Analysis of the "Data Collection" Task

Example Implementation and Explanations

---

1. Example Implementation of Data Collection via Telegram (Pyrogram)

1.1. Client Setup and Dependencies

```
from pyrogram import Client
import asyncio
import nest_asyncio
import csv

nest_asyncio.apply()  # Workaround for asyncio in Jupyter/Colab

# Initialize Pyrogram client
app = Client(
    "my_account",
    api_id="YOUR_API_ID",  # Obtain from https://my.telegram.org
    api_hash="YOUR_API_HASH"
)
```

Explanation:

- **Pyrogram**: Library for interacting with Telegram API.
- **api_id** and **api_hash**: Unique keys for API access.
- `nest_asyncio` allows async loops in environments with active event loops (e.g., Jupyter Notebook).

1.2. Data Collection Function

```
async def TelegramBatchLoader(author_url):
    async with app:
        async for message in app.get_chat_history(author_url, limit=300):
            # Extract text from message
            text = message.text or message.caption

            # Skip messages without text or short texts
            if not text or len(text) < 200:
                continue

            # Create record
            result = {
```

```
            'id': message.id,  # Unique Telegram message ID
            'text': text[:3000],  # Truncate to 3000 characters
            'category': 'other',  # Manually set category per channel
            'source': author_url,
            'length': len(text),
            'date': message.date.date().isoformat()  # Date in YYYY-MM-DD
        }

        # Write to CSV
        with open('collected_data.csv', 'a', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=result.keys())
            writer.writerow(result)

# Example execution for a channel
asyncio.run(TelegramBatchLoader("@olympic_russia"))
```

Key Points:

- **Text Extraction**: Uses `message.text` (message body) or `message.caption` (media captions).
- **Filtering**: Ignores messages shorter than 200 characters.
- **Unique ID**: Uses Telegram's native `message.id` instead of manual counters.
- **Categories**: Manually assign categories (e.g., `cybersport` for `@gabenews`).
- **Date Format**: Ensures `YYYY-MM-DD` format via `isoformat()`.

---

2. Data Post-Processing (Pandas)

2.1. Deduplication

```
import pandas as pd
df = pd.read_csv('collected_data.csv')
df = df.drop_duplicates(subset=['id'])  # Remove duplicate IDs
```

Explanation:

- `drop_duplicates` removes rows with identical `id` values.

2.2. Date Validation

```
# Convert string to datetime
df['date'] = pd.to_datetime(df['date'], errors='coerce')

# Filter data for 2023-2024
df = df[(df['date'] >= '2023-01-01') & (df['date'] <= '2024-12-31')]
```

Explanation:

- `errors='coerce'` converts invalid dates to `NaN`, which are later removed.
- Filters data within the specified date range.

---

3. Alternative Data Collection Methods

3.1. VK API Integration

```
import vk_api
from datetime import datetime

vk_session = vk_api.VkApi(token='VK_API_TOKEN')
vk = vk_session.get_api()

# Example: Fetch posts from a group
posts = vk.wall.get(domain='csru_official', count=100)['items']
for post in posts:
    text = post['text']
    date = datetime.fromtimestamp(post['date']).strftime('%Y-%m-%d')

    # Create record (similar to Telegram example)
    result = {
        'id': post['id'],  # Unique VK post ID
        'text': text[:3000],
        'category': 'cybersport',
        'source': 'https://vk.com/csru_official',
        'length': len(text),
        'date': date
    }
```

Explanation:

- **VK API** requires an access token from VK Dev.
- `post['id']` ensures uniqueness within the group.

3.2. Web Scraping (BeautifulSoup)

```
import requests
from bs4 import BeautifulSoup

url = "https://example-news-site.com/sport"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

# Parse news articles
```

```
articles = soup.find_all('div', class_='news-item')
for article in articles:
    text = article.find('p').text.strip()
    date = article.find('time')['datetime']

    # Create record
    result = {
        'id': hash(text),  # Use text hash as temporary ID
        'text': text[:3000],
        'category': 'sports',
        'source': url,
        'length': len(text),
        'date': date
    }
```

Explanation:

- **BeautifulSoup** parses HTML content.
- `hash(text)` generates a temporary unique ID (suitable for demos).
- For JavaScript-heavy sites, use **Selenium**.

Summary

The provided implementation collects data via Telegram but requires:

1. **Manual category assignment** per channel.
2. **ID uniqueness control** via `message.id`.
3. **Additional processing** for data balancing (e.g., equal category distribution).

Alternatives (VK API, web scraping) integrate into the same CSV structure. For large-scale data:

- Use databases (SQLite, PostgreSQL).
- Implement error handling (e.g., retries).
- Enable parallel data collection via async requests.

Analysis of the "Text Processing" Task

Example Implementation and Method Variations

---

1. Basic Text Cleaning

Goal: Remove HTML tags, URLs, emojis, and special characters.

Code:

```
def clean_text(self, text: str) -> str:
    text = BeautifulSoup(text, "html.parser").get_text()  # Remove HTML
    text = re.sub(r'http\S+|www.\S+', '', text)  # Remove URLs
    text = re.sub(r'[^\w\s\-\.]', ' ', text)  # Keep hyphens and periods
    text = ' '.join(text.split())  # Normalize whitespace
    return text
```

Alternatives:

- Use `lxml` for faster HTML parsing.
- Use `emoji` library to remove emojis.

---

2. Entity Extraction

Goal: Extract team names, tournaments, and person names.

Using Natasha:

```
def extract_entities(text: str) -> List[Dict]:
    doc = Doc(text)
    doc.segment(segmenter)  # Tokenize
    doc.tag_ner(ner_tagger)  # NER tagging
    return [
        {"text": span.text, "type": span.type}
        for span in doc.spans
    ]
```

Features:

- Recognizes `ORG` (organizations), `PER` (persons), `LOC` (locations).
- Returns normalized forms (e.g., "НХЛ" → "Национальная хоккейная лига").

Alternatives:

- Use `spaCy` with `ru_core_news_lg` model.
- Custom regex patterns (e.g., `r"Team [A-Za-z]+"`).

---

3. Lemmatization

Goal: Convert words to dictionary forms.

Code with pymorphy2:

```
def lemmatize_text(self, text: str) -> str:

    words = word_tokenize(text.lower())
    return ' '.join([
        morph.parse(word)[0].normal_form  # Lemmatize
        for word in words
        if word not in self.stopwords  # Remove stop words
    ])
```

Alternatives:

- Use `spaCy` for faster processing.
- Use `Stanza` for contextual lemmatization.

---

4. Technical Considerations

Compliance:

- **Formats**: `entities` saved as JSON strings (`json.dumps`).
- **Output**: CSV includes original texts (via `df.iterrows()` loop).

Saving Data:

```
pd.DataFrame(processed_data).to_csv('processed_data.csv', index=False)
```

---

5. Solution Variations

Enhancements:

1. **Entity Extraction**:
    - Combine Natasha with custom regex rules for tournaments (e.g., `r"Кубок [A-Za-z]+"`).
2. **Lemmatization**:
    - Replace `pymorphy2` with `Stanza` for context-aware processing.
3. **Abbreviation Handling**:
    - Add a custom dictionary (e.g., "НХЛ" → "Национальная хоккейная лига").

Analysis of the "Text Classification" Task

Example Implementation and Alternative Approaches

---

1. Data Preprocessing

Goal: Prepare texts for model training.

Steps:

- **Lemmatization**: Convert words to base forms (e.g., "scored" → "score").
- **Stop Word Removal**: Exclude non-informative words.
- **Lowercasing**.

Code:

processed_texts = [TextProcessor().lemmatize_text(text) for text in texts]

Alternatives:

- **Stemming** via `nltk.stem.SnowballStemmer` (faster but less accurate).
- **Deep Learning**: Use pretrained models like BERT for contextual processing.

---

2. Text Vectorization

Goal: Convert text to numerical features.

Example:

self.vectorizer = TfidfVectorizer(max_features=5000)
X = self.vectorizer.fit_transform(df['lemmatized_text'])

Explanation:

- **TF-IDF** reflects word importance relative to the corpus.
- `max_features=5000` limits dimensionality for faster training.

Alternatives:

- **Word2Vec/GloVe** for word embeddings.
- **BERT** embeddings via `transformers` library.

---

3. Model Selection

Example:

self.classifier = RandomForestClassifier(n_estimators=100)

Advantages:

- Resistant to overfitting.
- Handles sparse data (e.g., TF-IDF matrices).

Alternatives:

- **Logistic Regression**: Fast and interpretable.
- **CatBoost/XGBoost**: Gradient boosting for higher accuracy.
- **Neural Networks**: Implemented via `keras` or `pytorch`.

---

4. Training and Validation

Key Steps:

- **Data Split**: 80% training, 20% testing.
- **Cross-Validation**: 5-fold validation for stability assessment.

Code:

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
cv_scores = cross_val_score(self.classifier, X_train, y_train, cv=5)

---

5. Report Generation

Output Files:

predictions.csv:

pd.DataFrame({'id': test_data['id'], 'predicted_class': predictions}).to_csv('predictions.csv')

model_report.json:

{
  "model_description": {

```
    "vectorization": "TF-IDF",
    "classifier": "RandomForestClassifier",
    "preprocessing": ["lemmatization", "stop word removal"]
  },
  "training_process": {
    "split_ratio": "80/20",
    "cross_validation_scores": [0.85, 0.83, 0.86]
  }
}
```

Summary

The baseline solution works but can be enhanced by:

- Experimenting with vectorization methods (e.g., BERT).
- Testing advanced models (CatBoost, neural networks).
- Addressing class imbalance.