Certainly! I'll provide more detailed information on the reasons for permutation and the choice of dimensionality.

Permutation:

Why it's crucial:

1. Preserving Order Information:

- Without permutation, the order of symbols in an n-gram would be lost. For example, "ab" and "ba" would produce the same vector.

- Example:

Without permutation:

"ab" = v\_a \* v\_b = [ 1, -1, -1, -1, -1, 1, -1, 1, -1, -1]

"ba" = v\_b \* v\_a = [ 1, -1, -1, -1, -1, 1, -1, 1, -1, -1]

(identical results)

With permutation:

"ab" = v\_a \* permute(v\_b, 1) = [ 1, 1, 1, 1, 1, -1, -1, -1, -1, 1]

"ba" = v\_b \* permute(v\_a, 1) = [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

(different results)

2. Maintaining Vector Properties:

- Permutation preserves the statistical properties of the original vector, such as the number of 1s and -1s.

- This ensures that the permuted vectors remain in the same high-dimensional space and maintain their quasi-orthogonality.

3. Reversibility:

- The permutation operation is reversible, which allows for potential decoding or analysis of the encoded information if needed.

4. Capacity Increase:

- Permutation effectively increases the representational capacity of the system by allowing the same base vectors to represent different information based on their position.

Dimensionality (d):

Why different numbers of dimensions might be used:

1. Representational Capacity:

- Higher dimensionality increases the system's capacity to represent unique patterns.

- Example: With d=10, you have 2^10 = 1,024 possible binary vectors.

With d=1000, you have 2^1000 ≈ 10^301 possible binary vectors.

2. Noise Robustness:

- Higher dimensions provide more robustness against noise and errors.

- In high-dimensional spaces, random vectors are likely to be quasi-orthogonal, which helps in distinguishing between different encoded entities.

3. Trade-off with Computational Cost:

- Higher dimensions require more storage and computational resources.

- Example: For an alphabet of 26 letters:

With d=1000, item memory size: 26 \* 1000 = 26,000 bits

With d=10000, item memory size: 26 \* 10000 = 260,000 bits

4. Task Specificity:

- The optimal number of dimensions can depend on the specific task and dataset.

- For simpler tasks or smaller datasets, lower dimensions might suffice.

- More complex tasks or larger datasets might benefit from higher dimensions.

5. Similarity Preservation:

- Higher dimensions allow for better preservation of similarity relationships between encoded entities.

- Example: With d=10, two random vectors have about 50% chance of being orthogonal.

With d=10000, the dot product of two random vectors is likely to be very close to zero, allowing for finer distinctions between "somewhat similar" and "very similar" items.

6. Binding Capacity:

- Higher dimensions allow for more items to be bound together while maintaining the ability to recover individual components.

- This is particularly important for tasks that involve complex relationships or hierarchical structures.

In practice, dimensions in the thousands (e.g., 5000, 10000) are common in HDC applications, balancing representational power with computational feasibility. The exact choice often involves empirical tuning based on the specific application and available resources.

################################################################################

################################################################################

Of course! Here is the detailed explanation for each part of the HDCEncoder class and its methods:

### Detailed Explanation of the HDCEncoder Class

#### 1. Initialization of the HDCEncoder Class

python

Copy code

def \_\_init\_\_(self, d, alphabet\_size, n):

self.d = d # Dimensionality of the HD vectors

self.alphabet\_size = alphabet\_size # Size of the alphabet (27 for a-z and space)

self.n = n # Size of n-grams

# Initialize item memory with random ±1 vectors for each symbol

self.item\_memory = np.random.choice([-1, 1], size=(alphabet\_size, d))

# Create a mapping from characters to indices, including space

self.char\_to\_index = {chr(i): i - 97 for i in range(97, 123)} # 'a'-'z' -> 0-25

self.char\_to\_index[' '] = 26 # Space -> 26

* self.d: Specifies the dimensionality of the hyperdimensional (HD) vectors. High dimensionality (e.g., 1000) provides better representation and robustness.
* self.alphabet\_size: Size of the alphabet used for encoding, which includes 26 letters (a-z) plus space, making a total of 27 symbols.
* self.n: Size of the n-grams (substrings of length n) used to encode the text.
* self.item\_memory: A matrix of size (alphabet\_size, d) where each row is a randomly generated HD vector representing a symbol. This matrix is initialized with random ±1 values to ensure each symbol has a unique and high-dimensional representation.
* self.char\_to\_index: A dictionary mapping each character ('a' to 'z' and space) to its corresponding index (0-26). This mapping allows easy conversion of characters to their indices in item\_memory.

#### 2. Permutation Function

python

Copy code

def permute(self, vector, shift):

"""

Permute (rotate) the vector by the specified number of positions.

"""

return np.roll(vector, shift)

* Purpose: Permutes (or rotates) the elements of a vector by a specified number of positions. This permutation ensures that the positional information of each symbol in the n-gram is encoded into the HD vector.
* np.roll: This NumPy function efficiently performs the permutation. For example, permuting [1, 2, 3, 4] by 1 position results in [4, 1, 2, 3].

#### 3. Mapping n-gram to Hyperdimensional Vector

python

Copy code

def map\_ngram(self, ngram):

"""

Map an n-gram to a hyperdimensional vector using item memory and permutation.

"""

hd\_vector = np.ones(self.d) # Start with a vector of ones

for i, symbol in enumerate(ngram):

symbol\_vector = self.item\_memory[symbol] # Get the HD vector for the symbol

permuted\_vector = self.permute(symbol\_vector, i) # Permute based on position

hd\_vector \*= permuted\_vector # Combine via element-wise multiplication

return hd\_vector

* Purpose: Converts an n-gram into a single HD vector by combining the permuted vectors of its symbols.
* hd\_vector Initialization: Starts with a vector of ones.
* Permutation: Each symbol in the n-gram is permuted by its position index to encode positional information.
* Element-wise Multiplication: Combines the permuted vectors, ensuring that the final HD vector uniquely represents the n-gram.

#### 4. Encoding Text into HD Vector

python

Copy code

def encode(self, text):

"""

Encode text into a hyperdimensional vector by mapping n-grams.

"""

# Generate n-grams from the text

ngrams = [text[i:i+self.n] for i in range(len(text)-self.n+1)]

hd\_vector = np.zeros(self.d) # Initialize the HD vector

for ngram in ngrams:

# Map characters to indices (0-26) using a for loop

ngram\_indices = []

for c in ngram:

if c in self.char\_to\_index:

# 'ord(c) - 97' converts the character 'a' to 0, 'b' to 1, ..., 'z' to 25

# ' ' is mapped to 26

ngram\_indices.append(self.char\_to\_index[c])

if len(ngram\_indices) == self.n:

hd\_vector += self.map\_ngram(ngram\_indices) # Add the mapped n-gram

return np.sign(hd\_vector) # Binarize to ±1

#### Step-by-Step Breakdown of encode

1. Generate n-grams:

python

Copy code

ngrams = [text[i:i+self.n] for i in range(len(text)-self.n+1)]

* + This creates all possible n-grams of size n from the text. For example, for text "abcdef" and n = 3, this generates ['abc', 'bcd', 'cde', 'def'].

1. Initialize HD vector:

python

Copy code

hd\_vector = np.zeros(self.d)

* + Start with a zero vector to accumulate the contributions from each n-gram.

1. Map characters to indices:

python

Copy code

ngram\_indices = []

for c in ngram:

if c in self.char\_to\_index:

# 'ord(c) - 97' converts the character 'a' to 0, 'b' to 1, ..., 'z' to 25

# ' ' is mapped to 26

ngram\_indices.append(self.char\_to\_index[c])

* + Convert each character in the n-gram to its corresponding index using char\_to\_index. The ord(c) - 97 converts 'a' to 0, 'b' to 1, and so on, up to 'z' which converts to 25. The space character is mapped to 26.

1. Map n-gram to HD vector and accumulate:

python

Copy code

if len(ngram\_indices) == self.n:

hd\_vector += self.map\_ngram(ngram\_indices)

* + If the n-gram is valid (i.e., has the expected length), convert it to an HD vector using map\_ngram and add it to the overall hd\_vector.

1. Binarize HD vector:

python

Copy code

return np.sign(hd\_vector)

* + Convert the final HD vector to ±1 to maintain the high-dimensional properties. Binarization ensures the resultant vector is sparse and balanced, which is important for preserving the mathematical properties of high-dimensional spaces.

### Why These Steps Are Necessary

* Mapping Characters: Each character needs to be mapped to an index to fetch its corresponding HD vector from item\_memory. This ensures each symbol is uniquely represented.
* Permutation: Permuting the vectors based on their position in the n-gram ensures that the order of characters is encoded into the HD vector. This makes "abc" different from "bca".
* Element-wise Multiplication: Combines the contributions of each character's permuted vector, resulting in a unique representation of the n-gram. This aggregation maintains positional distinctions and encodes the n-gram uniquely.
* Binarization: Ensures the final HD vector has elements ±1, preserving sparsity and balance, which are crucial for high-dimensional representations.

This detailed explanation covers the entire process and rationale behind the HDCEncoder class, its methods, and how the text is encoded into hyperdimensional vectors.

################################################################################

################################################################################

Certainly! Let's break down this code and explain its components in detail:

1. Data Splitting:

```python

train\_data, test\_data = train\_test\_split(data, test\_size=0.3, random\_state=42, stratify=[label for \_, label in data])

```

- Why: Splitting the data allows us to evaluate the model's performance on unseen data.

- The 70/30 split is a common choice, balancing between having enough training data and a representative test set.

- Stratification ensures that the proportion of samples for each class is roughly the same in both sets.

2. Feature and Label Separation:

```python

X\_train, y\_train = zip(\*train\_data)

X\_test, y\_test = zip(\*test\_data)

```

- This separates the text samples (X) from their corresponding language labels (y).

3. Label Encoding:

```python

label\_encoder = LabelEncoder()

y\_train\_encoded = label\_encoder.fit\_transform(y\_train)

y\_test\_encoded = label\_encoder.transform(y\_test)

```

- Why encoding: Machine learning models typically require numerical input. Label encoding converts string labels to integers.

4. HD Vector Dimensions:

```python

d\_values = [100, 1000, 10000]

```

- Testing different dimensions allows us to find the optimal balance between performance and computational cost.

5. HDC Encoder Initialization:

```python

encoder = HDCEncoder(d=d, alphabet\_size=27, n=3)

```

- Why 27: This accounts for 26 lowercase letters plus a space character.

- n=3: Using trigrams (3-character sequences) which often capture language-specific patterns well.

6. Text Encoding:

```python

X\_train\_encoded = np.array([encoder.encode(text) for text in tqdm(X\_train, desc="Encoding train data")])

```

- This step converts each text sample into its HDC vector representation.

7. Ridge Classifier:

```python

clf = RidgeClassifier()

clf.fit(X\_train\_encoded, y\_train\_encoded)

```

- Why Ridge:

1. It's effective for high-dimensional data (which HDC produces).

2. It includes L2 regularization, helping to prevent overfitting.

3. It's computationally efficient compared to some other classifiers.

4. It performs well on many text classification tasks.

8. Prediction and Evaluation:

```python

y\_pred = clf.predict(X\_test\_encoded)

accuracy = accuracy\_score(y\_test\_encoded, y\_pred)

f1 = f1\_score(y\_test\_encoded, y\_pred, average='weighted')

```

- Accuracy gives an overall correctness measure.

- F1-score provides a balance between precision and recall, which is useful for multi-class classification.

9. Confusion Matrix:

```python

cm = confusion\_matrix(y\_test\_encoded, y\_pred)

cm\_df = pd.DataFrame(cm, index=label\_encoder.classes\_, columns=label\_encoder.classes\_)

```

- The confusion matrix helps visualize which languages are being confused with each other.

This code is designed to test the effectiveness of HDC encoding for language identification. By trying different dimensionalities (d\_values), it allows for finding an optimal balance between computational cost and performance. The use of Ridge classification on the HDC-encoded text leverages the high-dimensional nature of the encoding while providing regularization to prevent overfitting. The evaluation metrics (accuracy, F1-score, and confusion matrix) provide a comprehensive view of the model's performance across different languages.