Deep Dive - Predict Customer Churn with Clean Code

IBM

UDACITY

Agenda

1. Project Introduction
2. Project Walkthrough
   a. Unit Testing
   b. Pytest
   c. Coding the Library
   d. Polishing Up
3. Q & A
4. Feedback

Project Introduction

About This Project

1. **Objective:** Convert a machine learning notebook into a well-written library.
2. **Use Case:** Identify credit card customers that are most likely to churn.
3. **Data Source:** 10,000 customers, 18 features, 16.07% have churned. Kaggle link.

You don't really need to know much about the dataset or the algorithm. You just need to know enough to change the code that's now in a Jupyter notebook into a stand-alone Python library. This skill is useful for helping a data analysis team make their research into production-ready code.

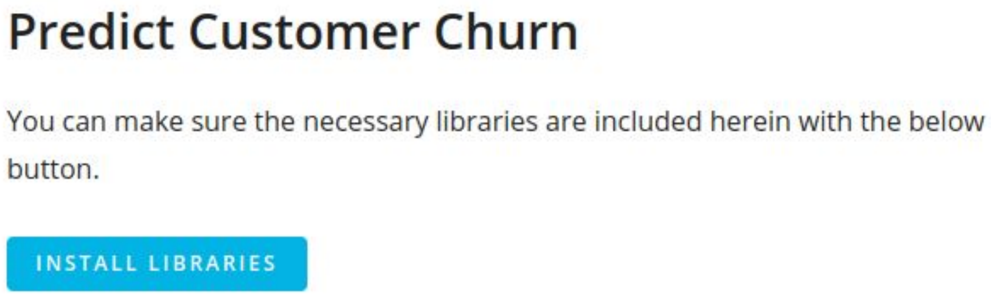
What Should be Submitted?

1. churn_library.py: The completed/refactored script
2. churn_script_logging_and_testing.py: Unit testing script
3. images/* and results/* : Plots and resulting model objects
4. README.md: A README file describing your project and how to run the scripts

Project Walkthrough

Before Starting the Project...

If you're working on the workspace, make sure to click the INSTALL LIBRARIES button from the Guide notebook:



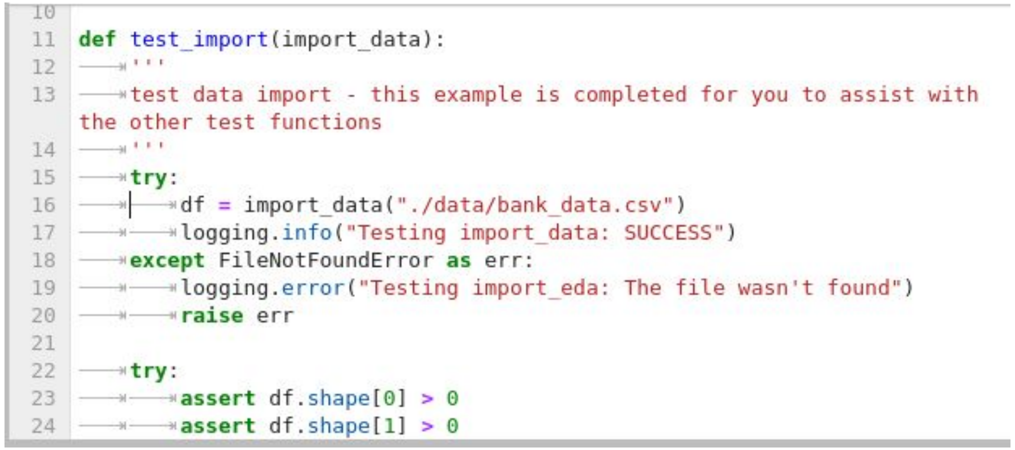
Predict Customer Churn

You can make sure the necessary libraries are included herein with the below button.

If you're attempting this project locally (as I did), create a Python 3.6.3 environment and then run `pip install -r requirements_py3.6.txt` or `conda install --file requirements_py3.6.txt`.

⚠️ Before Starting the Project...(2)

Some code lines are using tabs instead of spaces. They may cause issues. To be safe, convert them all into either tabs or spaces:



```
def test_import(import_data):
    '''
    test data import - this example is completed for you to assist with the other test functions
    '''
    try:
        df = import_data("./data/bank_data.csv")
        logging.info("Testing import_data: SUCCESS")
    except FileNotFoundError as err:
        logging.error("Testing import_eda: The file wasn't found")
        raise err

    try:
        assert df.shape[0] > 0
        assert df.shape[1] > 0
```

Unit Testing

Why Do Unit Testing?

Most learners jump straight into writing the main script, churn_library.py, and later write churn_script_logging_and_tests.py, often as an afterthought. While this could work, it's not the best approach for this project. The reason is that the part that runs the GridSearchCV process takes a very long time to run (about 30 minutes). Wouldn't it be nice to have a script that you can run to test a faster version of your code, and then run the main script only when everything works? That's where unit testing would be useful.

A software development method where you write unit tests as you write your software is called **Test Driven Development (TDD)**.

Note also that **writing testable code is much more important than the actual tests**. Read more about this here: Unit Tests Are Overrated: Rethinking Testing Strategies

Why is writing testable code more important than the actual tests?

- It is usually impossible, and generally impractical, to write unit tests for all parts of your application.
- Even if you managed to accomplish that feat, the application may still have issues when used by an actual user. It could be caused by integration issue i.e. how the components are talking to each other; other culprit might be the UI problem, which is very difficult to test programmatically.
- That said, unit testing is still very useful. For your code to be able to be unit tested, it needs to be well-written. It needs to be modular and not depending on global variables, for example, to test it properly.

Testable Code Example

Let's say we have the following function that simply draws a plot to a file:

```
def draw_plot(df):
    plt.figure(figsize=(8, 5))
    plot = df['Churn'].hist()
    plt.tight_layout()
    fig = plot.get_figure()
    fig.savefig('images/eda/age_hist.png')
    plt.close(fig)
```

The code looks innocent enough, but it is rather troublesome to unit test this code.

- One reasonable way to test is by checking if the file is created after running this function.
- Therefore, we need to delete the file before running this function so we may test its creation.
- On the other hand, we do not want to delete the file if it was created by the production code.
- Therefore, we are "forced" to add another parameter into our draw_plot() function to set the image path.

Testable Code Example (2)

The final code that can be easily unit tested would be as follows:

```
def draw_plot(df, image_pth='images/eda/age_hist.png'):
    plt.figure(figsize=(8, 5))
    plot = df['Churn'].hist()
    plt.tight_layout()
    fig = plot.get_figure()
    fig.savefig(image_pth)
    plt.close(fig)
```

With this code, you may have a unit testing function that passes a parameter `image_pth` that points to a test image path e.g. `test_images/eda/age_hist.png`.

Note that this function is a better function compared to the previous one, as it is more flexible i.e. you may set the image path through a parameter, which reinforces the importance of writing a testable code.

What Does a Test Code Look Like?

To test the previous function, you may create the following function:

```
import main_script as ms

def test_draw_plot():
    df = # ... create the DataFrame
    test_pth='images/eda/age_hist.png'
    ms.draw_plot(df, image_pth=test_pth)
    try:
        assert(os.path.isfile("./test_images/eda/age_hist.png"))
        logging.info("success message")
    except AssertionError as err:
        logging.error("error message")
        raise err
```

This unit test function can be called just as-is to test the draw_plot() function. However, to improve the development flow, I strongly recommend using a unit testing module such as **pytest**.

⚠️ Note About the Sample Test Code

Take a look at the sample `test_import()` function from Udacity provided `churn_script_logging_and_tests.py` script:

```
def test_import(import_data):
    '''
    test data import - this example is completed for you to assist with the other test functions
    '''
    try:
        df = import_data("./data/bank_data.csv")
        ...
```

Notice here, and in other template functions, they expect the function to test (in this case `import_data()`) as a parameter. As far as I know, this is not a common practice in unit testing. You may do away without them.

Pytest

Unit Testing Code with pytest

1. Use pytest.ini to set up logging.
2. Use fixture to create complex objects.
3. Create custom assertion functions.

To run tests, run the following command:

```
pytest churn_script_logging_and_tests.py
```

Use pytest.ini to Set Up Logging

A pytest.ini file may look like the following

```
[pytest]
log_cli = true
log_cli_level = INFO
log_file = ./churn_library.log
log_file_level = INFO
log_file_format = %(name)s - %(levelname)s - %(message)s
log_file_date_format = %Y-%m-%d %H:%M:%S
```

By having this file in the same directory as `churn_script_logging_and_tests.py`, you will store the logging message in `./churn_library.log` anytime you use `logging.info()` and `logging.error()`.

Use Fixture to Create Complex Objects

Complex objects such as pandas Dataframes can sometimes take multiple steps to recreate. Rather than rewriting the code, you may create a fixture for them then pass them onto the test function:

```
@pytest.fixture
def df():
    df = cls.import_data("./data/bank_data.csv")
    return df

# Then pass the fixture into a test function
def test_eda(df):
```

Better yet, you may pass in parameter `scope='module'` to the decorator to reuse the created `df` object:

```
@pytest.fixture(scope='module')
```

Create Custom Assertion Functions

One of the rubric specifications requires you to log both success and error messages.

It can take many repeated lines of code to do this for all assertions. Instead, consider creating a custom assertion function. For example, this function asserts truthness while writing log message upon success and failure:

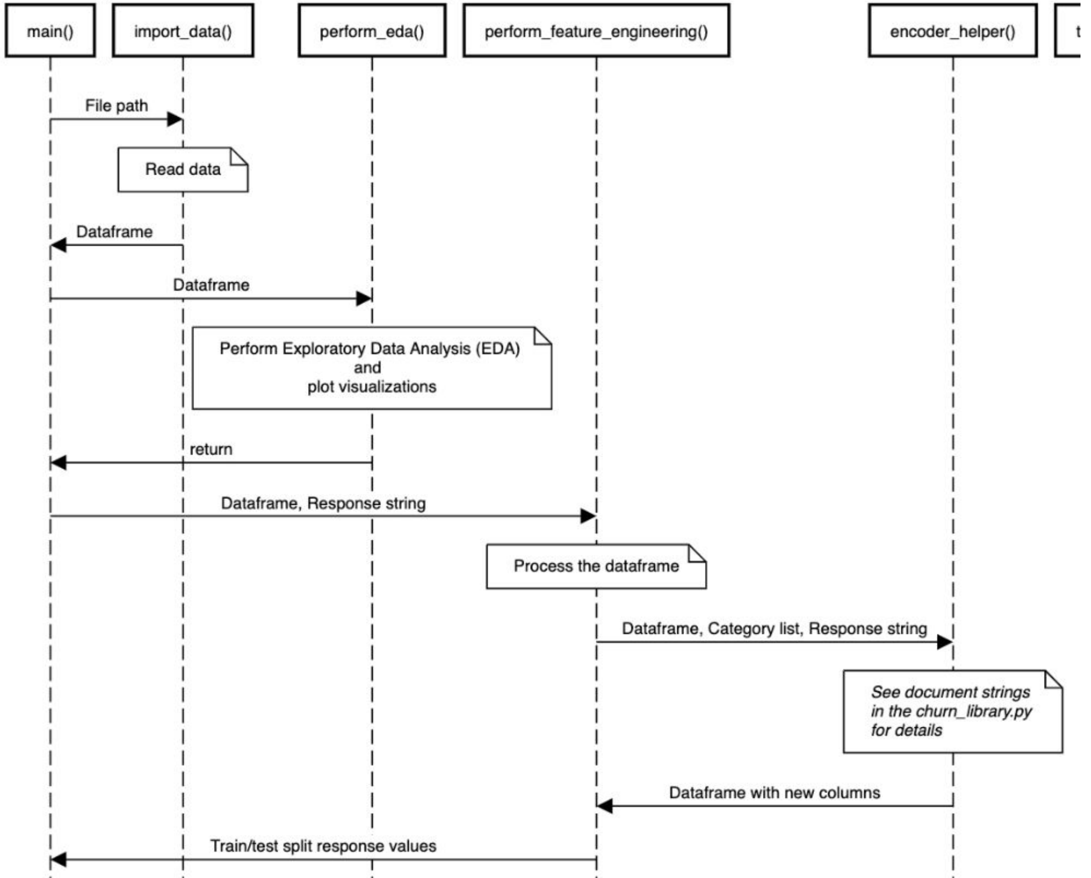
```
def assert_true(condition, message_when_succeed, message_when_fail):
    try:
        assert condition
        logging.info(message_when_succeed)
    except AssertionError as err:
        logging.error(message_when_fail)
        # err.args requires a set
        err.args = (message_when_fail,)
        raise err
```

Coding the Library

Sequence Diagram

Use the sequence diagram to guide your coding. The sequence on the right, for example, tells you that the code needs to do the following:

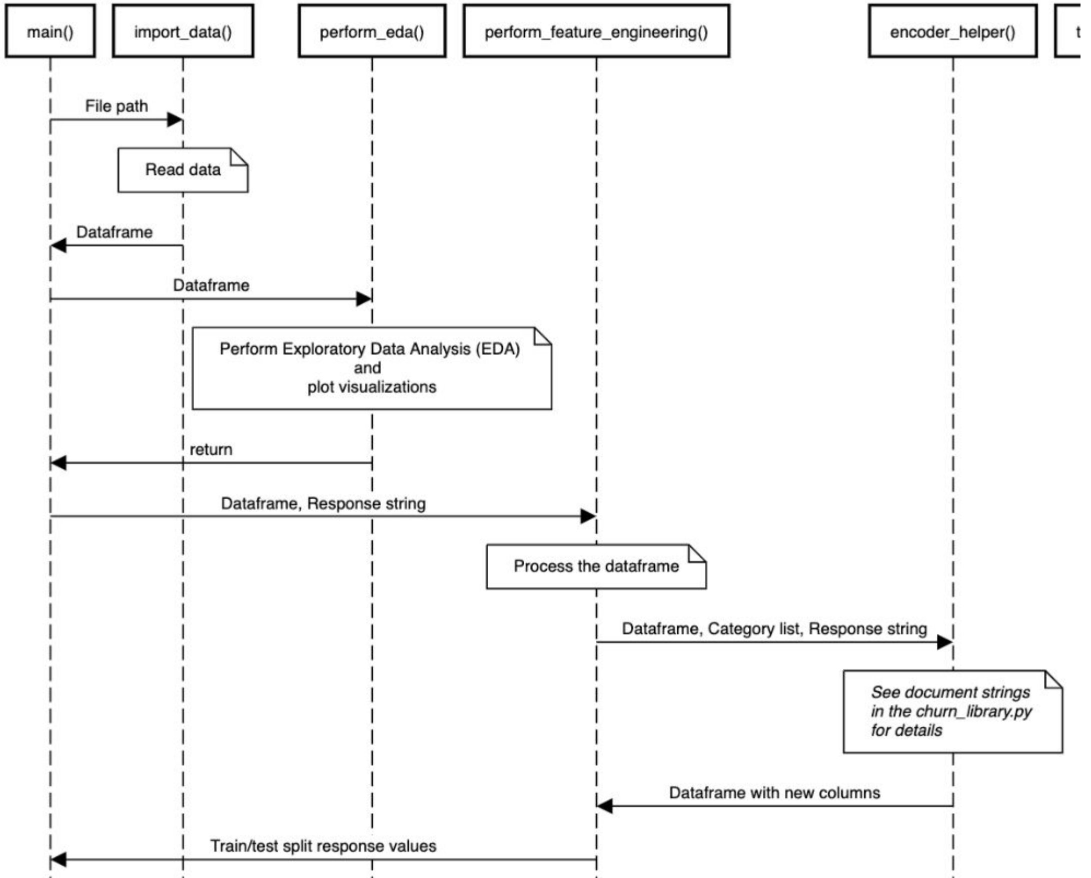
1. A main() function that passes File path to import_data() function, then returns a DataFrame object.
2. Pass the DataFrame object into perform_eda() function. Returns nothing.


main()
import_data()
perform_eda()
perform_feature_engineering()
encoder_helper()
File path
Read data
Dataframe
Dataframe
Perform Exploratory Data Analysis (EDA) and plot visualizations
return
Dataframe, Response string
Process the dataframe
Dataframe, Category list, Response string
See document strings in the churn_library.py for details
Dataframe with new columns
Train/test split response values

UDACITY

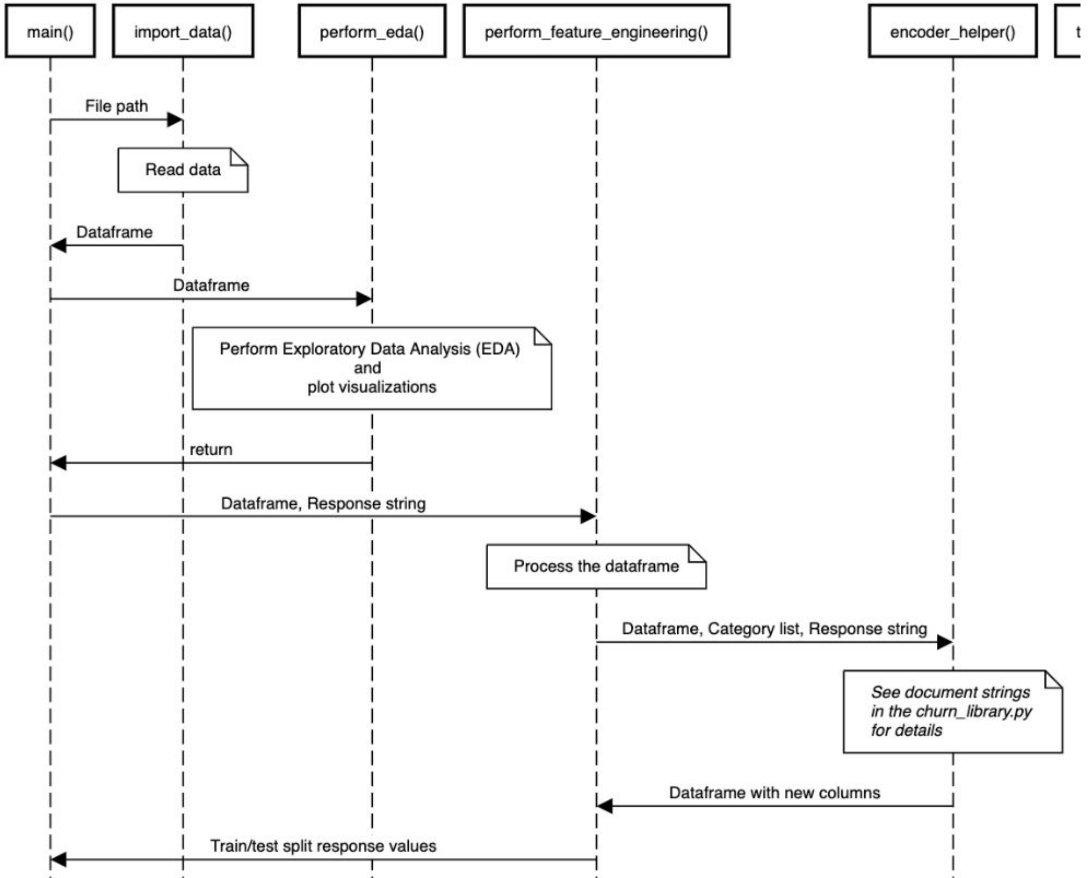
Sequence Diagram (2)

3. Pass the DataFrame object into perform_feature_engineering() function.
4. The perform_feature_engineering() function passes the DataFrame object along with Category list and Response string to encode_helper().


main()
import_data()
perform_eda()
perform_feature_engineering()
encoder_helper()
File path
Read data
Dataframe
Dataframe
Perform Exploratory Data Analysis (EDA) and plot visualizations
return
Dataframe, Response string
Process the dataframe
Dataframe, Category list, Response string
See document strings in the churn_library.py for details
Dataframe with new columns
Train/test split response values

Sequence Diagram (3)

5. And, finally, the encode_helper returns the encoded Dataframe object to the perform_feature_engineering() function, which is then split into train/test datasets, which are returned to the main() function.


main()
import_data()
perform_eda()
perform_feature_engineering()
encoder_helper()
File path
Read data
Dataframe
Dataframe
Perform Exploratory Data Analysis (EDA) and plot visualizations
return
Dataframe, Response string
Process the dataframe
Dataframe, Category list, Response string
See document strings in the churn_library.py for details
Dataframe with new columns
Train/test split response values

⚠️ How to Save Plot Images

The rubric requires you to save plot images instead of displaying them. Doing this can get confusing due to different plot types and libraries. In general, here's the code needed to save a figure:

```
import matplotlib.pyplot as plt

# Create the figure
plt.figure(figsize=(15, 8))

[plotting code]

# Save
plt.savefig(image_path)

# Don't forget to close the plot; otherwise, multiple plots are going to be printed on a single image.
plt.close()
```

Polishing Up

⚠️ Before Submitting the Project...

- Write a proper README documentation:
  - The project description must tell readers what this project is about. You may copy some text from the **Project Overview** page.
  - In the **Files and data description** section, list **all** of your files and add some explanations on them.
  - In the **Running Files** section, you may tell readers how to run your main and test scripts.
- Add a docstring to every single function in your scripts. The docstring must follow best practices in the **Docstrings** lesson.
- Add a documentation on top of every single script. Each documentation must have a description, Author, and Creation Date. Here's an example:

```
'''
This library has all the functions needed for churn prediction.

Author: Jay Teguh
Creation Date: 08/17/2023
'''
```

- Run pylint on your scripts and make sure the scores are 7.00+. For help with formatting, you may use the autopep8 application. See the lesson on **Auto-PEP8 & Linting** for more details.

Need More Help?

See my project as a reference:

https://github.com/jaycode/udacity-predict-customer-churn-with-clean-code

Q&A

Session Feedback Form

https://airtable.com/shr7WGplyaZIWq8pE

Tips (again)

1. Check the Project Rubric often.

https://review.udacity.com/#!/rubrics/2475/view

2. Have a quick question? Shoot an email to teguhwpurwanto@gmail.com.

Thank you

UDACITY