NER ON MOVIE TICKETS

I tried to perform NER on movie tickets using spacy (python library). I found that regular expressions could be a better option to recognize entities like date, time, price, seat no, ticket no. So I used regular expressions to recognize these entities in the input text. However entities like movie name and venue could not be recognized with regular expressions. I used lookup for the movie name. I also tried to get a trained model for the movie name recognition using spacy's entity ruler, but it was not very helpful as spacy uses context to recognize entities but text extracted from the movie tickets are just random strings which was insufficient to derive any context out of it. May be some imdb api could be used for movie name recognition , but I have not tried it here. Similarly for venue , training could not help much. For venue, my code relies completely on spacy's pretrained English model "en_core_web_sm". But this may be insufficient. May be for better accuracy we could use some google map API.

So my code works fine for date, time , seat no , price and ticket no but struggles to recognize movie name and venue.

Also , the format of templates of tickets booked at different websites like "BOOKMYSHOW" and "PVR" , could be used to recognize different entities.

For this project I followed spacy's documentation and also followed a youtube channel
https://youtu.be/E9h8qVm2uNY

for the same . I tried to deliver my best. Hope the information I gathered so far helps.

Here's breakdown of the code :

1) Importing the required libraries:

```
import spacy
from spacy.util import filter_spans
from spacy.tokens import Span
from spacy.language import Language
import re
import json
```

The code begins by importing necessary libraries :

- **Spacy** : This library is the core of NLP processing and provides the functionality for tokenization, parts of speech tagging and named entity recognition.
- **re** : This library is used for working with regular expressions, which are used to find patterns in the text.
- **Json** : This library is used for working with JSON data, specifically for loading the movie names from a JSON file.

2) Loading Spacy model :

```
nlp = spacy.load("en_core_web_sm")
```

The code loads the English language model provided by spaCy using the spacy.load() function. In this case, the "en_core_web_sm" model is used, which is a small English pipeline trained on web text. It provides basic NLP capabilities such as tokenization, part-of-speech tagging, and named entity recognition.

3) Defining Utility functions :

```
def filter_spans(spans):
    sorted_spans = sorted(spans, key=lambda span: (span.start, -span.end))
    result = []
    prev_start = -1
    prev_end = -1
    for span in sorted_spans:
        if span.start >= prev_start and span.end <= prev_end:
            continue  # Skip overlapping span
        result.append(span)
        prev_start = span.start
        prev_end = span.end
    return result
```

The code includes a utility function filter_spans() that helps filter and remove overlapping spans during the NER process. This function takes a list of spans and applies a filtering mechanism to keep only the non-overlapping spans. It ensures that the final list of entities contains the most relevant and non-overlapping entities.

4) Custom Components for NER :

i) For Seat No. :

```
def generate_seat_patterns(rows, seats):
    patterns = []
    for row in range(1, rows + 1):
        for seat in range(1, seats + 1):
            patterns.append(f"Row {row}, Seat {seat}")
            patterns.append(f"{chr(64 + row)}{seat}")
            patterns.append(f"{chr(64 + row)}-{seat}")
            patterns.append(f"{chr(64 + row)}{chr(96 + seat)}")
            patterns.append(f"{chr(96 + seat)}{chr(64 + row)}")
            patterns.append(f"{chr(96 + seat)}-{chr(64 + row)}")
            patterns.append(f"{row}{seat}")
            patterns.append(f"{row}-{seat}")
            patterns.append(f"{row}{chr(96 + seat)}")
            patterns.append(f"{chr(96 + seat)}{row}")
            patterns.append(f"{chr(64 + row)}{seat + 1}")
    return patterns
```

Here I tried to generate all possible patterns in which a seat no in a cinema hall could be represented. Below are the illustrations giving idea about all seat patterns generated by above code :

"Row 1, Seat 1" - Pattern: "Row {row}, Seat {seat}"
"A1" - Pattern: "{chr(64 + row)}{seat}"
"A-1" - Pattern: "{chr(64 + row)}-{seat}"
"Aa" - Pattern: "{chr(64 + row)}{chr(96 + seat)}"
"aA" - Pattern: "{chr(96 + seat)}{chr(64 + row)}"
"a-A" - Pattern: "{chr(96 + seat)}-{chr(64 + row)}"
"11" - Pattern: "{row}{seat}"
"1-1" - Pattern: "{row}-{seat}"
"1a" - Pattern: "{row}{chr(96 + seat)}"
"a1" - Pattern: "{chr(96 + seat)}{row}"
"A2" - Pattern: "{chr(64 + row)}{seat + 1}"

```python
@Language.component("find_seat_no")
def find_seat_no(doc):
    patterns = generate_seat_patterns(rows=10, seats=10)  # Update with the appropriate number of rows and seats
    seat_ents = []
    original_ents = list(doc.ents)

    seat_patterns = generate_seat_patterns(rows=10, seats=10)
    seat_regex = r"\b" + r"\b|\b".join(map(re.escape, seat_patterns)) + r"\b"

    seat_keywords = ["SEAT", "SEAT NO.", "SEAT:", "SEAT NO."]
    seat_keyword_regex = rf"(?:{'|'.join(seat_keywords)})[\s:]*([^\s:]+)"

    for match in re.finditer(seat_keyword_regex, doc.text, flags=re.IGNORECASE):
        start, end = match.span(1)
        span = doc.char_span(start, end)
        if span is not None:
            seat_ents.append((span.start, span.end, span.text))

    if not seat_ents:
        for match in re.finditer(seat_regex, doc.text):
            start, end = match.span()
            span = doc.char_span(start, end)
            if span is not None:
                seat_ents.append((span.start, span.end, span.text))

    pattern = r"\b([A-Z]\d{2})\b"
    for match in re.finditer(pattern, doc.text):
        start, end = match.span(1)
        span = doc.char_span(start, end)
        if span is not None:
            seat_ents.append((span.start, span.end, span.text))

    for ent in seat_ents:
        start, end, name = ent
        per_ent = Span(doc, start, end, label="SEAT_NO")
        original_ents.append(per_ent)

    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

The @Language.component("find_seat_no") decorator registers the following function find_seat_no as a custom pipeline component in spaCy. This component will be executed during the pipeline processing of a document.

The seat_regex variable is constructed using the seat_patterns to create a regular expression pattern. This pattern will be used to search for seat patterns in the document text.

The seat_keywords variable contains a list of common seat-related keywords like "SEAT", "SEAT NO.", "SEAT:", "SEAT NO.". These keywords are used to search for seat numbers in the document text.

A regular expression pattern is constructed using the seat_keywords to match the seat-related keywords followed by the seat number. The pattern is stored in the seat_keyword_regex variable.

The seat_ents list is initialized to store the found seat entities.

The document's original entities (found by spaCy's named entity recognition) are stored in the original_ents list.

The code iterates over the matches found by the re.finditer function using the seat_keyword_regex pattern. It retrieves the start and end positions of the seat number and creates a spaCy Span object for each match. If a span is successfully created, it is appended to the seat_ents list.

If no seat entities are found using the seat-related keywords, the code proceeds to search for seat patterns using the seat_regex pattern. It follows the same process as above to create spans and add them to seat_ents.

The code has a special case for a pattern like "A12". It uses a regex pattern to search for a letter followed by two digits. The matching spans are created and added to seat_ents.

After finding all the seat entities, they are converted into spaCy Span objects and appended to the original_ents list.

The filter_spans(original_ents) function is called to remove any overlapping spans in the original_ents list.

Finally, the doc.ents property is updated with the filtered spans.

The processed document (doc) is returned.

ii) For Price :

```
price_patterns = [
    r"(?<!\S)(rs\s?\d+(\.\d{2})?)",
    r"(?<!\S)(\d+(\.\d{2})?\s?rs)",
    r"(?<!\S)(\d+(\.\d{2})?\s?usd)",
    r"(?<!\S)(usd\s?\d+(\.\d{2})?)",
    r"(?<!\S)(\d+(\.\d{2})?\s?usd)",
    r"(?<!\S)(rs\.\s?\d+(\.\d{2})?)",
    r"(?<!\S)(\d+(\.\d{2})?\s?rs\.)",
]
```

First, a list of regular expression patterns called price_patterns is defined. These patterns are used to identify prices in the input text. The patterns use various combinations of digits, currency symbols (such as "rs" or "usd"), and decimal points to match prices formatted in different ways.

```python
@spacy.Language.component("find_price")
def find_price(doc):
    text = doc.text.lower()  # Lowercase the input text
    price_ents = []
    original_ents = list(doc.ents)
    for pattern in price_patterns:
        for match in re.finditer(pattern, text):
            start, end = match.span()
            span = doc.char_span(start, end)
            if span is not None:
                price_ents.append((span.start, span.end, span.text))
    for ent in price_ents:
        start, end, name = ent
        price_ent = Span(doc, start, end, label="PRICE")
        original_ents.append(price_ent)
    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

The find_price function is defined and decorated with @spacy.Language.component("find_price"). This makes the function a custom spaCy component that can be added to a spaCy pipeline.

The input doc is lowercased to make the regular expressions case-insensitive.

An empty list called price_ents is defined to hold the identified price entities.

A list of original entities is created to hold any existing entities in the input text.

Each regular expression pattern is iterated over using a for loop that finds all matches of the pattern in the input text using re.finditer(pattern, text). Each match has a start and end index in the input text.

If a valid Span object (i.e. a slice of the input text with a defined start and end index) can be created from the start and end indices using doc.char_span(start, end), the start, end, and text of the Span are appended to price_ents.

For each identified price entity, a new Span object with a label of "PRICE" is created using Span(doc, start, end, label="PRICE").

The newly identified price entities are added to the list of original entities using original_ents.append(price_ent).

Any overlapping entities are filtered out using filter_spans(original_ents).

The filtered list of entities is set as the new entities in the doc object using doc.ents = filtered.

The updated doc object is returned.

iii) For Date :

```python
date_patterns = [
    r"\b\d{1,2}(?:st|nd|rd|th)?\s(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|Jun(?:e)?|Jul(
    r"\b(?:Jan(?:uary)?|Feb(?:ruary)?|Mar(?:ch)?|Apr(?:il)?|May|Jun(?:e)?|Jul(?:y)?|Aug(?:ust)?|Sep(?:t
    r"\b\d{1,2}[./-]\d{1,2}[./-]\d{2,4}\b",  # Matches formats like dd/mm/yyyy, dd-mm-yyyy, dd.mm.yyyy
    r"\b\d{1,2}[./-]\d{1,2}[./-]\d{2}\b",  # Matches formats like dd/mm/yy, dd-mm-yy, dd.mm.yy
    r"\b\d{4}[./-]\d{1,2}[./-]\d{1,2}\b",  # Matches formats like yyyy/mm/dd, yyyy-mm-dd, yyyy.mm.dd
    r"\b\d{2}[./-]\d{1,2}[./-]\d{1,2}\b",  # Matches formats like yy/mm/dd, yy-mm-dd, yy.mm.dd
    r"\b\d{1,2}[./-]\d{4}[./-]\d{1,2}\b",  # Matches formats like dd/yyyy/mm, dd-yyyy-mm, dd.yyyy.mm
    r"\b\d{1,2}[./-]\d{2}[./-]\d{1,2}\b",  # Matches formats like dd/yy/mm, dd-yy-mm, dd.yy.mm
    r"(?:Mon|Tue|Wed|Thu|Fri|Sat|Sun), \d{1,2} (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec), \d{
    r"\b(?:Mon|Tue|Wed|Thu|Fri|Sat|Sun), \d{1,2} (?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\b"
]
```

Here I tried to generate all possible patterns in which a date could be represented.
Below are the illustrations giving idea about all date patterns represented by above regular expressions :

1. dd Month yyyy
2. Month dd yyyy
3. dd/mm/yyyy, dd-mm-yyyy, dd.mm.yyyy
4. dd/mm/yy, dd-mm-yy, dd.mm.yy
5. yyyy/mm/dd, yyyy-mm-dd, yyyy.mm.dd
6. yy/mm/dd, yy-mm-dd, yy.mm.dd
7. dd/yyyy/mm, dd-yyyy-mm, dd.yyyy.mm
8. dd/yy/mm, dd-yy-mm, dd.yy.mm
9. Matches format like Thu, 23 Mar

There may still be other patterns to represent a date. I tried to cover all possible patterns by adding new patterns as and when I encountered a new pattern.

```python
@spacy.Language.component("find_date")
def find_date(doc):
    text = doc.text
    date_ents = []
    original_ents = list(doc.ents)

    # Find dates using patterns
    for pattern in date_patterns:
        for match in re.finditer(pattern, doc.text):
            start, end = match.span()
            span = doc.char_span(start, end)
            if span is not None:
                date_ents.append((span.start, span.end, span.text))

    # Find dates preceded by "DATE", "Date", "DATE:", or "Date:"
    date_prefixes = ["DATE", "Date", "DATE:", "Date:"]
    for prefix in date_prefixes:
        pattern = r"\b" + re.escape(prefix) + r"\s*([\w.-]+)"
        for match in re.finditer(pattern, doc.text):
            start, end = match.span(1)
            span = doc.char_span(start, end)
            if span is not None:
                date_ents.append((span.start, span.end, span.text))

    # Add identified date entities to the original entities
    for ent in date_ents:
        start, end, name = ent
        per_ent = Span(doc, start, end, label="DATE")
        original_ents.append(per_ent)

    # Filter and update the entities
    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

The find_date function is defined as a spaCy component, which takes a doc object, which represents the processed text, and returns an updated doc object with the extracted date entities added as DATE entities.

The text variable is set to the text of the doc object.

A list called date_ents is initialized to store the date entities that are extracted later.

The original entities (doc.ents) are stored in a variable called original_ents.

Next, the function iterates over each regular expression pattern in the date_patterns list and tries to find matches for the pattern in the text using the re.finditer() function. If a match is found, the start and end indices of the match are retrieved, and doc.char_span() function is called to create a spaCy Span object that represents the identified date entity.

If a valid Span object is created, it is added to date_ents.

The code then searches for dates preceded by specific prefixes such as "DATE," "Date," "DATE:," or "Date:." If found, the date entity is extracted, and added to the date_ents list.

The function then creates spaCy Span objects for each of the identified date entities and adds them to the original_ents list.

The filter_spans() function is called to remove any overlapping or nested entities and to filter out any entities that are not recognized. The filtered entities are then added to doc.ents.

Finally, the updated doc object with the extracted DATE entities is returned.

iv) For Time :

```
time_patterns = [
    r"\b\d{1,2}:\d{2}\s?(?:AM|PM|am|pm)\b",  # Matches patterns like 4:00 PM, 10:30 am, etc.
    r"\b\d{1,2}:\d{2}\s?(?:A\.M\.|P\.M\.|a\.m\.|p\.m\.)\b",  # Matches patterns like 4:00 P.M., 10:30 a.m., etc.
    # Add more patterns as needed
]
```

Here I tried to generate all possible patterns in which time could be represented. Though spacy's pretrained model can identify time but still it adds to the working of the code.

Though I tried to cover all patterns in which usually time is represented. There might be some other patterns which can be added as and when encountered.

```
@spacy.Language.component("find_time")
def find_time(doc):
    text = doc.text
    time_ents = []
    original_ents = list(doc.ents)
    for pattern in time_patterns:
        for match in re.finditer(pattern, doc.text):
            start, end = match.span()
            span = doc.char_span(start, end)
            if span is not None:
                time_ents.append((span.start, span.end, span.text))
    for ent in time_ents:
        start, end, name = ent
        per_ent = Span(doc, start, end, label="TIME")
        original_ents.append(per_ent)
    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

Define a spaCy pipeline component called "find_time" that takes a doc (spaCy document object) as input and processes it to identify time-related entities.

Retrieve the input text from the doc object.

Initialize an empty list called "time_ents" to hold any matches found by the regular expression patterns.

Retrieve the original entity spans from the doc object and store them in a list called "original_ents".

Loop through each regular expression pattern in the "time_patterns" list and search for matches in the input text using the re.finditer() function.

For each match found, retrieve the start and end character positions of the match, and create a new spaCy span object using doc.char_span().

If the span object is not None (i.e., a valid span was created), append a tuple containing the span's start position, end position, and text to the "time_ents" list.

Loop through each entity tuple in the "time_ents" list.

Extract the start position, end position, and text of the entity tuple.

Create a new spaCy entity span object using the extracted start and end positions, with the label "TIME".

Append the new entity span object to the "original_ents" list.

Use the filter_spans() function to remove any overlapping entity spans from the "original_ents" list.

Replace the doc's existing entity spans with the filtered list.

Return the updated doc object with identified time entities.

v) For Ticket No. :

```
ticket_pattern = r"(?i)(?:booking id|ticket no)\s*[ :-]*\b([A-Za-z\d]+)\b|\b([A-Za-z\d]{8})\b"
```

The code defines a regular expression pattern called ticket_pattern. This pattern uses the r prefix to indicate a raw string, and the (?i) flag to enable case-insensitive matching. The pattern contains two alternative groups, separated by the | character. Each group captures a booking ID or ticket number, with different formats. The first group matches patterns like "booking id : ABC123" or "ticket no-DEF456", while the second group matches patterns like "XYZ789AB".

Though I tried to cover all ways in which usually a ticket no / booking id is represented but there may still be other ways which may be included in the pattern as and when they are encountered.

```python
@spacy.Language.component("find_ticket_no")
def find_ticket_no(doc):
    ticket_ents = []
    original_ents = list(doc.ents)
    lowercase_text = doc.text.lower()
    for match in re.finditer(ticket_pattern, lowercase_text):
        start, end = match.span(1)
        span = doc.char_span(start, end)
        if span is not None:
            ticket_ents.append((span.start, span.end, span.text))
    for ent in ticket_ents:
        start, end, name = ent
        per_ent = Span(doc, start, end, label="TICKET_NO")
        original_ents.append(per_ent)
    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

The code defines a new spaCy component using the @spacy.Language.component decorator. This component is called find_ticket_no. It takes a spaCy Doc object as input, and returns the same object with ticket number entities added as annotations.

The component initializes an empty list called ticket_ents, which will store the detected ticket number entities.

The component makes a copy of the original entity list in the doc object, and stores it in a new list called original_ents. This is done to preserve any existing named entity annotations in the Doc object.

The component converts the text of the doc object to lowercase, and stores it in a new variable called lowercase_text. This is done to enable case-insensitive matching with the ticket_pattern.

The component uses the re.finditer() function to search for all occurrences of the ticket_pattern in the lowercase_text. This function returns an iterator that generates Match objects.

The component iterates over all the Match objects generated by the re.finditer() function. For each Match, it extracts the starting and ending character positions of the first capturing group (i.e., the actual ticket number), and uses these positions to create a new spaCy Span object.

The component checks if the Span object is not None. This is because some matches from the ticket_pattern might not correspond to valid spans in the Doc object (e.g., if the match occurs in a stopword or punctuation mark). If the Span object is not None, the component appends a tuple of the form (start, end, text) to the ticket_ents list. This tuple stores the starting and ending character positions of the Span, as well as its raw text.

vi) For Movies :

For movies, there is clearly no regular expression . Also I could found machine learning could not do much in recognizing real time data. So I found it better to perform lookup from a large set of movie names which give accurate results to some extent .

I used the following piece of code to create movies.json file by scrapping movies from a popular website www.sacnilk.com .

```
import requests
from bs4 import BeautifulSoup
import json

# URL of the webpage containing the table
url = "https://www.sacnilk.com/entertainmenttopbar/Top_500_Bollywood_Movies_Of_All_Time"

# Send a GET request to the webpage
response = requests.get(url)

# Create a BeautifulSoup object to parse the HTML content
soup = BeautifulSoup(response.content, 'html.parser')

# Find the table element that contains the movie names
table = soup.find('table')

# Find all the rows in the table
rows = table.find_all('tr')

# Extract the movie names from each row
movie_names = []
for row in rows:
    # Assuming the movie name is in the first column (index 0) of each row
    columns = row.find_all('td')
    if columns:
        movie_name = columns[1].text.strip()
        movie_names.append(movie_name)

# Create a dictionary with the movie names
data = {"movies": movie_names}

# Save the movie list as JSON in a file
with open("movies.json", "w") as json_file:
    json.dump(data, json_file)

print("Movie list saved as movies.json.")
```

```python
def ignore_brackets(text):
    result = ""
    bracket_count = 0

    for char in text:
        if char == '(':
            bracket_count += 1
        elif char == ')':
            bracket_count -= 1
        elif bracket_count == 0:
            if char == '\n':
                result += ' '  # Join newline with a space
            else:
                result += char

    return result
```

The above 'ignore_brackets' is a utility function that ignores the strings inside brackets. This is done to remove unnecessary strings to get better results . It also replaces '\n' with space to get all possible strings from the input text so as to better match the movie names from the json file.

```python
# Load the movie names from the JSON file
with open("movies.json", "r") as f:
    movie_names = [name.lower() for name in json.load(f)]

@Language.component("find_movie_name")
def find_movie_name(doc):
    movie_ents = []
    original_ents = list(doc.ents)
    # Check if the extracted text forms a movie name
    for token in doc:
        extracted_text = token.text
        extracted_text = ignore_brackets(extracted_text)
        tokens = extracted_text.split()
        for i in range(len(tokens)):
            for j in range(i + 1, len(tokens) + 1):
                partial_name = " ".join(tokens[i:j])
                lowercase_partial_name = partial_name.lower()
                for movie_name in movie_names:
                    lowercase_movie_name = movie_name.lower()
                    if lowercase_partial_name == lowercase_movie_name:
                        movie_ent = Span(doc, token.i + i, token.i + j, label="MOVIE")
                        movie_ents.append(movie_ent)

    # Add identified movie name entities to the original entities
    original_ents.extend(movie_ents)

    filtered = filter_spans(original_ents)
    doc.ents = filtered
    return doc
```

The code opens up a JSON file called "movies.json" in read-only mode and loads its contents into a list called "movie_names". It also converts all movie names to lowercase, allowing for case-insensitive matching later on.

The code defines a function called "find_movie_name" and decorates it using the "@Language.component" decorator, indicating that it is a custom spaCy component that will be added to the natural language processing pipeline.

The function takes a spaCy document ("doc") as input and initializes an empty list called "movie_ents" to store identified movie name entities.

Next, it creates a copy of the document's original entities in a list called "original_ents".
The function then loops through each token in the document and extracts its text as "extracted_text".

The "ignore_brackets" function is called to remove any text within brackets from the extracted text, as this may interfere with movie name matching.

The extracted text is split into "tokens" based on whitespace, and a nested loop then generates all possible combinations of these tokens as "partial_name".

Each "partial_name" is converted to lowercase for case-insensitive matching and compared to all "movie_names" using another nested loop.

If a match is found between a "partial_name" and a "movie_name," a spaCy span is created from the token indices of the matching tokens and added to "movie_ents".
Finally, the "movie_ents" are appended to "original_ents," filtered to remove overlapping entities, and returned as the updated document's entity annotations.

vii) For Venue :

My code relies on pretrained English model of spacy for recognition of address. As address can be of wide variety , we won't be able to use regular expression for the purpose. We may use s lookup or use Google map API for this.

5) Adding Custom Pipes to Spacy's Pipeline :

```
nlp.add_pipe("find_seat_no", before="ner")
nlp.add_pipe("find_date", before="ner")
nlp.add_pipe("find_price", before="find_seat_no")
nlp.add_pipe("find_time", before="ner")
nlp.add_pipe("find_ticket_no", before="ner")
nlp.add_pipe("find_movie_name",before="ner")
```

The nlp.add_pipe() method is used to add these pipeline components to the pipeline with an optional positioning argument. The before parameter here means that the custom component will be placed before the specified built-in pipeline component (in this case, "ner").

Adding a pipe before another allows us to set priority for the entity recognition.

```
doc = nlp(test)

# Print the modified named entities after applying the custom components
for ent in doc.ents:
    if ent.label_ == "DATE":
        print(ent.text, ": DATE")
    elif ent.label_=="MOVIE":
        print(ent.text,": MOVIE")
    elif ent.label_ == "TIME":
        print(ent.text, ": TIME")
    elif ent.label_ == "SEAT_NO":
        print(ent.text, ": SEAT NO")
    elif ent.label_=="TICKET_NO":
        print(ent.text," : TICKET NO.")
    elif ent.label_ == "PRICE":
        print(ent.text, ": PRICE")
    elif ent.label_=="GPE":
        print(ent.text,": VENUE")
```

nlp is an instance of a spaCy model that has been loaded, representing natural language processing capabilities.
test is a string variable containing some text to be analyzed by nlp.
doc is assigned the value of the analyzed test text by calling the nlp model instance with the test string inside parentheses.
The code then performs some custom named entity recognition (NER) by checking for specific entity labels in doc.ents, which is a list of the named entities identified by the nlp model.
For each entity that matches a particular label (DATE, MOVIE, TIME, SEAT_NO, TICKET_NO, PRICE, or GPE), the code prints out the text of the entity and its corresponding label. For example, if a named entity with label DATE is found in doc.ents, the date and the label "DATE" will be printed.