameter determines the number of samples to be used in each iteration. In the last 2 lines, the device to be used, that is, the CPU or GPU, is set, and then our feed forward neural network model that we explained in the previous stages is created.

```
class ChatDataset(Dataset):
    def __init__(self, X, y):
        self.n_samples = len(X)
        self.x_data = X
        self.y_data = y

    # support indexing such that dataset[i] can be used to get i'th sample
    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    # we can call len(dataset) to return the size
    def __len__(self):
        return self.n_samples

# Eğitim veri setini DataLoader ile yaratın
train_dataset = ChatDataset(X_train, y_train)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, num_workers=0)

# Test veri setini DataLoader ile yaratın
test_dataset = ChatDataset(X_test, y_test)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, num_workers=0)

"""dataset = ChatDataset()
#used pytorch because can automatically iterate over code and use as batch training
train_loader = DataLoader(dataset=dataset, batch_size=batch_size, shuffle=True, num_workers=0)"""

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = NeuralNet(input_size, hidden_size, output_size).to(device)
```

Figure 2.15. ChatDataset sınıfı ve Data Loaders

Figure 2.16 contains the loss function and optimization algorithm that will be used in training the model. Additionally, accuracy, precision, and recall metrics to be used during training have been defined. The loss function is CrossEntropyLoss, and the loss value is found by performing operations between the model output and the real labels. We can say that as this value decreases, our model learns more. In the optimizer section, we use Adam optimizer and the learning_rate value is the value in the variable we specified in the previous sections.

Finally, there is the definition of accuracy, precision, and recall metrics. We are dealing with the multi-class problem and use these metrics to evaluate the performance of the model.

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

epoch_list=[]
loss_list=[]

accuracy_metric = torchmetrics.Accuracy(task="multiclass", num_classes=output_size)
precision_metric = torchmetrics.Precision(average='weighted', num_classes=output_size, task="multiclass")
recall_metric = torchmetrics.Recall(average='weighted', num_classes=output_size, task="multiclass")
```

Figure 2.16. Creation of Loss, Optimizer and Metrics

4. Model Training and Print Results

```
# Train the model
for epoch in range(num_epochs):
    total_accuracy = 0
    total_precision = 0
    total_recall = 0
    total_f1 = 0

    for (words, labels) in train_loader:
        words = torch.tensor(words, dtype=torch.float32).to(device)
        labels = torch.tensor(labels, dtype=torch.long).to(device)

        # Forward pass
        outputs = model(words)
        loss = criterion(outputs, labels)

        # Accuracy metriğini güncelle
        predictions = torch.argmax(outputs, dim=1)
        batch_accuracy = accuracy_metric(predictions, labels)
        total_accuracy += batch_accuracy.item()

        # Precision, Recall, F1 Score metriklerini güncelle
        precision_metric.update(predictions, labels)
        recall_metric.update(predictions, labels)
        # f1_metric.update(predictions, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Figure 2.17. Model Training Part

This section contains a loop that performs training of the model. The loop will run as many times as the number of epochs we specified earlier, that is, 1000 times. Each epoch represents the process of updating and learning the model using the training data set. The inner for loop operates on the train_loader, which is used to load the training dataset. In each iteration, one minibatch of data is taken. The words variable is given to our model and the outputs are compared with labels to obtain the loss value. With torch.argmax, the highest value is found on a row basis and this value is accepted as the predicted value, and then our metrics are updated. In the last part, the backpropagation operations required for training the model are performed. The gradients are reset, the backward gradients of the loss are calculated, and then the model parameters are updated using the optimization algorithm.

```
if (epoch+1) % 100 == 0:
    average_accuracy = total_accuracy / len(train_loader)
    average_precision = precision_metric.compute()
    average_recall = recall_metric.compute()
    # average_f1 = f1_metric.compute()
    f1_tensor = multiclass_f1_score(predictions, labels, num_classes=output_size)

    print("Train Results:")
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, Accuracy: {average_accuracy:.4f}, Precision: {average_pr
    epoch_list.append((epoch+1))
    loss_list.append(round(loss.item(), 4))
```

Figure 2.18. Update Average Values and Print the Results

The above section is used to print statistics about the history of the training process once in a certain number of 100 epochs. This is used to evaluate training performance and check for

possible cases of overlearning while following the training of the model. The f1-score metric from torcheval is used here because other libraries give errors and this part of the code was done differently. The print sections print the results of the metric values to the screen every 100 epochs.

```
# Test kısmı
model.eval()
test_accuracy = 0
with torch.no_grad():
    all_predictions = []
    all_labels = []
    for (words, labels) in test_loader:
        words = torch.tensor(words, dtype=torch.float32).to(device)
        labels = torch.tensor(labels, dtype=torch.long).to(device)

        # Forward pass
        outputs = model(words)

        # Accuracy metriğini güncelle
        predictions = torch.argmax(outputs, dim=1)
        batch_accuracy = accuracy_metric(predictions, labels)
        test_accuracy += batch_accuracy.item()

        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

average_test_accuracy = test_accuracy / len(test_loader)
print(f'Test Accuracy: {average_test_accuracy:.4f}')

# Calculate additional metrics
precision = precision_score(all_labels, all_predictions, average='weighted')
recall = recall_score(all_labels, all_predictions, average='weighted')
f1 = f1_score(all_labels, all_predictions, average='weighted')

print(f'Test Precision: {precision:.4f}')
print(f'Test Recall: {recall:.4f}')
print(f'Test F1-Score: {f1:.4f}')
```

Figure 2.19. Testing the Our Model and Print Results

This part of the code is used to evaluate the performance of the trained model on the test data set. It measures the performance on the test data set, using the parameters learned during training of the model. In the test part, that is, in the model.eval() part, operations such as dropout or backpropogation are not applied as in the train part. That's why we use no_grad. Other operations are very similar to the training section; the model is run for each mini batch and the metrics are updated using the results. Afterwards, the results are printed on the screen with the print section.

```
data = {
"model_state": model.state_dict(),
"input_size": input_size,
"hidden_size": hidden_size,
"output_size": output_size,
"all_words": all_words,
"tags": tags
}

FILE = "data.pth"
torch.save(data, FILE)

print(f'training complete. file saved to {FILE}')

fig, ax = plt.subplots(1,1, figsize=(10,5))
ax.plot(epoch_list, loss_list, linewidth=6, color='red')
ax.set_xlabel("Number of Epochs")
ax.set_ylabel("Loss")
ax.set_title('Training Loss for Chatbot Language Detection (TF-IDF)')
#change file name
plt.savefig('training_loss_for_chatbot.png',dpi=300, bbox_inches='tight')
```

Figure 2.20. Visualizing Part

The last part of the code for the neural network deals with recording the learned parameters of the trained model and some statistics of the training process. Additionally, the training process

is visualized by recording the loss values obtained during the training process as a graph. A dictionary named data is created and the parameters of the trained model are recorded here and then saved with torch using the .pth extension. The training process is visualized with the help of matplotlib.pyplot, using the loss_list and epoch_list we created while the model was being trained.

5. Alternative Method (RandomForestClassifier) with Same Structure

```
X = []
y = []

for (pattern_sentence, tag) in xy:
    tf_vector = tf(pattern_sentence, all_words)
    X.append(tf_vector)
    label = tags.index(tag)
    y.append(label)


le = LabelEncoder()
all_labels = np.array(y)  # Tüm etiketler
le.fit(all_labels)
y_encoded = le.transform(all_labels)

# Sınıf dengesizliğinin kontrolü
class_counts = {tag: y_encoded.tolist().count(tags.index(tag)) for tag in tags}
print("Class counts:", class_counts)

# Oversampling
oversample = RandomOverSampler(sampling_strategy='auto', random_state=42)
X_resampled, y_resampled = oversample.fit_resample(X, y_encoded)

class_counts_resampled = {tag: y_resampled.tolist().count(tags.index(tag)) for tag in tags}
print("Resampled Class counts:", class_counts_resampled)
```

Figure 2.21. LabelEncoder and Class Imbalance

The first parts of the code and the rest are very similar, but since we are using RandomForestClassifier instead of neural network here, we have made some changes accordingly. LabelEncoder is used to detect and evaluate all labels. If we had not used it, there would have been label differences between the train set and the test set, and this would have caused an error. Since our dataset was very small, we first checked the dataset and then performed data duplication to collect all data at the same level. This is something that is not recommended most of the time, but it worked for our code. Finally, we print the current data.

```
X_resampled = np.array(X_resampled)
y_resampled = np.array(y_resampled)
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)


rf_model = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42, warm_start=True)

cv_results = cross_val_score(rf_model, X_resampled, y_resampled, cv=skf, scoring='accuracy')

# Sonuçları yazdırma
print("Cross-validation results with oversampling:", cv_results)
print("Mean accuracy with oversampling:", np.mean(cv_results))

X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)
```

Figure 2.22. Split Dataset and Cross Validation

This section enables cross-validation using the Random Forest algorithm on a data set with class imbalance and prints the results with various performance metrics. StratifiedKFold does 5-fold cross-validation. For each floor, training and test sets are created using RandomForestClassifier, the model is trained and the accuracy score is calculated. The cross_val_score function returns the resulting accuracy scores as an array. The cross-validation results are printed to the screen. In the dataset division part, it is divided into 20% test and 80% train dataset, similar to the neural network part. random_state helps us always get the same distribution.

```
for epoch in range(1, epochs + 1):
    rf_model.n_estimators += 10
    rf_model.fit(np.array(X_train), np.array(y_train))

    # Eğitim seti üzerinde tahmin
    y_pred_train_rf = rf_model.predict(np.array(X_train))

    train_accuracy_rf = accuracy_score(y_train, y_pred_train_rf)
    train_precision_rf = precision_score(y_train, y_pred_train_rf, average='weighted', zero_division=1)
    train_recall_rf = recall_score(y_train, y_pred_train_rf, average='weighted', zero_division=1)
    train_f1_rf = f1_score(y_train, y_pred_train_rf, average='weighted', zero_division=1)

    train_accuracy_list.append(train_accuracy_rf)
    train_precision_list.append(train_precision_rf)
    train_recall_list.append(train_recall_rf)
    train_f1_list.append(train_f1_rf)

    # Test seti üzerinde tahmin
    y_pred_test_rf = rf_model.predict(np.array(X_test))

    test_accuracy_rf = accuracy_score(y_test, y_pred_test_rf)
    test_precision_rf = precision_score(y_test, y_pred_test_rf, average='weighted', zero_division=1)
    test_recall_rf = recall_score(y_test, y_pred_test_rf, average='weighted', zero_division=1)
    test_f1_rf = f1_score(y_test, y_pred_test_rf, average='weighted', zero_division=1)
```

Figure 2.23. Training the RandomForestClassifier

This section is a set of procedures to evaluate the performance of the Random Forest model by iteratively updating its training. The n_estimators parameter is increased by 10 units in each iteration, increasing the complexity of the model and adding more trees. Then, in each iteration, predictions are made on the training and test sets and accuracy, precision, recall, f1-score metrics are calculated.

```
    # Her 10 epoch'ta sonuçları yazdırma
    if epoch % print_interval == 0:
        print(f'Epoch [{epoch}/{epochs}], Train Accuracy: {train_accuracy_rf:.4f}, '
              f'Train Precision: {train_precision_rf:.4f}, '
              f'Train Recall: {train_recall_rf:.4f}, '
              f'Train F1 Score: {train_f1_rf:.4f}')
        print(f'Test Accuracy: {test_accuracy_rf:.4f}, Test Precision: {test_precision_rf:.4f}, '
              f'Test Recall: {test_recall_rf:.4f}, Test F1 Score: {test_f1_rf:.4f}')

# Eğitim sonuçları
print("Final Train Results:")
print(f'Train Accuracy: {train_accuracy_list[-1]:.4f}, Train Precision: {train_precision_list[-1]:.4f}, '
      f'Train Recall: {train_recall_list[-1]:.4f}, Train F1 Score: {train_f1_list[-1]:.4f}')

# Test sonuçları
print("Final Test Results:")
print(f'Test Accuracy: {test_accuracy_list[-1]:.4f}, Test Precision: {test_precision_list[-1]:.4f}, '
      f'Test Recall: {test_recall_list[-1]:.4f}, Test F1 Score: {test_f1_list[-1]:.4f}')
```

Figure 2.24. Print Results in Every 10 Epochs

The first part is in the for loop and prints performance metrics such as accuracy, precision, recall and F1-score obtained on the training and test sets every 10 epochs. This is used to observe changes in the model's performance as the training process continues.

The part interpreted as training and testing results prints the final performance metrics obtained when the training process is completed. These metrics show the final performance of the model on the training and test sets.

2.7. Recommendation System

2.7.1. Dataset

The dataset we used in our project is adapted version of U.S. Department of Agriculture nutrition dataset for Turkish meals. The U.S. Department of Agriculture (USDA) provides various nutrition datasets that offer information about the nutritional content of foods. One of the primary datasets is the USDA National Nutrient Database for Standard Reference (SR). This database includes information on the nutritional composition of a wide range of foods, including raw, cooked, and prepared items.The SR database provides data on macronutrients (such as carbohydrates, proteins, and fats), micronutrients (vitamins and minerals), and other components like fiber and cholesterol. The information is valuable for researchers, nutritionists, dietitians, and the general public who want to understand the nutritional content of different foods. The USDA National Nutrient Database for Standard Reference (SR) has several properties that make it a valuable resource for understanding the nutritional content of foods. Here are some key properties:

Comprehensive Coverage: The dataset covers a wide range of foods, including raw, cooked, and prepared items. It includes both single-ingredient foods and composite dishes.

Nutrient Composition: Provides detailed information on the nutritional composition of foods, including macronutrients (carbohydrates, proteins, and fats), micronutrients (vitamins and minerals), and other components such as fiber and cholesterol.

Standardized Units: The nutrient values are typically presented in standardized units (e.g., grams, milligrams, micrograms per 100 grams or per serving) to facilitate comparisons between different foods.

Reference Values: The dataset often includes reference values for standard serving sizes, which aids in comparing the nutritional content of similar portions of different foods.

Data Source Documentation: Provides information on the sources of data, including the methods used for analysis and the specific food products or samples analyzed.

Regular Updates: The USDA regularly updates the database to reflect changes in food composition and to include new foods that become available in the market.

Searchable Interface: Users can access the dataset through a searchable interface, allowing for easy retrieval of information on specific foods or nutrients.

Public Accessibility: The dataset is typically made publicly accessible, allowing researchers, nutritionists, policymakers, and the general public to use and benefit from the information.

These properties contribute to the usefulness and reliability of the USDA National Nutrient Database for Standard Reference in various applications, including nutrition research, dietary planning, and public health initiatives.

Unfortunately, we could not find a dataset containing the nutritional values of Turkish food, so we decided to produce our own dataset. Every single data we collect is from Turkish National food composition database. It includes three different class according to nutrient species.

1	Food_iten	Breakfast	Lunch	Dinner	VegNovV	Calories	Fats	Proteins	Iron	Calcium	Sodium	Potassiun	Carbohyd	Fibre	VitaminD	Sugars
2	Baklava	0	0	1	0	300	20	5	1	50	10	100	30	2	0	20
3	Kebab (Mi	0	0	1	0	400	25	20	2	30	20	150	15	3	0	2
4	Meze Plat	0	1	1	0	250	15	10	1	20	15	120	10	2	0	5
5	Hummus	0	1	1	1	150	10	5	1	15	5	70	10	4	0	1
6	Kofte	0	1	1	0	350	15	18	2	25	15	130	20	3	0	3
7	Pide	0	1	1	0	280	12	10	1	20	10	100	25	2	0	1
8	Manti	0	0	1	0	300	15	12	1	30	12	120	30	4	0	2
9	Menemer	1	0	1	1	200	15	8	1	20	10	80	15	2	0	5
10	Turkish De	0	0	1	0	200	0	2	0.5	10	5	80	50	1	0	40

Figure 2.25. A part of dataset

2.7.2. What is a Recommendation System?

A recommendation system is a type of software application or algorithm designed to suggest items or content to users based on their preferences, behavior, or characteristics. These systems are commonly used in various online platforms, such as e-commerce websites, streaming services, social media, and more. Recommendation systems aim to enhance user experience by providing personalized suggestions, thereby helping users discover new items, products, or content that align with their interests and preferences. They typically leverage user data, such as past behavior, ratings, and demographics, as well as item characteristics to generate relevant recommendations.

2.7.3. Use of Recommendation Systems in Daily Life

Recommendation systems contribute to a more personalized and efficient user experience by delivering content, products, and services that align with individual preferences and behaviors. There are dozens of different recommendation systems, let's list them as follows:

E-commerce Platforms: Recommendation systems are widely used on platforms like Amazon, eBay, and others to suggest products based on users' browsing history, purchase behavior, and preferences.

Streaming Services: Platforms such as Netflix, Spotify, and YouTube utilize recommendation systems to suggest movies, TV shows, music, or videos based on users' viewing or listening history, ratings, and preferences.

Social Media: Social networking sites like Facebook, Instagram, and Twitter employ recommendation systems to suggest friends, connections, and content based on users' interactions, interests, and engagement history.

Online News and Content Platforms: News websites, blogs, and content aggregators use recommendation systems to suggest articles, posts, or videos tailored to users' interests and reading habits.

Online Travel Platforms: Websites like Expedia, Booking.com, and Airbnb leverage recommendation systems to suggest hotels, flights, or vacation rentals based on users' past bookings, preferences, and search history.

Food Delivery Apps: Services like Uber Eats, DoorDash, and Grubhub utilize recommendation systems to suggest restaurants and food items based on users' past orders, reviews, and preferences.

Job Portals: Platforms like LinkedIn and Indeed use recommendation systems to suggest relevant job postings, connections, and professional content based on users' skills, experience, and career interests.

Health and Fitness Apps: Apps for fitness tracking or wellness often incorporate recommendation systems to suggest personalized workout routines, meal plans, or health tips based on users' goals and progress.

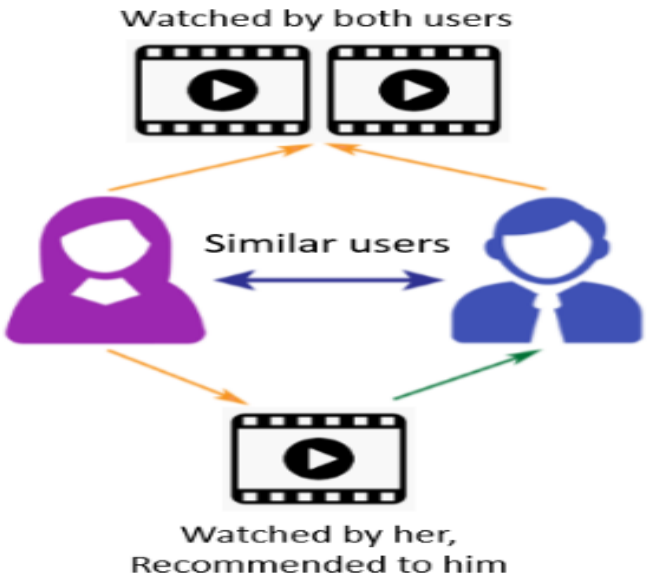


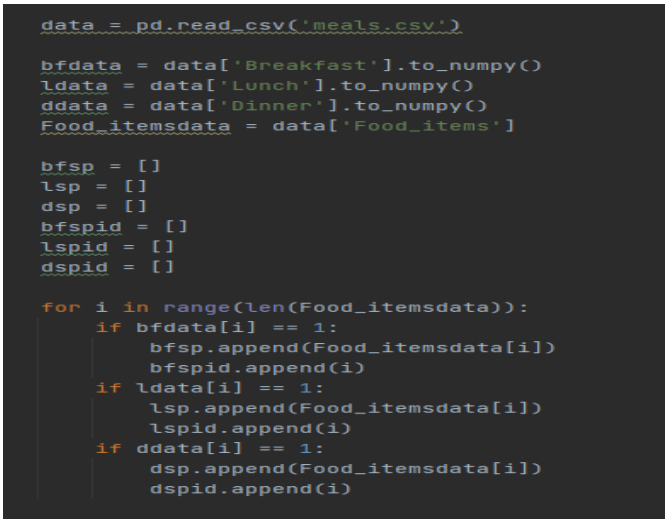
Figure 2.26. Sample movie recommendation system architecture

2.7.4. Data Preprocessing

Data preprocessing is a critical aspect of the data science workflow, encompassing a series of steps aimed at refining raw data to make it suitable for analysis and machine learning applications. In addition to cleaning and transforming data, feature engineering is often employed to create new informative features, enhancing a model's

ability to capture patterns. Handling imbalanced data ensures fair representation of classes, while techniques like dimensionality reduction streamline the dataset for more efficient processing. The choice of encoding categorical variables, dealing with duplicates, and appropriate data splitting into training and testing sets contribute to the overall robustness of the analysis or model. Effective data visualization during preprocessing not only aids in understanding the dataset's structure but also helps in identifying outliers or patterns that might influence subsequent analyses. Ultimately, meticulous data preprocessing is fundamental to obtaining accurate, reliable, and interpretable results in the realm of data science and machine learning.

Separating the items into 3 groups according to their breakfast, lunch and dinner features. The separation of food items for breakfast, lunch, and dinner based on their presence in the data serves the purpose of categorizing and analyzing the user's dietary choices during different meals throughout the day. This segregation allows the program to perform specific analyses and make personalized recommendations for each meal based on the user's input and dietary preferences.



```
data = pd.read_csv('meals.csv')

bfdata = data['Breakfast'].to_numpy()
ldata = data['Lunch'].to_numpy()
ddata = data['Dinner'].to_numpy()
Food_itemsdata = data['Food_items']

bfsp = []
lsp = []
dsp = []
bfspid = []
lspid = []
dspid = []

for i in range(len(Food_itemsdata)):
    if bfdata[i] == 1:
        bfsp.append(Food_itemsdata[i])
        bfspid.append(i)
    if ldata[i] == 1:
        lsp.append(Food_itemsdata[i])
        lspid.append(i)
    if ddata[i] == 1:
        dsp.append(Food_itemsdata[i])
        dspid.append(i)
```

Figure 2.27. Item list separation

Transpose data and select specific rows and columns. In the provided code, the iloc (integer-location based indexing) method is used to retrieve specific rows of meal data for breakfast, lunch, and dinner. The iloc method is a way to access rows and columns in a DataFrame by using their integer indices. using iloc for row selection depends on the specific structure of the DataFrame and the goals of the analysis.

```
bfspid_data = data.iloc[bfspid]
bfspid_data = bfspid_data.T
val = list(np.arange(5, 15))
Valapnd = [0] + val
bfspid_data = bfspid_data.iloc[Valapnd]
bfspid_data = bfspid_data.T
```

Figure 2.28. Having transpose of data

2.7.5. User Input

The diet list given in this project is based on the food preferences, age and body mass index of the users. A food recommendation system made through Body Mass Index (BMI) can utilize individual health and dietary information to provide personalized food suggestions.

BMI Calculation:The system should start by calculating the user's BMI based on their height and weight. BMI is a measure of body fat and is commonly used to categorize individuals into different weight status categories (underweight, normal weight, overweight, and obese).

Personalized Nutritional Goals:Set personalized nutritional goals based on the user's BMI category. For example, someone aiming to lose weight might have different nutritional requirements compared to someone trying to maintain their current weight.

Dietary Preferences and Restrictions:Allow users to input their dietary preferences, restrictions, and allergies. This could include vegetarian or vegan preferences, gluten intolerance, or specific food allergies.

Health Conditions:Consider any existing health conditions or medical issues that may influence dietary choices. For instance, individuals with diabetes may have specific dietary needs, and the system should take this into account.

Meal Planning:Provide personalized meal plans that align with the user's BMI, nutritional goals, and dietary preferences. This may involve suggesting the appropriate

balance of macronutrients (carbohydrates, proteins, and fats) and micronutrients (vitamins and minerals).

```
bmi = weight / ((height / 100) ** 2)

for lp in range(0, 80, 20):
    test_list = np.arange(lp, lp + 20)
    for i in test_list:
        if (i == age):
            agecl = round(lp / 20)

if (bmi < 16):
    clbmi = 4
elif (bmi >= 16 and bmi < 18.5):
    clbmi = 3
elif (bmi >= 18.5 and bmi < 25):
    clbmi = 2
elif (bmi >= 25 and bmi < 30):
    clbmi = 1
elif (bmi >= 30):
    clbmi = 0
```

Figure 2.27. BMI and Age Intervals

2.7.6. Clustering Method

In our project we use K-Means clustering method for classifying data according to their calories. K-Means clustering is a populer unsupervised machine learning algorithm used for partitioning a dataset into K distinct, non-overlapping subsets (clusters). The goal of K-means is to group similar data points into clusters, where similar is defined by proximity of data points to the mean (centroid) of the cluster.

```
Dinner_calorie = Dinner_ID[1:, 1:len(Dinner_ID)]
X = np.array(Dinner_calorie)
kmeans = KMeans(n_clusters=3, random_state=0, n_init=5).fit(X)
dinner_label = kmeans.labels_
```

Figure 2.28. K-Means Clustering for Dinner Calorie

2.7.7. Data and Label Regularization

This code block is preparing labeled data for three different categories, each with five repetitions. The data is being stored in separate arrays (wl_nut(weight loss nutritions), wg_nut(weight gain nutritions), and h_nut(healthy_nutritions)), and the labels are

appended to each row of data. The labels come from the clbmi(Range of users' BMI) variable.

```
for i in range(5):
    for j in range(len(wl)):
        valloc = list(wl[j])
        valloc.append(bmicls[i])
        valloc.append(agecls[i])
        wl_nut[t] = np.array(valloc)
        if j < len(brklbl):
            yt.append(brklbl[j])
        t += 1
    for j in range(len(wg)):
        valloc = list(wg[j])
        valloc.append(bmicls[i])
        valloc.append(agecls[i])
        wg_nut[r] = np.array(valloc)
        if j < len(lnchlbl):
            yr.append(lnchlbl[j])
        r += 1
    for j in range(len(h)):
        valloc = list(h[j])
        valloc.append(bmicls[i])
        valloc.append(agecls[i])
        h_nut[s] = np.array(valloc)
        if j < len(dnrlbl):
            ys.append(dnrlbl[j])
        s += 1
```

Figure 2.29. Regularization

2.7.8. Classification Model

According to the user's preference, a diet recommendation is made using RandomForestClassifier. Random Forest Classifier is an ensemble learning algorithm commonly used for both classification and regression tasks. Random Forest Classifier is particularly effective for creating robust and versatile models in the field of diet recommendations. Its ability to aggregate predictions from multiple decision trees not only enhances accuracy but also mitigates the risk of individual trees making overly specific or biased predictions. This ensemble learning approach allows the model to generalize well to diverse user preferences and dietary needs, ensuring a more personalized and reliable diet recommendation. Additionally, the inherent feature importance analysis provided by Random

Forest can offer valuable insights into the key factors influencing dietary choices, aiding both users and healthcare professionals in understanding the rationale behind the suggested recommendations. It operates by constructing a multitude of decision trees during training and outputs the mode (classification) or mean (regression) prediction of the individual trees for a given input. A random forest is nothing more

than a series of decision trees with their findings combined into a single final result. They are so powerful because of their capability to reduce overfitting without massively increasing error due to bias.

```
clf = RandomForestClassifier(n_estimators=100)

X_test = np.zeros((len(h) * 5, 9), dtype=np.float32)

for i in range(len(h)):
    valloc = list(h[i])
    valloc.append(agecl)
    valloc.append(clbmi)
    X_test[i] = np.array(valloc) * ti

X_train = h_nut[:len(ys)]
y_train = ys

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Figure 2.30. Random Forest Classifier Model

3. CONCLUSION

We trained our Neural Network structure to comply with a standard. These standards were first divided into 20% test and 80% train. Each was trained with the same hyperparameters. For example, the same values for learning rate, epochs, batch size and others were used. TF-IDF, TF, Bag of Words were used during vector creation and the values for each were calculated separately.

The naming NeuralNetworkwithTFIDF means the neural network trained using the TF-IDF function.

The naming NeuralNetworkwithTF means the neural network trained using the TF function.

The naming NeuralNetworkwithBoW means the neural network trained using the BoW function.

In the RandomForestClassifier section, NeuralNetwork was not used and RandomForestClassifier was used instead. During this process, two different samples were created. The first one produced very bad results due to the small amount of data, and in our second model, the results were brought to a good point by using different techniques such as data multiplication and cross validation, and the results of the best model were shared.

Model Name (Train Set)	Accuracy	Precision	Recall	F1
NeuralNetworkwithTFIDF	1.0	0.92	0.91	1.0
NeuralNetworkwithTF	1.0	0.94	0.93	1.0
NeuralNetworkwithBoW	0.60	1.0	0.60	0.66
RandomForestClassifier	0.90	0.97	0.90	0.90

Figure 3.1. Our Models Train Set Results

When the Train results are evaluated, the results of the neural network trained using TF-IDF and TF seem to be the best and compete with each other. RandomForestClassifier can be interpreted as a model that follows these values and gives good results. However, unfortunately, the results of the model trained using BoW were quite low compared to the others. The biggest reason behind the models giving such results is that the dataset is very small. Developing the intent file and developing the Contextual Chatbot structure is of great importance. In addition, weights can be given to metrics during training, and changing this weights value while training may have affected the results completely differently. Looking at these results, it seems that TF-IDF, TF, RandomForestClassifier can be used.

Model Name (Test Set)	Accuracy	Precision	Recall	F1
NeuralNetworkwithTFIDF	0.83	0.80	0.70	0.73
NeuralNetworkwithTF	0.22	0.46	0.40	0.42
NeuralNetworkwithBoW	0.77	0.70	0.60	0.64
RandomForestClassifier	0.6	1.0	0.6	0.66

Figure 3.2. Our Models Test Set Results

When we evaluate the test results, it can be said that the TF-IDF, BoW and RandomForestClassifier results are usable, but the TF results seem extremely bad and completely inefficient. Looking at the results, it can easily be said that some models are prone to overlearning or that there are inconsistencies in the data. The most important factors here are that there is a limited number of data and the model is very prone to memorizing them, so unfortunately the results cannot be obtained in the way we want. However, if we are to

comment on the train and test results we have, it can be commented that the TF-IDF method gives non-negligible results for both datasets and is useful.

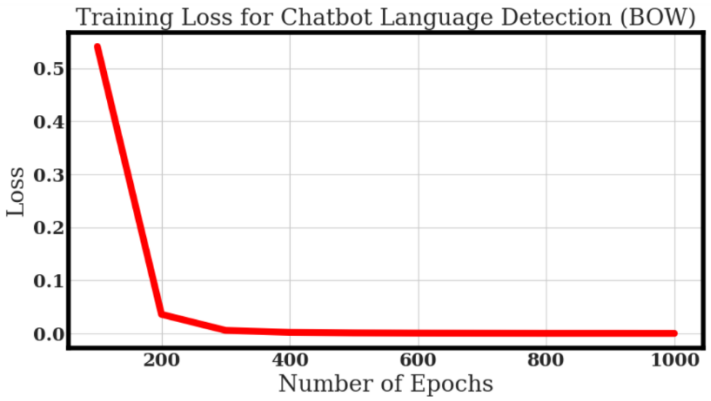


Figure 3.3. BoW Model Training Loss-Epoch Graph

When we look at the Loss-Epochs chart of the BoW model, we can see that there is a rapid decrease until the first 200 epochs, then there is a normal decrease until 400, and then the Loss value stops and reaches an acceptable limit. Here, running 1000 epochs may be unnecessary training and may force the model to overfit, so training this model around 300-500 epochs would be the most logical decision. Apart from this, there does not appear to be any problem in training the model and there are no bifurcations.

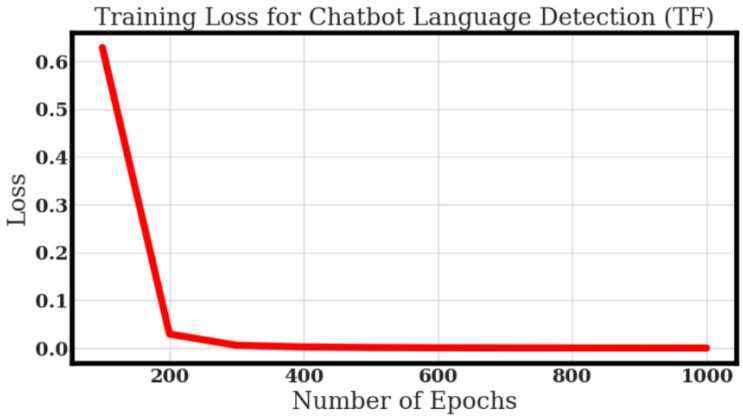


Figure 3.4. TF Model Training Loss-Epoch Graph

When analyzing the Loss-Epochs chart of the TensorFlow model, we observe a sharp decrease in the Loss value until the first 200 epochs. After that, the decrease becomes normal until the range of 300-400 epochs. Beyond this point, the Loss value stops decreasing and reaches an acceptable limit. Therefore, training the model for 1000 epochs may be unnecessary and may lead to overfitting. It is more logical to train the model for around 300-

500 epochs. Additionally, there do not seem to be any problems with training the model, and there are no bifurcations.

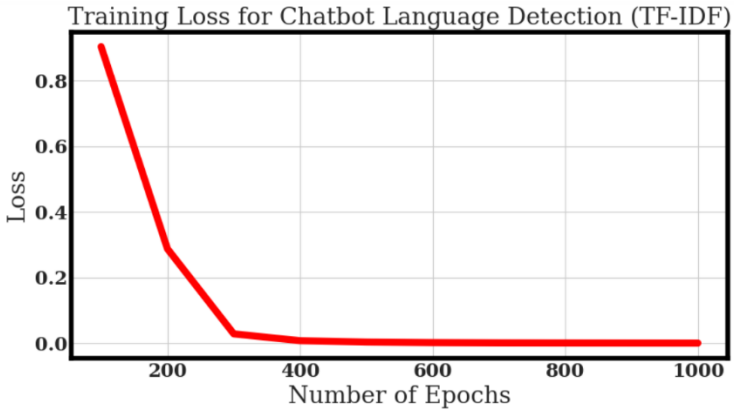


Figure 3.5. TF-IDF Model Training Loss-Epoch Graph

Looking at the Loss-Epochs graph of the TF-IDF model, we can see that the loss value sharply decreases until the first 200 epochs. After that, the slope of the decrease slows down in the 200-300 range, but the loss value continues to decrease slowly until approximately 450-500 epochs. The decrease then stops, and the loss value reaches an acceptable limit. Running 1000 epochs would be unnecessary and may cause the model to overfit, so training the model for around 500 epochs would be the most logical decision. Additionally, there are no problems or bifurcations in training the model.

With our DietBot project, we managed to execute all the concepts we had in mind from start to finish. So far, most projects of this kind have relied on classical machine learning techniques, such as the collaborative filtering method, using well-designed datasets, while there have been very few examples of ChatBots that people can interact with. Looking at these applications developed by researchers, it became clear that they had several shortcomings, and mobile applications built in this structure also had significant issues. We aimed to create a Contextual Chatbot using ChatGPT and similar technologies to help people familiarize themselves with Chatbots. This project was quite niche, and there were barely any models trained with neural networks, besides the Charlie Bot project. Recognizing this gap, we decided to diversify the NLP techniques applied to the project as much as possible, and we achieved different results by implementing them. We also trained an additional machine learning-based model to understand how much of a difference we could make with ensemble model types such as RandomForestClassifier. Once we understood the user's problem and request, we provided answers based on the input while running a recommendation system in

the background. The data used in the system is as important as the design, and developers can opt for datasets containing geography-specific foods depending on the target audience of the chatbot. At the end of the day, we managed to create a high-quality Chatbot that can respond to the user's problem using various techniques. Our neural network structures seem to provide better results than classical machine learning techniques, and we believe they can compete with the research in this field.

When we evaluate previous examples, Gao and his colleagues presented a Bayesian personalized diet recommendation system, Butti Gouthami and Malige Gangappa (2020) developed a system that recommends users' favorite meals using the USDA nutrition data set, Thi Ngoc Trang Tran et al (2021) published a research on healthcare recommendation systems, and many similar studies are also found in the literature. However, when we look at our own work, we can say that we have dealt with a job that is not very common in today's world and that can overcome the prejudices in this field, and we believe that the results of this are satisfactory.

Bibliography

W. Zhou and X. Peng, "Linear Programming Method and Diet Problem," *2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*, Changchun, China, 2022, pp. 761-764, doi: 10.1109/ICETCI55101.2022.9832221

Butti Gouthami and Malige Gangappa, “NUTRITION DIET RECOMMENDATION SYSTEM USING USER’S INTEREST”, International Journal of Advanced Research in Engineering and Technology (IJARET), vol. 11, 2020, pp. 2910-2919, doi: 10.34218/IJARET.11.12.2020.272

C. A. Iheanacho and O. R. Vincent, "Classification and recommendation of food intake in West Africa for healthy diet using Deep Learning," *2022 5th Information Technology for Education and Development (ITED)*, Abuja, Nigeria, 2022, pp. 1-6, doi: 10.1109/ITED56637.2022.10051387.

D. Chowdhury, A. Roy, S. R. Ramamurthy and N. Roy, "CHARLIE: A Chatbot That Recommends Daily Fitness and Diet Plans," *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom*

Workshops), Atlanta, GA, USA, 2023, pp. 116-121, doi: 10.1109/PerComWorkshops56833.2023.10150359.

R. Sajith, C. Khatua, D. Kalita and K. B. Mirza, "Nutrient Estimation from Images of Food for Diet Management in Diabetic Patients," *2023 World Conference on Communication & Computing (WCONF)*, RAIPUR, India, 2023, pp. 1-6, doi: 10.1109/WCONF58270.2023.10235177.

Thi Ngoc Trang Tran, Alexander Felfernig, Christoph Trattner and Andreas Holzinger, "Recommender systems in the healthcare domain: state-of-the-art and research issues", Journal of Intelligent Information Systems, vol. 57, 2020, pp. 171–201, doi: 10.1007/s10844-020- 00633-6.

A. S. Fauziah, F. Renaldi and I. Santikarama, "Combining Medical Records, Daily Diets, and Lifestyle to Detect Early Stage of Diabetes using Data Mining Random Forest Method," *2022 IEEE Creative Communication and Innovative Technology (ICCIT)*, Tangerang, Indonesia, 2022, pp. 1-5, doi: 10.1109/ICCIT55355.2022.10118711.

Jun Gao, Ninghao Liu, Mark Lawley, and Xia Hu, "An Interpretable Classification Framework for Information Extraction from Online Healthcare Forums", Journal of Healthcare Engineering, Cui Tao, vol. 2017, Article ID 2460174

A. Muenzberg, J. Sauer, A. Hein and N. Roesch, "Checking the Plausibility of Nutrient Data in Food Datasets Using KNIME and Big Data," *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, Barcelona, Spain, 2019, pp. 1-4, doi: 10.1109/WiMOB.2019.8923233.

Q. Miao, R. Fang and Y. Meng, "Extracting and integrating nutrition related linked data," *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, Anaheim, CA, USA, 2015, pp. 365-372, doi: 10.1109/ICOSC.2015.7050835.

J. Kalra, D. Batra, N. Diwan and G. Bagler, "Nutritional Profile Estimation in Cooking Recipes," *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, Dallas, TX, USA, 2020, pp. 82-87, doi: 10.1109/ICDEW49219.2020.000-3.

M. -L. Chiang, C. -A. Wu, J. -K. Feng, C. -Y. Fang and S. -W. Chen, "Food Calorie and Nutrition Analysis System based on Mask R-CNN," *2019 IEEE 5th International Conference*

on Computer and Communications (ICCC), Chengdu, China, 2019, pp. 1721-1728, doi: 10.1109/ICCC47050.2019.9064257.

İ. Berkan Aydilek, "Approximate estimation of the nutritions of consumed food by deep learning," *2017 International Conference on Computer Science and Engineering (UBMK)*, Antalya, Turkey, 2017, pp. 160-164, doi: 10.1109/UBMK.2017.8093588.

S. Sadhasivam, M. S. Sarvesvaran, P. Prasanth and L. Latha, "Diet and Workout Recommendation Using ML," *2023 2nd International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, Coimbatore, India, 2023, pp. 1-4, doi: 10.1109/ICAECA56562.2023.10199540.

S. Santhosham and C. P. Sah, "Advanced Healthcare Chat Bot using Python," *2023 2nd International Conference for Innovation in Technology (INOCON)*, Bangalore, India, 2023, pp. 1-4, doi: 10.1109/INOCON57975.2023.10101239.

V. Singh, Y. Rohith, B. Prakash and U. Kumari, "ChatBot using Python Flask," *2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, 2023, pp. 1182-1185, doi: 10.1109/ICICCS56967.2023.10142484.

B. Kohli, T. Choudhury, S. Sharma and P. Kumar, "A Platform for Human-Chatbot Interaction Using Python," *2018 Second International Conference on Green Computing and Internet of Things (ICGCIoT)*, Bangalore, India, 2018, pp. 439-444, doi: 10.1109/ICGCIoT.2018.8753031.

T. T. Tran, J. W. Choi, C. Van Dang, G. SuPark, J. Y. Baek and J. W. Kim, "Recommender System with Artificial Intelligence for Fitness Assistance System," *2018 15th International Conference on Ubiquitous Robots (UR)*, Honolulu, HI, USA, 2018, pp. 489-492, doi: 10.1109/URAI.2018.8441895.

M. Donciu, M. Ionita, M. Dascalu and S. Trausan-Matu, "The Runner -- Recommender System of Workout and Nutrition for Runners," *2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, 2011, pp. 230-238, doi: 10.1109/SYNASC.2011.18.

S. Ono, Y. Yotsuya, N. Takahashi, T. Sakamoto and T. Kato, "Physical-exercise recommendation system based on individual daily schedule," *2022 Joint 12th International Conference on Soft Computing and Intelligent Systems and 23rd International Symposium on*

Advanced Intelligent Systems (SCIS&ISIS), Ise, Japan, 2022, pp. 1-3, doi: 10.1109/SCISISIS55246.2022.10001861.