Федеральное государственное автономное образовательное учреждение высшего образования
«МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий
Кафедра «Информатика и вычислительная техника»

Направление подготовки/ специальность: Системная и программная инженерия

ОТЧЕТ

по проектной практике

Студент: Смирнов Андрей Сергеевич Группа: 241-326

Место прохождения практики: Московский Политех, кафедра «Информатика и вычислительная техника»

Отчет принят с оценкой _____ Дата _____

Руководитель практики: _____

Москва 2025

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современные методы обработки естественного языка (NLP) позволяют эффективно анализировать большие объемы текстовых данных, извлекать ключевую информацию, классифицировать контент и выявлять закономерности. Однако использование NLP-алгоритмов часто требует специализированных знаний в программировании и машинном обучении, что ограничивает доступность этих технологий для широкого круга пользователей.

Разработка графического интерфейса для NLP-анализа текстовых данных призвана сделать инструменты обработки естественного языка более удобными и доступными. Такой интерфейс позволит пользователям без глубоких технических навыков загружать тексты, выбирать методы анализа и визуализировать результаты в интуитивно понятном формате.

1. Общая информация о проекте

BindWord XP - десктопное приложение для NLP-анализа текстовых данных, созданное на основе, основанное на существующем Python/Spacy скрипте, и возможностью обработки текста. **Цель проекта** - создание рабочего десктопного оффлайн приложения с графическом интерфейсом.

Актуальность

Многие люди при работе с большим объемом информации сталкиваются со снижением продуктивности и риском ошибок из-за ручного анализа, который занимает много времени. Отсутствие программного обеспечения для анализа больших объемов текстовых данных создает значительные трудности для пользователей.

Задачи

Для достижения цели поставлены следующие задачи:

- определение необходимого функционала для целевой аудитории;
- написание графического интерфейса;
- оптимизация кода;
- дополнение функционала;
- тестирование и публикация бинарных файлов и исходного кода.

2. Общая характеристика деятельности организации *(заказчика проекта)*

Наименование заказчика

Московский политехнический университет (Московский Политех) – ведущий технический вуз России, осуществляющий подготовку специалистов в области инженерии, информационных технологий, транспорта, дизайна и других высокотехнологичных направлений.

Организационная структура

Университет имеет разветвленную организационную структуру, включающую:

- **Ректорат** (ректор, проректоры по направлениям деятельности);
- **Институты и факультеты** (например, Институт информационных технологий, Транспортный институт, Факультет урбанистики и дизайна и др.);
- **Кафедры** (профильные подразделения, отвечающие за образовательные программы);
- **Научно-исследовательские центры и лаборатории**;
- **Административно-управленческие подразделения** (учебный отдел, отдел международного сотрудничества, HR-служба и др.).

Описание деятельности

Московский политехнический университет осуществляет:

1. **Образовательную деятельность** – реализация программ бакалавриата, магистратуры, аспирантуры, дополнительного образования.
2. **Научно-исследовательскую работу** – проведение фундаментальных и прикладных исследований в сфере инновационных технологий, участие в грантах и collaborations с промышленными партнерами.
3. **Инновационное развитие** – поддержка стартапов, технологическое предпринимательство, взаимодействие с индустриальными компаниями.
4. **Международное сотрудничество** – программы обмена, совместные проекты с зарубежными вузами и научными центрами.

Университет активно внедряет современные подходы к обучению, включая проектно-ориентированные методы, и является ключевым участником развития инженерного образования в России.

3. Описание задания по проектной практике

Наша команда получила несколько заданий по проектной практике, в том числе:

- Настройка Git и репозитория;
- Написание документов в Markdown;
- Создание статического веб-сайта;
- Взаимодействие с организацией-партнёром;
- Вариативное задание (в нашем случае, собственный чат-бот в Telegram)

Для начала, нами был создан групповой репозиторий на GitHub на основе предоставленного шаблона, где в будущем были размещены результаты выполнения других заданий. При создании мы освоили базовые команды Git: клонирование, коммит, пуш и создание веток.

Далее, после создания репозитория, мы начали изучать синтаксис Markdown, параллельно с этим создавая и заполняя необходимые документы. В данном формате были оформлены все материалы проекта (описание проекта, процесс выполнения заданий, журнал прогресса и др.).

Также нами был разработан статический веб-сайт для проекта BindWord XP, представляющего собой инструмент для NLP-анализа текстовых данных. Сайт создавался с использованием современных веб-технологий, включая HTML5, CSS3 и JavaScript, что позволило реализовать интерактивный и удобный интерфейс.

Основной задачей при создании сайта было наглядно представить функциональные возможности проекта и обеспечить удобный доступ к информации для пользователей. Главная страница содержит краткое описание проекта с выделением его ключевых преимуществ, таких как анализ частотности слов, распознавание именованных сущностей и определение тональности текста.

Страница "О проекте" детально раскрывает концепцию разработки, включая технические особенности и области применения. Особое внимание уделено описанию алгоритмов работы системы и их практической пользе для различных категорий пользователей.

Раздел "Команда" был реализован с функцией поиска, что значительно упрощает навигацию по списку участников проекта. Каждый член команды представлен с

указанием группы, что отражает вклад студентов разных специальностей в разработку.

Хронология работы над проектом представлена в разделе "Журнал" в формате временной шкалы. Такой подход позволяет наглядно продемонстрировать этапы разработки и достигнутые результаты на каждом из них.

Техническая реализация сайта включает несколько важных аспектов. Адаптивная вёрстка обеспечивает корректное отображение на устройствах с различными разрешениями экрана. Для стилизации элементов использовались CSS-переменные, что упрощает поддержку и модификацию дизайна. Интерактивные элементы, такие как кнопки и карточки, имеют анимационные эффекты при наведении, что улучшает пользовательский опыт.

Параллельно с выполнением других заданий, мы посетили мероприятие от заказчика – Карьерный Марафон. Это был очень ценный опыт для нас, ведь мы смогли познакомиться с деятельностью многих ведущих компаний России. Это помогло нам понять структуру современных крупных организаций и определиться с выбором будущей профессии.

И после этого мы занялись выполнением вариативного задания: созданием собственного чат-бота в Telegram. Для написания кода для бота был выбран стек **aiogram**, который является самым популярным стеком для создания телеграмм-ботов, благодаря поддержки асинхронности. Бота, созданного в рамках выполнения вариативного задания для Проектной Практики, мы назвали **ToYouFromNow**. Наш проект представляет собой аналог сайта *FutureMe* в обёртке телеграмм-бота, который позволяет пользователю отправить сообщения "в будущее", и пользователь получит их обратно в выбранное время. Для создания бота мы изучили видеокурс на YouTube по созданию телеграмм-ботов на aiogram 3.0 от *sudo teach IT*. Следую инструкции из видео-уроков мы создали рабочего бота, отвечающий на сообщения пользователей, имеющий фоновый процесс и работающий с БД. Исходный код нашего проекта можно посмотреть в папке *src*.

Туториал по созданию телеграмм-бота.

Для выполнения задания по Проектной практике мы написали туториал для начинающих, который позволит каждому создать своего бота, зная азы языка программирования *Python*. Руководство разбито на несколько последовательных шагов для удобства читающего.

1. Получение токена

Для работы бота в Telegram необходим токен - уникальный ключ, который используется для аутентификации и идентификации бота. Токен можно получить в чате с телеграмм-ботом **@BotFather**, написав команду */newbot*.

После ввода имени бота и его юзернейма, вы получите тот самый **токен**.

2. Загрузка библиотек

Важная часть любой разработки на *Python* - **библиотеки**. Загрузить библиотеку можно при помощи консоли через **установщика пакетов Python**. Пример команды для установки библиотек:

pip install aiogram

Для разработки бота понадобятся следующие библиотеки:

- aiogram
- asyncio
- sqlaclhemy
- dotenv

Также пользователи среды разработки *PyCharm* могут подгружать необходимые через инструменты этой среды (Settings -> Project -> Python Intepreter -> *нажать плюсик (+)* -> *выбрать библиотеку* -> Install Package)

3. Скелет бота и скрытие токена

Создав новый .py-файл, например *main.py*, в проекте импортируем необходимые на данные этапы библиотеки.

```
import asyncio
from aiogram import Bot, Dispatcher
from dotenv import load_dotenv
import os
```

Ранее упоминалось, что токен не стоит вставлять напрямую в код проекта, особенно, в OpenSource, загруженный в публичный удалённый репозиторий на GitHub. Всю конфиденциальную информацию, включаю ключи API - токены, программисты хранят в файлах окружения формата. env. В папке с проектом нужно создать файл с расширением. env. Файл окружения хранит информацию в парах ключ-значение. Ключ принято называть заглавными буквами **TOKEN**, и через оператор присваивания (=) записывается токен. Теперь в коде проекта вместо настоящего токена, будем писать ключ TOKEN.

TOKEN='ваш токен'

Вернёмся в *main.py*. Функцией *load_dotenv()* загружаем токен. Затем создаём объект класса Bot, в аргумент которого пишем *os.getenv ('TOKEN')*. Наконец, создаём объект класса Dispatcher.

```
load_dotenv ()
bot = Bot(os.getenv('TOKEN'))
dp = Dispatcher()
```

Далее нужно описать функцию main, которая будет служить точкой входа при запуске бота. В этой функции нужно запустить **поллинг** - регулярное взаимодействие между ботом и пользователем. Важно! Поллинг в aiogram - асинхронная функция, и все асинхронные функции нужно вызывать с ключевым словом **await** (есть одно исключение, о которым мы упомянем отдельно).

```
async def main():
        await dp.start_polling(bot)


if __name__ == '__main__':
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("Exit")
```

4. Обработка сообщений

Чтобы наш бот заговорил используются **обработчики**(*handlers*). Для обработчиков создадим новый .py-файл, назовём *handlers.py*. Импортируем нужные библиотеки и модули:

```
from aiogram import Router, Bot
from aiogram.filters import CommandStart, Command
from aiogram.types import Message
```

Диспетчер нельзя импортировать в другой файл из main.py, поэтому в файле handlers.py мы создадим аналог диспетчера - **роутер**. По сути, диспетчер просто является главным роутером, остальные роутеры используется, если диспетчер не смог обработать сообщение от пользователя.

```
router = Router()
```

Разберём пример обработчика, который отправляет сообщение пользователю при старте бота, или отправки команды */start*

```
@router.message(CommandStart())
async def start(message: Message):
    await message.answer('sup <3! Этот бот позволяет отправить письмо себе в будущее! Хочешь попробовать? напиши команду'
                         ' /help, чтобы узнать доступные команды')
```

В аргументах декоратора @router.message() указывается при обработке какой команды, следующая функция вызовется. Для обработки старта бота есть отдельная функция *CommandStart()*, для обработки других команд в аргументах декоратора пишется функция Command() с аргументом в виде строки без слеша, например для обработки команды /help будет выглядеть так: Command('help'). После объявления декоратора создаётся асинхронная функция с объектом класса **Message** в аргументе. Для отправки сообщения у этого объекта вызываем метод **answer**, в аргументах которого пишем нужное сообщение для отправки пользователю. Метод асинхронный, поэтому перед ним ставим *await*. Последний штрих - перейти обратно в файл *main.py*, импортируем *router* из файла *handlers*, а в функцию *main* пишем *dp.include_router(router)*.

```
from handlers import router
```

...

```
async def main():
        dp.include_router(router)
        await dp.start_polling(bot)
```

Готово! Наш бот теперь нам отвечает.

5. Конечный автомат состояний (FSM)

Во время разработки бота возникнет необходимость получать информацию от пользователя, например, для регистрации или сохранения в БД. Возникают трудности, как дать понять боту что в этом сообщении - логин, а в другом - номер телефона? Для этого используется **FSM**. Начнём с импорта нужных модулей в *handlers.py*.

```
from aiogram.fsm.state import StatesGroup,State
from aiogram.fsm.context import FSMContext
```

Далее задаём состояние, какие будут использоваться в дальнейшем. Определение состояний записываем в отдельном классе. Например, для своего проекта нам важно было записывать от пользователя текст сообщения и дату, когда оно придёт пользователю. Класс состояний должен быть подклассом **StatesGroup**, а состояния - объекты класса **State**.

```
class Parser(StatesGroup):
    text = State()
    date = State()
```

Создадим обработчик для команды */send*. В аргументы функции добавляем объект класса **FSMContext**, через который и будут управлять состояниями. В этом обработчике меняем состояние на *text* асинхронной методом **set_state()**, а аргументе которого добавляем нужное состояние.

```
@router.message(Command('send'))
async def send(message: Message, state: FSMContext):
    await state.set_state(Parser.text)
    await message.answer("Отлично! Напиши сообщение, которое ты получишь в будущем! (>100 символов)")
```

Теперь в аргументе декоратора для следующего обработчика вместо сообщения и команд записываем состояние, которое мы установили ранее. После отправки сообщения пользователем, вызовется именно этот обработчик. Кстати, мы же хотели сохранить текст сообщения пользователя? Для сохранения данных используется метод **update_data**. После сохранения данных можно снова изменить состояние на следующее.

```
@router.message(Parser.text)
async def getText(message: Message, state: FSMContext):
    if (len(message.text) >= 100):
        await state.update_data(text=message.text)
        await state.set_state(Parser.date)
        await message.answer('Золотые слова! Теперь напиши, когда ты снова увидишь своё сообщение.(Например, "через год", "через 3 месяца", "2026 год 3 сентября в 15:49")')
    else:
        await message.answer("Так мало слов! Уверен, тебе есть что сказать :D")
```

Чтобы получить записанные данные обратно в переменную, нужно использовать метод **get_data()**. Теперь полученные данные можно использовать дальше в коде: записать в базу данных или отправить их пользователю обратно. В конце нужно обязательно очистить данные **clear()**.

```
@router.message(Parser.date)
async def getDate(message: Message, state: FSMContext,bot: Bot):
    try:
        await state.update_data(date=parse_date(message.text))
        data = await state.get_data()
        await addMessage(message.from_user.id, message.chat.id, data["date"], data["text"])
        await message.answer(f"Увидимся {(data["date"]).strftime("%d %b %Y")} в {(data["date"]).strftime("%H:%M")} !")
        await state.clear()
```

```
    except ValueError:
        await message.answer("Ой! Не смог распознать дату :<. Попробуй ещё раз!")
```

6. Работа с базой данных

В течение текущего учебного года мы изучали проектирование и устройство баз данных, инструменты для работы с ними, SQL, поэтому мы решили добавить в наш телеграмм-бот взаимодействие с базой данной. Для телеграмм-бота, написанного на aiogram, нужно использовать асинхронные функции для взаимодействия с БД. Для этого в расширениях для библиотеки sqlalchemy с asyncio. Код для взаимодействия баз данных разделим на 2 файла: первый - для подключения к БД, а второй - для запросов. В файл с подключением к БД подгружаем нужные модули.

```
from sqlalchemy import BigInteger, TIMESTAMP, Text
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column
from sqlalchemy.ext.asyncio import AsyncAttrs, async_sessionmaker, create_async_engine
from datetime import datetime
```

После погрузки модулей нужно создать **асинхронный движок для работы с базой данных** при помощи функции *create_async_engine*. На основе этого движка создаётся подключение при помощи *async_sessionmaker()*.

```
engine = create_async_engine(url='sqlite+aiosqlite:///db.sqlite3')

async_session = async_sessionmaker(engine)
```

Таблицы проектируется через классы-наследников от класса Base. Имя таблицы создаётся через присваивание переменной __tablename__, а атрибуты таблицы - через поля класса. Для таблицы обязателен первичный ключ!

```
class Base(AsyncAttrs, DeclarativeBase):
    pass


class Message(Base):
    __tablename__ = 'messages'
    id: Mapped[int] = mapped_column(primary_key=True)
```

```
    user_id: Mapped[int] = mapped_column(BigInteger)
    chat_id: Mapped[int] = mapped_column(BigInteger)
    sending_time: Mapped[datetime]= mapped_column(TIMESTAMP)
    message_text: Mapped[str] = mapped_column(Text)
    is_sent: Mapped[int] = mapped_column()
```

Функция **async_main()** нужна для запуска подключения к базе данных. Её импортируем в файл *main.py*, и вызываем в функции *main()*.

```
async def async_main():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.create_all)
from databaseInterface import async_main
...
async def main():
    await async_main()
    dp.include_router(router)
    await dp.start_polling(bot)
```

Как упоминалось, запросы мы будем писать в отдельном файле, например, request.py. Из предыдущего файла импортируем подключение и класс, описывающий таблицу. SQL-запрос описывается как асинхронная функция, использующая асинхронное подключение. Для примера, метод *add* работает как *INSERT INTO* в языке SQL. После вызова метода *add* нужно вызвать асинхронный метод *commit*.

```
import datetime
from databaseInterface import async_session
from databaseInterface import Message

async def addMessage(user_id: int,chat_id: int,sending_time: datetime,message_text: str):
```

```
    async with async_session() as session:
session.add(Message(user_id=user_id,chat_id=chat_id,sending_time=sending_time,mes
sage_text=message_text,is_sent=0))
    await session.commit()
```

7. Фоновые процессы

Одна из самых частых задач для телеграмм-бота - отправлять какое-либо сообщение пользователю без предварительного сообщения от пользователя, например, рассылка всем пользователям бота или отправка сообщения, которое было сохранено в БД (как в нашем проекте). Для начала создадим асинхронную функцию, которая работает пока глобальная переменная истинна. Чтобы не нагружать сервер функция повторяется с перерывом в какое-то время. Важно использовать именно асинхронную функцию sleep, так как обычная бы остановило выполнения остальных функций в коде. Далее в асинхронной функции, пусть будет *on_startup*, вызывается метод *create_task* с ранее написанной функции в аргументе. Важно! Среда разработки может выдавать предупреждение и настойчиво просить поставить *await* перед *asyncio.create_task*. Но ставить ненужно, ибо фоновый процесс не даст остальным функциям, как поллинг, работать. Также нужно описать функцию, которая будет вызываться после завершение работы бота. В ней переменной для бесконечного цикла присваивается false, чтобы прервать фоновый процесс без ошибок. Финальный штрих - в функции main() указать функции для старта и завершения фонового процесса методом *register*().

```
stop_flag = True

async def check_messages():
    logger.info("Checking messages started")
    while stop_flag:
        ... //логика функции
        await asyncio.sleep(60)
```

```
async def on_startup():
    asyncio.create_task(check_messages())


async def on_shutdown():
    stop_flag = False


async def main():
    await async_main()
    dp.include_router(router)
    dp.startup.register(on_startup)
    await dp.start_polling(bot)
    dp.shutdown.register(on_shutdown)
```

4. Описание достигнутых результатов по проектной практике

Подводя итог, можно с уверенностью сказать, что наша команда успешно справилась со всеми поставленными задачами. Мы смогли создать репозиторий на GitHub, оформили все документы в формате Markdown, создали статический веб-сайт, а также взаимодействовали с организацией-партнёром и создали собственного чат-бота в Telegram.

Наша команда получила существенный опыт создания проектов на каждом из этапов работы, а также нашла практическое применение теоретическим знаниям, полученным в процессе обучения.

ЗАКЛЮЧЕНИЕ

В ходе проектной практики наша команда успешно выполнила все поставленные задачи, достигнув значимых результатов. Были освоены ключевые инструменты и технологии, такие как Git, Markdown, Python, aiogram, SQLAlchemy и другие, что позволило создать функционального Telegram-бота **ToYouFromNow**, аналог сервиса FutureMe. Бот предоставляет пользователям возможность отправлять сообщения в будущее, сочетая удобный интерфейс с надёжным хранением данных в базе SQLite.

Кроме того, был разработан подробный туториал по созданию Telegram-ботов, который может быть полезен для начинающих разработчиков. Все материалы проекта, включая документацию и исходный код, были размещены в GitHub-репозитории, что демонстрирует навыки командной работы и управления версиями.

Проделанная работа имеет практическую ценность, так как созданный бот решает реальную задачу — упрощает организацию личных напоминаний. Освоенные технологии и методы могут быть применены в будущих проектах, а опыт взаимодействия с организацией-партнёром и работы в команде стал важным этапом профессионального роста.

Таким образом, цели проектной практики достигнуты, а полученные результаты подтверждают готовность команды к реализации сложных и актуальных IT-проектов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Введение в CSS верстку: https://developer.mozilla.org/ru/docs/Learn_web_development/Core/CSS_layout/Introduction
2. DevTools для «чайников»: https://habr.com/ru/articles/548898/
3. Элементы HTML: https://developer.mozilla.org/ru/docs/Web/HTML/Element
4. Основы HTML: https://developer.mozilla.org/ru/docs/Learn_web_development/Getting_started/Your_first_website/Creating_the_content
5. Основы CSS: https://developer.mozilla.org/ru/docs/Web/CSS
6. https://doka.guide/
7. Официальная документация Git: https://git-scm.com/book/ru/v2
8. https://skillbox.ru/media/code/chto_takoe_git_obyasnyaem_na_skhemakh/
9. Бесплатный курс на Hexlet по Git: https://ru.hexlet.io/courses/intro_to_git
10. Уроки по Markdown: https://ru.hexlet.io/lesson_filters/markdown
11. Видеокурс по созданию ботов в Telegram: https://www.youtube.com/playlist?list=PLV0FNhq3XMOJ31X9eBWLIZJ4OVjBwb-KM

ПРИЛОЖЕНИЯ

Репозиторий на GitHub: https://github.com/Tavvex/project-practice-2025

Статический веб-сайт: https://tavvex.github.io/project-practice-2025/site