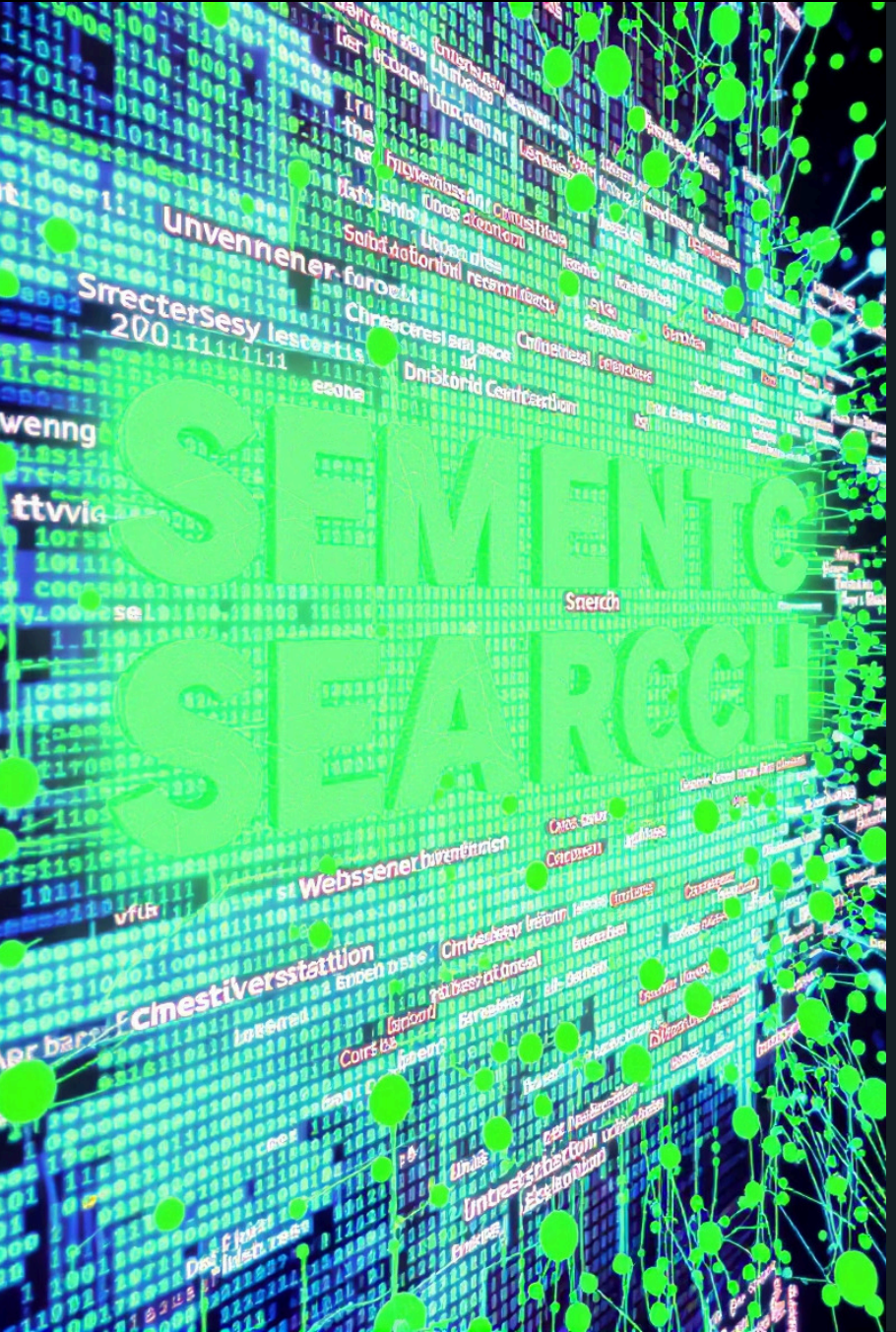
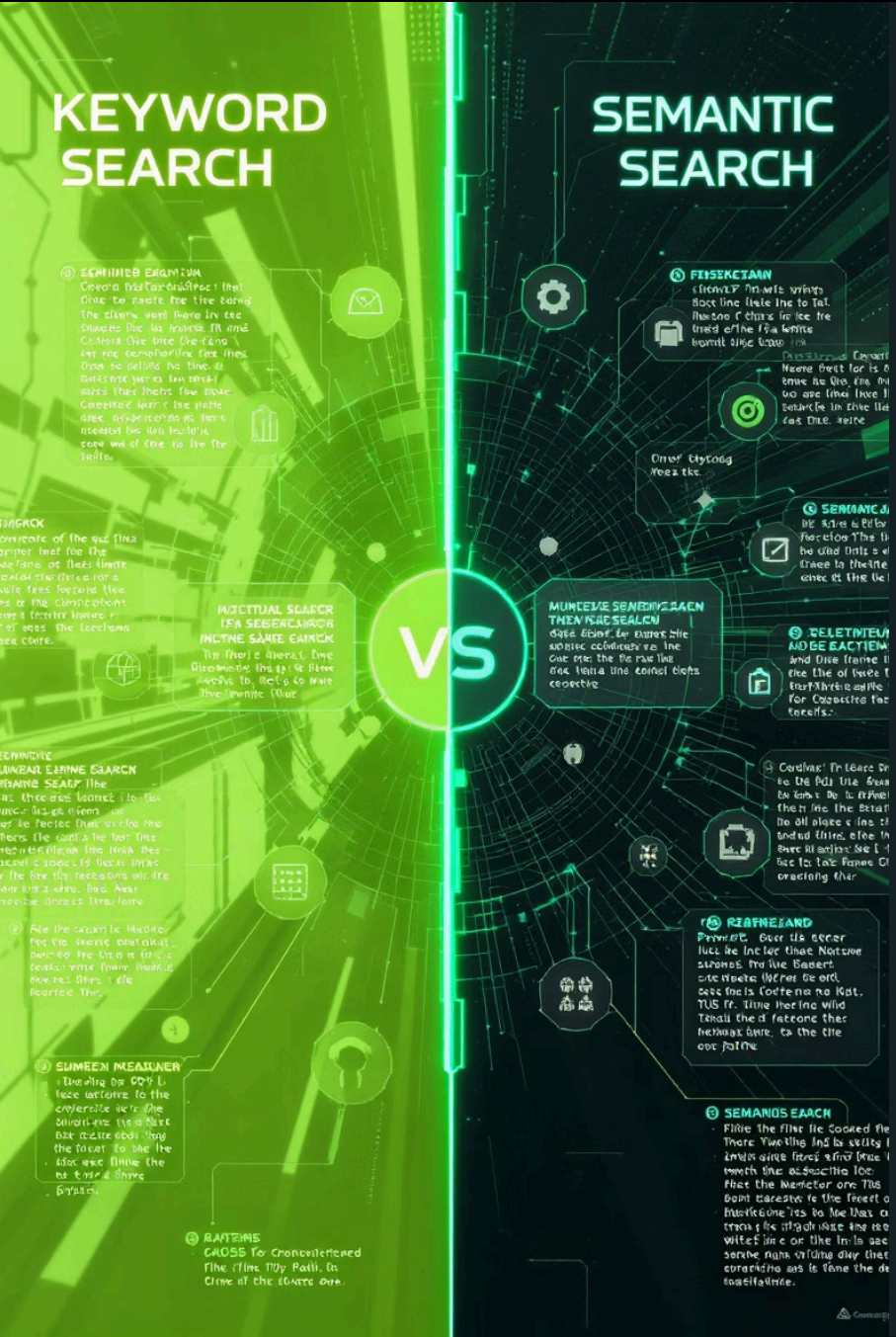


Ricerca Semantica di Frasi e Similitudini in Python

Benvenuti a questo corso introduttivo sulla ricerca semantica di frasi utilizzando Python. Esploreremo come creare un sistema che comprenda il significato delle parole oltre la semplice corrispondenza letterale, utilizzando modelli di embeddings avanzati. Questo approccio ci permetterà di trovare frasi simili basandoci sul loro significato, non solo sulle parole condivise.



Che Cos'è la Ricerca Semantica?

La ricerca semantica va oltre la semplice corrispondenza di parole chiave per comprendere l'intento e il contesto delle query. Mentre la ricerca tradizionale cerca corrispondenze esatte di termini, la ricerca semantica identifica il significato sottostante.

Questo approccio utilizza modelli di machine learning avanzati che trasformano il testo in rappresentazioni numeriche (embeddings) che catturano relazioni semantiche. Così, una query su "auto economiche" può trovare risultati relativi a "veicoli a basso costo" anche senza parole in comune.

Ricerca Tradizionale

Basata su corrispondenza esatta delle parole chiave. Limitata da sinonimi e contesto.

Ricerca Semantica

Comprende significato e contesto. Trova relazioni concettuali tra frasi diverse.

Preparazione del Dataset di Frasi

Il primo passo per costruire un sistema di ricerca semantica è preparare un dataset di frasi su cui lavorare. Questo dataset costituirà la base del nostro "corpus" di conoscenza che verrà interrogato durante le ricerche.

È importante che il dataset sia rappresentativo del dominio in cui vogliamo operare. Per un corso introduttivo, possiamo utilizzare esempi semplici ma significativi che coprano diversi argomenti e stili linguistici.

Esempio di Dataset Base

- La programmazione in Python è divertente
- I linguaggi di programmazione moderni sono versatili
- Il machine learning richiede molti dati
- Gli algoritmi di ricerca migliorano l'esperienza utente
- L'intelligenza artificiale sta cambiando il mondo

Considerazioni Importanti

- Varietà di lunghezze delle frasi
- Diversità di argomenti
- Inclusione di sinonimi e concetti correlati
- Bilanciamento tra frasi simili e diverse

Creazione degli Embeddings con OpenAI

Gli embeddings sono rappresentazioni vettoriali che catturano il significato semantico delle frasi. Per crearli, utilizzeremo l'API di OpenAI, che offre modelli di embedding potenti e pre-addestrati.

La funzione **openai.resources.embeddings.create()** ci permette di trasformare il testo in vettori numerici ad alta dimensionalità. Questi vettori posizionano frasi con significati simili vicine nello spazio vettoriale.

Testo di Input

Frasi in linguaggio naturale dal nostro dataset

API OpenAI

Elaborazione tramite modelli pre-addestrati

Embeddings

Vettori numerici multidimensionali (es. 1536 dimensioni)

```
import openai

response = openai.resources.embeddings.create(
    model="text-embedding-ada-002",
    input="La programmazione in Python è divertente"
)
embedding = response.data[0].embedding
```

Conversione in Vettori NumPy

Una volta ottenuti gli embeddings dall'API di OpenAI, dobbiamo convertirli in vettori NumPy per facilitare le operazioni matematiche successive. NumPy ci offre strumenti potenti per manipolare array multidimensionali e calcolare similitudini tra vettori.

Questa conversione è semplice ma essenziale, poiché ci permette di sfruttare le funzionalità avanzate di NumPy per l'elaborazione vettoriale efficiente, particolarmente importante quando lavoriamo con dataset di grandi dimensioni.

```
import numpy as np

# Converti l'embedding in un array NumPy
embedding_np = np.array(embedding)

# Stampa la forma del vettore
print(f"Dimensioni: {embedding_np.shape}")

# Output esempio: Dimensioni: (1536,)
```

Vantaggi degli Array NumPy

- Operazioni vettoriali ottimizzate
- Funzioni matematiche integrate
- Efficienza di memoria
- Compatibilità con librerie scientifiche
- Operazioni di broadcasting

Calcolo della Similarità con Cosine Similarity

La similarità del coseno è la metrica più utilizzata per misurare quanto due embeddings siano semanticamente vicini. Questo metodo calcola il coseno dell'angolo tra due vettori: più il valore si avvicina a 1, più i vettori sono simili.

Questa metrica è particolarmente adatta per gli embeddings perché si concentra sulla direzione dei vettori piuttosto che sulla loro magnitudine, rendendo il confronto più robusto rispetto a differenze di lunghezza delle frasi.

Formula

$sim(A,B) = \cos(\theta) = (A \cdot B)/(||A|| \cdot ||B||)$

Dove $A \cdot B$ è il prodotto scalare e $||A||$ è la norma del vettore

Intervallo

Da -1 (opposti) a 1 (identici)

0 indica ortogonalità (nessuna relazione)

Implementazione

Facilmente calcolabile con scikit-learn o NumPy

Efficiente anche con vettori ad alta dimensionalità

```
from sklearn.metrics.pairwise import cosine_similarity

similarity = cosine_similarity([embedding_1], [embedding_2])[0][0]
print(f"Similarità: {similarity:.4f}")
```

Funzione di Ricerca Semantica Base

Costruiamo ora una funzione di ricerca semantica che, data una query, restituisca le frasi più simili dal nostro dataset. Questa funzione rappresenta il cuore del nostro sistema e combina tutti gli elementi visti finora.

La funzione calcola l'embedding della query, lo confronta con tutti gli embeddings del dataset utilizzando la similarità del coseno, e restituisce i risultati ordinati per rilevanza. Questo approccio ci permette di trovare frasi semanticamente simili anche quando non condividono parole chiave.

```
def semantic_search(query, dataset, embeddings, top_n=3):
    # Ottieni l'embedding della query
    query_response = openai.resources.embeddings.create(
        model="text-embedding-ada-002",
        input=query
    )
    query_embedding = np.array(query_response.data[0].embedding)

    # Calcola la similarità con tutti gli embeddings del dataset
    similarities = cosine_similarity([query_embedding], embeddings)[0]

    # Ordina i risultati per similarità e prendi i top_n
    top_indices = np.argsort(similarities)[::-1][:top_n]

    # Restituisci le frasi più simili con i loro punteggi
    results = [
        (dataset[i], similarities[i])
        for i in top_indices
    ]

    return results
```

Normalizzazione dei Vettori

La normalizzazione dei vettori è un passaggio importante per migliorare l'efficienza e l'accuratezza del calcolo della similarità. Un vettore normalizzato ha una lunghezza (norma) pari a 1, mantenendo inalterata la sua direzione nello spazio.

Quando normalizziamo gli embeddings, il calcolo della similarità del coseno si semplifica in un semplice prodotto scalare, risparmiando tempo di elaborazione. Inoltre, la normalizzazione può migliorare la qualità dei risultati eliminando l'influenza della lunghezza del testo sulle similitudini.

Vantaggi della Normalizzazione

- Calcoli di similarità più efficienti
- Rimozione dell'influenza della lunghezza
- Maggiore stabilità numerica
- Confronti più equi tra frasi

```
from sklearn.preprocessing import normalize

# Normalizza gli embeddings
normalized_embeddings = normalize(embeddings)

# Con vettori normalizzati, il coseno
# si riduce al prodotto scalare
def cosine_sim_normalized(vec1, vec2):
    return np.dot(vec1, vec2)
```

Dopo la normalizzazione, ogni vettore avrà una norma euclidea pari a 1, preservando le informazioni direzionali che sono cruciali per la similarità semantica.

Gestione Batch per Chiamate Multiple

Quando lavoriamo con dataset più grandi, diventa essenziale ottimizzare le chiamate API per generare gli embeddings. La gestione a batch ci permette di inviare multiple frasi in una singola richiesta, riducendo il numero di chiamate e migliorando significativamente l'efficienza.

L'API di OpenAI supporta la generazione di embeddings per più testi contemporaneamente, consentendoci di elaborare interi lotti di frasi con una sola chiamata. Questo non solo riduce i tempi di attesa ma anche i costi associati alle chiamate API.

Preparazione Batch

Suddividi il dataset in gruppi di dimensione appropriata (es. 100 frasi per batch)

Chiamata API

Invia ciascun batch come array all'API di OpenAI

Gestione Risposta

Elabora la risposta mantenendo la corrispondenza tra frasi ed embeddings

```
def get_embeddings_batch(texts, batch_size=100):
    all_embeddings = []

    for i in range(0, len(texts), batch_size):
        batch = texts[i:i+batch_size]
        response = openai.resources.embeddings.create(
            model="text-embedding-ada-002",
            input=batch
        )
        batch_embeddings = [item.embedding for item in response.data]
        all_embeddings.extend(batch_embeddings)

    return np.array(all_embeddings)
```

Ordinamento dei Risultati per Similarità

Una volta calcolate le similarità tra la query e tutte le frasi del dataset, dobbiamo ordinare i risultati per mostrare prima quelli più rilevanti. Questo processo è essenziale per offrire all'utente i migliori risultati in cima alla lista.

In Python, possiamo utilizzare funzioni di ordinamento avanzate che ci permettono di definire criteri personalizzati. Per la ricerca semantica, ordineremo le frasi in base al loro punteggio di similarità in ordine decrescente, mostrandole dal più al meno rilevante.

```
# Calcolo delle similarità
similarities = cosine_similarity(
    [query_embedding],
    embeddings
)[0]

# Creazione di coppie (indice, similarità)
indexed_scores = list(enumerate(similarities))

# Ordinamento per similarità decrescente
sorted_results = sorted(
    indexed_scores,
    key=lambda x: x[1],
    reverse=True
)

# Estrazione dei risultati ordinati
top_results = [
    (dataset[idx], score)
    for idx, score in sorted_results[:5]
]
```

Visualizzazione dei Risultati

Per una migliore esperienza utente, è utile presentare i risultati in un formato leggibile, includendo il punteggio di similarità:

```
for i, (text, score) in enumerate(top_results):
    print(f"{i+1}. {text}")
    print(f"   Similarità: {score:.4f}")
    print()
```

Questo formato permette all'utente di vedere chiaramente la rilevanza di ciascun risultato e di valutare quanto il sistema abbia compreso la sua query.

Test con Frasi Simili e Diverse

Per valutare l'efficacia del nostro sistema di ricerca semantica, è fondamentale testarlo con frasi che presentano diversi gradi di similarità. Questo ci aiuta a capire se il modello sta catturando correttamente le relazioni semantiche e non solo le somiglianze lessicali.

Possiamo creare un insieme di test con coppie di frasi che sono semanticamente simili ma lessicalmente diverse, e altre che sono lessicalmente simili ma semanticamente diverse. Questo ci darà un'idea chiara delle capacità e dei limiti del nostro sistema.

Frasi Semanticamente Simili

- "Python è un linguaggio facile da imparare"
- "Imparare a programmare in Python è semplice"

Similarità attesa: Alta (>0.8)

Frasi Lessicalmente Simili ma Semanticamente Diverse

- "La rete neurale ha molti nodi"
- "La rete da pesca ha molti nodi"

Similarità attesa: Media (0.4-0.7)

Frasi Semanticamente Diverse

- "La cucina italiana è deliziosa"
- "La programmazione in Python è divertente"

Similarità attesa: Bassa (<0.3)

Creazione di un'Interfaccia CLI

Per rendere il nostro sistema di ricerca semantica più accessibile, possiamo creare un'interfaccia a riga di comando (CLI) che permetta agli utenti di effettuare ricerche senza dover modificare il codice. Questo è particolarmente utile per test rapidi e dimostrazioni.

L'interfaccia CLI dovrebbe permettere all'utente di inserire una query, specificare il numero di risultati desiderati e visualizzare chiaramente i risultati con i relativi punteggi di similarità. Possiamo utilizzare il modulo argparse di Python per gestire facilmente i parametri da riga di comando.

```
import argparse

def main():
    parser = argparse.ArgumentParser(description='Ricerca semantica di frasi')
    parser.add_argument('query', type=str, help='La frase da cercare')
    parser.add_argument('--top', type=int, default=3, help='Numero di risultati da mostrare')
    parser.add_argument('--dataset', type=str, default='frasi.txt', help='File del dataset')

    args = parser.parse_args()

    # Carica dataset ed embeddings
    dataset = load_dataset(args.dataset)
    embeddings = load_embeddings(args.dataset + '.embeddings')

    # Esegui la ricerca
    results = semantic_search(args.query, dataset, embeddings, args.top)

    # Mostra i risultati
    print(f"\nRisultati per: '{args.query}'\n")
    for i, (text, score) in enumerate(results):
        print(f"{i+1}. {text}")
        print(f"   Similarità: {score:.4f}")
        print()

if __name__ == '__main__':
    main()
```

Gestione dell'API Key e Parametri

La gestione sicura delle chiavi API e la configurazione dei parametri sono aspetti fondamentali per un'applicazione di ricerca semantica robusta. Le chiavi API non dovrebbero mai essere hardcoded nel codice sorgente per evitare esposizioni accidentali.

Un approccio migliore è utilizzare variabili d'ambiente o file di configurazione separati, possibilmente crittografati. Inoltre, è utile permettere all'utente di configurare parametri come il modello di embedding da utilizzare o le dimensioni del batch.

Gestione della Chiave API

Utilizza variabili d'ambiente per memorizzare la chiave API di OpenAI:

```
import os
from dotenv import load_dotenv

# Carica variabili da file .env
load_dotenv()

# Accedi alla chiave API
api_key = os.getenv("OPENAI_API_KEY")
openai.api_key = api_key
```

Configurazione Parametri

Crea un file di configurazione in formato JSON o YAML:

```
import json

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

# Accedi ai parametri
model = config.get('embedding_model', 'text-embedding-ada-002')
batch_size = config.get('batch_size', 100)
```

Salvataggio degli Embeddings su File

Il calcolo degli embeddings è un'operazione costosa in termini di tempo e risorse API. Per ottimizzare il nostro sistema, è fondamentale salvare gli embeddings già calcolati su file, in modo da poterli riutilizzare in sessioni future senza doverli ricalcolare.

Python offre diverse opzioni per la serializzazione di dati complessi come gli array NumPy. Il formato più comune è il pickle, ma per gli embeddings è spesso preferibile utilizzare formati più specializzati come NPY di NumPy, che è ottimizzato per array numerici di grandi dimensioni.

Salvataggio degli Embeddings

```
import numpy as np

# Salva gli embeddings in formato NPY
def save_embeddings(embeddings, filename):
    np.save(filename, embeddings)
    print(f"Embeddings salvati in {filename}")

# Esempio di utilizzo
embeddings = get_embeddings_batch(dataset)
save_embeddings(embeddings, "embeddings.npy")
```

Caricamento degli Embeddings

```
# Carica gli embeddings da file
def load_embeddings(filename):
    try:
        embeddings = np.load(filename)
        print(f"Embeddings caricati da {filename}")
        return embeddings
    except FileNotFoundError:
        print(f"File {filename} non trovato")
        return None

# Esempio di utilizzo
embeddings = load_embeddings("embeddings.npy")
```

Per dataset più grandi, è possibile implementare anche un sistema di cache incrementale che aggiorni gli embeddings solo per le frasi nuove o modificate, risparmiando ulteriormente sui costi API.

Utilizzo di Database Vettoriali (FAISS)

Per applicazioni di ricerca semantica su larga scala, i database vettoriali come FAISS (Facebook AI Similarity Search) offrono prestazioni notevolmente superiori rispetto all'approccio naive. FAISS è specializzato nell'indicizzazione e ricerca efficiente di vettori ad alta dimensionalità, come i nostri embeddings.

L'utilizzo di FAISS può ridurre i tempi di ricerca da ore a millisecondi su dataset di milioni di frasi, rendendo possibili applicazioni in tempo reale. La libreria implementa algoritmi avanzati come gli indici IVF (Inverted File) e PQ (Product Quantization) per ottimizzare sia la velocità che l'utilizzo della memoria.

1 **Installazione**

Installa FAISS con pip: **pip install faiss-cpu** (o faiss-gpu per versione CUDA)

2 **Creazione Indice**

Crea un indice per gli embeddings normalizzati e addestralo

```
import faiss

# Converti in float32 (richiesto da FAISS)
embeddings_float32 = embeddings.astype('float32')

# Crea un indice
dimension = embeddings_float32.shape[1]
index = faiss.IndexFlatIP(dimension)  # IP = Inner Product

# Aggiungi vettori all'indice
index.add(embeddings_float32)
```

3 **Ricerca**

Esegui ricerche veloci con k-nearest neighbors

```
# Converti query in float32
query_float32 = query_embedding.astype('float32').reshape(1, -1)

# Cerca i k elementi più simili
k = 5  # numero di risultati
distances, indices = index.search(query_float32, k)

# Accedi ai risultati
for i, idx in enumerate(indices[0]):
    print(f"{i+1}. {dataset[idx]}")
    print(f"   Similarità: {distances[0][i]}")
```

Per dataset estremamente grandi, FAISS offre anche la possibilità di costruire indici compositi che bilanciano precisione e velocità secondo le esigenze specifiche dell'applicazione.