PRODUCT RATINGS AND RECOMMENDATION SYSTEM

AUTHOR - PINAK SHOME

AFFILIATION - UNIVERSITY OF DENVER

INSTRUCTOR - DON DALTON

COURSE - DEEP LEARNING

DEPARTMENT - COMPUTER SCIENCE

DATE - 06/08/2024

INTRODUCTION

DATASET DESCRIPTION:

Data Fields

For User Reviews

Field	Type	Explanation
rating	float	Rating of the product (from 1.0 to 5.0).
title	str	Title of the user review.
text	str	Text body of the user review.
images	list	Images that users post after they have received the product. Each image has different sizes (small, medium, large), represented by the small_image_url, medium_image_url, and large_image_url respectively.
asin	str	ID of the product.
parent_asin	str	Parent ID of the product. Note: Products with different colors, styles, sizes usually belong to the same parent ID. The "asin" in previous Amazon datasets is actually parent ID. **Please use parent ID to find product meta.**
user_id	str	ID of the reviewer
timestamp	int	Time of the review (unix time)
verified_purchase	bool	User purchase verification
helpful_vote	int	Helpful votes of the review

For Item Metadata

Field	Type	Explanation
main_category	str	Main category (i.e., domain) of the product.
title	str	Name of the product.
average_rating	float	Rating of the product shown on the product page.
rating_number	int	Number of ratings in the product.
features	list	Bullet-point format features of the product.
description	list	Description of the product.
price	float	Price in US dollars (at time of crawling).
images	list	Images of the product. Each image has different sizes (thumb, large, hi_res). The "variant" field shows the position of image.
videos	list	Videos of the product including title and url.
store	str	Store name of the product.
categories	list	Hierarchical categories of the product.
details	dict	Product details, including materials, brand, sizes, etc.
parent_asin	str	Parent ID of the product.
bought_together	list	Recommended bundles from the websites.

BUSINESS OBJECTIVES:

- Predict Product Ratings from given user reviews (No Latency Constraints, Accuracy Highly Important)
- Build a Product Recommendation System Based on given features to boost Sales
- Avoid Situations when ratings and reviews do not reflect the content of each other (Logistical Issues, Sparse Ratings, Miss Clicks)
- Help Detect unusual anomalies or patterns in ratings. A sudden influx of low ratings with positive reviews might indicate a coordinated spam or attack
- Business can use product ratings to identify highly positive reviews, using this to promote products based on customer sentiment rather than just numeric ratings

RESEARCH OBJECTIVES:

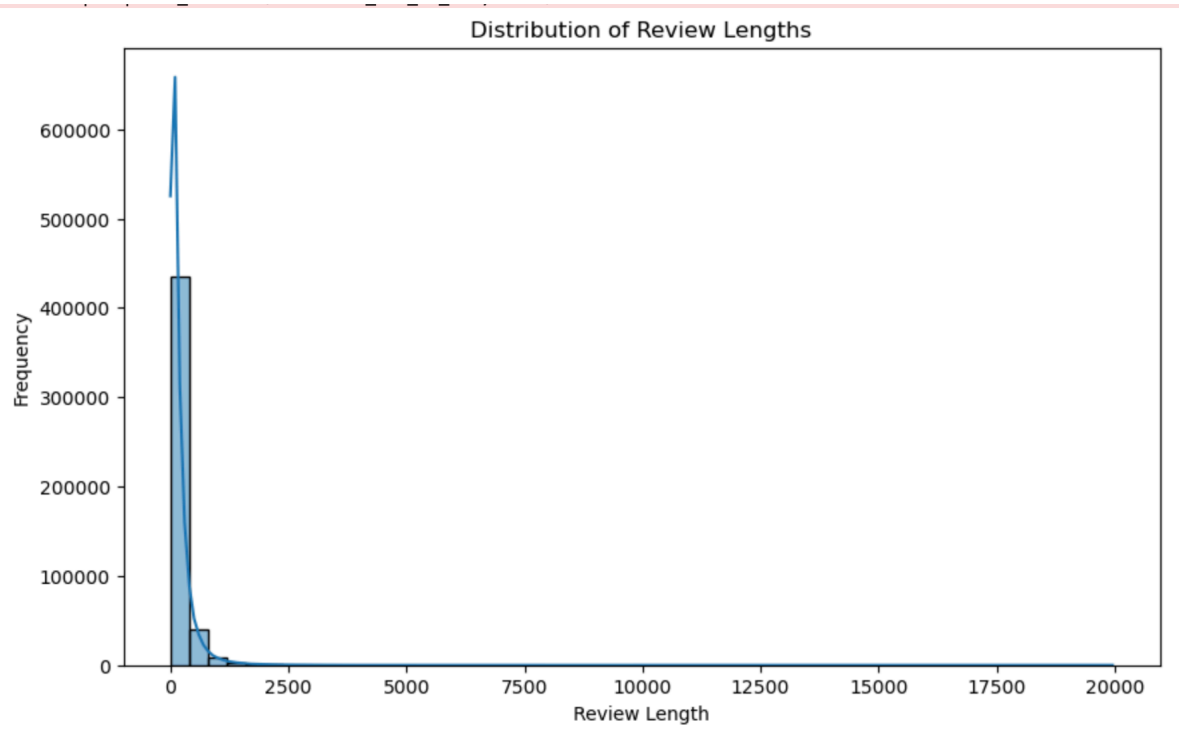
- Exploring Ability of neural-networks to form meaningful connections on very limited set of data
- Exploring the use of treating 'reviews' which are inherently static as sequential data and evaluating the performance by using Recurrent Neural Networks.
- Exploring the ability of NN's to understand the relationships between features and target variables better than classical Machine learning models when features don't follow assumptions that classical machine learning models benefit from
- Analyzing the importance of dynamic dataset specific word embedding's over pre-trained.

Preprocessing & Exploratory Data Analysis:

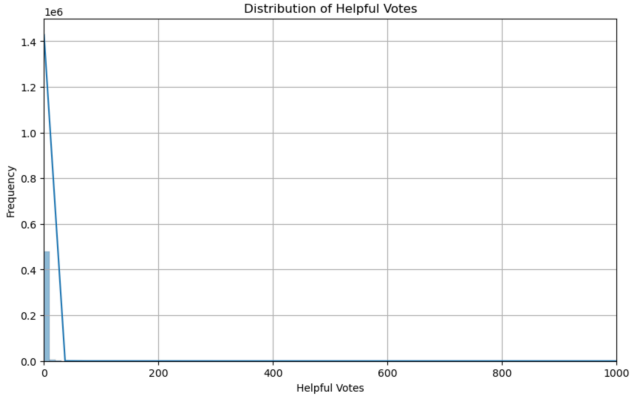
Preprocessing workflow steps:

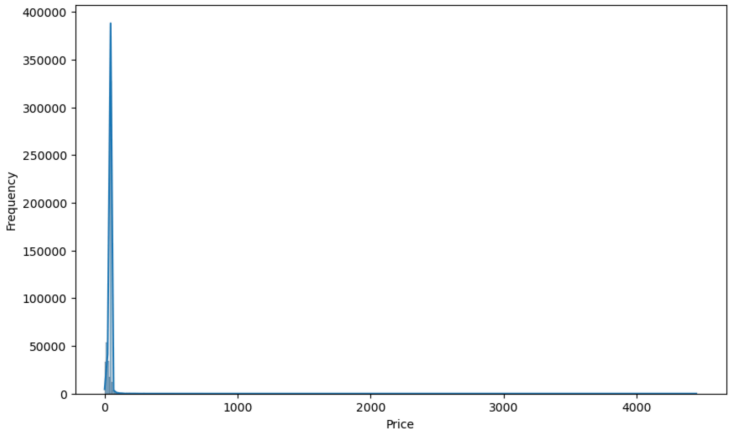
- Both Jsonl files converted to Pandas DataFrame
- Dropped columns across both data frames: [‘images’,’videos','categories','features','description','main_category','bought_together']
- Checked and Dropped Duplicate rows across both DataFrames
- Check for unique Column Values across both columns
- Mean imputed Price since majority of price distribution was in the same range
- Flattened specific columns in meta_data because they existed in form of nested lists or nested dictionaries
- Fill all empty columns ‘NA’ or ‘[]’ with placeholder values
- Feature Engineered 4 new features - Review Length, title length, summary length, Details Length
- Merged Both Dataframes product ID
- Tried Extracting keywords from reviews, titles and details with spacy models - No highly significant features extracted

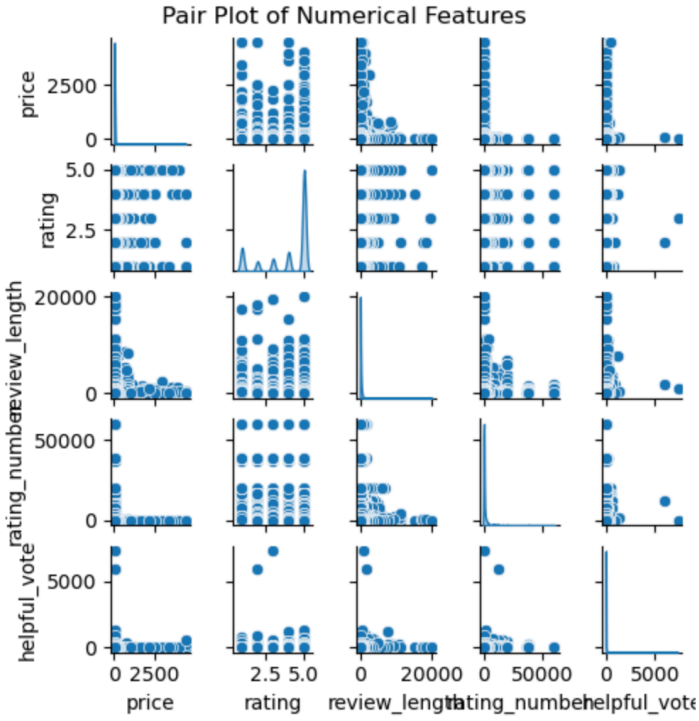
PLOTS:

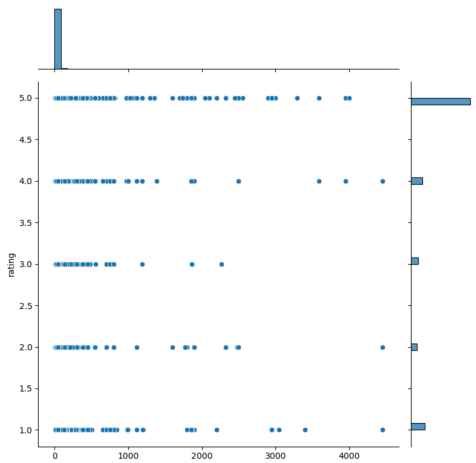


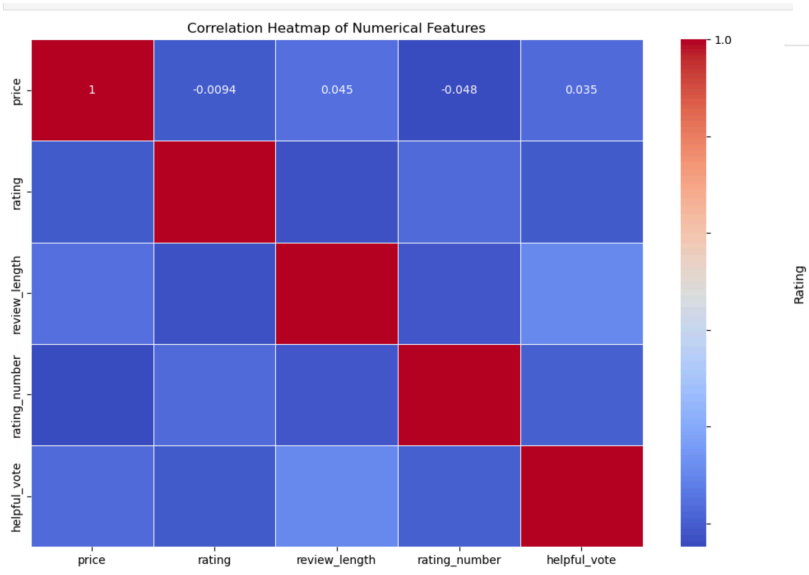

Most Common Words in Review Texts
recommend
one
keep
needed
even
really
good
will
use
received
problem
Great
product
purchase
Want
put
buy
go
easy
used
purchased
ordered
think
comfortable
soft
seem
perfect
much
using
looking
review
fit
smell
clean
now
s
product
see
need
nice
far
hold
hand
made
brush
taste
little
hard
wear
thing
bought
got
week
thought
small
don t
tried
time
love
help
bag
give
come
feet
make
skin
back
way
going
works
great
br
br
price
item
find
money
try
say
able
still
take
look
sure
know
year
water
happy
found
best
great
work
feel
well
lot
day
first
expected
hair
better
bottle

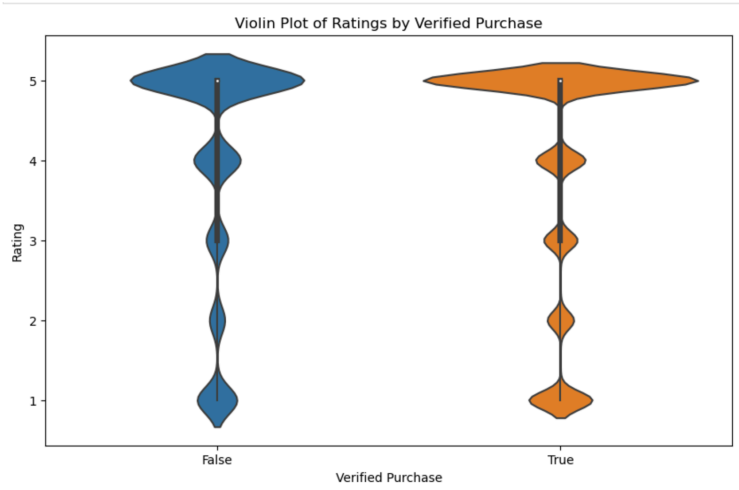

Distribution of Helpful Votes
1e6
Frequency
Helpful Votes

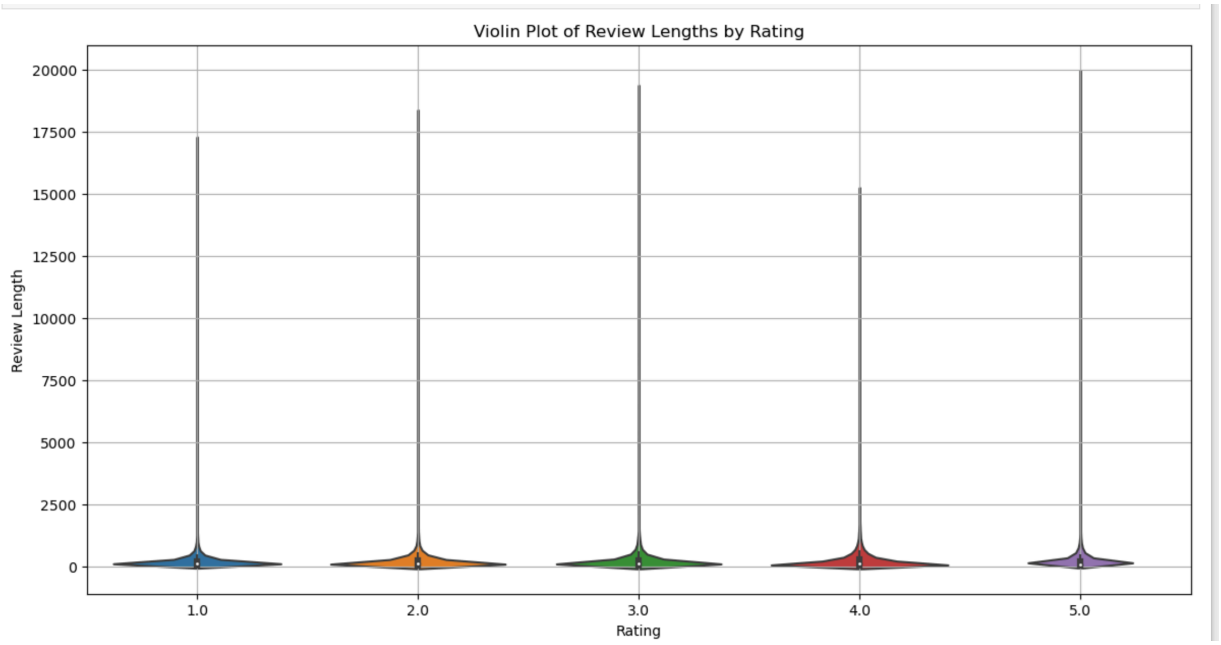

Distribution of Product Prices
Frequency
Price


Pair Plot of Numerical Features
price
rating
review_length
rating_number
helpful_vote


rating







EDA OBSERVATIONS:

- Low Correlation Amongst Features
- Imbalanced Class Problem with highest concentration on Ratings 5 & 1
- Review Length across all ratings are similar
- Pair plots tell us non-linear relationships exist
- Product Prices are heavily concentrated under 250
- Review lengths are generally under 1250 words
- Several rows of features have empty lists in place of values, which were dropped

MODELING:

BASELINE MODEL - RANDOM FOREST REGRESSOR:

```python
# Define the target variable
y = sample_df['rating'].values

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_sparse, y, test_size=0.2, random_state=42)

# Train a Random Forest Regressor
model = RandomForestRegressor(n_estimators=50, random_state=42, n_jobs=-1)  # Use all cores
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Calculate and print the Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

```
Mean Squared Error: 1.1258917467203804
```

MODEL 2: Hyper-parameter Tuned Neural Networks with tf-idf embeddings:

```python
)

# Perform the hyperparameter search
tuner.search(X_train, y_train, epochs=10, validation_split=0.2, verbose=1)

# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Print the best hyperparameters
print(f"""
The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is {best_hps.get('units')} and the optimal dropout rate is {best_hps.get('dropout')}.
""")

# Build the best model
model = tuner.hypermodel.build(best_hps)

# Train the model
history = model.fit(X_train, y_train, epochs=20, validation_split=0.2, verbose=1)

# Predict on the test set
y_pred = model.predict(X_test)

# Calculate and print the Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

```
Trial 10 Complete [00h 01m 06s]
mse: 0.3182298094034195

Best mse So Far: 0.0954609289765358
Total elapsed time: 00h 12m 54s

The hyperparameter search is complete. The optimal number of units in the first densely-connected
layer is 32 and the optimal dropout rate is 0.0.
```

MODEL 3: RNNs treating static ‘reviews’ as sequential using Keras Tokenizer:

Layer (type)	Output Shape	Param #	Connected to
text_input (InputLayer)	(None, 100)	0	–
title_input (InputLayer)	(None, 100)	0	–
embedding (Embedding)	(None, 100, 50)	500,000	text_input[0][0]
embedding_1 (Embedding)	(None, 100, 50)	500,000	title_input[0][0]
lstm (LSTM)	(None, 64)	29,440	embedding[0][0]
lstm_1 (LSTM)	(None, 64)	29,440	embedding_1[0][0]
numerical_input (InputLayer)	(None, 7)	0	–
concatenate (Concatenate)	(None, 135)	0	lstm[0][0], lstm_1[0][0], numerical_input[…
dense (Dense)	(None, 64)	8,704	concatenate[0][0]
dense_1 (Dense)	(None, 1)	65	dense[0][0]

Total params: 1,067,649 (4.07 MB)
Trainable params: 1,067,649 (4.07 MB)
Non-trainable params: 0 (0.00 B)

MODEL 4: BERT FINE-TUNED FOR REGRESSION

```python
# Define a PyTorch dataset class
class ReviewDataset(Dataset):
    def __init__(self, texts, numerical_features, targets, tokenizer, max_len):
        self.texts = texts
        self.numerical_features = numerical_features
        self.targets = targets
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, index):
        text = self.texts[index]
        numerical_features = self.numerical_features[index]
        target = self.targets[index]

        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_len,
            return_token_type_ids=False,
            padding='max_length',
            truncation=True,
            return_attention_mask=True,
            return_tensors='pt',
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'numerical_features': torch.tensor(numerical_features, dtype=torch.float),
            'target': torch.tensor(target, dtype=torch.float)
        }
```

```python
# Define the Transformer model for regression
class TransformerRegressor(torch.nn.Module):
    def __init__(self, transformer_model_name, n_numerical_features):
        super(TransformerRegressor, self).__init__()
        self.transformer = BertModel.from_pretrained(transformer_model_name)
        self.drop = torch.nn.Dropout(p=0.3)
        self.out = torch.nn.Linear(self.transformer.config.hidden_size + n_numerical_features, 1)

    def forward(self, input_ids, attention_mask, numerical_features):
        transformer_outputs = self.transformer(
            input_ids=input_ids,
            attention_mask=attention_mask
        )
        pooled_output = transformer_outputs[1]
        combined = torch.cat((pooled_output, numerical_features), dim=1)
        output = self.out(self.drop(combined))
        return output
```

```python
# Training function
def train_epoch(model, data_loader, loss_fn, optimizer, device, scheduler, n_examples):
    model = model.train()
    losses = []
    for batch in data_loader:
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        numerical_features = batch['numerical_features'].to(device)
        targets = batch['target'].to(device)

        outputs = model(input_ids=input_ids, attention_mask=attention_mask, numerical_features=numerical_features)
        loss = loss_fn(outputs, targets.unsqueeze(1))
        losses.append(loss.item())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()

    return np.mean(losses)
```

```python
# Evaluation function
def eval_model(model, data_loader, loss_fn, device, n_examples):
    model = model.eval()
    losses = []
    preds = []
    with torch.no_grad():
        for batch in data_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            numerical_features = batch['numerical_features'].to(device)
            targets = batch['target'].to(device)

            outputs = model(input_ids=input_ids, attention_mask=attention_mask, numerical_features=numerical_features)
            loss = loss_fn(outputs, targets.unsqueeze(1))
            losses.append(loss.item())
            preds.append(outputs.cpu().numpy())

    return np.mean(losses), np.concatenate(preds, axis=0)

# Training the model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

optimizer = AdamW(model.parameters(), lr=2e-5)
total_steps = len(train_loader) * 10

scheduler = get_linear_schedule_with_warmup(
    optimizer,
    num_warmup_steps=0,
    num_training_steps=total_steps
)

loss_fn = torch.nn.MSELoss().to(device)
```

ACCURACY RESULTS:

Random Forest (OverSampled) — 1.12 (With hyper-parameter tuning goes down to 0.8)
Neural Networks (OverSampled) - 0.33
Hyper-Parameter Tuned NN (Without OverSampling) - 0.811
Hyper-Parameter Tuned NN (With OverSampling) - 0.04
RNNs using Keras Tokenizer (Without OverSampling) - 0.8
BERT(Without Oversampling) - 1.4

OBSERVATIONS:

- Class Imbalance is significantly affecting how neural-networks learn
- OverSampled Neural Networks outperform OverSampled random forest, implying that deep learning models are better at learning relationships between features when they don't follow classical machine learning model suitability assumptions
- Hyper-parameter tuning over-sampled NN's significantly increases performance
- Treating Review Text as sequential (when it was inherently static) and using RNNs to capture sequential context when dataset is small and vocabulary is limited does not improve performance
- RNN's are better at handling inherently sequential data and may outperform feed forward networks on static data as well but need a larger dataset and a bigger vocabulary to be able to capture relationships better
- Fine-tuning Bert for regression does not indicate any significant improvement in performance as well
- This suggests that even though Bert embeddings are trained on a large corpus on data, fine-tuning it to the specific dataset in contention is highly important for it to capture semantic meanings better

PRODUCT RECOMMENDATION SYSTEM

Workflow Steps:

- Select a subset of 10,000 samples from the merged dataset.
- Combine text columns into a single `combined_text` column.
- Load BERT tokenizer and model from the `transformers` library.
- Define `get_bert_embeddings` function to generate BERT embeddings.

- Apply the function to the combined_text column and stack embeddings.
- Select numerical features for normalization.
- Normalize numerical features using StandardScaler.
- Combine BERT embeddings and normalized numerical features into X.
- Define the target variable y as the rating column.
- Split data into training and testing sets (80-20 split).
- Define a regression model with multiple dense layers.
- Compile the model with adam optimizer and mean_squared_error loss.
- Train the model on the training data for 20 epochs with batch size 32.
- Predict ratings for all products using the trained model.
- Combine BERT embeddings, normalized numerical features, and predicted ratings into X_with_ratings.
- Calculate cosine similarity between products using X_with_ratings.
- Define get_similar_products function to retrieve top K similar products.
- Use the function to find the top 5 similar products to an example product and print results.

```
subset_df['combined_text'] = subset_df['text'] + ' ' + subset_df['title_x'] + ' ' + subset_df['title_y'] + ' ' + subset_df['details']

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

# Function to get BERT embeddings
def get_bert_embeddings(text):
    inputs = tokenizer(text, return_tensors='pt', max_length=512, truncation=True, padding=True)
    outputs = bert_model(**inputs)
    embeddings = outputs.last_hidden_state[:, 0, :].detach().numpy()
    return embeddings

# Generate BERT embeddings for the text data
bert_embeddings = np.vstack([get_bert_embeddings(text) for text in subset_df['combined_text']])

# Normalize numerical features
numerical_features = subset_df[['price', 'average_rating', 'rating_number', 'helpful_vote', 'x_length', 'y_length', 'de_length', 'review_length']]
scaler = StandardScaler()
normalized_numerical_features = scaler.fit_transform(numerical_features)

# Combine BERT embeddings with normalized numerical features
X = np.hstack([bert_embeddings, normalized_numerical_features])

# Define the target variable
y = subset_df['rating'].values

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert the input data to dense format
X_train = X_train.astype(np.float32)
X_test = X_test.astype(np.float32)

# Build the regression model
input_shape = X_train.shape[1]
```

```
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1)  # Output layer for regression
])

model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(X_train, y_train, epochs=20, validation_split=0.2, batch_size=32, verbose=1)

# Predict ratings for all products
predicted_ratings = model.predict(X).flatten()

# Combine BERT embeddings, normalized numerical features, and predicted ratings
X_with_ratings = np.hstack([bert_embeddings, normalized_numerical_features, predicted_ratings.reshape(-1, 1)])

# Calculate cosine similarity between products
cosine_sim_with_ratings = cosine_similarity(X_with_ratings, X_with_ratings)

# Function to get top K similar products considering predicted ratings
def get_similar_products(product_idx, k=5):
    sim_scores = list(enumerate(cosine_sim_with_ratings[product_idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:k+1]  # Exclude the product itself
    similar_products = [(subset_df.iloc[i[0]]['title_y'], i[1]) for i in sim_scores]
    return similar_products

# Example usage
example_product_idx = 0  # Using the first product as an example
similar_products = get_similar_products(example_product_idx, k=5)
print(f"Top 5 similar products to '{subset_df.iloc[example_product_idx]['title_y']}':")
for product, score in similar_products:
    print(f"Product: {product}, Similarity Score: {score}")
```

```
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.4428 - val_loss: 1.6609
Epoch 6/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.3497 - val_loss: 1.9762
Epoch 7/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.3966 - val_loss: 1.3653
Epoch 8/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.3365 - val_loss: 1.3940
Epoch 9/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.3325 - val_loss: 1.3315
Epoch 10/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.2878 - val_loss: 1.4779
Epoch 11/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.2937 - val_loss: 1.6386
Epoch 12/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1717 - val_loss: 1.4884
Epoch 13/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1753 - val_loss: 1.4788
Epoch 14/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1432 - val_loss: 1.3524
Epoch 15/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1291 - val_loss: 1.4721
Epoch 16/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.0787 - val_loss: 1.3747
Epoch 17/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.0047 - val_loss: 1.3140
Epoch 18/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1242 - val_loss: 1.3915
Epoch 19/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.0042 - val_loss: 1.3625
Epoch 20/20
200/200 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 1.1237 - val_loss: 1.3921
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Top 5 similar products to 'AlbaChem® PSR II Powdered Dry Cleaning Fluid':
Product: Alba Chem PSR Powdered Dry Cleaning Fluid 12.5 oz, Similarity Score: 0.9774683389647245
Product: Alba Chem PSR Powdered Dry Cleaning Fluid 12.5 oz, Similarity Score: 0.9766651665965533
Product: Bright Air Odor Eliminator - Cool and Clean , 14 Ounce Jar (Pack of 3), Similarity Score: 0.9760934781712813
Product: Hydrosilex Recharge 1000 ML/32OZ, Universal Hydrophobic Coating Finish Spray-On Protection for Paint, Vinyl, Rubber, Plastic, Similarity Score:
0.9753441667909953
Product: WD40 300080 Specialist Electrical Contact Cleaner 11oz, Similarity Score: 0.9749399656579174
```