Preprocessing Data to Build Receipt Understanding Model

In this notebook we will focus on preprocessing data to build the receipt understanding model. We will finetune Microsoft's open-source LayoutLMv3 model on the Scanned Receipts OCR and Information Extraction (SROIE) dataset. Our goal is to create train and test datasets that ready to be consumed by the model.

Data Loading Source

The SROIE dataset was released in the Robust Reading Competetion. The dataset is shared with us as a Google Drive link containing all files. To help simplify the dataset loading process, we will use the dataset published on Kaggle which can be found here.

There are other datasets avillable on huggingface and kaggle that are also copies of the original SROIE dataset. I will be using this particular dataset as it contains the files in the same format as the one they released on Google Drive.

Before we load the dataset, we will need to download some neccessary packages and libraries.

Note: The downloaded packages are the ones not available on Google Colab by default. Downloading them is added here to make it simple and easy to quickly try the code in colab. If you are running this notebook locally, ensure that you're running the notebook in a dedicated conda environment or a virtual environment so that you don't pollute your base environment. If the notebook fails to run locally, make sure you have the other dependencies installed.

```
%%capture
! pip install datasets
! pip install transformers
! pip install kaggle
```

Now we are going to use the official Kaggle API to download the dataset in the data/unprocessed folder.

```
%%capture
! kaggle datasets download -d urbikn/sroie-datasetv2

%%capture
! mkdir -p data/unprocessed
! unzip -o sroie-datasetv2.zip -d data/unprocessed
! rm sroie-datasetv2.zip

import os

len(os.listdir('data/unprocessed/SROIE2019/train/img')),
len(os.listdir('data/unprocessed/SROIE2019/test/img'))
```

```
(626, 347)
```

The train dataset contains 626 images and the test dataset contains 347 images.

Initial Data Structure

The downloaded dataset contains two major folders:

1. **train**: This folder contains the training data.
2. **test**: This folder contains the test data.

Each of these folders contains the following subfolders:

1. **img**: This folder containing the images of the receipts.
2. **box**: This folder containing the bounding box information of the text in the images and the text inside the bounding boxes. The file is a text file with the same name as the image file but with a .txt extension. The format of the file is as follows: `x1, y1, x2, y2, x3, y3, x4, y4, text` where (x1, y1), (x2, y2), (x3, y3), and (x4, y4) are the coordinates of the bounding box and text is the text inside the bounding box.
3. **entities**: This folder contains the ground truth information of the entities in the receipts. The file is a text file with the same name as the image file but with a .txt extension. The file contains json data with the following format: `json     {         "company": "COMPANY_NAME",         "date": "DATE",         "address": "ADDRESS",         "total": "TOTAL",     }`

Data Preprocessing Steps

We will divide the data preprocessing steps into two major parts:

1. **Data Collection**: In this step we will gather all the information spread across the three subfolders and create a single dataframe with the following columns:
   - `image_path`: The path to the image file.
   - `bboxes`: A list of bounding boxes in the image. Each bounding box is a list of 4 coordinates (x1, y1, x2, y2) where (x1, y1) is the top-left corner and (x2, y2) is the bottom-right corner of the bounding box. The coordinates are normalized between 0 and 1000 as required by the LayoutLMv3 model.
   - `words`: A list of words in the image corresponding to the bounding boxes.
   - `ner_tags`: A list of named entity recognition tags for each word in the image. The tags will encoded as numbers. The possible tags are:
     - **O**: The word is not part of any named entity. (Other)
     - **B-COMPANY**: The word is the company name.
     - **B-DATE**: The word is the date.
     - **B-ADDRESS**: The word is the address.
     - **B-TOTAL**: The word is the total.
2. **Data Processing**: The goal of the step is to create a huggingface dataset that can be consumed by the LayoutLMv3 model. The steps involved in this process are:
   - **Reading Images**: We will read the images and resize them to a fixed size.

- **Tokenization**: We will tokenize the words in the images using the LayoutLMv3 tokenizer.
- **Adding Padding**: We will add padding to the tokenized words to make them of the same length.

Data Collection

We will start by creating a list of all possible ner tags and then create a dictionary to map the tags to their corresponding indices and vice versa.

```
labels_list = ['O', 'B-COMPANY', 'B-DATE', 'B-ADDRESS', 'B-TOTAL']
ids2labels = {k: v for k, v in enumerate(labels_list)}
labels2ids = {v: k for k, v in enumerate(labels_list)}
```

The biggest challenge in preprocessing the data is to map each bounding box to the corresponding label. The text provided in the entities file is not always the same as the text in the bounding boxes. Thus, we will use the following approach to achieve this:

1. If SequenceMatcher ratio between the bounding box text and an entity is greater than 0.8, we will assign the corresponding tag to the bounding box text.
2. If a word in bounding box includes the date or total, we will assign the corresponding tag to the word. This is because the date and total are usually written in the same line with other words. For example, the total is usually written as "Total: 100.00" or "Total 100.00" and the date is usually written as "Date: 2022-01-01" or "Date 2022-01-01".
3. If a word in bounding box is a part of the company name or address, we will assign the corresponding tag to the word. This is because the company name and address are usually written in multiple lines and the bounding boxes may not cover the entire text.

```
from difflib import SequenceMatcher

def get_word_tag(word, entities):
  for entity in entities:
    if SequenceMatcher(None, word.lower(),
entities[entity].lower()).ratio() >= 0.8:
      return labels2ids[f'B-{entity}']
    elif entity in ['ADDRESS', 'COMPANY'] and word.lower() in
entities[entity].lower():
      return labels2ids[f'B-{entity}']
    elif entity in ['DATE', 'TOTAL'] and entities[entity].lower() in
word.lower():
      return labels2ids[f'B-{entity}']
  return labels2ids['O']
```

Each line in the box file contains the the bbox coordinates and the text in it seperated by a comma. We will create helper functions to parse the line and extract the bbox coordinates and the text. This function will also normalize the bbox coordinates between 0 and 1000.

```
def get_normalized_bbox(line, img_width, img_height):
  line = line.strip().split(',')
  x1, y1, x2, y2 = int(line[0]), int(line[1]), int(line[6]),
int(line[7])
  return [x1 / img_width * 1000, y1 / img_height * 1000, x2 /
img_width * 1000, y2 / img_height * 1000]

def get_word(line):
  line = line.strip().split(',')
  return ','.join(line[8:])
```

The entities file may not contain all the entities in the image. Thus, we will create a helper function to fill in the missing entities with empty strings. This will allow us to easily access all entities without having to check if the entity exists in the dictionary.

```
import json

def get_entities_dict(entities_file_path):
  entities_file = open(entities_file_path, 'r')
  entities = json.load(entities_file)
  entities_file.close()
  res_dict ={
    'COMPANY': entities.get('company', ''),
    'DATE': entities.get('date', ''),
    'ADDRESS': entities.get('address', ''),
    'TOTAL': entities.get('total', '')
  }
  return res_dict
```

Now we will create a function creates a dataframe from the data in the `train` and `test` folders. The function will take the path to the `train` and `test` folders as input and return a dataframe with the columns discussed earlier.

```
from PIL import Image
import pandas as pd
from pathlib import Path

def get_image_dimensions(img_path):
  img = Image.open(img_path)
  return img.size

def create_df(data_dir):
  box_path = os.path.join(data_dir, 'box')
  img_path = os.path.join(data_dir, 'img')
  entities_path = os.path.join(data_dir, 'entities')

  image_path_list = []
  bboxes_list = []
  words_list = []
  ner_tags_list = []
  for file in os.listdir(img_path):
    id = Path(file).stem

    box_file_path = os.path.join(box_path, f'{id}.txt')
    entities_file_path = os.path.join(entities_path, f'{id}.txt')

    img_width, img_height =
get_image_dimensions(os.path.join(img_path, file))

    try:
      with open(box_file_path, 'r') as f:
        lines = f.readlines()
        words = [get_word(line) for line in lines]
        bboxes = [get_normalized_bbox(line, img_width, img_height) for
line in lines]

      entities = get_entities_dict(entities_file_path)
      ner_tags = [get_word_tag(word, entities) for word in words]

      image_path_list.append(os.path.join(img_path, file))
      bboxes_list.append(bboxes)
      words_list.append(words)
      ner_tags_list.append(ner_tags)
    except:
      pass

  df = pd.DataFrame({
      'image_path': image_path_list,
      'bboxes': bboxes_list,
      'words': words_list,
      'ner_tags': ner_tags_list
  })

  return df

train_df = create_df('data/unprocessed/SROIE2019/train')
test_df = create_df('data/unprocessed/SROIE2019/test')
len(train_df), len(test_df)
```

```
(626, 345)
```

We notice that one test receipt has been dropped. This file contains Euro sign which isn't supported by utf-8. I decided to drop this image by adding the try except block around the code that reads the file. This will allow us to skip the file and continue with the rest of the images.

To get a sense of our labelling stragey, we will calculate how many of each tag we have assigned to the words in the images.

```
def create_stats_df(df):
  stats_not_other = df['ner_tags'].apply(lambda x: sum([1 for tag in x
if tag != labels2ids['O']]))
  stats_total = df['ner_tags'].apply(lambda x: sum([1 for tag in x if
tag == labels2ids['B-TOTAL']]))
  stats_address  = df['ner_tags'].apply(lambda x: sum([1 for tag in x
if tag == labels2ids['B-ADDRESS']]))
  stats_company = df['ner_tags'].apply(lambda x: sum([1 for tag in x
if tag == labels2ids['B-COMPANY']]))
  stats_date = df['ner_tags'].apply(lambda x: sum([1 for tag in x if
tag == labels2ids['B-DATE']]))
  stats_df = pd.DataFrame({
      'total': stats_total,
      'address': stats_address,
      'company': stats_company,
      'date': stats_date,
      'not_other': stats_not_other
  })
  return stats_df

stats_df = create_stats_df(train_df)
stats_df.describe()

            total     address     company        date   not_other
count  626.000000  626.000000  626.000000  626.000000  626.000000
mean     3.107029    4.966454    1.305112    1.214058   10.592652
std      2.065267    4.195673    1.103249    0.440575    4.888374
min      1.000000    0.000000    0.000000    0.000000    2.000000
25%      2.000000    3.000000    1.000000    1.000000    8.000000
50%      3.000000    4.000000    1.000000    1.000000    9.000000
75%      4.000000    5.750000    1.000000    1.000000   12.000000
max     33.000000   25.000000    9.000000    3.000000   39.000000
```

From the results in the table, we find

- The median number of bounding boxes assigned to `company` tag is 1. This implies the success of our labelling strategy.
- The median number of bounding boxes assigned to `date` tag is 1. This also implies the success of our labelling strategy.
- The median number of bounding boxes assigned to `address` tag is a whooping 4. The most likely reason for this is that the address is usually written in multiple lines and the bounding boxes may not cover the entire text. Moreover, the address may be written multiple times in the receipt.
- The median number of bounding boxes assigned to `total` tag is 3. One possible explanation is that the total is usually written multiple times in the receipt. For example, if you buy only one item, the total value will be written once as item price and once as the total price. Other receipts may include total price after discount and total price before discount which are very likely to be the same.

We want our model to take an image and return a json object with the following format:

```
{
    "company": "COMPANY_NAME",
    "date": "DATE",
    "address": "ADDRESS",
    "total": "TOTAL",
}
```

Our model may predict the same entity multiple times. Thus we need a stragey to pick only one bounding box for each entity. We will use the following stragey:

- **Company**: We will pick the largest bounding box assigned to the company tag. This is because the company name is usually written with a larger font size than other text in the receipt.
- **Date**: We will pick the bounding box assigned to the date tag that is closest to the top of the image. This is because the date is usually written at the top of the receipt.
- **Address**: We will once again pick the largest bounding box assigned to the address tag. The LayoutLMv3 processor won't read text on seperate lines in the same box. Hence, it is unlikely for us to have one bounding box including the entire address. Our best bet is to pick the largest bounding box hoping it has the most important part of the address.
- **Total**: We will pick the bounding box assigned to the total tag that is closest to the bottom of the image. This is because the total is usually written at the bottom of the receipt. Moreover, if some of the cases where the total is written multiple times occur, the total at the bottom is the most likely to be the final total.

To enhance the performance of our model, we will add this information the training and test datasets. We will create a function that takes the bounding boxes and the ner tags and recreates the tags such that at most one bounding box is assigned to each entity.

```
def recompute_ner_tags(row):
  ner_tags = row['ner_tags']
  bboxes = row['bboxes']

  new_ner_tags = [labels2ids['O'] for _ in ner_tags]

  # Give company tag only to the largest bounding box
  company_idx = [i for i, tag in enumerate(ner_tags) if tag ==
labels2ids['B-COMPANY']]
  if len(company_idx) > 0:
    max_idx = max(company_idx, key=lambda x: (bboxes[x][2] - bboxes[x]
[0]) * (bboxes[x][3] - bboxes[x][1]))
    new_ner_tags[max_idx] = labels2ids['B-COMPANY']

  # Give address tag only to the largest bounding box
  address_idx = [i for i, tag in enumerate(ner_tags) if tag ==
labels2ids['B-ADDRESS']]
  if len(address_idx) > 0:
    max_idx = max(address_idx, key=lambda x: (bboxes[x][2] - bboxes[x]
[0]) * (bboxes[x][3] - bboxes[x][1]))
    new_ner_tags[max_idx] = labels2ids['B-ADDRESS']
  # Give date tag to the earliest bounding box
  date_idx = [i for i, tag in enumerate(ner_tags) if tag ==
labels2ids['B-DATE']]
  if len(date_idx) > 0:
    min_idx = min(date_idx, key=lambda x: (bboxes[x][1] + bboxes[x]
[3])/2)
    new_ner_tags[min_idx] = labels2ids['B-DATE']

  # Give total tag to the last bounding box
  total_idx = [i for i, tag in enumerate(ner_tags) if tag ==
labels2ids['B-TOTAL']]
  if len(total_idx) > 0:
    max_idx = max(total_idx, key=lambda x: (bboxes[x][1] + bboxes[x]
[3])/2)
    new_ner_tags[max_idx] = labels2ids['B-TOTAL']

  return new_ner_tags

train_df['ner_tags'] = train_df.apply(recompute_ner_tags, axis=1)
test_df['ner_tags'] = test_df.apply(recompute_ner_tags, axis=1)

train_stats_df = create_stats_df(train_df)
train_stats_df.describe()
```

```
       total     address     company        date   not_other
count  626.0  626.000000  626.000000  626.000000  626.000000
mean     1.0    0.992013    0.880192    0.996805    3.869010
std      0.0    0.089085    0.324997    0.056478    0.342365
min      1.0    0.000000    0.000000    0.000000    2.000000
25%      1.0    1.000000    1.000000    1.000000    4.000000
50%      1.0    1.000000    1.000000    1.000000    4.000000
75%      1.0    1.000000    1.000000    1.000000    4.000000
max      1.0    1.000000    1.000000    1.000000    4.000000
```

```
test_stats_df = create_stats_df(test_df)
test_stats_df.describe()
```

```
            total     address     company        date   not_other
count  345.000000  345.000000  345.000000  345.000000  345.000000
mean     0.997101    0.985507    0.881159    0.991304    3.855072
std      0.053838    0.119684    0.324071    0.092979    0.360691
min      0.000000    0.000000    0.000000    0.000000    2.000000
25%      1.000000    1.000000    1.000000    1.000000    4.000000
50%      1.000000    1.000000    1.000000    1.000000    4.000000
75%      1.000000    1.000000    1.000000    1.000000    4.000000
max      1.000000    1.000000    1.000000    1.000000    4.000000
```

Now that's much better. The table shows the median number of bounding boxes assigned to any tag is 1. This will make our model better able to understand the receipt.

Data Processing

I will start by directly converting the dataframes to huggingface datasets.

```
from datasets import Dataset

train_dataset = Dataset.from_pandas(train_df)
test_dataset = Dataset.from_pandas(test_df)

train_dataset, test_dataset
```

```
(Dataset({
     features: ['image_path', 'bboxes', 'words', 'ner_tags'],
     num_rows: 626
 }),
 Dataset({
     features: ['image_path', 'bboxes', 'words', 'ner_tags'],
     num_rows: 345
 }))
```

Now, we will instantiate the LayoutLMv3 processor to tokenize the words in the images, resize the images, and add padding to the tokenized words.

We will add the argument `apply_ocr=False` to the processor to avoid applying OCR on the images. While fine-tuning the model, we will train the head of the model on the annotated data. If we apply OCR on the images, we won't have the labels for the extracted words. However, we will apply OCR on the images when we use the model for inference.

```
from transformers import AutoProcessor

model_id = "microsoft/layoutlmv3-base"

processor = AutoProcessor.from_pretrained(model_id, apply_ocr=False)
```

```
/home/ibrahim/anaconda3/envs/d2l/lib/python3.9/site-packages/
transformers/tokenization_utils_base.py:1601: FutureWarning:
`clean_up_tokenization_spaces` was not set. It will be set to `True`
by default. This behavior will be depracted in transformers v4.45, and
will be then set to `False` by default. For more details check this
issue: https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
```

We are going to create a function that maps the unprocessed data to the processed data. The function will take the unprocessed examples and use the processor to create the processed examples.

```
def prepare_examples(examples):
  images = [Image.open(path).convert('RGB') for path in
examples["image_path"]]
```

```
  words = examples["words"]
  bboxes = examples["bboxes"]
  ner_tags = examples["ner_tags"]
  encoding = processor(images, words, boxes=bboxes,
word_labels=ner_tags, padding="max_length", truncation=True)
  return encoding
```

Before we apply the mapping, we are going to define the features of the processed dataset.

Note that 224 x 224 is the image size expected by the LayoutLMv3 model and 512 is the maximum sequence length expected by the model.

```
from datasets import Features, Sequence, Value, Array2D, Array3D

features = Features({
  'pixel_values': Array3D(dtype="float32", shape=(3, 224, 224)),
  'input_ids': Sequence(feature=Value(dtype='int64')),
  'attention_mask': Sequence(Value(dtype='int64')),
  'bbox': Array2D(dtype="int64", shape=(512, 4)),
  'labels': Sequence(feature=Value(dtype='int64')),
})
```

Finally, we will apply the mapping to the train and test datasets and save them to disk.

Notice that I have changed the default `batch_size` to 32. The deafult batch size is 1000. Loading a 1000 image simultaneously will consume a lot of memory. I have reduced the batch size to 32 to avoid memory issues. You can increase the batch size if you have enough memory.

```
train_dataset = train_dataset.map(prepare_examples, batched=True,
remove_columns=train_dataset.column_names, batch_size=32,
features=features)
test_dataset = test_dataset.map(prepare_examples, batched=True,
remove_columns=test_dataset.column_names, batch_size=32,
features=features)
```

{"model_id":"3acdac40e12f4eb2aedb60e1973120e7","version_major":2,"version_minor":0}

{"model_id":"298af4dff97b44e68e35c61c239a0a5c","version_major":2,"version_minor":0}

```
train_dataset.save_to_disk('data/train')
test_dataset.save_to_disk('data/test')
```

{"model_id":"2cb4c017ed9c4a11b6624bbd9ee3c179","version_major":2,"version_minor":0}

{"model_id":"10d63d3105a34750b3e6ad2804871eed","version_major":2,"version_minor":0}

Summary

In this notebook, we have preprocessed the SROIE dataset to create a train and test dataset that can be consumed by the LayoutLMv3 model. We have created a dataframe with the necessary information and then processed the data to create a huggingface dataset. The processed dataset is saved to disk and can be used to fine-tune the LayoutLMv3 model. Now we are ready to the next step where we build the receipt understanding model.