Universidade do Minho

# Inteligência Ambiente: Tecnologias e Aplicações

MIEI - 4º ANO - 1º SEMESTRE
UNIVERSIDADE DO MINHO

## TRABALHO PRÁTICO DE GRUPO PARTE 2


Bernardo Viseu
A74618

Pedro Medeiros
A80580

Marco Gonçalves
A75480

Helena Martins
A82500


Braga, 10 de janeiro de 2021

# Conteúdo

# 1. Introdução

No âmbito da Unidade Curricular de Inteligência Ambiente: Tecnologias e Aplicações, foi proposta a realização de um segundo trabalho, como forma de desenvolver conhecimentos e aplicar os mesmos acerca do Processamento de Linguagem Natural no contexto de sistemas inteligentes capazes de reconhecer e interpretar interações textuais, uma vez que a inteligência artificial usa o processamento de linguagem natural para entender a linguagem humana e simulá-la.

# 2. Desenvolvimento

## 2.1 Servidor

O primeiro passo na realização deste projecto foi o desenvolvimento de um servidor que receba as mensagens dos *Users* que estão em conversa.

Para isto, decidimos seguir a abordagem comum de utilização de uma *socket*. Fizemos então em *Python* um servidor simples denominado `chat_socket.py` na porta 7000 (no `localhost`).

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# localhost/7000
IP_address = '127.0.0.1'
Port = 7000
server.bind((IP_address, Port))

"""
Para ouvir 50 clientes maximo
"""
server.listen(50)
list_of_clients = []

#(mais codigo...}

while True:
    # Aceita nova connection, conn = socket do user,
    # addr = ip address do user
    conn, addr = server.accept()

    # Lista de clients
    list_of_clients.append(conn)

    # Print address de novo user
    print(addr[0] + " connected")

    # Cria thread para novo user
    start_new_thread(clientthread, (conn, addr))
```

**Listing 2.1:** Inicialização da Socket

Utilizamos uma lista `list_of_clients` em que está guardada todas as conexões actuais. Decidimos ter a possibilidade de estarem até 50 clientes ligados ao servidor, de maneira que várias pessoas podem estar a conversar neste *chat-room*.

A lista de *clients* é utilizada para facilitar a remoção de clientes que tenham abandonado o servidor, utilizando a função `remove(connection)`:

```
def remove(connection):
    if connection in list_of_clients:
        list_of_clients.remove(connection)
```

**Listing 2.2:** Função de remoção de uma conexão

### 2.1.1 Envio de Mensagens para os Clientes

No nosso servidor, cada *client* que se conecte vai inicializar uma *thread*.

Para isso utilizamos a função `start_new_thread(clientthread,(conn, addr))`. Esta thread vai inicializar a função `clientthread`:

```
def clientthread(conn, addr):
    name = conn.recv(2048).decode()
    # Envia welcome message
    welcomemsg = 'Welcome, your random name is ' + name + "!"
    conn.send(welcomemsg.encode())

    while True:
        try:
            message = conn.recv(2048)
            if message:
                decodedmsg = message.decode()

                #Print da mensagem com o nome do client
                print("<" + name + "> " + decodedmsg)

                # Broadcast da mensagem
                message_to_send = "< " + name + " > " + decodedmsg
                broadcast(message_to_send.encode(), conn)
            else:
                """
                Caso a msg esteja vazia, remove a connection
                """
                remove(conn)
        except:
            continue
```

**Listing 2.3:** Função clientthread

Esta função inicialmente vai receber um nome aleatoriamente escolhido para o utilizador, que vai utilizar como prefixo em todas as mensagens que esse utilizador enviar. Depois disto entra num *loop* em que vai esperar por mensagens por esse *client*.

Cada mensagem recebida aqui é enviada para todos os outros utilizadores através de uma função `broadcast` que recebe a mensagem e a *connection* do cliente, e envia esta mensagem para todas as outras *connections*, de maneira a que todos os outros utilizadores a recebam. Para isto basta percorrer a `list_of_clients` e enviar a *string* para todas as conexões que não sejam a que foi recebida pela função.

Esta função também identifica se algum *client* não está a receber as mensagens, e nesse caso remove o da lista de *clients*.

```
def broadcast(message, connection):
    for clients in list_of_clients:
        if clients != connection:
            try:
                clients.send(message)
            except:
                clients.close()

                # remove client se estiver erro
                remove(clients)
```

**Listing 2.4:** Função broadcast

## 2.2 Cliente

Feito o código para o servidor, desenvolvemos também em *Python* um `client.py`, onde se encontra todas as funções relevantes para a interacção com o servidor.

Primeiramente, criamos de maneira semelhante a que foi feita no servidor, a socket que faz ligação com o servidor:

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

IP_address = '127.0.0.1'
Port = 7000
server.connect((IP_address, Port))
```

**Listing 2.5:** Inicialização de socket cliente

A primeira mensagem que o cliente manda para o servidor é feita automaticamente, e representa um nome aleatório que foi escolhido para essa utilizador. Para isto utilizamos uma escolha aleatória de uma linha de um ficheiro de texto que contém vários nomes de animais (`randomanimals.txt`):

```
# get random animal name
lines = open("randomanimals.txt").readlines()
randomname = random.choice(lines)

server.send(randomname[:-1].encode())
```

**Listing 2.6:** Método de obter nome aleatório para um user

Depois disto, entra-se num *loop* para que sejam enviadas e recebidas as várias mensagens trocadas entre o cliente e o servidor. Isto é feito a partir de uma lista de *input streams*, que, utilizando a *library* `Select` de *Python*, permite um controlo de quando deve ser lido o que o servidor enviou para o cliente.

## 2.3 Utilização de AIML

No cliente existe também código para lidar com um dos objectivos deste projecto, e isso é a sugestão de respostas que o cliente pode enviar, dependendo do que os outros utilizadores dizem.

Para isto, e seguindo o que nos foi aconselhado nas aulas, utilizamos *AIML* (*Artificial Intelligence Markup Language*) de maneira a ter um *ChatBot* a ler as mensagens enviadas, e assim damos como sugestão aos utilizadores o que o *ChatBot* responderia a essas mensagens.

Seguindo os passos desta fonte ( `https://github.com/paulovn/python-aiml` ), conseguimos inicializar um *ChatBot* funcional:

```
BRAIN_FILE = "brain.brn"

kernel = aiml.Kernel()

if os.path.exists(BRAIN_FILE):
    print("Loading from brain file: " + BRAIN_FILE)
    kernel.loadBrain(BRAIN_FILE)
else:
    print("Parsing aiml files")
    kernel.bootstrap(learnFiles="aiml/learningFileList.aiml",
                     commands="load aiml")
    print("Saving brain file: " + BRAIN_FILE)
    kernel.saveBrain(BRAIN_FILE)
```

**Listing 2.7:** Inicialização de ChatBot para sugestão de respostas

Neste excerto de código inicializamos o *kernel* que vai criar um *Brain File* que controla as respostas para certos *patterns* indicados nos ficheiros *AIML*.

Para obtermos a resposta para as mensagens recebidas, temos apenas de pedir uma *response* ao *kernel*, para isto temos de fazer um *cleanup* da mensagem recebida, de maneira a retirar o nome do utilizador que a enviou (que vem entre "<" e ">"), e utilizamos também a *library* `AutoCorrect` para corrigir os erros gramáticos das mensagens, de maneira a o kernel as poder 'entender' correctamente.

Isto é feito de cada vez que o cliente recebe uma mensagem do servidor:

```
message = socks.recv(2048)

if not message.decode():
    print('Servidor abaixo, vou sair...')
    exit()
decodedmsg = message.decode()
print(decodedmsg)

# retira o que esta entre "<>"
msg = re.sub('<[^>]+>', '', decodedmsg)
msg = [spell(w) for w in (msg.split())]
question = " ".join(msg)
response = kernel.respond(question)
suggestion = '[Suggestion: ' + response + ' ]'
print(suggestion)
```

**Listing 2.8:** Método para receber respostas do kernel

### 2.3.1 A.L.I.C.E. AIML Set

Para obtermos boas sugestões para vários tipos de mensagens, utilizamos um conjunto de ficheiros *AIML* distribuidos gratuitamente ( aqui ), que são a base de um famoso *ChatBot* denominado *A.L.I.C.E.* (*Artificial Linguistic Internet Computer Entity*). Com estes ficheiros *AIML* conseguimos obter sugestões de respostas de muito boa qualidade, para vários temas de conversa.

É de constatar que a utilização de este *set* de ficheiros *AIML* limita a utilização deste projecto para conversas em inglês, dado que noutras línguas, apesar de o sistema de conversa funcionar normalmente, o sistema de sugestões de respostas não irá funcionar.

## 2.4 Exemplo de Utilização

Aqui demonstramos um exemplo do produto final em utilização, em vários terminais a simular vários utilizadores diferentes:


**Figura 2.1:** Servidor a correr.

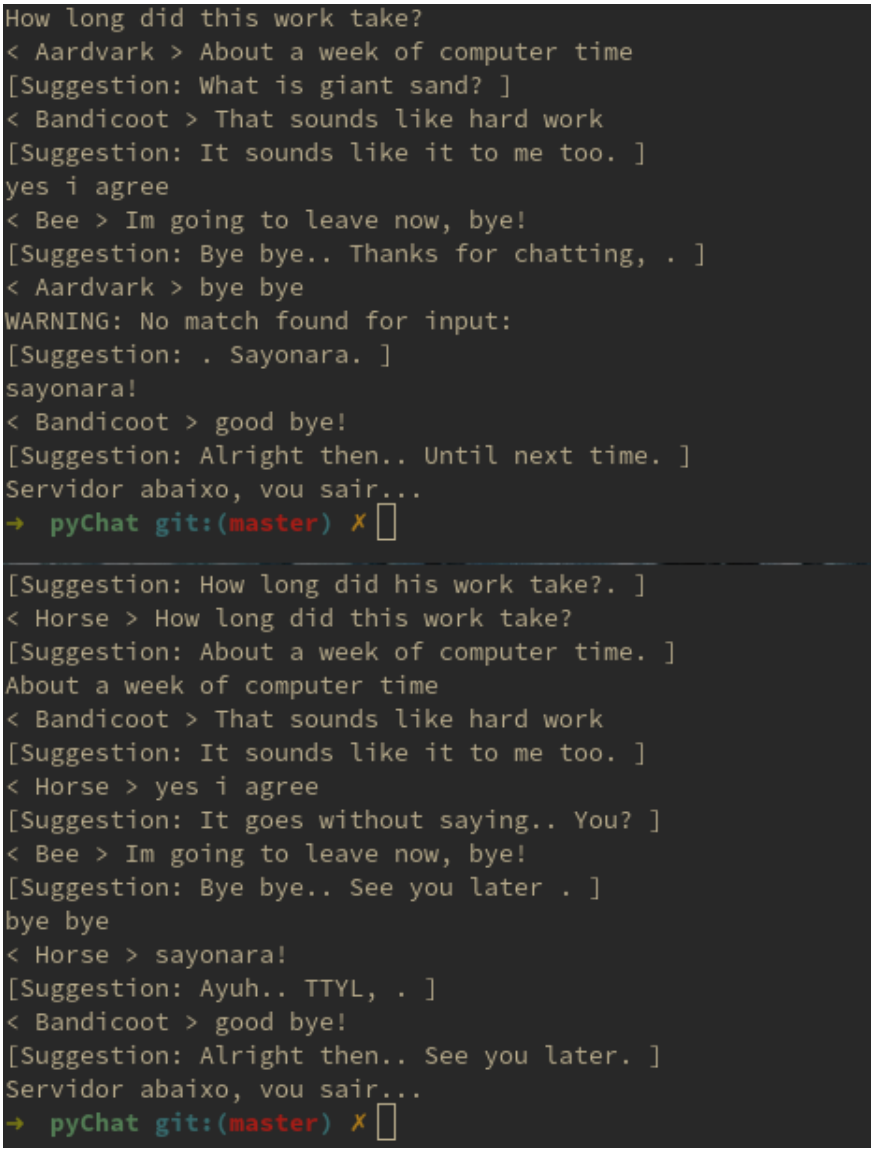


**Figura 2.2:** Utilizadores a enviarem mensagens.

# 3. Conclusão

Através do sistema que foi desenvolvido fomos capazes de implementar um sistema inteligente com suporte de Processamento de Linguagem Natural, ou seja, através de um conjunto de respostas automáticas, quando um utilizador pretender responder a uma dada mensagem numa conversa, aparecerá uma resposta sugerida que o utilizar pode usar para enviar na resposta.

Achamos que existe ainda funcionalidades com mais potencial, como melhor sugestão de respostas para línguas outras que inglês, e uma avaliação das emoções dos utilizadores, que se mostrou ser a parte que nos deu mais dificuldades e por isso não a conseguimos implementar de maneira satisfatória.

Este trabalho foi muito esclarecedor e ajudou muito no entendimento de muitos conceitos de programação que até agora nos eram desconhecidos (principalmente na forma como *ChatBots* funcionam), o que fez com que este trabalho fosse muito didáctico.

Com isto conseguimos então dar por terminado este trabalho que nos fez ter um melhor entendimento sobre *Processamento de Linguagem Natural*, no contexto de ambientes inteligentes capazes de reconhecer e interpretar interacções textuais.