Universidad Nacional Autónoma de México
Facultad de Ciencias
Modelado y Programación | 7054
Proyecto 1
Jacome Delgado Alejandro
Jimenez Sanchez Emma Alicia
Labonne Arizmendi Raúl Emiliano
8 de octubre de 2023

1. Proposito

Se deberá entregar una aplicación gráfica la cual acepte como entrada, un ticket o el nombre de la ciudad a la cual se esta interesado en conocer el clima. La entrada debe ser capaz de aceptar errores, por ejemplo: Monterye, Montery, Monterey y MTY, cada uno de estos deberá contestar con el clima de Monterey.

2. Desarrollo

La primer problematica a enfretarnos seria la de saber como conseguir la informacion del clima de una ciudad dada, para esto se utilizo los servicios de OpenWeather[6], esta es una empresa que pone a nuestra disposicion una API a la cual llamas con unas coordenadas y una llave, esta devuelve el clima de las coordenadas proporcionadas.

Nesesitas una API key valida proporcionada por OpenWeather, en su pagina estan las instrucciones para conseguirla, por motivos obvios no la podemos proporsionar nosotros.

Ya que se posee la una forma de acceder al clima de cualquier parte del mundo, se tendra que ver como es que se obtienen los datos, para esto se nos proporciono un dataset(en formato csv), de la cual sacar los tickets y las coordenadas de las ciudades, asi pues ya tenemos la base de datos. Sin embargo, como el programa puede recibir el IATA code de un aeropuerto, y ademas el nombre de la ciudad donde se encuentra, se contara con un dataset independiente en el que se almacena el IATA code, el nombre de la ciudad y las coordenadas de la misma (tambien en formato csv).

Como contamos con archivos csv en los cuales tenemos almacenada la informacion, una forma de poder leerlos y hacer busquedas en el de forma relativamente simple, sera utilizando la libreria de pandas, esta nos permite leer datasets en diversos formatos y manipularlos de forma sencilla para su implementacion basta con leer la documentacion [7]. Asi pues, ya es posible leer los datasets proporcionados y resactar de ellos la informacion que nesitamos.

Ahora bien, ya tenemos como conseguir las coordenadas nesesarias para el clima, es nesesario buscar con una forma de llamar a la API con los datos correspondentes para conseguir la información que se requeire. para lograr esto de una forma en la que sea sencillo capturar posibles errores a la hora de hacer la peticion, y que ademas nos de la informacion en un formato facil de manipular, se decidio por usar la libreria de requests[5], ya que esta solo requiere como argumento una url y a partir del objeto instanciado es facil verificar el estatus que mando el servidor de la API y recuperar la informacion que este regresa en formato json es muy facil.

Como se sigue el patron de diselño MVC (Modelo Vista Controlador) y ya tenemos toda la parte del modelo cubierta, es nesesario determinar con que se realizara la parte de vista (ya que esto determina

como se hara el controlador que comunicara ambas partes), asi pues basado en facilidad de uso y practicidad, se opto por utilizar flask [4], el cual facilita la creacion de la interfaz grafica, asi pues, el programa sera una aplicacion web.

Con el framework de Flask, se pudo crear plantillas de html de manera eficiente y de cierta manera con una mejor estructurada, facilitar la comunicación entre la parte visual y el controlador, favorecer la colaboración en el proyecto y propocionar una estructura al proyecto. El framework posibilito la entrada y salida de datos desde las plantillas html, con apoyo de las librerias importadas de request y $render_template$, se logro acceder a los a los datos que se introducen mediante la bara de navegacion de la plantilla index.html, como en la bara de navegación pueden ingresar cualquier dato, se requiere comparar si el tamaño del dato para poder determinar si es ticket o ciudad o IATA code.

Después del proceso de datos y la obtencion de informacion de la api OpenWeather [6], dependiendo de la busqueda los resultados se van a presentar de manera amigable en las plantillas $result_city.html$ o $result_ticket.html$.

Para poder hacer las diferentes plantillas de html, se ocupo componentes del framework de Bootstrap [2], como la barra de navegación y la carta que se despliega tanto en las plantillas html de los errores y los resultados, además de usar grid [3] para el acomodamiento de los datos en las plantillas de los resultados.

3. Resolucion del problema

Lo primero a tratar seria como es que funcionaria la app por el lado del modelo, para esto se diseñaron los siguientes pseudocodigos:

Algoritmo para obtener los datos necesarios de un ticket.

```
funcion read(ticket):
    data := dataDB()
    datoTicket := buscaTicket(ticket)
    if (datoTicket is empty):
        return error
    ciudad_1 := datoTicket.loc[0, 'origen']
    ciudad_2 := datoTicket.loc[0, 'destino]
    clima := [buscaClima(ciudad_1), BuscaClima(ciudad_2)]
    return clima
```

basicamente, esta funcion lo que hace es mandar a llamar a la funcion que busca el clima para la ciudad de destino y la de llegada, para luego mandar una tupla que contiene los json con el clima de llegada y de destino.

Algoritmo para obtener el clima de una ciudad en formato JSON

```
funcion buscaClima(ciudad):
    fila := filaCiudad(ciudad)
    llave := fila.loc[0,'IATA']
    coord := buscaIATA(fila)
    clima := buscaEnCache(llave)
    if (clima is none)
        clima := solicita(coord, llave)
    return clima
```

este es el algoritmo principal de toda la logica del programa, ya que este es el que prepara los datos nesesarios para conseguir el clima de la ciudad, ya sea pidiedoselo al cache o haciendo una llamada al API de OpenWeather.

Algoritmo para obtener el clima

```
funcion solicita(coord, llave):
    clima := llamaAPI(coord, API_KEY)
    cache[llave] := clima
    return clima
```

este algoritmo lo que hace es revisar si el clima se encuentra en el cache del sistema, si no esta; hace una llamada a la API con la funcion siguiente:

Algoritmo para hacer la llamada a la API

```
funcion llamadaAPI(coord, API_KEY):
    url := 'https://api.openweathermap.org/data/2.5/weather?lat='
        + coord[0] + '&lon=' + coord[1] + '&appid=' + llave
        + '&units=metric'
    api := request.get(url)
    try:
        verifica(api)
    catch ValueError:
        lanza ValueError
    catch SyntaxError:
        lanza SyntaxError
    catch UserWarning:
        lanza UserWarning
    catch FutureWarning:
        lanza FutureWarning
    return api.json()
```

aqui se utiliza la libreria requests para poder hacer la llamada, como se ve, se llama a la funcion verifica para revisar que la llamada fue hecha correctamente. en caso contrario, lanza una excepcion

Algoritmo para buscar en la cache

```
funcion buscaEnCache(iata):
    clima := cache.get(iata)
    return clima
```

este algoritmo sencillamenmte revisa en el cache si esta el json con el clima buscado, en caso nulo regresa un elemento vacio

Algoritmo para obtener la latitud y longittud de una ciudad

```
funcion buscaIATA(datoCiudad):
    coord := [datoCiudad.loc[0, latitud], datoCiudad.loc[0, longitud]]
    return coord
```

lo que hace este algoritmo es que dado un DataFrame con solo una fila, regresa los valores guardados en las columnas de latitud y longitud

Algoritmo para obtener los datos necesarios para el funcionamiento de la busqueda en la base de datos

```
funcion filaCiudad(cadena):
    if(longitud(cadena) > 3):
        datosCiudad = BuscaLevenstein(cadena)
    else:
        datos := datoCiudad()
        datosCiudad := datos[datos['IATA'] == cadena]
        if (datosCiudad is empty):
            lanza TypeError
        return datosCiudad
```

lo que hace este algoritmo es buscar en el dataframe con la informacion la fila que contiene la informacion nesesaria para la busqueda, si no la encuentra regresa un TypeError

Algoritmo que verifica el estatus a la llamada de la API

```
funcion verifica(api):
    if(status(api) == 401)
        lanza ValueError
    if(status(api) == 400)
        lanza SyntaxError
    if(status(api) == 429)
        lanza UserWarning
    if(status(api) == 500)
        lanza FutureWarning
    return ''
```

lo que hace es evaluar el codigo de estatus que almacena el obejto requests generado por la llamada a la API y lanzar una execpcion especifica para cualquier tipo de fallo.

Algoritmo de levenstein

```
funcion levenstein(cadena1, cadena2):
    dicc := {}
    for i in range longitud(cadena1) + 1):
        dicc[i] := {}
        dicc[i][0] := i
    for i in range longitud(cadena2) + 1):
        dicc[0][i] := i
    for i in range 1 <= longitud(cadena1) + 1):
        for j in range 1 <= longitud(cadena2) + 1):
            dicc[i][j] := min(dicc[i][j-1]+1, dicc[i-1][j-1]
                              + (!(cadena1[i-1] == cadena2[j-1])))
    return dicc[longitud(cadena1)][longitud(cadena2]
```

este algoritmo no da la distancia de levenstein para 2 strings cualquiera

Algoritmo para buscar la informacion de una ciudad en la base de datos con el algortimo de levenstein.

```
funcion buscaLevenstein(cadena):
```

```
datos := datoCiudad()
linea := dato[dato['IATA] == 'zzz']
minimo := 99
fila := 0
for i in range longitud(datos):
    distancia := levenstein (normaliza(datos.loc[i, 'cities],
                  normaliza(cadena))
    if (distancia == 0):
        return datos.iloc[[i]]
    if (distancia < minimo):
        minimo = distancia
        fila = i
if (minimo > 10):
    return linea
return datos.iloc[[fila]]
```

lo que hace este algoritmo es buscar en el dataset la fila que contenga el dato que tiene la menor distancia de levenstein, regresa esa fila, en caso de que la distancia sea mayor que cierto umbral, regresa una fila vacia

Algoritmo que carga el dataset del archivo CSV al archivo "dataset2.csv".

```
funcion dataDB()
    directorio_actual := os.path.dirname(os.path.realpath(__archivo__))
    nombre_del_archivo := os.path.join(directorio_actual,"dataset2.csv")
    dato := pd.read_csv(nombre_del_archivo,header=0)
    return dato
```

lo unico que hace esta funcion es leer los datos del dataset y cargarlos en un DataFrame

Algoritmo que carga el dato de la ciudad del archivo CSV al archivo çities3.csv".

```
funcion dataCity()
    directorio_actual := os.path.dirname(os.path.realpath(__archivo__))
    nombre_del_archivo := os.path.join(directorio_actual,"cities3.csv")
    dato := pd.read_csv(nombre_del_archivo, header=0)
    return data
```

hace lo mismo que el anterior, solo que con el dataset de las ciudades.

en estos se puede observar a detalle que es lo que hacen los algoritmos del programa, claro esta que en la documentacion del propio codigo se dan mas detalles acerca de cada uno.

4. Código de otras fuentes

en el proyecto hicimos uso de codigo externo, especificamnete tomamos la implementacion del algoritmo de Levenshtein que se encuentra dsiponible en wikipedia[1], esto fue hecho asi porque es una version mas eficiente de la que nosotros podriamos hacer y ademas como tal solo calcula la distancia de Levenshtein de 2 strings, la implentacion de este algoritmo para que nos ayudara a buscar coincidencias fue totalmente original. el codigo en cuestion es la funcion Levenstein()

Referencias

[1] Algoritimo de levenshtein. https://es.wikipedia.org/wiki/Distancia$_d e_L evenshtein$.

[2] Bootstrap. https://getbootstrap.com/.

[3] Bootstrap. https://getbootstrap.com/docs/5.3/layout/grid/.

[4] Flask. https://flask.palletsprojects.com/en/2.3.x/.

[5] Librería request. https://pypi.org/project/requests/.

[6] Openweather. https://openweathermap.org.

[7] Pandas. https://pandas.pydata.org.