

Monument info

MEMBRI:

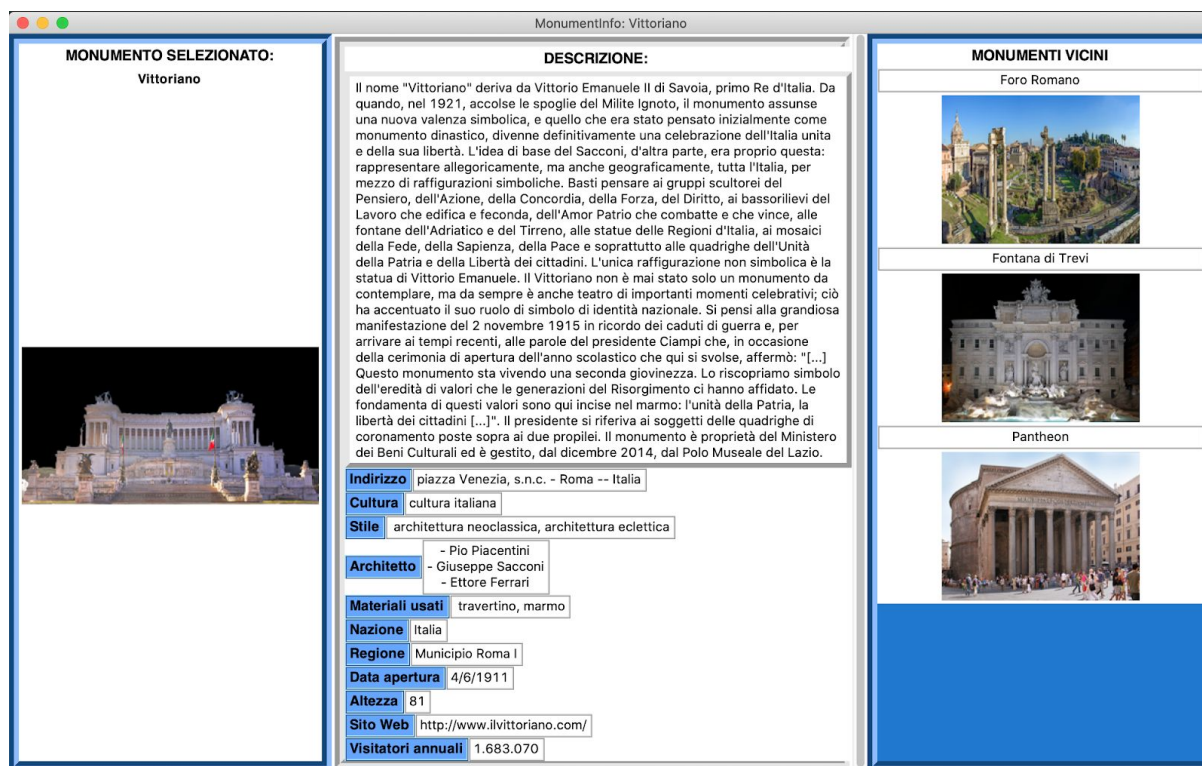
1. Piccolo Nicola MAT. **679594**
2. Nacci Oronzo MAT. **681044**

Intro:

MonumentInfo è un progetto software che si pone come obiettivo di riconoscere un monumento, data l'immagine fornita dall'utente, il programma fornisce una descrizione e raccomanda altri luoghi vicini visitabili.

Gui:











All'avvio del programma è possibile selezionare un monumento che sarà oggetto del riconoscimento, il monumento selezionabile è nella cartella *test* del repository.



Il programma implementa un'interfaccia grafica dove sulla colonna a sinistra viene fornita l'immagine del monumento riconosciuto, nella colonna centrale troviamo una breve descrizione seguita da altre proprietà, infine sulla colonna più a destra vengono riportati i tre luoghi raccomandati da visitare nella zona.

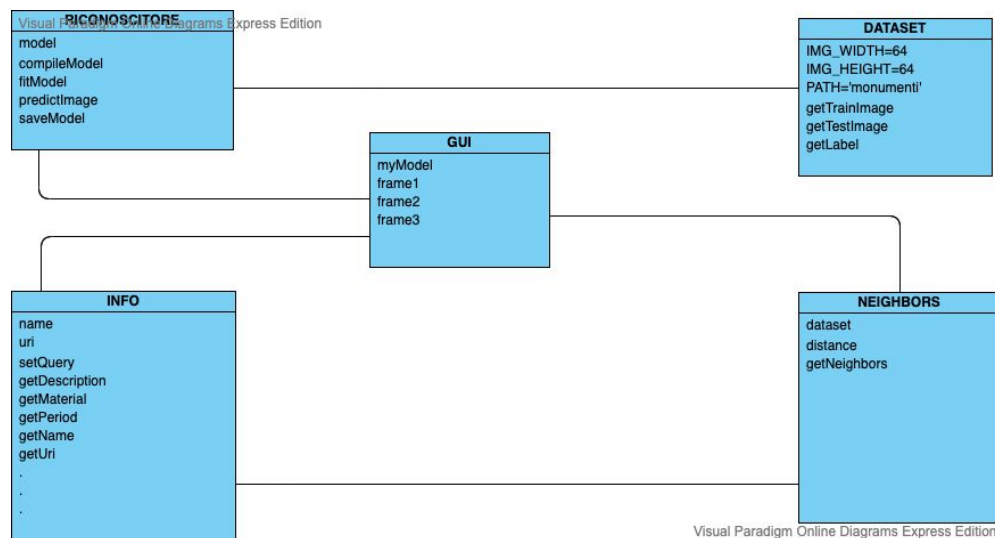
Descrizione progettuale:

repository:

	modello	14/02/2020 11:48	Cartella di file	
	monumenti	14/02/2020 11:48	Cartella di file	
	test	14/02/2020 11:48	Cartella di file	
	dataset.py	14/02/2020 11:48	File PY	2 KB
	gui.py	14/02/2020 11:48	File PY	8 KB
	infoMonumento.py	14/02/2020 11:48	File PY	11 KB
	kfCV.py	14/02/2020 11:48	File PY	5 KB
	myModel.py	14/02/2020 11:48	File PY	2 KB
	neighbors.py	14/02/2020 11:48	File PY	3 KB
	trainMyModel.py	14/02/2020 11:48	File PY	1 KB

- la cartella modello contiene file "mymodel.h5" corrispondente al modello salvato del riconoscitore;
- la cartella monumenti contiene altre sottocartelle di immagini di monumenti classificati utilizzati come train e due file .csv:
 - label.csv che contiene i nomi dei monumenti, utilizzato per effettuare le query;
 - monuments.csv contiene le (LAT, LON) dei monumenti italiani con la proprietà "attrazione turistica" su WikiData. Il file è utilizzato nel k-NN;
- la cartella test contiene le immagini utilizzabili per il test programma.

classi:



-La classe **GUI (gui.py)**, che implementa l'interfaccia, richiama le altre classi a supporto dell'esecuzione del programma

- La classe **RICONOSCITORE (myModel.py)** implementa il riconoscitore delle immagini, addestrato su un training set composto da immagini di monumenti già classificati. La classe compila e crea un modello che potrà essere salvato e usato come riconoscitore.

-La classe **INFO (infoMonumento.py)** implementa una serie di query per estrarre informazioni relative al monumento riconosciuto.

-La classe **DATASET (dataset.py)** dato un path, estrae le immagini dalle sotto cartelle per creare i subset di train e validation utili all'addestramento del riconoscitore.

-La classe **NEIGHBORS (neighbors.py)** implementa il recommender system.

Dettagli implementativi:

Il programma è stato sviluppato in python 3.7

Il programma implementa una **rete neurale convoluzionale** (CNN) per la classificazione delle immagini. Inoltre, grazie all'uso di **ontologie online** e **Linked Open Data**, possiamo ritrovare informazioni strutturate, estratte dal **Semantic Web**, per poter fornire all'utente una completa descrizione del monumento riconosciuto. In aggiunta forniamo all'utente una serie di monumenti vicini visitabili. Utilizzando l'algoritmo di **apprendimento supervisionato case-based K-NN** ($k=3$) ritroviamo la lista dei neighbors sulla base di un nuovo dataset appositamente creato, utilizzando altre info estratte da una ontologia online (**WIKIDATA**).

Dataset:

esempio di alcune immagini presenti nel dataset:



TRAINING SET

Il dataset è stato da noi creato, utilizzando immagini prese dal web, successivamente scontornate per ovviare al numero limitato di immagini.

Per ogni monumento disponiamo di circa 15 foto. Il 90% viene utilizzato per il training e il restante 10% per validation. Le immagini vengono: ridotte ad una dimensione di 64x64 e vengono trasformate in array (matrice) di pixel con valori scalati tra 0 e 1.

Per fare ciò utilizziamo la libreria *preprocessing.image* di Keras (sottolibreria di tensorflow) che ci fornisce metodi per il preprocessing delle immagini e contenitori (**ImageDataGenerator**) che saranno istanziati con l'insieme di immagini di train e quelle di validation.

Riconoscitore:

La rete neurale è stata creata utilizzando le librerie di keras.

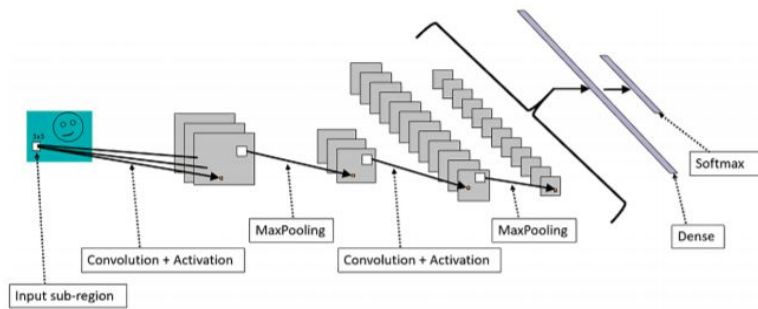
In particolare con `keras.models` abbiamo importato la classe fondamentale che rappresenta la sequenza di layers del nostro modello. Il nostro modello, ossia l'insieme dei "pezzi" che andranno a comporre la nostra rete neurale artificiale sarà composto da una sequenza lineare di layers.

Sequential è appunto la classe che ci permette di aggiungere via via layer dopo layer dall'input all'output della sequenza.

Dalla libreria "`tensorflow.keras.layers`" abbiamo importato i vari livelli: **Dense, Conv2D, Flatten, MaxPooling2D**

- **Conv2D:** è il livello che si occupa di individuare schemi, come ad esempio curve, angoli, circonferenze o quadrati raffigurati in un'immagine con elevata precisione. Per attuare il processo di Convoluzione si fa scorrere un certo Filtro (chiamato Kernel) sul tensore (l'immagine). Un filtro è un insieme di pesi rappresentati sotto forma di un tensore di dimensione $M \times N$ e la cui profondità è la medesima dell'immagine di input. Quando questo filtro viene fatto scorrere sull'immagine, la regione su cui si trova in quel momento è detta *Campo Recettivo*. Durante questo processo, i valori del filtro vengono moltiplicati ai pixel originali, ed infine sommati in un singolo valore scalare che rappresenta l'output di quel passo di convoluzione. Questo processo viene ripetuto fino ad arrivare alla fine dell'immagine. Il tensore risultante è la rappresentazione di tutti gli output del processo di convoluzione e si chiama *Activation Map* (Mappa di attivazione)
- **MaxPooling2D:** dopo ogni layer convoluzionale si può applicare un'operazione chiamata Pooling che consiste nel creare una versione ridimensionata dell'output dello strato precedente in modo da ridurre le dimensioni del risultato ed ottenere una rappresentazione più compatta delle feature iniziali.
- **Flatten:** è un livello che si occupa di ridurre la dimensionalità dell'output dei vari strati convoluzionali in modo tale da ottenere in uscita un unico vettore. Questo vettore sarà l'input dell'ultima parte della rete neurale.
- **Dense:** Il vettore finale del Flatten viene usato come input per una rete neurale 'classica' in cui vengono utilizzati strati completamente interconnessi (dense layers) di neuroni che creano la rappresentazione finale dell'output.

esempio rappresentazione grafica CNN:



MyModel:

Il nostro modello di rete neurale è il seguente:

```
self.model = Sequential([
    Conv2D(16, 3, padding='same', activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D(),
    Conv2D(32, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Conv2D(64, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(train.num_classes, activation='softmax')
])
```

E' formato da 9 layers:

- Il primo layer convoluzionale prende come input un'immagine nelle dimensioni specificate (64x64x3), ha 16 neuroni di output e crea una finestra kernel 3x3 per creare la mappa di attivazione. La funzione di attivazione utilizzata è la ReLU:

- *Rectified Linear Unit (ReLU)*:

$$f(x) = \max(0, x)$$



- A seguire abbiamo un livello di Pooling.
- Troviamo poi altre 2 coppie di layers: <Conv2D e MaxPooling2D> che aumentano progressivamente le dimensioni dell'output, da 16 a 32 e da 32 a 64.
- Abbiamo poi un livello Flatten per appiattare l'output del livello precedente.
- Seguono poi livelli completamente connessi: col fine di connettere tutti i neuroni del livello precedente e di stabilire le varie classi identificative secondo una determinata probabilità. Di fatto l'ultimo layer Dense ha come output il numero di classi in cui classificare gli esempi e la funzione di attivazione utilizzata è la *softmax* che

restituisce un un vettore di probabilità. Esso contiene, per ogni esempio, il valore di probabilità che l'esempio ha di appartenere ad una certa classe.

Una volta creato il modello della CNN bisogna compilarlo:

```
def compileModel(self):  
    self.model.compile(optimizer='adam',  
                        loss='binary_crossentropy',  
                        metrics=['accuracy'])
```

Noi abbiamo scelto di utilizzare l'ottimizzatore *ADAM*, e come metrica di errore da minimizzare la *binary_crossentropy*.

Utilizziamo poi l'*accuratezza* per confrontare l'efficacia del nostra rete sul validation set.

- L'**OTTIMIZZATORE** specifica l'algoritmo scelto per effettuare la **discesa di gradiente**
- La **LOSS** utilizzata è la log loss:

$$\log_loss = -(y \log(p) + (1-y) \log(1-p))$$

dove y è la classe reale, mentre p è quella predetta dalla CNN

Per effettuare il train richiamiamo il metodo *fit*:

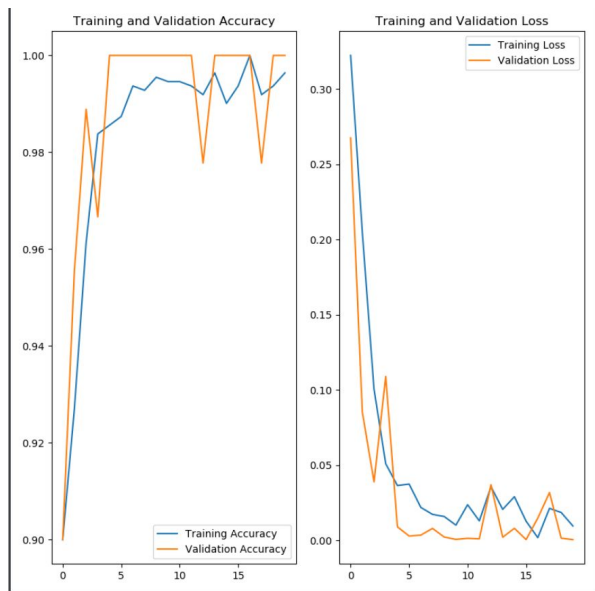
```
def fitModel(self):  
    total_train = self.train.samples  
    batch_train = self.train.batch_size  
  
    total_val = self.val.samples  
    batch_val = self.val.batch_size  
  
    self.model.fit_generator(  
        self.train,  
        steps_per_epoch=total_train // batch_train,  
        epochs=self.epochs,  
        validation_data=self.val,  
        validation_steps=total_val // batch_val  
    )
```

Qui specifichiamo al modello l'insieme di train, l'insieme di validazione, e il parametro *epochs* che indica il numero di 'epoche' (volte) in cui effettuare il train sui dati.

Impostazione parametri:

Dopo aver effettuato una **K-fold Cross Validation**, con k = 5, abbiamo scelto i parametri migliori per *batch-size* e *epochs* (rispettivamente 3 e 20), ottenendo una accuratezza media di circa 98%.

qui un grafico, ricavato direttamente dal programma, sull'andamento dell'errore del nostro modello sull'insieme di validazione e su quello di train:



Query:

Dato il monumento riconosciuto per estrapolare le informazioni necessarie abbiamo utilizzato le basi di conoscenza di WikiData e DBpedia.

WikiData è un base di conoscenza online che raccoglie dati strutturati. L'archivio di WikiData è costituito principalmente da triple RDF soggetto-predicato-oggetto.

Per estrapolare le informazioni dal web semantico usiamo la libreria **SPARQLWrapper** che ci permette di effettuare query **SPARQL** specificando un apposito *endpoint* da cui ricavare le informazioni strutturate. Gli endpoint utilizzati sono quello di DBPedia e quello di WikiData.

```
def setQuery(self, query, wrapper):
    sparql = SPARQLWrapper(wrapper)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    while True:
        i = 1
        try:
            results = sparql.query().convert()
            return results
        except urllib.error.HTTPError:
            time.sleep(i)
            i += 1
```

Il metodo **setQuery** prende in input l'endpoint che specifica la fonte da cui prelevare le informazioni e la query sotto forma di stringa.

Attraverso il metodo **setQuery (query)** riusciamo ad effettuare le query, mentre con **setReturnFormat(JSON)** riusciamo a specificare in che formato verranno restituiti i risultati, per poi trattare l'output di conseguenza.

k-NN:

Utilizzando l'algoritmo di apprendimento supervisionato KNN, nella fase del calcolo dei neighbors di una dato esempio, riusciamo a ritrovare luoghi vicini all'esempio, da poter visitare.

La similarità usata nel calcolo dei neighbors è data dalla distanza geografica del monumento riconosciuto e gli altri elementi del dataset.

Il dataset è stato creato precedentemente, sulla base di una query che ritorna tutti i luoghi/monumenti presi da WikiData con la proprietà "attrazione turistica" presenti in Italia.

Il dataset ha come campi la locazione geografica del luogo (LAT e LON) e l'URI, in modo tale da poter derivare le altre info con opportune query sull'URI.