Проект ООП: GlossingTool

прототип интерфейса для глоссинга на локальном хосте

python: FastAPI

ML: CNN

+ небольшой фронт

Все, что касается классов

Классы

- **LemmaInserter** основной класс для работы с подстановкой лемм и морфем
- **DataEncoder** базовый класс для энкодеров
- **EntryDataset** датасет для работы с данными
- **GlossingPreprocessor** препроцессор данных
- **CharTokenizer** токенизатор
- **Trainer** класс для обучения

критерий 1

Наследование от собственных классов

- **MorphBIOEncoder** от DataEncoder
- **SegmDataModule** от DataModule
- **MorphSegmentationCNN** от BasicNeuralClassifier

критерий 3

Наследование от стандартных классов

- **EntryDataset** от Dataset
- **TokenType** от Enum
- **BasicNeuralClassifier** от nn.Module
- Для “Duck Typing”: **DataclassLike** от Protocol

критерий 2

Наследование от абстрактных классов

наследование от ABC:

- **DataEncoder**
- **SystemPipeline**
- **BaseInference**
- **DataModule**

критерий 4

Используемые библиотеки

ML-составляющая

torch
numpy
scikit-learn

Дополнительно

jaxtyping для типизации тензоров
tqdm
python-dotenv
pyyaml
pytest

критерий 5

Декораторы

@abstractmethod в абстрактных классах
@classmethod ExperimentConfig.from_yaml()
InferenceConfig.from_yaml()
@staticmethod EntryDataset.collate_fn()

критерий 6

Собственные декораторы

регистрация модели/датамодуля/пайплайна для трейна с помощью декоратора, пример:
@register_model

критерий 7

Перегрузка операторов

перегрузка операторов для истории обучения модели

Пример для TrainingHistory

```python
class TrainingHistory:
    def __getitem__(self, idx: int) -> Dict[str, Any]:
        """Доступ к данным конкретной эпохи по индексу."""
        if idx < 0 or idx >= len(self.epochs):
            raise IndexError(f"Индекс {idx} вне диапазона: всего {len(self.epochs)} эпох")

        return {
            'epoch': self.epochs[idx],
            'train_loss': self.train_losses[idx],
            'val_loss': self.val_losses[idx],
            'val_accuracy': self.val_accuracies[idx],
            'val_word_accuracy': self.val_word_accuracies[idx]
        }

    def __iadd__(self, epoch_data: Dict[str, Any]) -> 'TrainingHistory':
        """Добавление данных новой эпохи через +=."""
        self.epochs.append(epoch_data['epoch'])
        self.train_losses.append(epoch_data['train_loss'])
        self.val_losses.append(epoch_data['val_loss'])
        self.val_accuracies.append(epoch_data['val_accuracy'])
        self.val_word_accuracies.append(epoch_data['val_word_accuracy'])
        return self
```

критерий 8

Исключения

NotImplementedError

RuntimeError IndexError ValueError

критерий 9

Кастомные исключения

LengthMismatch DataMissing

ComponentInitializationError

```python
class DataError(Exception):
    """Базовое для ошибок с данными"""
    pass

class LengthMismatch(DataError):
    """Ошибка соответствия по количеству в GlossingPipeline"""
    def __init__(self, text: str, sent: str, value_one: Any, value_two: Any):
        super().__init__(f"Количество слов в тексте {text} в предложении {sent} не совпадает с данными: {len(value_one)} vs {len(value_two)}")

class DataIsMissing(DataError):
    """Ошибка отсутствия данных"""
    def __init__(self):
        super().__init__(f"Данные не были предоставлены")

class ComponentError(Exception):
    """Базовое для ошибок компонентов"""

class ComponentInitializationError(ComponentError):
    """Ошибка инициализации компонента"""
    def __init__(self, component_name: str, reason: str):
        self.component_name = component_name
        self.reason = reason
        super().__init__(f"Не получилось инициализировать модель {component_name}: {reason}")
```

критерий 10

Синтаксический сахар питона

Использование моржовых операторов

в регулярных выражениях re.match для group

```
def _check_content_type(self, line: str) -> Tuple[str, str]:
        if segmentation := re.match(r"\d+(?:_\d+)*>\s*(.*)$", line):
            return 'segmented', segmentation.group(1)
        if glossing := re.match(r"\d+(?:_\d+)*<\s*(.*)$", line):
            return 'glossed', glossing.group(1)

        if line.startswith("#"):
            meta = line.lstrip("# \t")
            return 'metadata', meta
        if translation_line := re.match(r"(\d+(?:_\d+)*)=(.*)$", line):
            num, translation = translation_line.groups()
            self._add_line('line_num', num)
            return 'translation', translation
```

Match-Case statements

```
for char, l in zip(ex, preds):
    # print(char, l)
    match l:
        case 1:
            if prev_label is None or prev_label == 0:
                final_string += char
                prev_label = l
            else:
                final_string += f'-{char}'
        case 2:
            final_string += char
            prev_label = l
            if prev_label is None or prev_label == 0:
                prev_label = 1
        case 0:
            final_string += '\t'
            prev_label = l
final_data.append(final_string)
```

Распаковка словарей конфигов через **

```
@classmethod
def from_yaml(cls, path: str) -> 'InferenceConfig':
    """Загрузить конфигурацию из YAML"""
    with open(path) as f:
        config_dict = yaml.safe_load(f)

    task = config_dict['task']
    model_dict = config_dict['model']

    if task == 'segmentation':
        model = SegmentationConfig(**model_dict)
    else:
        model = ModelConfig(**model_dict)

    return cls(
        name=config_dict['name'],
        task=task,
        model=model,
        training=TrainingConfig(**config_dict.get('training', {})),
        tokenizer=TokenizerConfig(**config_dict.get('tokenizer', {}))
    )
```

Использование f-strings

Типизация кода

критерий 11

Юнит-тесты

test_tokenizer.py

tests > test_tokenizer.py > TestCharTokenizer > test_get_tokens_ids_unknown_char

```python

@pytest.fixture
def tokenizer():
    """Фикстура для создания экземпляра токенизатора"""
    return CharTokenizer(symbols)

class TestCharTokenizer:
    """Тесты для класса CharTokenizer"""

    def test_initialization(self, tokenizer):
        """Тест инициализации токенизатора"""
        assert tokenizer is not None
        assert hasattr(tokenizer, 'id2char')
        assert hasattr(tokenizer, 'char2id')
        assert isinstance(tokenizer.id2char, dict)
        assert isinstance(tokenizer.char2id, dict)

    def test_tokenize_simple_word(self, tokenizer):
        """Тест токенизации простого слова"""
        result = tokenizer.tokenize(['слово'])
        assert isinstance(result, list)
        assert len(result) == 1
        assert isinstance(result[0], list)

    def test_tokenize_word_with_hyphen(self, tokenizer):
        """Тест токенизации слова с дефисом"""
        result = tokenizer.tokenize(['слово-морфема-ещё-одна'])
        assert isinstance(result, list)
        assert len(result) == 1
        # Дефис должен быть удален при токенизации
        chars = ''.join(result[0])
        assert '-' not in chars or '<PUNC>' in result[0]

    def test_tokenize_multiple_words(self, tokenizer):
        """Тест токенизации нескольких слов"""
        words = ['слово', 'второе', 'третье']
        result = tokenizer.tokenize(words)
        assert len(result) == 3
        assert all(isinstance(chars, list) for chars in result)
```

lunadunkel (13 minutes ago) Ln 97, Col 37 Spaces: 4 UTF-8 LF

test_preprocessing.py

tests > test_preprocessing.py > TestGlossingPreprocessor > test_classify_token_speaker

```python
class TestGlossingPreprocessor:
    """Тесты для класса GlossingPreprocessor"""

    def test_tokenize_simple_line(self, preprocessor):
        """Тест токенизации простой строки"""
        line = "слово второе третье"
        tokens = preprocessor._tokenize(line)
        assert len(tokens) == 3
        assert tokens == ["слово", "второе", "третье"]

    def test_tokenize_empty_line(self, preprocessor):
        """Тест токенизации пустой строки"""
        line = ""
        tokens = preprocessor._tokenize(line)
        assert tokens == []

    def test_tokenize_single_word(self, preprocessor):
        """Тест токенизации одного слова"""
        line = "слово"
        tokens = preprocessor._tokenize(line)
        assert len(tokens) == 1
        assert tokens[0] == "слово"

    def test_tokenize_extra_spaces(self, preprocessor):
        """Тест токенизации строки с лишними пробелами"""
        line = "  слово   через  много пробелов     "
        tokens = preprocessor._tokenize(line)
        # После strip и split не должно остаться пустых токенов
        assert all(token for token in tokens)
```

```python
class TestToken:
    """Тесты для класса Token"""

    def test_token_creation(self):
        """Тест создания токена"""
        token = Token(text="слово", type=TokenType.WORD, mask_for_bio=False)
        assert token.text == "слово"
        assert token.type == TokenType.WORD
        assert token.mask_for_bio == False

    def test_token_repr(self):
        """Тест строкового представления токена"""
        token = Token(text="слово", type=TokenType.WORD, mask_for_bio=False)
        repr_str = repr(token)
        assert "слово" in repr_str
        assert "WORD" in repr_str
        assert "False" in repr_str
```

критерий 12

Структура

```
backend/
├── src/
│   ├── core/
│   │   ├── base_classes.py          # Базовые классы
│   │   ├── config.py                # Конфигурации
│   │   ├── initializing.py          # Регистрация компонентов
│   │   ├── lemma_inserter.py
│   │   └── data/                    # Работа с данными
│   ├── models/                      # Нейросетевые модели
│   ├── encoders/                    # Энкодеры данных
│   ├── pipelines/                   # Пайплайны обработки
│   ├── training/                    # Обучение
│   ├── inference/                   # Инференс
│   └── tokenization/                # Токенизация
├── vocabularies/                    # Словари
└── requirements.txt
```

Структура классов

- есть разделение на абстрактные и конкретные классы
- используется паттерн для регистрации компонентов –> при расширение пайплайна ничего не сломается
- есть типизация с использованием type hints + jaxtyping

критерий 13

Шаблоны проектирования

Template Method (Шаблонный метод)

```python
class SystemPipeline(ABC):
    @abstractmethod
    def run(self, inputs: Sequence[Any]) -> Sequence[Any]:
        raise NotImplementedError
```

Registry Pattern (Реестр)

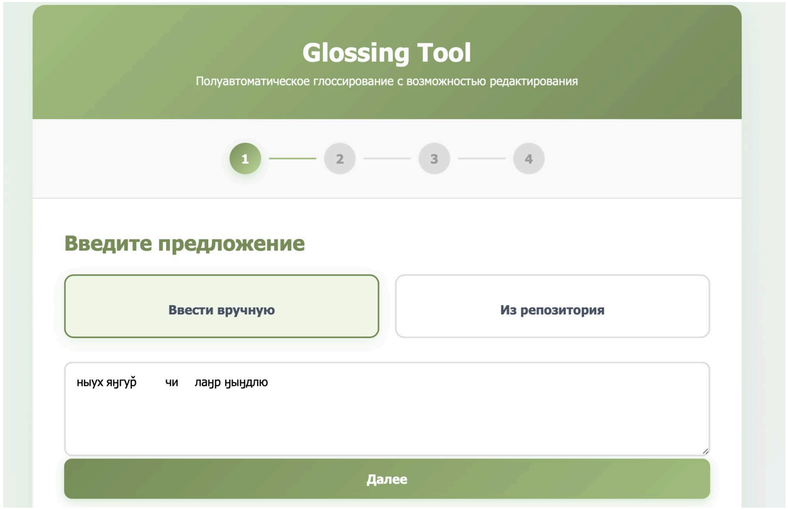
```python
MODEL_REGISTRY: Dict[str, Type[BasicNeuralClassifier]] = {}
DATAMODULE_REGISTRY: Dict[str, Type[DataModule]] = {}

@register_model("lemma_tagger")
class LemmaAffixTagger(BasicNeuralClassifier):
    ...
```

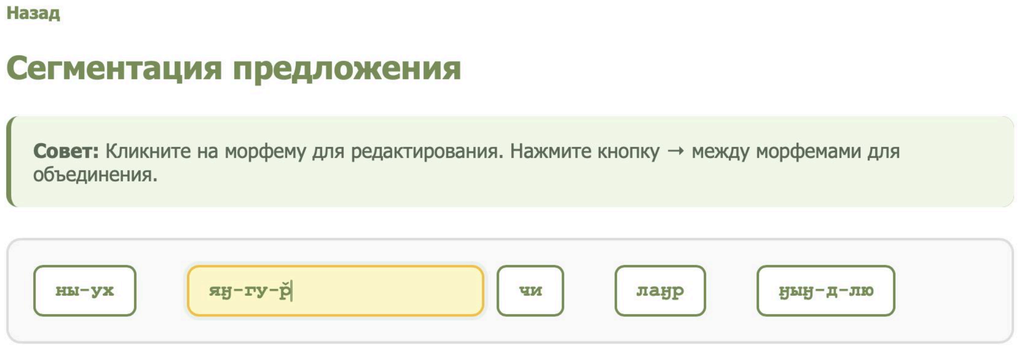
Factory Method Pattern?

Финальный продукт

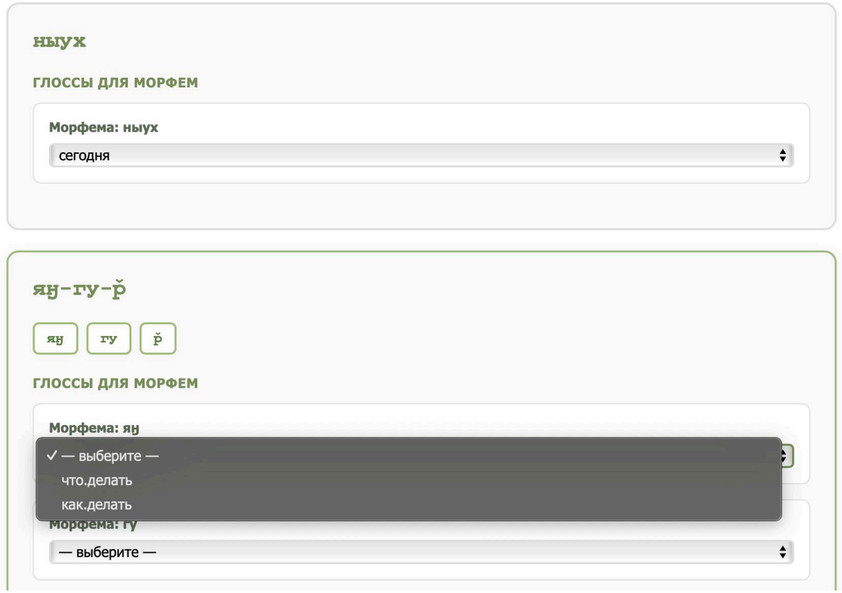
1



2



3



4