

Web Mining & Web Search

Wikipedia Article Suggestion & Recommendation

Authored by:

Hesam Koucheqi

Erfan Eftekhari

Supervisor:

Dr. Khanjari

Fall 2024 | Tehran, Iran

Iran University of Science and Technology

Contents

Introduction	2
Dataset	3
Loading Data	4
Preprocessing.....	5
Batch Processing	6
Garbage Collection.....	8
Transformer	9
Embedding.....	10
Variance	12
Recommend Articles Logic	13
Testing Recommender System	14
Results.....	15

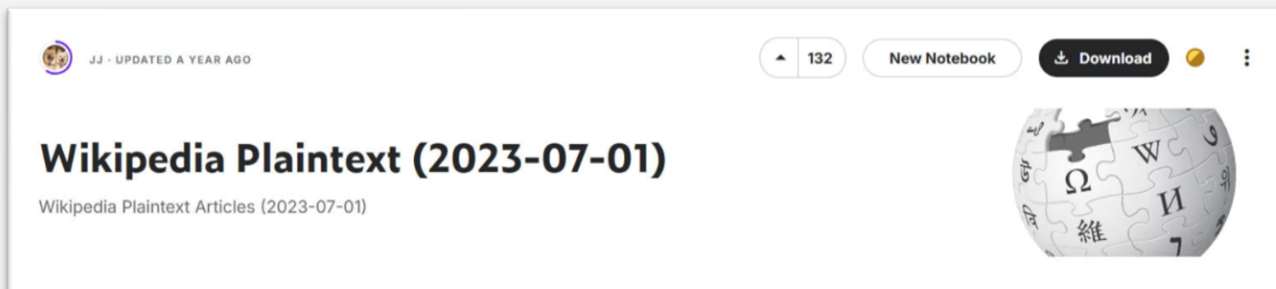
Introduction

In this project, we built a recommender system that helps users find relevant articles based on a given input article. The model was trained on a dataset and works by analyzing the input article, understanding its key points, and finding similar articles from the dataset. It then ranks these suggestions, making sure that the most relevant ones appear at the top.

To improve accuracy, the system doesn't just look for exact word matches but also tries to understand the overall meaning of the text. This helps it find articles that are not only similar in words but also in context and category. We also made sure that the recommendations are diverse, so users don't see repetitive suggestions.

In the following sections of this report, we'll explain how each part of the system works, including how we prepared the data, trained the model, measured similarity, and evaluated the results. We'll also talk about some of the challenges we faced and how we solved them to make the system better.

Dataset



We have used the *Wikipedia Plaintext dataset*¹ for this project. While there are other high-quality Wikipedia datasets available, most of them are outdated, with the latest ones dating back to 2020. This dataset, however, is an updated version based on the Wikipedia dump from July 1st, 2023. It contains 6,286,775 articles along with their titles, full text, and categories, with a total size of 15.1 GB. The dataset is structured as follows:

- **id**: A unique identifier for each article
- **text**: The full content of the article
- **title**: The article's title
- **category**: The category or categories assigned to the article


To make data access and processing more efficient, the articles are sorted in alphanumeric order and stored in **parquet** files. The dataset is divided into different partitions based on the first character of the article title:

- **a-z**: Articles whose titles start with letters
- **number**: Articles with titles that begin with numbers
- **other**: Articles with titles that start with symbols

This structured format makes it easier to search, retrieve, and process the data efficiently.

¹ <https://www.kaggle.com/datasets/jjinho/wikipedia-20230701>

Loading Data



```
# File paths for your Parquet files
file_paths = [
    '/kaggle/input/wikipedia-20230701/a.parquet',
    '/kaggle/input/wikipedia-20230701/b.parquet',
    '/kaggle/input/wikipedia-20230701/c.parquet'
]

# Read each file and store them in a list of DataFrames
dfs = [pd.read_parquet(file) for file in file_paths]

# Concatenate the DataFrames into one
df = pd.concat(dfs, ignore_index=True)

# Display the first few rows
print(df.head())


# Check the columns
print(df.columns)
```

This code snippet is responsible for loading and merging Wikipedia articles from multiple **Parquet** files into a single **DataFrame** for further processing. It begins by defining a list of file paths, each corresponding to a partitioned dataset containing Wikipedia articles starting with specific letters (e.g., "a", "b", "c"). The code then reads each file into a **pandas DataFrame** using `pd.read_parquet()` and stores them in a list. After loading the data, it concatenates all the DataFrames into a single unified dataset using `pd.concat()`, ensuring that the index is reset with `ignore_index=True`.

Once the merged dataset is ready, the script displays the first few rows using `df.head()` to provide a preview of the data. Additionally, `df.columns` is printed to check the structure and available fields. This process ensures that the dataset is

properly loaded and formatted before further analysis, such as extracting article text, filtering by category, or running similarity computations.

Preprocessing



```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('stopwords')
# Get the list of stop words
stop_words = set(stopwords.words('english'))

# Combine the fields into one text column
def combine_fields(row):
    categories = ', '.join(row['categories']) # Convert list to string

    # Tokenize the text
    words = word_tokenize(row["text"])

    # Remove stop words
    filtered_text = ' '.join(list(filter(lambda word: word.lower() not in
stop_words, words)))

    return f"Title: {row['title']}. Categories: {categories}. Text:
{filtered_text}"
    # return f"Title: {row['title']} Category: {row['categories']}"
```

This code snippet processes Wikipedia articles by combining relevant fields into a single text format while filtering out common stop words. It begins by importing necessary **NLTK** libraries and downloading the predefined set of **stopwords** for English. These stop words are stored in a set for efficient lookup.

The function `combine_fields(row)` takes a **row** from a DataFrame as input and performs the following steps:

1. Converts the list of **categories** into a comma-separated string.
2. Tokenizes the **text** field using `word_tokenize()`, breaking it into individual words.

3. Filters out **stop words** (such as "the", "is", "and") to reduce noise and improve meaningful content extraction.
4. Constructs a formatted string including the **title**, **categories**, and the **processed text**, ensuring it remains concise and structured.

This function is particularly useful for preparing text data for **natural language processing (NLP)** tasks such as text similarity, keyword extraction, and machine learning-based article recommendations.

Batch Processing

```
# Batch processing function
def batch_process(df, batch_size=100):
    combined_results = []

    # Calculate the number of batches
    num_batches = (len(df) // batch_size) + (1 if len(df) % batch_size != 0
    else 0)

    # Process each batch with tqdm for progress tracking
    for batch_num in tqdm(range(num_batches), desc="Processing Batches"):
        start_idx = batch_num * batch_size
        end_idx = min((batch_num + 1) * batch_size, len(df))

        # Get the current batch
        batch = df.iloc[start_idx:end_idx]

        # Apply the function to the batch
        batch_results = batch.apply(combine_fields, axis=1)

        # Append the results
        combined_results.extend(batch_results)

    return combined_results
```

This function, `batch_process(df, batch_size=100)`, is designed to efficiently process large datasets in manageable chunks, improving memory usage and performance.

It takes a **DataFrame** as input and processes it in batches of a specified size (default: **100 rows per batch**).

Key Steps:

1. **Initialize storage:** An empty list, `combined_results`, is created to store processed text data.
2. **Batch calculation:** The total number of batches is determined by dividing the DataFrame length by the batch size. If there are remaining rows, an extra batch is added.
3. **Batch iteration with progress tracking:**
 - The function loops through the dataset in chunks using `tqdm`, which provides a progress bar for better tracking.
 - It extracts the relevant batch of rows using `iloc[start_idx:end_idx]`.
 - It applies `combine_fields()` to transform the data in the batch.
 - The results are appended to `combined_results`, ensuring all processed text is stored efficiently.

Purpose:

This function is particularly useful for handling **large Wikipedia datasets**, as it prevents memory overload by processing smaller chunks instead of the entire DataFrame at once. It is ideal for tasks like **text preprocessing**, **feature extraction**, or **preparing inputs for machine learning models**.

The running of this code on 3 files (a, b, and c) takes about 45 minutes.

```
modules.json: 100% ██████████ 349/349 [00:00<00:00, 43.7kB/s]
config_sentence_transformers.json: 100% ██████████ 116/116 [00:00<00:00, 14.8kB/s]
README.md: 100% ██████████ 10.7k/10.7k [00:00<00:00, 1.23MB/s]
sentence_bert_config.json: 100% ██████████ 53.0/53.0 [00:00<00:00, 6.18kB/s]
config.json: 100% ██████████ 612/612 [00:00<00:00, 71.6kB/s]
model.safetensors: 100% ██████████ 90.9M/90.9M [00:00<00:00, 222MB/s]
tokenizer_config.json: 100% ██████████ 350/350 [00:00<00:00, 41.4kB/s]
vocab.txt: 100% ██████████ 232k/232k [00:00<00:00, 1.17MB/s]
tokenizer.json: 100% ██████████ 466k/466k [00:00<00:00, 2.38MB/s]
special_tokens_map.json: 100% ██████████ 112/112 [00:00<00:00, 14.1kB/s]
1_Pooling/config.json: 100% ██████████ 190/190 [00:00<00:00, 21.6kB/s]
Generating Embeddings: 100% ██████████ 2313/2313 [43:46<00:00, 1.14s/it]
```


Garbage Collection



```
import gc

# Drop unnecessary columns
df = df.drop(columns=['title', 'text', 'categories'])

# Force garbage collection
gc.collect()

# Optionally reset index to clean up any lingering references
df = df.reset_index(drop=True)

# Check DataFrame memory usage
print(df.info())
```

This code optimizes memory by removing unnecessary columns (title, text, categories), forcing garbage collection (`gc.collect()`), and resetting the index to clean up references. Finally, `df.info()` checks memory usage, ensuring efficient handling of large datasets.

Transformer

```
from sentence_transformers import SentenceTransformer
from sklearn.preprocessing import normalize
from tqdm import tqdm
tqdm.pandas(disable=True)
# Load the SentenceTransformer model
model = SentenceTransformer('all-MiniLM-L6-v2') # Lightweight & effective

# Convert the combined text column to a list
text_list = df['combined'].tolist()

# Initialize a list to store embeddings
embeddings = []

# Define batch size
batch_size = 512
tqdm.pandas(disable=True)
```

This code loads the **SentenceTransformer** model (all-MiniLM-L6-v2) to generate embeddings for the processed text. It converts the **combined** text column into a list and prepares for batch-wise embedding computation. The batch size is set to **512** for efficient processing. **TQDM** is used for progress tracking but is disabled for a cleaner output.

Embedding

```
for i in tqdm(range(0, len(text_list), batch_size), desc="Generating Embeddings"):
    # Process a batch of text
    batch = text_list[i:i + batch_size]

    # Generate embeddings for the current batch
    batch_embeddings = model.encode(batch, convert_to_numpy=True,
show_progress_bar=False)

    # Append the batch embeddings to the main list
    embeddings.extend(batch_embeddings)

# Normalize embeddings for better similarity scores
embedding_matrix = normalize(embeddings)

# Add normalized embeddings to the DataFrame
df['embeddings'] = list(embedding_matrix) # Convert 2D array to a list of 1D
arrays for Pandas compatibility

# Print the first few rows and verify an embedding's shape
print(df.head())
print(df['embeddings'].iloc[0].shape) # Should print (384,) for each
embedding
```

This code generates and stores **normalized embeddings** for the text data using **SentenceTransformer** in an efficient, batch-wise manner. The process ensures scalable computation while maintaining performance.

Step-by-Step Explanation:

1. Batch Processing with TQDM:

- The loop iterates through `text_list` in increments of `batch_size` (512) to avoid memory overload.
- `tqdm` provides a progress bar labeled "**Generating Embeddings**", helping track execution.

2. Generating Embeddings:

- Each batch (subset of `text_list`) is extracted and passed to `model.encode()`, which converts the text into numerical embeddings.
- `convert_to_numpy=True` ensures the output is in NumPy format, making it efficient for computations.
- `show_progress_bar=False` is used to keep the output clean.

3. Storing and Normalizing Embeddings:

- The generated embeddings are **appended** to the embeddings list, ensuring all batches are merged into a single structure.
- After processing all text, the **normalize()** function scales the embeddings, improving similarity calculations.

4. Saving Embeddings in the DataFrame:

- The **normalized embeddings** are assigned to a new column, 'embeddings', in the **DataFrame (df)**.
- Since embeddings are NumPy arrays, they are converted to lists to ensure compatibility with Pandas.

5. Verification:

- `df.head()` prints the first few rows to confirm embeddings are added.
- `df['embeddings'].iloc[0].shape` checks the size of an individual embedding, confirming it has the expected **384 dimensions**.

Purpose:

This method efficiently computes **semantic embeddings** for Wikipedia articles while ensuring **memory efficiency**, **scalability**, and **improved similarity scoring** for downstream tasks like **article recommendations**.

```
[nltk_data] Downloading package stopwords to /usr/share/nltk_data...
[nltk_data] Package stopwords is already up-to-date!
Processing Batches: 100% ██████████ 11840/11840 [1:11:45<00:00, 4.67it/s]
0    Title: A & B High Performance Firearms. Catego...
1    Title: A & C Black. Categories: Encyclopædia B...
2    Title: A & F Harvey Brothers. Categories: Cott...
3    Title: A & G Price. Categories: Locomotive man...
4    Title: A & M Karagheusian. Categories: 1904 es...
Name: combined, dtype: object
```

Running this part of the code takes more than 1 hour. We have run this part more than once to check whether the different combinations of hyperparameters works well or not.

After running this part, we have the dataframe in this format:

```
df.head()
```

	id	combined	embeddings
0	49495844	Title: A & B High Performance Firearms. Catego...	[0.005076993220085449, 0.014148154193647369, -...
1	3579086	Title: A & C Black. Categories: Encyclopædia B...	[-0.020476288057262163, -0.09883869838271596, ...
2	62397582	Title: A & F Harvey Brothers. Categories: Cott...	[-0.08427029833660497, -0.01244955600900288, -...
3	15547032	Title: A & G Price. Categories: Locomotive man...	[-0.10381282871141384, -0.01448750310363473, 0...
4	8021609	Title: A & M Karagheusian. Categories: 1904 es...	[-0.053385633809100275, -0.017941486034577337, ...

Variance

The variance of the data:

```
# # Check variance of embeddings
print(f"Variance of Embeddings: {np.var(embedding_matrix, axis=0).mean()}")
```

Variance of Embeddings: 0.0024311788196501563

Recommend Articles Logic

```
from sklearn.metrics.pairwise import cosine_similarity

def recommend_articles(input_title, top_n=10):
    if input_title not in df['combined'].values:
        return "Article not found."

    input_idx = df.index[df['combined'] == input_title].item()
    input_embedding = embedding_matrix[input_idx]

    # Compute cosine similarity
    similarities = cosine_similarity([input_embedding],
    embedding_matrix).flatten()

    # Exclude the input article and get top matches
    sorted_indices = np.argsort(similarities)[::-1]
    valid_indices = [i for i in sorted_indices if i != input_idx][:top_n]

    return df.iloc[valid_indices]
```

This function, `recommend_articles(input_title, top_n=10)`, suggests similar articles based on **cosine similarity** of their embeddings.

Step-by-Step Breakdown:

1. Check if the Article Exists:

- If `input_title` is not in the dataset (`df['combined'].values`), it returns "Article not found."

2. Retrieve the Embedding of the Input Article:

- Finds the index of the article in the DataFrame (`input_idx`).
- Retrieves its corresponding **embedding vector** from `embedding_matrix`.

3. Compute Similarity Scores:

- Uses **cosine similarity** to compare the input article's embedding with all others in `embedding_matrix`.

- The result is a **flattened array** of similarity scores.

4. Sort and Filter Results:

- **Sorts indices** in descending order to rank articles from most to least similar.
- **Excludes the input article** from recommendations.
- Selects the top `top_n` matches.

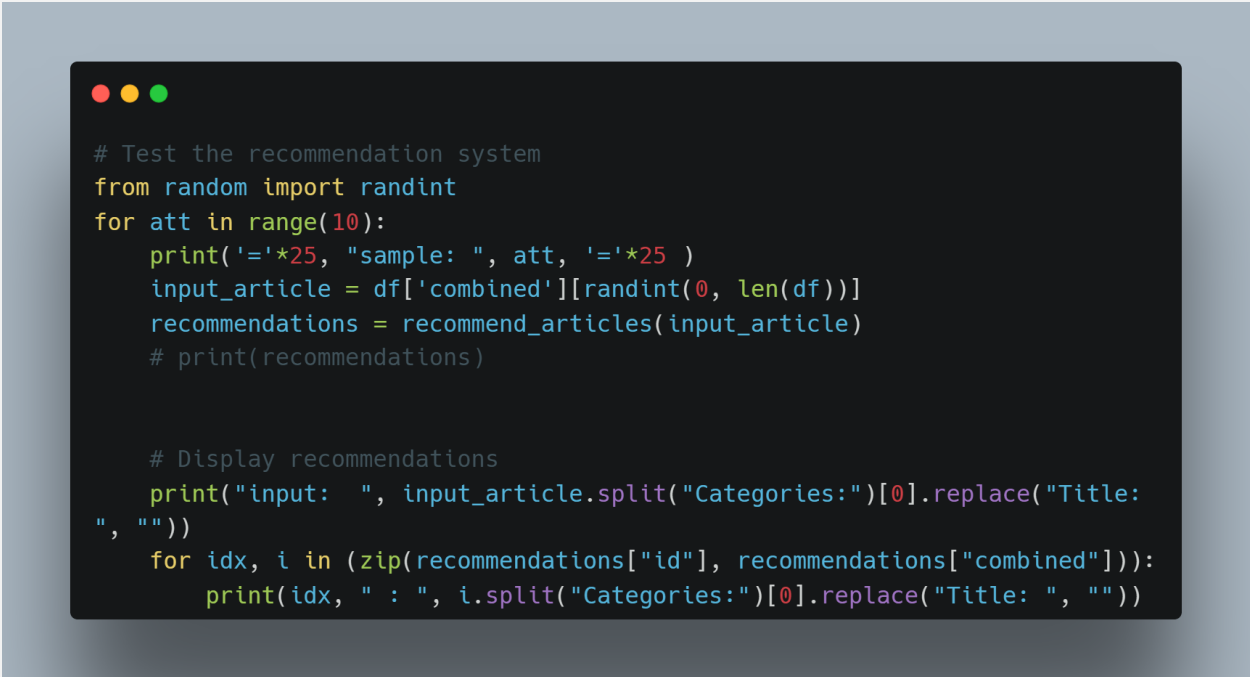
5. Return Recommended Articles:

- The function returns the corresponding **DataFrame rows** of the most similar articles.

Purpose:

This approach allows **efficient and scalable** article recommendations, helping users discover relevant Wikipedia content based on semantic similarity.

Testing Recommender System



```
# Test the recommendation system
from random import randint
for att in range(10):
    print('='*25, "sample: ", att, '='*25 )
    input_article = df['combined'][randint(0, len(df))]
    recommendations = recommend_articles(input_article)
    # print(recommendations)

# Display recommendations
print("input: ", input_article.split("Categories:")[0].replace("Title: ", ""))
for idx, i in (zip(recommendations["id"], recommendations["combined"])):
    print(idx, " : ", i.split("Categories:")[0].replace("Title: ", ""))
```

This code randomly selects **10 articles** and tests the recommendation system by retrieving similar articles for each. It follows these steps:

1. Randomly Select an Article:

- Chooses a random article from `df['combined']` using `randint(0, len(df))`.

2. Generate Recommendations:

- Calls `recommend_articles(input_article)` to find similar articles.

3. Display Results:

- Prints the **input article title** (before "Categories").
- Iterates over recommended articles, showing their **IDs and titles**.

This ensures the system is working correctly by providing real-time examples of suggested articles.

Results

Here are some of the inputs and their corresponding outputs and suggestions:

```
===== sample: 0 =====  
input:  Brouilletts Creek Covered Bridge.  
51089904 : Brownsville Covered Bridge.  
45357103 : Coal Creek Covered Bridge.  
50338322 : Cavanaugh Bridge.  
13189266 : Billie Creek Covered Bridge.  
13092346 : Big Rocky Fork Covered Bridge.  
21326891 : Baker's Camp Covered Bridge.  
47728334 : Aqueduct Bridge (Clay City, Indiana).  
47731095 : County Bridge No. 45.  
13144895 : Crooks Covered Bridge.  
13135571 : Conley's Ford Covered Bridge.
```



```
===== sample: 9 =====
input:  Counterslip Baptist Church.
61269318 : Broadmead Baptist Church.
11074571 : City Road Baptist Church.
23470840 : Cornerstone Church Liverpool.
38058802 : Broad Street Baptist Church.
11572035 : Buckingham Baptist Chapel.
10420401 : Church of Holy Trinity, Hotwells.
53502481 : Albany Road Baptist Church.
36464241 : Bristol Jamia Mosque.
64396240 : Carrington Baptist Church, Nottingham.
2845449 : Bristol Community Church.
```

```
===== sample: 6 =====
input:  Canadian Environmental Sustainability Indicators.
14602493 : Canadian Council of Ministers of the Environment.
56873964 : Context-Based Sustainability.
10685699 : Canadian Environmental Assessment Act, 2012.
30034641 : Committee on Sustainability Assessment.
12402182 : California Environmental Protection Agency.
19748815 : California Environmental Resources Evaluation System.
49598146 : Criteria and indicators of sustainable forest management.
37514747 : Climate Change Performance Index.
44154342 : Alberta Environment and Protected Areas.
29592212 : BSI PAS 2060.
```

```
input:  ABS-3.
46694726 : Agila-1.
55702740 : ABS-7.
50688532 : ABS-2.
40565206 : ABS-CBN Convergence.
2503405 : ABS-CBN.
45525154 : ABS-3A.
8522558 : AsiaSat 3S.
17632955 : Broadcasting Satellite System Corporation.
45686536 : ABS-CBN TV Plus.
21684356 : Signal.
```

```
input:  Australian Railway Historical Society.
56680499 : Australian Transport Officers' Federation.
4806206 : Australian National Railways Commission.
56346778 : Australian Federated Union of Locomotive Employees.
2004088 : Australian Rail Track Corporation.
6958687 : Australian Railway History.
26300855 : Australian Railways Union.
2495693 : Australian Railroad Group.
1966395 : Australian Rail Tram and Bus Industry Union.
25367604 : Australian Society of Section Car Operators.
39900318 : AN Tasrail.
```

```
input:  Australian women's cricket team in New Zealand in 2015-16.
52708581 : Australian women's cricket team in New Zealand in 2016-17.
69105027 : Australian women's cricket team in New Zealand in 2008-09.
69105026 : Australian women's cricket team in New Zealand in 2007-08.
68264001 : Australian women's cricket team in New Zealand in 1994-95.
69073129 : Australian women's cricket team in New Zealand in 1993-94.
51517728 : Australian women's cricket team in Sri Lanka in 2016-17.
56110939 : Australian women's cricket team in India in 2017-18.
69073132 : Australian women's cricket team in New Zealand in 1996-97.
69073136 : Australian women's cricket team in New Zealand in 1998-99.
64821981 : Australia women's cricket team in New Zealand in 2020-21.
```

```
input:  All In (TV series).
2375408 : All About Eve (South Korean TV series).
73754882 : All That We Loved.
51747707 : A Style For You.
47669633 : All About My Mom.
39057102 : All the K-pop.
38614584 : All About My Romance.
64658180 : All That Cube.
22676622 : April Kiss.
64308826 : All of Us Are Dead.
31984409 : All My Love For You.
```

```
input:  A Good Girl's Guide to Murder.
An Expert in Murder.
A Murder Is Announced.
Act of Murder (novel).
A Daughter's Deadly Deception.
A Mind to Murder.
Assassin's Fate.
Arsenic for Tea.
A Beautiful Crime.
A Gentleman's Guide to Love and Murder.
A Catalogue of Crime.
```

```
input:  Angela Maria Guidi Cingolani.
37470274 : Angelo Maria Cicolani.
65558912 : Angela Ianaro.
58659217 : Anna Gabriella Ceccatelli.
59130886 : Anna Rossomando.
28437379 : Anna Finocchiaro.
71152973 : Anna Sanna.
61000622 : Anna Cinzia Bonfrisco.
11321021 : Adele Faccio.
70888662 : Anna Maria Bernasconi.
29247272 : Andrea Causin.
```

```
input:  APPG on Agriculture and Food for Development.
24662445 : Agriculture and Horticulture Development Board.
20264398 : Agriculture in the United Kingdom.
38050724 : Agricultural Engineers Association.
19255711 : Agricultural and Marketing Research and Development Trust.
21105306 : Agriculture in England.
29343003 : Assured Food Standards.
21132828 : Agricultural society.
26658597 : Australasian Agricultural and Resource Economics Society.
5272089 : Agricultural Development Bank.
21422588 : Agricultural & Applied Economics Association.
```