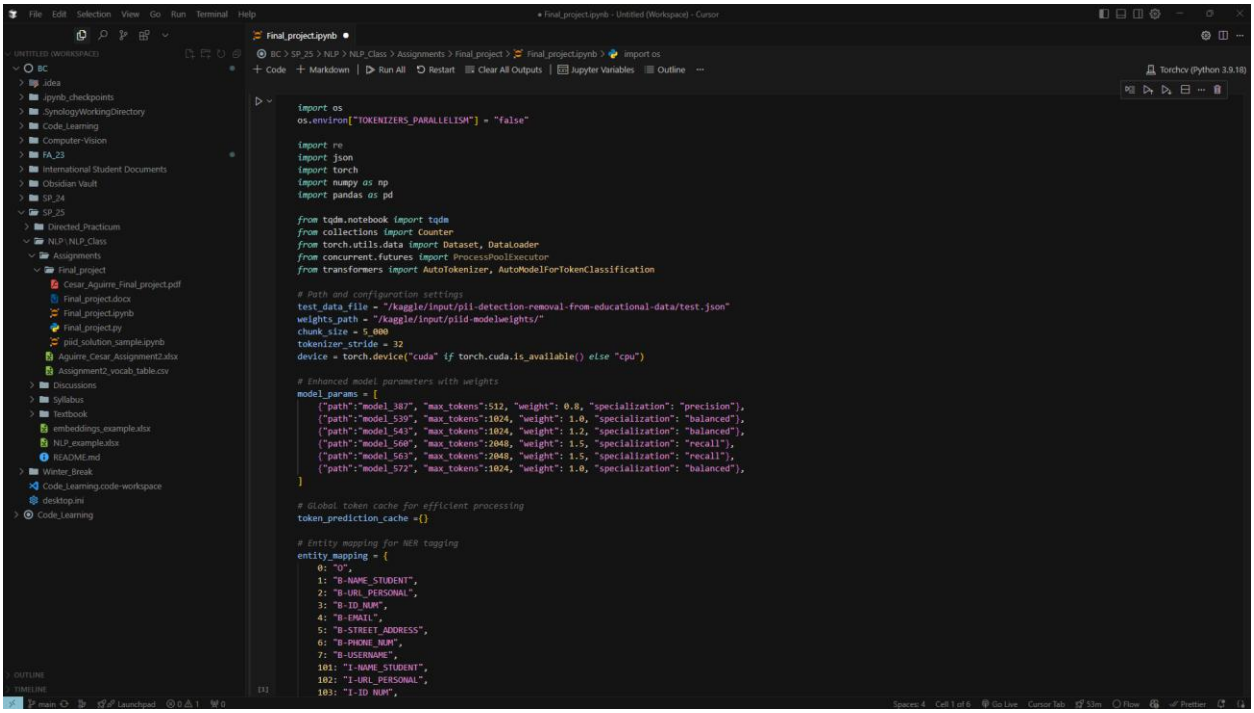
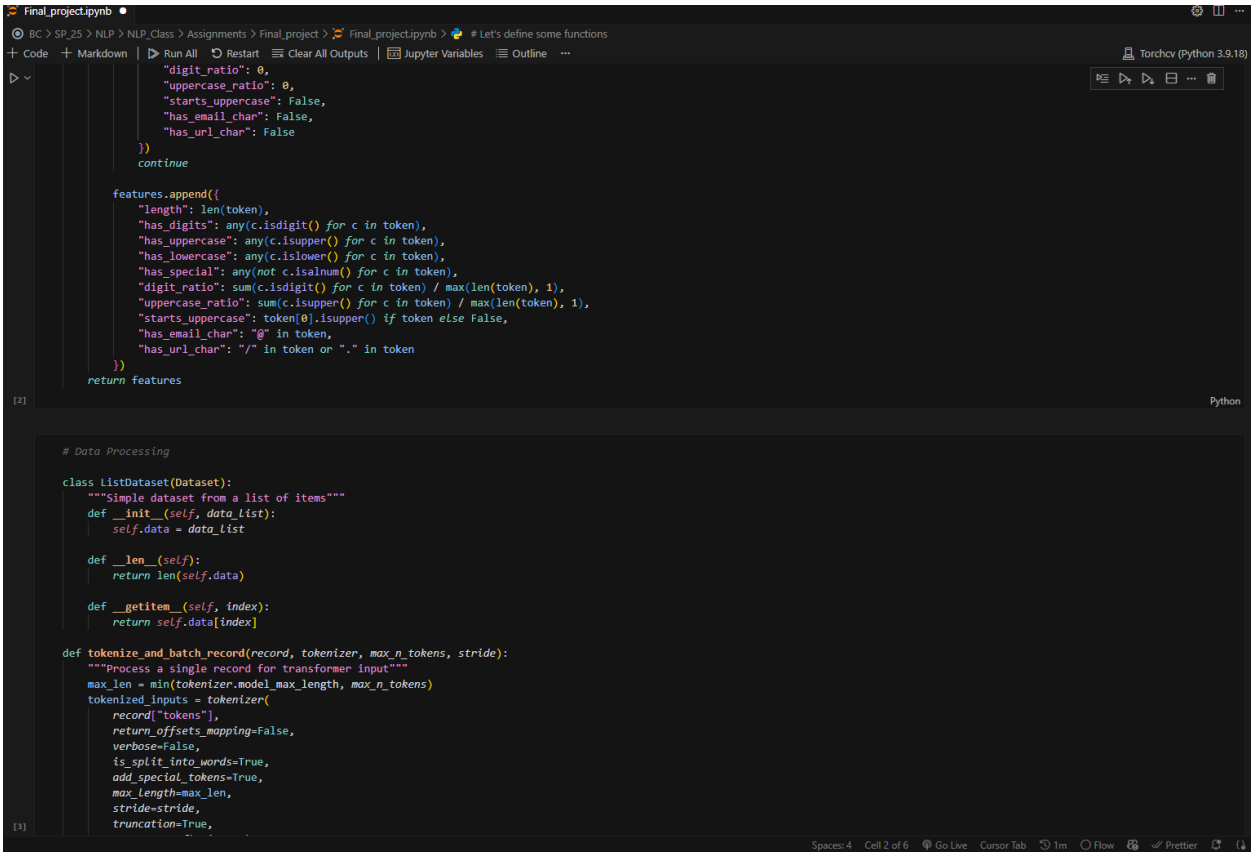
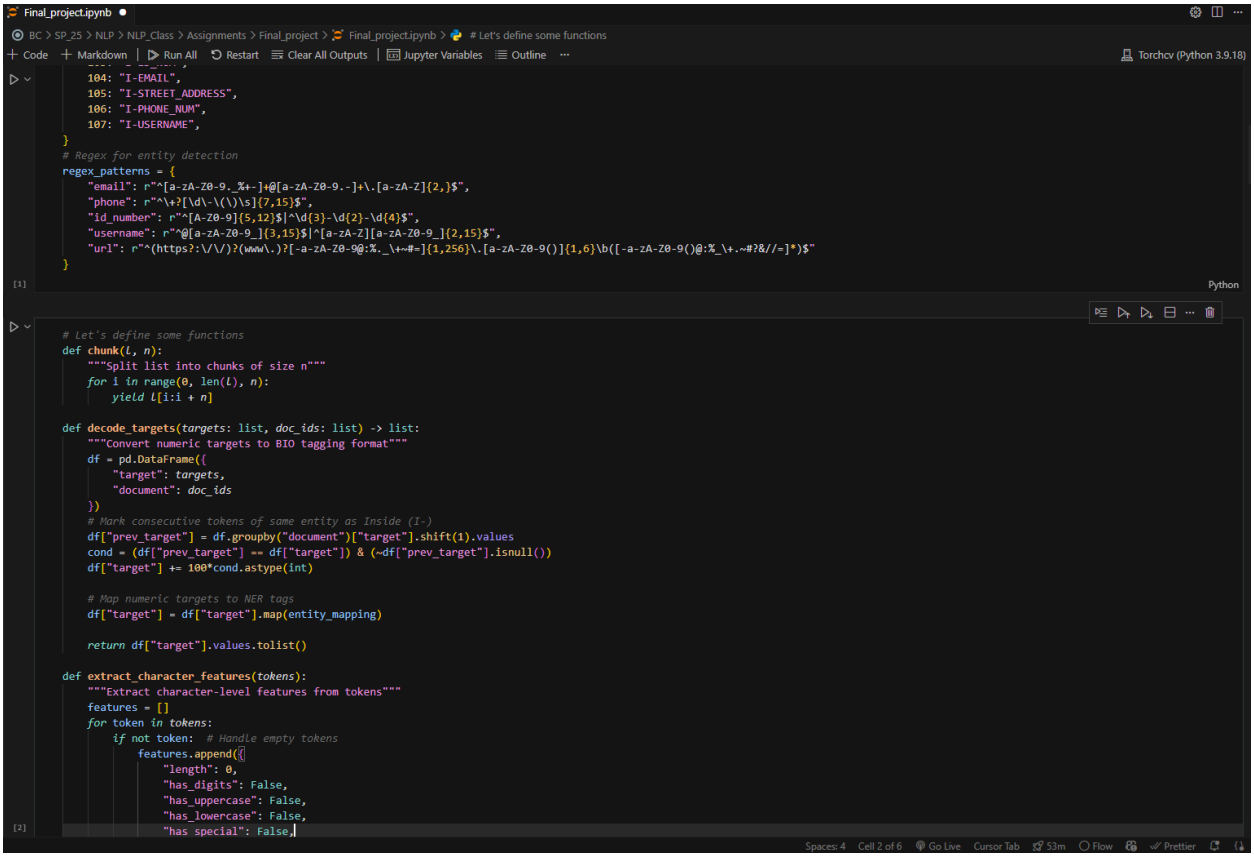


```
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"

import re
import json
import torch
import numpy as np
import pandas as pd

from tqdm.notebook import tqdm
from collections import Counter
from torch.utils.data import Dataset, DataLoader
from concurrent.futures import ProcessPoolExecutor
from transformers import AutoTokenizer, AutoModelForTokenClassification

# Path and configuration settings
test_data_file = "/kaggle/input/pii-detection-removal-from-educational-data/test.json"
weights_path = "/kaggle/input/piid-modelweights/"
chunk_size = 5_000
tokenizer_stride = 32
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Enhanced model parameters with weights
model_params = [
    {"path":"model_387", "max_tokens":512, "weight": 0.8, "specialization": "precision"},
    {"path":"model_539", "max_tokens":1024, "weight": 1.0, "specialization": "balanced"},
    {"path":"model_543", "max_tokens":1024, "weight": 1.2, "specialization": "balanced"},
    {"path":"model_560", "max_tokens":2048, "weight": 1.5, "specialization": "recall"},
    {"path":"model_563", "max_tokens":2048, "weight": 1.5, "specialization": "recall"},
    {"path":"model_572", "max_tokens":1024, "weight": 1.0, "specialization": "balanced"},
]

# Global token cache for efficient processing
token_prediction_cache ={}

# Entity mapping for NER tagging
entity_mapping = {
    0: "O",
    1: "B-NAME_STUDENT",
    2: "B-URL_PERSONAL",
    3: "B-ID_NUM",
    4: "B-EMAIL",
    5: "B-STREET_ADDRESS",
    6: "B-PHONE_NUM",
    7: "B-USERNAME",
    101: "I-NAME_STUDENT",
    102: "I-URL_PERSONAL",
    103: "I-ID_NUM",
```



```
                "digit_ratio": 0,
                "uppercase_ratio": 0,
                "starts_uppercase": False,
                "has_email_char": False,
                "has_url_char": False
            })
            continue

        features.append({
            "length": len(token),
            "has_digits": any(c.isdigit() for c in token),
            "has_uppercase": any(c.isupper() for c in token),
            "has_lowercase": any(c.islower() for c in token),
            "has_special": any(not c.isalnum() for c in token),
            "digit_ratio": sum(c.isdigit() for c in token) / max(len(token), 1),
            "uppercase_ratio": sum(c.isupper() for c in token) / max(len(token), 1),
            "starts_uppercase": token[0].isupper() if token else False,
            "has_email_char": "@" in token,
            "has_url_char": "/" in token or "." in token
        })
    return features
```

```
# Data Processing

class ListDataset(Dataset):
    """Simple dataset from a list of items"""
    def __init__(self, data_list):
        self.data = data_list

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

    def __getitem__(self, index):
        return self.data[index]

def tokenize_and_batch_record(record, tokenizer, max_n_tokens, stride):
    """Process a single record for transformer input"""
    max_len = min(tokenizer.model_max_length, max_n_tokens)
    tokenized_inputs = tokenizer(
        record["tokens"],
        return_offsets_mapping=False,
        verbose=False,
        is_split_into_words=True,
        add_special_tokens=True,
        max_length=max_len,
        stride=stride,
        truncation=True,
```


Final_project.ipynb
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
Torchcv (Python 3.9.18)
"I-EMAIL",
"I-STREET_ADDRESS",
"I-PHONE_NUM",
"I-USERNAME",
}
# Regex for entity detection
regex_patterns = {
"email": r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$",
"phone": r"^\+?[\d\-\(\)\s]{7,15}$",
"id_number": r"^[A-Z0-9]{5,12}$|^\d{3}-\d{2}-\d{4}$",
"username": r"^@[a-zA-Z0-9_]{3,15}$|^[a-zA-Z][a-zA-Z0-9_]{2,15}$",
"url": r"^(https?:\/\/)?(www\.)?[-a-zA-Z0-9@:%._\+~#=]{1,256}\.[a-zA-Z0-9()]{1,6}\b([-a-zA-Z0-9()@:%_\+.~#?&//=]*)$"
}
[1]
Python
# Let's define some functions
def chunk(l, n):
"""Split list into chunks of size n"""
for i in range(0, len(l), n):
yield l[i:i + n]
def decode_targets(targets: list, doc_ids: list) -> list:
"""Convert numeric targets to BIO tagging format"""
df = pd.DataFrame({
"target": targets,
"document": doc_ids
})
Mark consecutive tokens of same entity as Inside (I-)
df["prev_target"] = df.groupby("document")["target"].shift(1).values
cond = (df["prev_target"] == df["target"]) & (~df["prev_target"].isnull())
df["target"] += 100*cond.astype(int)
Map numeric targets to NER tags
df["target"] = df["target"].map(entity_mapping)
return df["target"].values.tolist()
def extract_character_features(tokens):
"""Extract character-level features from tokens"""
features = []
for token in tokens:
if not token: # Handle empty tokens
features.append({
"length": 0,
"has_digits": False,
"has_uppercase": False,
"has_lowercase": False,
"has special": False,
[2]
Spaces: 4
Cell 2 of 6
Go Live
Cursor Tab
53m
Flow
Prettier

Final_project.ipynb

BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions

+ Code + Markdown | Run All Restart Clear All Outputs | Jupyter Variables Outline ... Torchcv (Python 3.9.18)

```
            truncation=True,
            return_overflowing_tokens=True
        )

        # Extract character-level features
        char_features = extract_character_features(record["tokens"])

        # Create a targets map if labels exist
        if "labels" in record:
            # Note: encode_targets function is assumed to be defined elsewhere
            targets_map = {i:v for i,v in enumerate(encode_targets(record["labels"]))}
        else:
            targets_map = {}

        # Create batches from tokenized inputs
        batch = [{
            "input_ids": tokenized_inputs["input_ids"][i],
            "attention_mask": tokenized_inputs["attention_mask"][i],
            "word_ids": [-100 if x is None else x for x in tokenized_inputs.word_ids(i)],
            "targets": [targets_map.get(x, -100) for x in tokenized_inputs.word_ids(i)],
            "document": [record["document"]] * len(tokenized_inputs["input_ids"][i]),
            # Add token texts for rule-based processing
            "tokens": [record["tokens"][x] if x is not None and x < len(record["tokens"]) else ""
                       for x in tokenized_inputs.word_ids(i)],
        } for i in range(len(tokenized_inputs["input_ids"]))]

        return batch

    def tokenize_and_batch(sample, tokenizer, max_n_tokens, stride):
        """Process a batch of records with progress tracking"""
        tokenized_sample = [tokenize_and_batch_record(rec, tokenizer, max_n_tokens, stride) for rec in tqdm(sample)]
        tokenized_sample = [x for xs in tokenized_sample for x in xs]
        return tokenized_sample

    def collate_fn(batch):
        """Combine individual samples into batches with padding"""
        keys = batch[0].keys()
        # Special handling for non-tensor data
        non_tensor_keys = ["tokens"]
        tensor_keys = [k for k in keys if k not in non_tensor_keys]

        # Process tensor data
        seq = {k:[torch.tensor(x[k]) for x in batch] for k in tensor_keys}
        result = {k:torch.nn.utils.rnn.pad_sequence(v, True, 0) for k,v in seq.items()}

        # Process non-tensor data
        for k in non_tensor_keys:
            result[k] = [item for x in batch for item in x[k]]

[3]
        return result
```

Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow

Final_project.ipynb

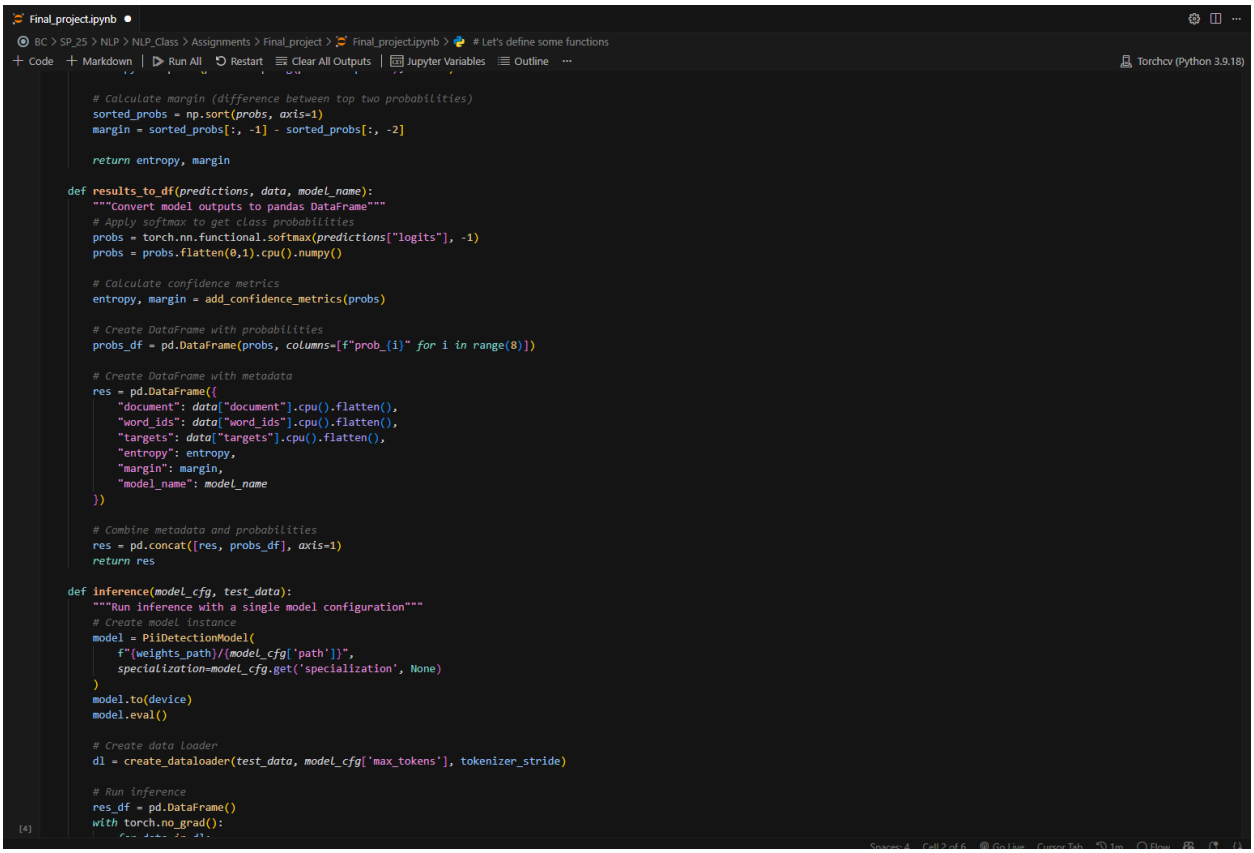
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions

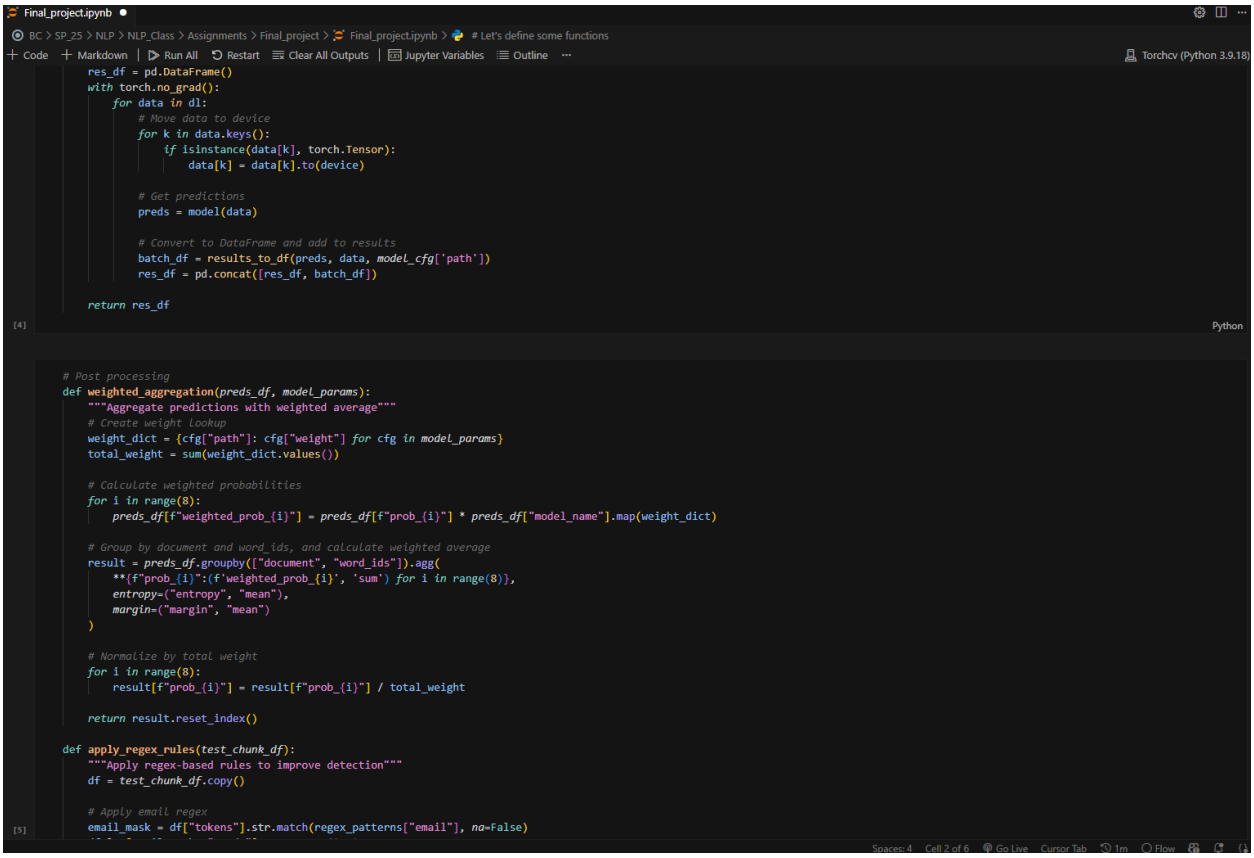
+ Code + Markdown | Run All Restart Clear All Outputs | Jupyter Variables Outline ... Torchcv (Python 3.9.18)

```
    def create_dataloader(test_data, max_n_tokens, tokenizer_stride):
        """Create a PyTorch DataLoader"""
        tokenizer = AutoTokenizer.from_pretrained(weights_path + "tokenizer")
        tokenized_data = tokenize_and_batch(test_data, tokenizer, max_n_tokens, tokenizer_stride)
        return DataLoader(
            ListDataset(tokenized_data),
            batch_size = 4,
            num_workers = 2,
            pin_memory = True,
            shuffle = False,
            collate_fn = collate_fn,
        )
```

[3] Python

```
    # Model Definition
    class PiiDetectionModel(torch.nn.Module):
        """Enhanced wrapper around pre-trained token classification model"""
        def __init__(self, load_path, specialization=None):
            super().__init__()
            self.backbone = AutoModelForTokenClassification.from_pretrained(load_path)
            self.specialization = specialization

        def forward(self, d):
            """Run inference through the model"""
            preds = self.backbone(
                input_ids=d["input_ids"],
                attention_mask=d["attention_mask"]
            )

            # Apply specialization adjustments if needed
            logits = preds.logits
            if self.specialization == "precision":
                # Slightly reduce probability of positive predictions to favor precision
                logits[:, :, 1:] -= 0.2
            elif self.specialization == "recall":
                # Slightly boost probability of positive predictions to favor recall
                logits[:, :, 1:] += 0.1

            return {"logits": logits}

    def add_confidence_metrics(probs):
        """Calculate confidence metrics for predictions"""
        # Calculate entropy (lower means more confident)
        epsilon = 1e-10  # To avoid log(0)
        entropy = -np.sum(probs * np.log(probs + epsilon), axis=1)

[4]
        # Calculate margin (difference between top two probabilities)
```


Final_project.ipynb
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
Torchcv (Python 3.9.18)
# Calculate margin (difference between top two probabilities)
sorted_probs = np.sort(probs, axis=1)
margin = sorted_probs[:, -1] - sorted_probs[:, -2]
return entropy, margin
def results_to_df(predictions, data, model_name):
"""Convert model outputs to pandas DataFrame"""
# Apply softmax to get class probabilities
probs = torch.nn.functional.softmax(predictions["logits"], -1)
probs = probs.flatten(0,1).cpu().numpy()
# Calculate confidence metrics
entropy, margin = add_confidence_metrics(probs)
# Create DataFrame with probabilities
probs_df = pd.DataFrame(probs, columns=[f"prob_{i}" for i in range(8)])
# Create DataFrame with metadata
res = pd.DataFrame({
"document": data["document"].cpu().flatten(),
"word_ids": data["word_ids"].cpu().flatten(),
"targets": data["targets"].cpu().flatten(),
"entropy": entropy,
"margin": margin,
"model_name": model_name
})
# Combine metadata and probabilities
res = pd.concat([res, probs_df], axis=1)
return res
def inference(model_cfg, test_data):
"""Run inference with a single model configuration"""
# Create model instance
model = PiiDetectionModel(
f"{weights_path}/{model_cfg['path']}",
specialization=model_cfg.get('specialization', None)
)
model.to(device)
model.eval()
# Create data loader
dl = create_dataloader(test_data, model_cfg['max_tokens'], tokenizer_stride)
# Run inference
res_df = pd.DataFrame()
with torch.no_grad():
[4]


Final_project.ipynb
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions
Code
Markdown
Run All
Restart
Clear All Outputs
Jupyter Variables
Outline
Torchcv (Python 3.9.18)
res_df = pd.DataFrame()
with torch.no_grad():
for data in dl:
# Move data to device
for k in data.keys():
if isinstance(data[k], torch.Tensor):
data[k] = data[k].to(device)
# Get predictions
preds = model(data)
# Convert to DataFrame and add to results
batch_df = results_to_df(preds, data, model_cfg['path'])
res_df = pd.concat([res_df, batch_df])
return res_df
[4]
Python
# Post processing
def weighted_aggregation(preds_df, model_params):
"""Aggregate predictions with weighted average"""
# Create weight lookup
weight_dict = {cfg["path"]: cfg["weight"] for cfg in model_params}
total_weight = sum(weight_dict.values())
# Calculate weighted probabilities
for i in range(8):
preds_df[f"weighted_prob_{i}"] = preds_df[f"prob_{i}"] * preds_df["model_name"].map(weight_dict)
# Group by document and word_ids, and calculate weighted average
result = preds_df.groupby(["document", "word_ids"]).agg(
**{f"prob_{i}":(f'weighted_prob_{i}', 'sum') for i in range(8)},
entropy=("entropy", "mean"),
margin=("margin", "mean")
)
# Normalize by total weight
for i in range(8):
result[f"prob_{i}"] = result[f"prob_{i}"] / total_weight
return result.reset_index()
def apply_regex_rules(test_chunk_df):
"""Apply regex-based rules to improve detection"""
df = test_chunk_df.copy()
# Apply email regex
email_mask = df["tokens"].str.match(regex_patterns["email"], na=False)
[5]

Final_project.ipynb

BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions

Code | Markdown | Run All | Restart | Clear All Outputs | Jupyter Variables | Outline | ... | Torchcv (Python 3.9.18)

```
        # Apply email regex
        email_mask = df["tokens"].str.match(regex_patterns["email"], na=False)
        df.loc[email_mask, "preds"] = 4  # Email class

        # Apply phone regex
        phone_mask = df["tokens"].str.match(regex_patterns["phone"], na=False)
        df.loc[phone_mask, "preds"] = 6  # Phone class

        # Apply ID regex
        id_mask = df["tokens"].str.match(regex_patterns["id_number"], na=False)
        df.loc[id_mask, "preds"] = 3  # ID class

        # Apply username regex
        username_mask = df["tokens"].str.match(regex_patterns["username"], na=False)
        df.loc[username_mask, "preds"] = 7  # Username class

        # Apply URL regex
        url_mask = df["tokens"].str.match(regex_patterns["url"], na=False)
        df.loc[url_mask, "preds"] = 2  # URL class

        return df

    def context_aware_processing(df):
        """Apply rules that consider token context"""
        result = df.copy()

        # Create columns for previous and next tokens
        result["prev_token"] = result.groupby("document")["tokens"].shift(1)
        result["next_token"] = result.groupby("document")["tokens"].shift(-1)

        # Rule 1: Improve name detection with common name parts
        common_titles = ["Mr", "Mrs", "Ms", "Dr", "Prof"]
        is_after_title = result["prev_token"].isin(common_titles)
        is_capitalized = result["tokens"].str.istitle()
        likely_name = is_after_title & is_capitalized & (result["preds"] == 0)
        result.loc[likely_name, "preds"] = 1  # Mark as name

        # Rule 2: Improve address detection
        address_indicators = ["St", "Ave", "Rd", "Blvd", "Ln", "Dr", "Cir", "Apt"]
        preceded_by_number = result["prev_token"].str.match(r"^\d+$", na=False)
        followed_by_address_indicator = result["next_token"].isin(address_indicators)
        likely_street = preceded_by_number & followed_by_address_indicator
        result.loc[likely_street, "preds"] = 5  # Mark as address

        # Rule 3: Mark newlines as address components
        result.loc[result["tokens"] == "\n", "preds"] = 5

        # Rule 4: Clean up predictions - names should start with capital letter
        result.loc[(result["preds"] == 1) & (~result["tokens"].str.istitle()), "preds"] = 0
```

[5]

Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow

Final_project.ipynb

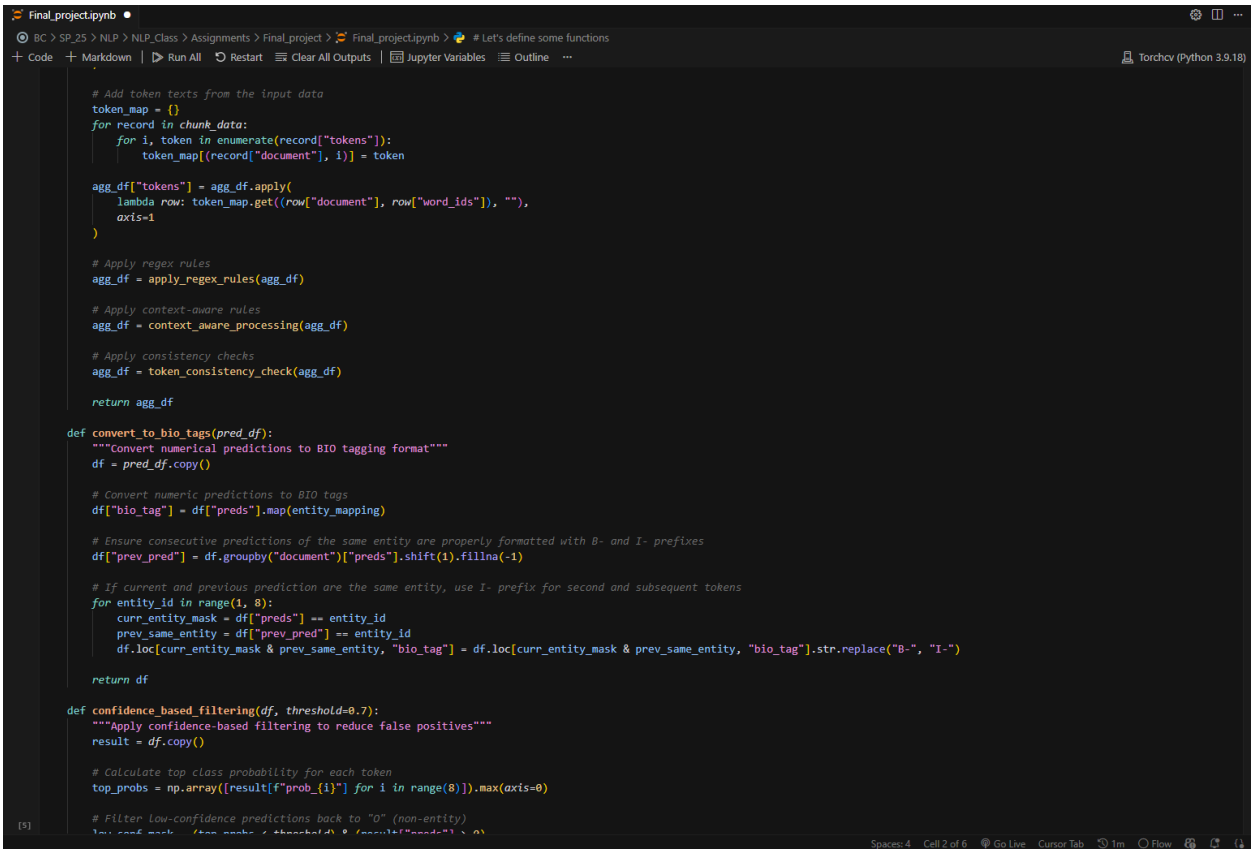
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions

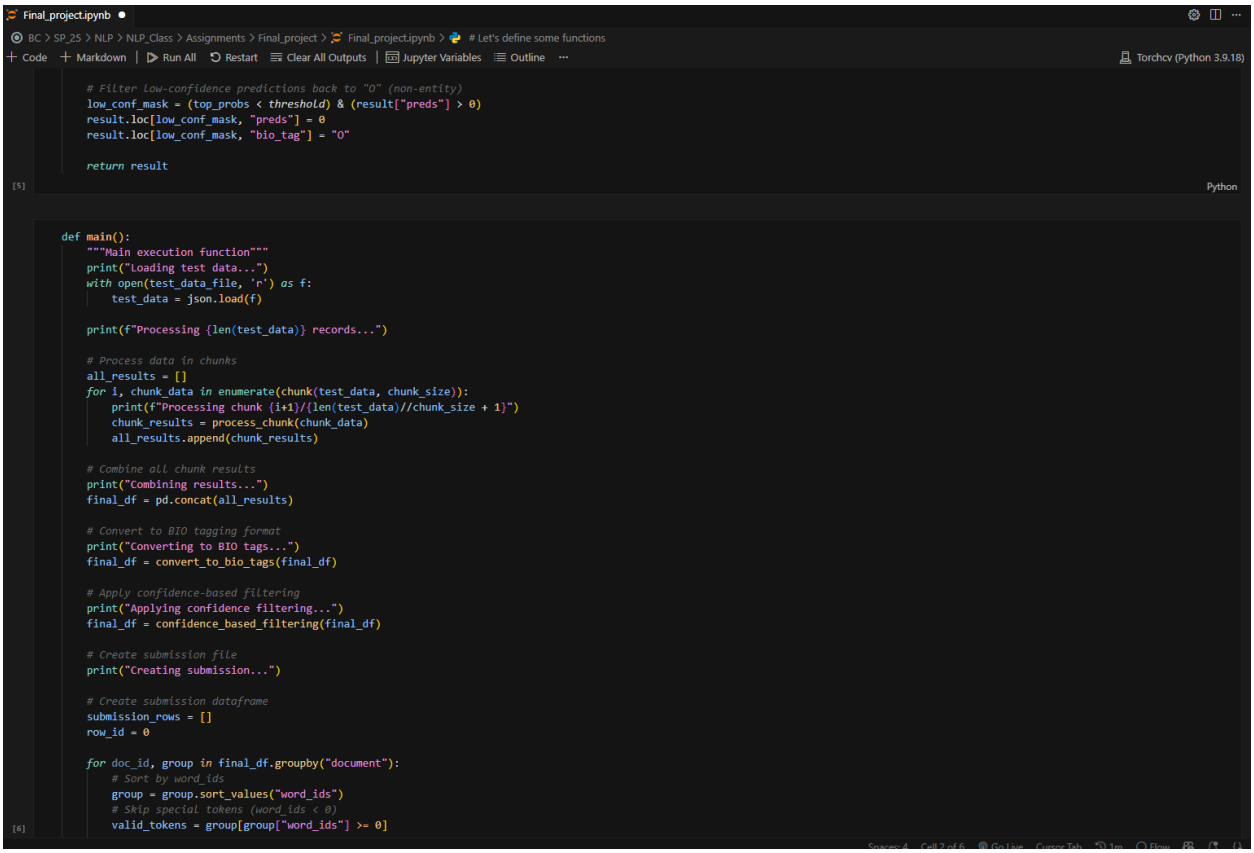
Code | Markdown | Run All | Restart | Clear All Outputs | Jupyter Variables | Outline | ... | Torchcv (Python 3.9.18)

```
        # Rule 5: Consecutive tokens with same entity should be marked consistently
        # Propagate entity labels to neighboring tokens if they look like part of the same entity
        for entity_id in [1, 2, 3, 4, 5, 6, 7]:  # All entity types
            entity_mask = result["preds"] == entity_id

            # Forward propagation
            for _ in range(2):  # Apply twice to handle longer entities
                next_entity_mask = result.groupby("document")["preds"].shift(-1) == entity_id
                propagate_forward = (~entity_mask) & next_entity_mask & result["tokens"].notna()
                result.loc[propagate_forward, "preds"] = entity_id + 100  # Mark as I-entity

            # Backward propagation
            for _ in range(2):
                prev_entity_mask = result.groupby("document")["preds"].shift(1) == entity_id
                propagate_backward = (~entity_mask) & prev_entity_mask & result["tokens"].notna()
                result.loc[propagate_backward, "preds"] = entity_id + 100  # Mark as I-entity

        return result

    def token_consistency_check(df):
        """Apply consistency checks across documents"""
        # Create a mapping of commonly detected tokens to their most frequent entity
        token_entity_counter = df.groupby("tokens")["preds"].apply(lambda x: Counter(x).most_common(1)[0] if len(x) > 0 else (0, 0))

        # Filter only for tokens that appear multiple times and are consistently predicted as entities
        token_entity_map = {
            token: entity for token, (entity, count) in token_entity_counter.items()
            if count >= 3 and entity > 0 and token and len(token) > 2
        }

        # Apply consistent labeling
        result = df.copy()
        for token, entity in token_entity_map.items():
            token_mask = (result["tokens"] == token) & (result["preds"] == 0)
            result.loc[token_mask, "preds"] = entity

        return result

    def process_chunk(chunk_data):
        """Process a chunk of test data through all models"""
        chunk_results = []

        # Run inference with each model configuration
        for model_cfg in model_params:
            chunk_df = inference(model_cfg, chunk_data)
            chunk_results.append(chunk_df)

        # Combine results from all models
        combined_df = pd.concat(chunk_results)
```

[5]

Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow


Final_project.ipynb
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions
+ Code + Markdown | Run All Restart Clear All Outputs | Jupyter Variables Outline ...
Torchcv (Python 3.9.18)
# Add token texts from the input data
token_map = {}
for record in chunk_data:
for i, token in enumerate(record["tokens"]):
token_map[(record["document"], i)] = token
agg_df["tokens"] = agg_df.apply(
lambda row: token_map.get((row["document"], row["word_ids"]), ""),
axis=1
)
# Apply regex rules
agg_df = apply_regex_rules(agg_df)
# Apply context-aware rules
agg_df = context_aware_processing(agg_df)
# Apply consistency checks
agg_df = token_consistency_check(agg_df)
return agg_df
def convert_to_bio_tags(pred_df):
"""Convert numerical predictions to BIO tagging format"""
df = pred_df.copy()
# Convert numeric predictions to BIO tags
df["bio_tag"] = df["preds"].map(entity_mapping)
Ensure consecutive predictions of the same entity are properly formatted with B- and I- prefixes
df["prev_pred"] = df.groupby("document")["preds"].shift(1).fillna(-1)
# If current and previous prediction are the same entity, use I- prefix for second and subsequent tokens
for entity_id in range(1, 8):
curr_entity_mask = df["preds"] == entity_id
prev_same_entity = df["prev_pred"] == entity_id
df.loc[curr_entity_mask & prev_same_entity, "bio_tag"] = df.loc[curr_entity_mask & prev_same_entity, "bio_tag"].str.replace("B-", "I-")
return df
def confidence_based_filtering(df, threshold=0.7):
"""Apply confidence-based filtering to reduce false positives"""
result = df.copy()
# Calculate top class probability for each token
top_probs = np.array([result[f"prob_{i}"] for i in range(8)]).max(axis=0)
# Filter low-confidence predictions back to "O" (non-entity)
[5]
Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow


Final_project.ipynb
BC > SP_25 > NLP > NLP_Class > Assignments > Final_project > Final_project.ipynb > # Let's define some functions
+ Code + Markdown | Run All Restart Clear All Outputs | Jupyter Variables Outline ...
Torchcv (Python 3.9.18)
# Filter low-confidence predictions back to "O" (non-entity)
low_conf_mask = (top_probs < threshold) & (result["preds"] > 0)
result.loc[low_conf_mask, "preds"] = 0
result.loc[low_conf_mask, "bio_tag"] = "O"
return result
[5]
Python
def main():
"""Main execution function"""
print("Loading test data...")
with open(test_data_file, 'r') as f:
test_data = json.load(f)
print(f"Processing {len(test_data)} records...")
# Process data in chunks
all_results = []
for i, chunk_data in enumerate(chunk(test_data, chunk_size)):
print(f"Processing chunk {i+1}/{len(test_data)//chunk_size + 1}")
chunk_results = process_chunk(chunk_data)
all_results.append(chunk_results)
# Combine all chunk results
print("Combining results...")
final_df = pd.concat(all_results)
# Convert to BIO tagging format
print("Converting to BIO tags...")
final_df = convert_to_bio_tags(final_df)
# Apply confidence-based filtering
print("Applying confidence filtering...")
final_df = confidence_based_filtering(final_df)
# Create submission file
print("Creating submission...")
# Create submission dataframe
submission_rows = []
row_id = 0
for doc_id, group in final_df.groupby("document"):
# Sort by word_ids
group = group.sort_values("word_ids")
# Skip special tokens (word_ids < 0)
valid_tokens = group[group["word_ids"] >= 0]
[6]
Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow

```python
    for doc_id, group in final_df.groupby("document"):
        # Sort by word_ids
        group = group.sort_values("word_ids")
        # Skip special tokens (word_ids < 0)
        valid_tokens = group[group["word_ids"] >= 0]

        # Add rows for tokens with entity labels (non-O tags)
        for _, row in valid_tokens[valid_tokens["bio_tag"] != "O"].iterrows():
            submission_rows.append({
                "row_id": row_id,
                "document": int(row["document"]),
                "token": int(row["word_ids"]),
                "label": row["bio_tag"]
            })
            row_id += 1

    # Create DataFrame and save as CSV
    submission_df = pd.DataFrame(submission_rows)
    submission_df.to_csv("submission.csv", index=False)

    print(f"Done! Submission contains {len(submission_df)} PII detections.")

if __name__ == "__main__":
    main()
```

[6] Python

```
Loading test data...
Processing 10 records...
Processing chunk 1/1
```

Spaces: 4 Cell 2 of 6 Go Live Cursor Tab 1m Flow