A Comprehensive Study and Implementation of Record Linkage Using Advanced Techniques

Alessio Marinucci, 546778

1 Introduction to Record Linkage

1.1 Definition

Record linkage, also known as entity resolution, involves identifying and consolidating records that represent the same entity across different datasets. This process is crucial for achieving a unified and accurate view of data, which enhances data quality and supports reliable analysis. For instance, in the healthcare sector, linking patient records from various hospitals can provide a comprehensive medical history, thereby improving patient care and enabling more effective research.

1.2 Challenges

Several challenges are inherent in the record linkage process:

- **Data Heterogeneity**: Diverse data sources often differ in format, structure, and schema. For example, date formats may vary between "MM/DD/YYYY" and "DD-MM-YYYY".
- **Data Quality Issues**: Data can be incomplete, erroneous, or outdated. Typographical errors in names and addresses can further complicate the matching process.
- **Scalability**: Managing large datasets efficiently is a major concern, as the computational complexity of comparing each record with all others increases exponentially with data size.
- **Privacy Concerns**: Maintaining data privacy and security during the linkage process is essential, especially when handling sensitive information such as medical or financial records.

1.3 Recent Developments

Recent advancements in record linkage have seen a significant integration of machine learning techniques, particularly leveraging deep learning architectures and Large Language Models (LLMs) such as GPT (Generative Pre-trained Transformer) models. These sophisticated models have revolutionized the field by offering unprecedented capabilities in discerning intricate patterns and relationships within datasets.

A big improvement is the utilization of deep learning algorithms for feature extraction and representation learning. Traditional record linkage methods often relied on handcrafted features, which could be limited in capturing the complexities of real-world data. Deep learning models learn hierarchical representations of data autonomously and so capture the similarities beetween records, even in the presence of noise and variability. In addition the deployment of distributed computing frameworks and parallel processing techniques dramatically improved the scalability and efficiency of record linkage systems. Now large-scale datasets with millions of records can be processed in a fraction of the time compared to conventional methods; this facilitates rapid and accurate linkage across extensive databases. Another notable trend is the integration of domain-specific knowledge into machine learning models through techniques such as transfer learning and domain adaptation. By pre-training models on relevant datasets or fine-tuning them on specific record linkage tasks, researchers have achieved remarkable improvements in accuracy and robustness, particularly in domains with sparse or heterogeneous data.

Furthermore, progresses in data privacy and security have been essential to ensure the confidentially of sensitive information during the record linkage process. Techniques such as differential privacy and secure multi-party computation are increasingly being incorporated to mitigate the risks associated with data linkage, thereby fostering greater trust and compliance with regulatory standards.

Overall, the synergy between machine learning, distributed computing, and domain expertise has propelled record linkage capabilities to unprecedented heights, offering unparalleled accuracy, scalability, and efficiency in matching records across diverse datasets. These developments hold immense potential for various applications ranging from healthcare and finance to e-commerce and beyond, paving the way for a more interconnected and data-driven future.

2 Approaches to Record Linkage

2.1 Traditional Methods

Traditional approaches to record linkage are typically grouped into rule-based and probabilistic methods.

2.1.1 Rule-Based Approaches

Rule-based methods utilize predefined criteria to determine if two records match. Common techniques include:

- **String Matching**: Techniques such as Levenshtein distance (edit distance), Jaccard similarity (intersection over union of token sets), and Soundex (phonetic algorithm) are employed to compare textual attributes.

- **Domain-Specific Rules**: Custom rules based on domain knowledge. For instance, in healthcare, exact matches on Social Security Number (SSN) may be prioritized, while allowing for some variability in names and addresses.

```python
def jaccard_similarity(set1, set2):
    intersection = len(set1.intersection(set2))
    union = len(set1.union(set2))
    return intersection / union

def soundex(name):
    """
    Implementation of the Soundex algorithm.
    """
    # Convert the name to uppercase
    name = name.upper()

    # Soundex conversion table
    soundex_mapping = {
        'A': '', 'B': '1', 'C': '2', 'D': '3', 'E': '', 'F': '1', 'G': '2', 'H': '',
        'I': '', 'J': '2', 'K': '2', 'L': '4', 'M': '5', 'N': '5', 'O': '', 'P': '1',
        'Q': '2', 'R': '6', 'S': '2', 'T': '3', 'U': '', 'V': '1', 'W': '', 'X': '2',
        'Y': '', 'Z': '2'
    }

    # Keep the first letter
    soundex_code = name[0]

    # Convert the rest of the letters to Soundex digits
    for char in name[1:]:
        digit = soundex_mapping.get(char, '')
        if digit != '' and digit != soundex_code[-1]:
            soundex_code += digit

    # Remove the first letter if it has been encoded
    soundex_code = soundex_code[0] + soundex_code[1:].replace(soundex_code[0], '')

    # Pad with zeros or truncate to ensure a length of 4
    soundex_code = (soundex_code + '000')[:4]

    return soundex_code

# Rule-based matching function
def match_records(record1, record2):
    if jaccard_similarity(set(record1['name']), set(record2['name'])) > 0.8:
        if soundex(record1['surname']) == soundex(record2['surname']):
            return True
    return False
```

Listing 1: Sample Rule-Based Matching

The provided code defines functions for rule-based matching of records. The `jaccard_similarity` function calculates the Jaccard similarity between two sets by computing the ratio of the size of the intersection to the size of the union of the two sets. The `soundex` function implements the Soundex algorithm, which encodes names phonetically. The `match_records` function compares two records by first checking if the Jaccard similarity between the sets of characters in their `name` fields is greater than 0.8. If this condition is met, it then compares the Soundex encodings of their `surname` fields. If both conditions are satisfied, it returns `True`, indicating that the records match; otherwise, it returns `False`.

2.1.2 Probabilistic Approaches

Probabilistic methods, such as the Fellegi-Sunter model, estimate the probability that two records represent the same entity based on the similarity of their attributes, using statistical techniques to manage uncertainty and variability in the data.

```python
from scipy.stats import norm

def probabilistic_match(record1, record2, field_weights,
    threshold):
    score = 0
    for field, weight in field_weights.items():
        similarity = jaccard_similarity(set(record1[field]),
             set(record2[field]))
        score += weight * similarity

    return score > threshold
```

Listing 2: Sample Probabilistic Matching

The provided code defines a function for probabilistic matching of records. The `probabilistic_match` function calculates a similarity score between two records based on weighted similarities of their fields. For each field, the Jaccard similarity between the sets of characters in the field values of the two records is computed. This similarity is then multiplied by a predefined weight for that field and added to the total score. If the final score exceeds a specified threshold, the function returns `True`, indicating a match; otherwise, it returns `False`.

2.2 Machine Learning Approaches

Machine learning techniques are increasingly being employed in record linkage to automate the process of determining whether two records from different datasets refer to the same entity. These approaches leverage labeled datasets, where pairs of records are annotated as matches or non-matches, to train predictive models.

One common machine learning approach used in record linkage is supervised learning. In supervised learning, models are trained on labeled data, where each record pair is labeled as either a match or a non-match. These models learn patterns and relationships in the data, enabling them to predict the likelihood that a given pair of records refers to the same entity. Supervised learning algorithms such as logistic regression, random forests, and support vector machines have been successfully applied to record linkage tasks.

Another approach is semi-supervised learning, which combines labeled and unlabeled data for training. In record linkage, semi-supervised learning techniques can leverage a small set of labeled record pairs along with a larger set of unlabeled pairs. These models learn from both the labeled and unlabeled data to improve their predictive accuracy, making them particularly useful when labeled

data is difficult or expensive to obtain.
Moreover, recent advancements in deep learning and Large Language Models (LLMs) have enabled the development of more sophisticated record linkage models. These models can leverage the contextual information present in textual data to make more accurate linkage decisions, even in the presence of noise and variability.
Overall, machine learning approaches offer powerful tools for automating record linkage tasks, improving accuracy, scalability, and efficiency. By leveraging labeled data and advanced modeling techniques, researchers can develop highly effective record linkage systems capable of handling diverse datasets across various domains.

2.2.1 Supervised Learning

Supervised learning methods train classifiers on features derived from record pairs. Popular algorithms include:

- **Decision Trees**: Trees that split records based on attribute values, handling both numerical and categorical data, and providing interpretable matching rules.

- **Support Vector Machines (SVMs)**: Classifiers that find the optimal hyperplane to separate matching and non-matching pairs. Effective for high-dimensional data but requires careful hyperparameter tuning.

- **Neural Networks**: Models that learn complex data patterns through multiple layers of neurons, capturing non-linear relationships but requiring significant data and computational resources.

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

# Features extracted from record pairs
features = [
    # Similarity scores for attributes
]

# Labels (1 for match, 0 for non-match)
labels = [
    # Ground truth labels
]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features
    , labels, test_size=0.2)

# Train a decision tree classifier
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

```python

# Evaluate the classifier
accuracy = clf.score(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
```

Listing 3: Sample Decision Tree Classifier

The provided code defines a process for training and evaluating a Decision Tree classifier using the scikit-learn library. Features representing similarity scores for attributes are extracted from record pairs, and labels indicating matches (1) and non-matches (0) are provided. The data is split into training and testing sets using `train_test_split`. A `DecisionTreeClassifier` is trained on the training set and evaluated on the testing set, with the accuracy of the classifier being printed as the output.

2.2.2 Unsupervised Learning

Unsupervised learning techniques do not require labeled data, using clustering algorithms to group similar records:

- **K-Means Clustering**: Groups records into a predefined number of clusters based on similarity. Simple and efficient but requires specifying the number of clusters.
- **DBSCAN**: Density-based clustering that identifies clusters based on the density of data points, handling clusters of varying shapes and sizes, though it requires careful parameter tuning.

```python
from sklearn.cluster import DBSCAN
import numpy as np

# Data (feature vectors representing records)
data = np.array([
    # Feature vectors
])

# Cluster data using DBSCAN
db = DBSCAN(eps=0.5, min_samples=5).fit(data)

# Extract clusters
labels = db.labels_
```

Listing 4: Sample DBSCAN Clustering

The provided code demonstrates DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering using scikit-learn. The feature vectors representing records are stored in the ‘data‘ array. DBSCAN is applied to the data specifying the maximum distance between two samples for one to be considered as in the neighborhood of the other, and the number of samples in a neighborhood for a point to be considered as a core point, respectively. The resulting cluster labels are stored in the ‘labels‘ array.

2.3 Deep Learning Approaches

Deep learning methods have emerged as powerful tools in record linkage, leveraging neural network architectures to automatically learn complex features from data, thereby enhancing accuracy and scalability.

One prominent application of deep learning in record linkage is the use of convolutional neural networks (CNNs) and recurrent neural networks (RNNs). CNNs are particularly effective for processing structured data such as tabular records, where they can learn hierarchical representations of the data's features. By applying filters to input data, CNNs can capture local patterns and relationships, enabling them to make accurate linkage decisions even in the presence of noise. On the other hand, RNNs are well-suited for handling sequential data, making them suitable for record linkage tasks involving textual or temporal data. RNNs can learn dependencies and relationships between records over time, allowing them to effectively model the contextual information present in textual data. This makes them particularly useful for tasks such as entity resolution in natural language processing applications.

Moreover, the advent of Large Language Models (LLMs) such as GPT (Generative Pre-trained Transformer) has opened up new possibilities for record linkage. LLMs are pre-trained on large corpora of text data and can generate contextual representations of input data, enabling them to capture subtle similarities between records. By fine-tuning LLMs on record linkage tasks, researchers can leverage their powerful language understanding capabilities to achieve state-of-the-art performance in matching records across diverse datasets.

Furthermore, deep learning methods offer scalability advantages due to their ability to leverage parallel processing and distributed computing frameworks. This enables the processing of large-scale datasets with millions of records, making deep learning approaches well-suited for applications requiring high throughput and efficiency.

Overall, deep learning approaches can contribute a lot for advancing record linkage capabilities, offering superior accuracy, scalability, and efficiency compared to traditional methods. By leveraging neural network architectures and large-scale pre-trained models, researchers can develop highly effective record linkage systems capable of handling diverse datasets across various domains. .

2.3.1 Siamese Networks

Siamese networks consist of two identical subnetworks processing two input records to produce a similarity score, effectively capturing complex relationships between records. Each subnetwork typically includes several layers of convolutional or recurrent neural networks that learn hierarchical data representations.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Lambda
import tensorflow.keras.backend as K

# Define base network
```

```
def create_base_network(input_shape):
    input = Input(shape=input_shape)
    x = Dense(128, activation='relu')(input)
    x = Dense(128, activation='relu')(x)
    x = Dense(128, activation='relu')(x)
    return Model(input, x)

# Input layers for the Siamese network
input_a = Input(shape=(input_shape,))
input_b = Input(shape=(input_shape,))

# Create identical subnetworks
base_network = create_base_network(input_shape)
processed_a = base_network(input_a)
processed_b = base_network(input_b)

# Compute L1 distance between subnetworks' outputs
distance = Lambda(lambda tensors: K.abs(tensors[0] - tensors[1]))([processed_a, processed_b])

# Output layer
output = Dense(1, activation='sigmoid')(distance)

# Define Siamese network model
model = Model([input_a, input_b], output)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit([data_a, data_b], labels, epochs=10, batch_size=128)
```

Listing 5: Sample Siamese Network

The provided code implements a Siamese network using TensorFlow and Keras. The network consists of two identical branches, each taking an input (input_a and input_b) and producing an output after passing through a set of dense layers. The L1 distance between the outputs of the two branches is computed using a Lambda function. Subsequently, the distance is passed through a single dense layer with sigmoid activation, which returns a value between 0 and 1. This value represents the probability that the provided inputs are similar. The model is then compiled using the Adam optimizer and binary crossentropy loss function. Finally, the model is trained on the input data (data_a and data_b) with their respective labels (labels) for a specified number of epochs.

2.3.2 Transformer Models

Transformers, like BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), capture semantic similarities between records by understanding context and language. These models are pre-trained on large text corpora and fine-tuned for specific tasks such as record linkage.

```python
from transformers import BertTokenizer, BertModel
import torch

# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Tokenize and encode input records
records = ["Record 1 text", "Record 2 text"]
inputs = tokenizer(records, return_tensors='pt', padding=True, truncation=True, max_length=128)

# Extract BERT embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state[:, 0, :].numpy()
```

Listing 6: Using BERT for Embeddings

The provided code demonstrates how to use BERT (Bidirectional Encoder Representations from Transformers) for generating embeddings of input records. Firstly, it loads a pre-trained BERT model and tokenizer from the ‘transformers‘ library. Then, it tokenizes and encodes the input records using the tokenizer, ensuring padding and truncation to a maximum length of 128 tokens. Finally, it extracts the BERT embeddings from the last hidden state of the model's output, taking only the first token (CLS token) embedding for each record, and converts them to NumPy arrays.

3 Proposed Approach: Enhanced Siamese Network with BERT Embeddings

3.1 Overview

My proposed approach aims to enhance the record linkage process using an advanced Siamese network architecture integrated with BERT embeddings. Record linkage is a critical practice for integrating and correlating information from different data sources, but it often faces challenges related to data variety and complexity. My proposal focuses on leveraging BERT embeddings, derived from the Bidirectional Encoder Representations from Transformers (BERT) model, to capture semantic and contextual information from records, enabling a better understanding of semantic similarities between them.

The Siamese network is a neural structure that has proven effective in calculating the similarity between pairs of inputs. My proposed architecture has adopted an enhanced Siamese Network that processes the BERT embeddings of input records. These embeddings provide a dense and contextual representation of the data, allowing the network to capture complex semantic and contextual information. This approach helps overcome limitations of traditional record linkage methods, which often rely on shallow representations of data.

Utilizing BERT embeddings offers several advantages. Firstly, they capture semantic relationships between words in the records, enabling the network to grasp subtle and complex meanings. This is particularly useful for addressing variations in data, such as spelling errors, abbreviations, and synonyms, which can hinder accurate record matching. Additionally, BERT embeddings are pre-trained on large amounts of text, meaning the network benefits from linguistic and semantic knowledge already acquired during the model's pre-training.

This approach will allow us to fully leverage the deep learning capabilities of BERT embeddings, significantly improving the accuracy and effectiveness of the record linkage process.

3.2 Architecture

The architecture of my proposed solution includes the following components:

3.2.1 Input Layer

The input layer accepts pairs of records to be compared, tokenized using BERT's tokenizer.

3.2.2 Embedding Layer

BERT embeddings transform the input records into dense vectors encapsulating semantic context. This is achieved using a pre-trained BERT model fine-tuned on a domain-specific dataset.

3.2.3 Siamese Network

Identical subnetworks process the BERT embeddings of each record. Each subnetwork includes several dense layers to learn relevant features from the embeddings.

3.2.4 Similarity Layer

A similarity score is computed using a distance metric, such as cosine similarity or L1 distance, between the feature representations of the two records.

3.2.5 Output Layer

The output layer uses a sigmoid activation function to produce a binary classification output, indicating whether the records match.

```python
from transformers import BertTokenizer, TFBertModel
import tensorflow as tf

# Load BERT tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bert_model = TFBertModel.from_pretrained('bert-base-uncased')

# Define base network
def create_base_network(input_shape):
    input = tf.keras.layers.Input(shape=input_shape)
    x = tf.keras.layers.Dense(128, activation='relu')(input)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    x = tf.keras.layers.Dense(128, activation='relu')(x)
    return tf.keras.Model(input, x)

# Define Siamese network
input_a = tf.keras.layers.Input(shape=(768,))
input_b = tf.keras.layers.Input(shape=(768,))

base_network = create_base_network((768,))
processed_a = base_network(input_a)
processed_b = base_network(input_b)

# Compute L1 distance
distance = tf.keras.layers.Lambda(lambda tensors: tf.abs(tensors[0] - tensors[1]))([processed_a, processed_b])

# Output layer
output = tf.keras.layers.Dense(1, activation='sigmoid')(distance)

# Define Siamese network model
```

```python
model = tf.keras.Model([input_a, input_b], output)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Example input records
records = ["Record 1 text", "Record 2 text"]
inputs = tokenizer(records, return_tensors='tf', padding=True, truncation=True, max_length=128)

# Extract BERT embeddings
outputs = bert_model(inputs)
embeddings = outputs.last_hidden_state[:, 0, :].numpy()

# Train the model (example with dummy data)
data_a = embeddings[:1]
data_b = embeddings[1:]
labels = tf.constant([1])  # Example label (1 for match, 0 for non-match)

model.fit([data_a, data_b], labels, epochs=10, batch_size=1)
```

Listing 7: Enhanced Siamese Network Implementation

The provided code implements an enhanced Siamese network using BERT embeddings for input records. Firstly, it loads a pre-trained BERT tokenizer and model using the Hugging Face Transformers library. Then, it defines a base network and constructs the Siamese network architecture with the base network sharing weights for both input branches. L1 distance between the processed inputs is computed, followed by a sigmoid activation layer to produce a similarity score. The model is compiled with the Adam optimizer and binary crossentropy loss. Example input records are tokenized and encoded using the BERT tokenizer, and BERT embeddings are extracted from the last hidden state. Finally, the model is trained with dummy data, where the embeddings of two records are used as input pairs, and a label indicating a match (1) or non-match (0) is provided.

4 Improvements and Implementation

4.1 Potential Enhancements

To further enhance my proposed approach, are considered several improvements:

4.1.1 Data Augmentation

Generate synthetic data by introducing variations and noise to existing records, improving the model's ability to generalize.

4.1.2 Fine-Tuning BERT

Fine-tune the BERT model on domain-specific data to enhance the quality of embeddings, improving the model's performance in record linkage tasks.

4.1.3 Ensemble Methods

Combine the Siamese network with other approaches, such as rule-based or probabilistic methods, to enhance robustness and accuracy. An ensemble approach leverages the strengths of different models, providing more reliable results.

4.2 Implementation Steps

The implementation involves several key steps:

4.2.1 Data Preprocessing

Preprocessing is essential to prepare data for training. This includes tokenization, normalization, and handling missing values.

```python
from transformers import BertTokenizer

# Load BERT tokenizer
tokenizer = BertTokenizer.from_pretrained(’bert-base-uncased’)

# Example records
records = ["John Doe, 123 Main St, Anytown, USA", "Jon Doe, 123 Main Street, Anytown, US"]

# Tokenize records
tokenized_records = tokenizer(records, return_tensors=’pt’, padding=True, truncation=True, max_length=128)

# Normalize records
normalized_records = [record.lower() for record in records]
```

Listing 8: Data Preprocessing Example

4.2.2 Model Training

The training process includes data splitting, training the network, and hyperparameter tuning to optimize performance.

```
from sklearn.model_selection import train_test_split

# Sample features from record pairs
features = [
    # Similarity scores
]

# Labels (1 for match, 0 for non-match)
labels = [
    # Ground truth labels
]

# Split data
X_train, X_val, y_train, y_val = train_test_split(features, labels, test_size=0.2, random_state=42)

# Train Siamese network
history = model.fit([X_train[:, :768], X_train[:, 768:]], y_train, validation_data=([X_val[:, :768], X_val[:, 768:]], y_val), epochs=10, batch_size=32)

# Evaluate model
val_accuracy = model.evaluate([X_val[:, :768], X_val[:, 768:]], y_val)[1]
print(f'Validation Accuracy: {val_accuracy:.2f}')
```

Listing 9: Model Training Example

4.2.3 Evaluation

Evaluate the model using metrics such as precision, recall, F1-score, and accuracy. Visualization tools like ROC and precision-recall curves aid in understanding performance.

```
from sklearn.metrics import confusion_matrix, roc_curve, auc, precision_recall_curve
import matplotlib.pyplot as plt

# Predictions on validation set
y_pred = model.predict([X_val[:, :768], X_val[:, 768:]])

# Binarize predictions
y_pred_bin = (y_pred > 0.5).astype(int)

# Confusion matrix
```

```python
conf_matrix = confusion_matrix(y_val, y_pred_bin)
print('Confusion Matrix:')
print(conf_matrix)

# ROC curve
fpr, tpr, _ = roc_curve(y_val, y_pred)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# Precision-Recall curve
precision, recall, _ = precision_recall_curve(y_val, y_pred)
plt.figure()
plt.plot(recall, precision, color='blue', lw=2)
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.show()
```

Listing 10: Model Evaluation Example

4.2.4 Deployment

Deploy the model as a web service using frameworks like Flask or FastAPI to handle real-time record linkage requests.

```python
from flask import Flask, request, jsonify
import tensorflow as tf

# Initialize Flask app
app = Flask(__name__)

# Load trained model
model = tf.keras.models.load_model('siamese_model.h5')

@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    record1 = data['record1']
    record2 = data['record2']

    # Tokenize and encode records
    inputs = tokenizer([record1, record2], return_tensors=’tf’, padding=True, truncation=True, max_length=128)

    # Extract BERT embeddings
    outputs = bert_model(inputs)
    embeddings = outputs.last_hidden_state[:, 0, :].numpy()

    # Predict match
    prediction = model.predict([embeddings[0].reshape(1, -1), embeddings[1].reshape(1, -1)])
    match = prediction[0][0] > 0.5

    return jsonify({’match’: match})

if __name__ == ’__main__’:
    app.run(debug=True)
```

Listing 11: Flask Deployment Example

5 Conclusion

Record linkage is a pivotal task in data integration, critical for constructing a comprehensive and accurate view of entities across disparate datasets. While traditional methods offer foundational solutions, modern approaches leveraging machine learning and deep learning, particularly BERT embeddings and Siamese networks, provide enhanced accuracy and scalability. My proposed approach demonstrates the potential of integrating advanced techniques to address the complexities of record linkage, ensuring robust and efficient entity resolution.

6 Link to GitHub Repository

For more details and the complete implementation, please visit my GitHub Repository.

References

[1] Christen P. (2020). "Deep Learning for Record Linkage: A Review." *Data Mining and Knowledge Discovery.*

[2] Xie S, Liu Z, Han Y, Sun G, Li X. (2021). "BERT-Link: A BERT-based Approach for Unsupervised Record Linkage." *arXiv preprint arXiv:2101.02457.*

[3] Shrivastava A, Neubig G. (2021). "Deep Record Linkage." *arXiv preprint arXiv:2107.03259.*

[4] Dong X, Tang X, Tan A, Chang KC-C, Yu PS. (2022). "Learning to Clean, Match, and Integrate Noisy Data with Deep Learning." *Proceedings of the VLDB Endowment.*

[5] Hsieh P-H, Ratner A, Sahami M, Ré C. (2021). "Improving Record Linkage Through Active Learning and Human-in-the-loop." *arXiv preprint arXiv:2109.09442.*