TF-IDF vectorizer

```
def recommend_jobs(resume_df, jobs_df, top_n=5):
  """Recommends jobs based on a resume using TF-IDF."""
```

- This defines the function `recommend_jobs` which takes two DataFrames, `resume_df` and `jobs_df` , and an optional argument `top_n` (defaulting to 5) specifying the number of top recommendations to return.

```
def process_column(value):
    """Flattens nested lists and converts everything to a string."""
    if isinstance(value, list):
      # Flatten only if it's a list of lists
      flat_list = []
      for item in value:
        if isinstance(item, list):
          flat_list.extend(item)  # Unpack sublist
        else:
          flat_list.append(item)  # Directly add non-list items
      return ' '.join(map(str, flat_list))  # Convert to string
    return str(value)  # Convert non-lists to string
```

- This function is crucial for handling the potentially nested list structure of data (e.g., lists of skills, lists of positions).
- **It does two main things:**
  1. **Flattens Nested Lists:**
     - If a value is a list, it checks if the elements of that list are also lists.
     - If they are (a list of lists), it flattens the inner lists into a single list.
       - For example, `[['skill1', 'skill2'], 'skill3']` , it correctly unpacks the inner list to become `['skill1', 'skill2', 'skill3']`
     - If the list elements are not lists themselves, they are simply added to the `flat_list` .
  2. **Converts to String:**
     - Regardless of the input type (list or not), it converts the value to a string.
     - This is essential because TF-IDF works with text, it joins the elements of the flattened list (if it was a list) with spaces.

```
# 1. Process resume data
  resume_text = resume_df.apply(lambda row: ' '.join([
    process_column(row['skills']),
```

```
        process_column(row['institution']),
        process_column(row['degree_names']),
        process_column(row['field_of_study']),
        process_column(row['experience_related_skills']),
        process_column(row['experience_positions']),
        process_column(row['experience_responsibilities']),
    ]), axis=1).values
```

- This section applies the `process_column` function to each column of the `resume_df`.
- It then joins the resulting strings for each row (representing a single resume) into one large string.
- The `axis=1` in `apply` ensures that the function is applied row-wise.
- The `.values` extracts the NumPy array of the resulting strings.
- So, `resume_text` becomes an array where each element is the combined text representation of a resume.

```
# 2. Process job postings
    jobs_text = jobs_df.apply(lambda row: ' '.join([
        process_column(row['position']),
        process_column(row['job_role_and_duties']),
        process_column(row['requisite_skill']),
        process_column(row['offer_details'])
    ]), axis=1).values
```

- This does the same thing as the resume processing, but for the `jobs_df`.
- `jobs_text` becomes an array of strings, each representing a job posting.

```
# 3. Combine resume and job descriptions
    all_text = pd.Series(list(resume_text) + list(jobs_text))
```

- This combines the `resume_text` and `jobs_text` arrays into a single Pandas Series.
- This is necessary because the TF-IDF vectorizer needs to be fit on all the text data (resumes and jobs) to learn the vocabulary and term frequencies.

```
# 4. TF-IDF Processing
    tfidf = TfidfVectorizer(stop_words='english')
    tfidf_matrix = tfidf.fit_transform(all_text)

    resume_tfidf = tfidf_matrix[:len(resume_df)]
    jobs_tfidf = tfidf_matrix[len(resume_df):]
```

- **This is the core of the recommendation system:**
    - **`TfidfVectorizer(stop_words='english')`:**

- Creates a TF-IDF vectorizer, which will convert the text into numerical vectors.
- `stop_words='english'` removes common English words (like "the", "a", "is") that don't carry much meaning.

- `tfidf.fit_transform(all_text)`:
  - Fits the vectorizer on all the text data and transforms it into a TF-IDF matrix.
  - This matrix represents the importance of each word in each document (resume or job posting).
- `resume_tfidf = tfidf_matrix[:len(resume_df)]`:
  - Extracts the TF-IDF vectors for the resumes.
- `jobs_tfidf = tfidf_matrix[len(resume_df):]`:
  - Extracts the TF-IDF vectors for the job postings.

```
# 5. Calculate Cosine Similarity
    cosine_similarities = cosine_similarity(resume_tfidf, jobs_tfidf)
```

- This calculates the cosine similarity between each resume vector and each job posting vector.
- Cosine similarity measures the angle between two vectors; a higher cosine similarity indicates a greater degree of similarity.
- The result is a matrix where `cosine_similarities[i][j]` represents the similarity between the i-th resume and the j-th job.

```
# 6. Get Recommendations
    if cosine_similarities.size > 0:
        recommended_job_indices = cosine_similarities.argsort(axis=1)[:, ::-1][0]
        num_recommendations = min(top_n, len(recommended_job_indices))
        recommended_job_indices = recommended_job_indices[:num_recommendations]

        recommended_jobs = jobs_df.iloc[recommended_job_indices.tolist()]
    else:
        recommended_jobs = pd.DataFrame()
        print("No similar jobs found.")

    return recommended_jobs
```

- **This section determines the top recommendations:**
  - `cosine_similarities.argsort(axis=1)[:, ::-1][0]`:
    - For the first resume (index 0), it finds the indices of the jobs sorted by cosine similarity in descending order (`[::-1]`).
    - `argsort` returns the indices that would sort the array.
  - `num_recommendations = min(top_n, len(recommended_job_indices))`:
    - Ensures that we don't try to recommend more jobs than available.

- `recommended_job_indices = recommended_job_indices[:num_recommendations]`:
  - Takes the indices of the top `n` recommended jobs.
- `recommended_jobs = jobs_df.iloc[recommended_job_indices.tolist()]`:
  - Uses the indices to retrieve the corresponding job postings from `jobs_df`.
- The `else` block handles the case where no similar jobs are found.