CSE 6748: Applied Analytics Practicum

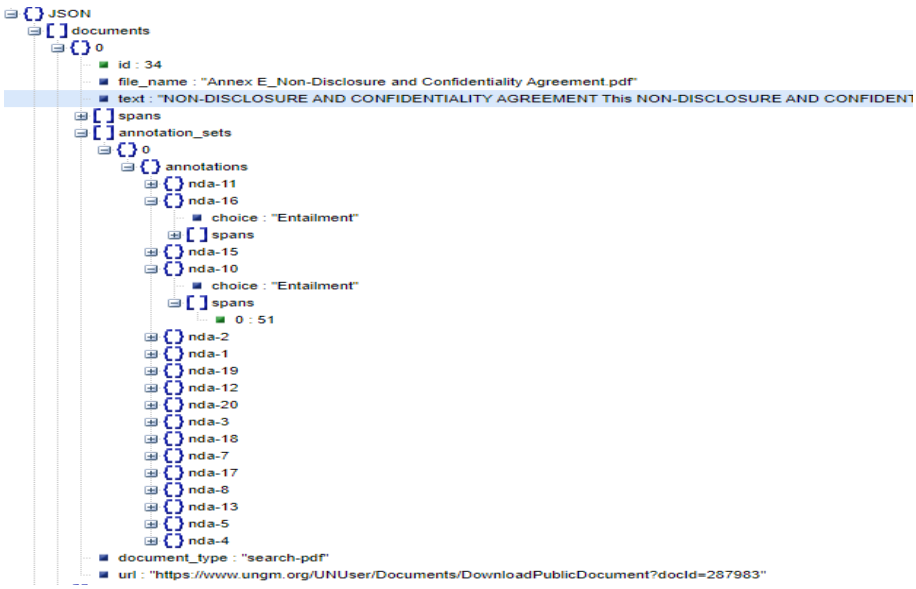
Siddharth Gudiduri
GT ID: 903464761

Overview

I was tasked with developing an end-to-end process to automate contract reviews for the CSE 6748 project. Our goal was to create a process that resembled that of an experienced attorney, but with increased speed and accuracy. This type of service will not only maximize document review but will also reduce the risk of human error by mitigating unacceptable or missing clauses. Demonstration of concept The source code for this project can be found at https://github.com/sgudiduri/CSE-6748. Because there are many details within the algorithm and check-in omitted, this code is only for your preview and not the company's active repo check-in. Contract Review Automation (CRA) is divided into three mandatory parts and three optional parts. CRA stages include data analysis, model development, productionizing code, and creating model API, Deploy to PaaS, Testing. Bonus tasks involve implementing Pocket base, Caching via Redis and Scaling concepts.

1. Data Analysis

Data is received json format. Training data contains 423 documents, and test 123 documents. structure shown below



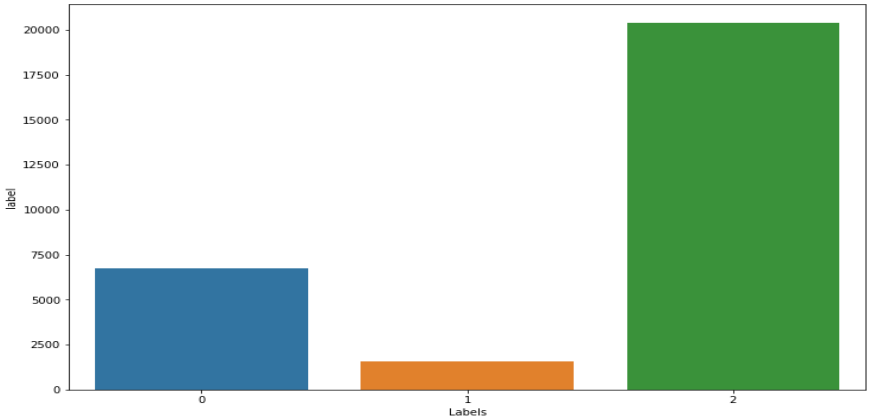
At the document level, we have the "text" key, which contains the entire document text, and the "spans" key, which divides the text into a list of premises. The key term "annotation sets" refers to a list of multiple annotations for a given document. At the annotation level, each key "nda-1," "nda-2," and so on represents a hypothesis that either implies, contradicts, or is neutral to the

given document. The "spans" key is indexed at the document level under each hypothesis. For example, "nda-1" includes the spans 1, 13, and 91. At the document level, span 1 corresponds to sentence text indexed between characters [25, 89]. Each hypothesis' text sequence is described by the "labels" key.

The next step in data analysis is to process the data and extract the features that I will need for building. This feature engineer step was recorded and will be used later in the machine learning pipeline to process incoming data. The extracted Tibble used for Model building is shown below.

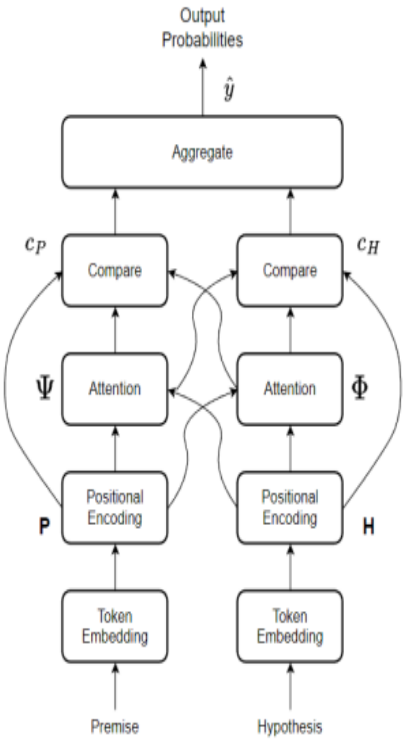
	hypotheis	premise	label
31	Receiving Party shall not reverse engineer any...	i obtained by the Recipient without restrictio...	2
56	Receiving Party shall not reverse engineer any...	For and on behalf of UNHCR For and on behalf o...	2
13	Receiving Party shall not reverse engineer any...	NOW THEREFORE the Parties agree as follows	2
39	Receiving Party shall destroy or return some C...	5 All Confidential Information in any form and...	0
40	Receiving Party shall destroy or return some C...	a if a business relationship is not entered in...	0
...	...	...	...
10	Receiving Party shall not use any Confidential...	a make use of any of the Provider s Confidenti...	0
16	Receiving Party shall not use any Confidential...	Neither the Recipient nor any of the Recipient...	0
89	Receiving Party shall not use any Confidential...	c irrevocably and unconditionally waives the r...	2
6	Receiving Party shall not use any Confidential...	This Agreement sets forth the Parties obligati...	2
48	Receiving Party shall not use any Confidential...	a of this Section 7	2

I discovered that the data was imbalanced, which will play a role in selecting hyperparameters for model building. The screenshot below depicts a higher frequency of neutral cases, followed by entitlement and contradiction. Here's a research code:



2. Model Building

I modified one of the existing architectures Decomposable Attention Model with some minor changes like adding additional layers, dropouts, and attention mechanism.



Token Embedding:

continuing for word representation, I used Global Vectors (GloVe) embedding, an unsupervised learning algorithm for obtaining vector representations for words, with training performed on aggregated global word-word co-occurrence statistics from a corpus. GloVe 6B 100d is used to perform word embedding.

Positional Encoding:

Below formula uniquely encodes information about the position of a token.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Here, d is the embedding dimension, pos is the position of the token in the sequence, and i maps to sin and cosine functions.

Attending:

We perform soft alignment of the premise and hypothesis essentially achieved by passing the input premise and hypothesis through a multi-layer perceptron and then computing soft attention weights

$$w_{ij} = F(p_i)^T F(h_j)$$

where F is the multi-layer perceptron with ReLU nonlinear activation that maps pi, hj to a hidden dimension space. This allows us to calculate the projection of the premise over the hypothesis. The intuition behind the alignment model is based on a bidirectional RNN used as an encoder and decoder

$$\Phi_i = \sum_{j=1}^{b} \frac{exp(w_{ij})}{\sum_{k=1}^{b} exp(w_{ik})} h_j$$

$$\Psi_j = \sum_{i=1}^{a} \frac{exp(w_{ij})}{\sum_{k=1}^{a} exp(w_{kj})} p_i$$

Comparing:

In the compare section, all the tokens from one sequence, with their corresponding weights are compared with a token in the other sequence.

$$c_{P,i} = G([p_i, \Phi_i]), i = 1...a$$

$$c_{H,j} = G([h_j, \Psi_j]), j = 1...b$$

The representation is the concatenation of premise token pi and the softly aligned weight representation for that token Φi . A similar operation is performed for the hypothesis as well. As the concatenation operation is performed along the embedding dimension, the multi-layer perceptron G maps input dimension equal to twice the embedding dimension, to the number of hidden units.

Aggregating:

The final step performed by the decomposable attention model is aggregating the information obtained from the comparison step. The information in the comparison vectors is aggregated through a summation operation. The summed-up results are now fed into a multi-layer perceptron H and are mapped to the number of outputs - Entailment, Contradiction and Neutral. Below are learnable parameters:

$$c_P = \sum_{i=1}^{a} c_{P,i}$$

$$c_H = \sum_{j=1}^{b} c_{H,j}$$

$$\hat{y} = H([c_P, c_H])$$

Focal Loss:

While training the decomposable attention model, we use focal loss as there exists class imbalance among the 3 classes - Entailment, Contradiction and Neutral.

$$CB_{focal}(z, y) = -\frac{1-\beta}{1-\beta^{n_y}} \sum_{i=1}^{C} (1 - p_i^t)^{\gamma} log(p_i^t)$$

The β hyper-parameter can be tuned to perform reweighting. When pt is small and consequently, (1 − pt)γ is close to 1, then Focal loss becomes classic cross entropy, and would result in incorrect classification by the model. As the model adjusts its weights, Focal Loss scales down the contribution of easy examples during training and instead focuses on the harder examples, resulting in an improvement in prediction accuracy for the minor classes.

Results:

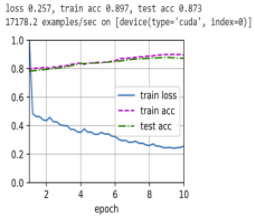


Figure 3. Loss (CE) and Accuracy Curves (Standard Dataset)

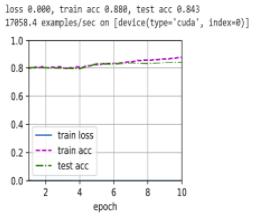


Figure 4. Loss (Focal) and Accuracy Curves (Standard Dataset)

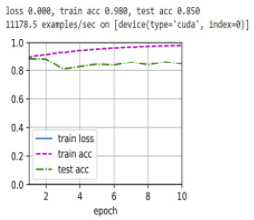


Figure 5. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

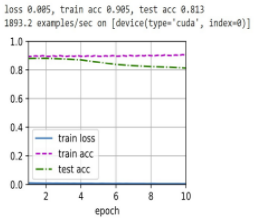


Figure 6. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

Optimizer [SGD, ADAM]	Batch Size [64, 128, 256]	Learning Rate [1e-2, 1e-3, 1e-4]	Regularization [1e-3, 1e-4, 1e-5]	Entailment F1 Score	Contradiction F1 Score	Neutral F1 Score	Accuracy
ADAM	256	1e-2	1e-3	0.08	0.28	0.54	0.88

Table 1. Optimal Hyper-parameters and Validation Results for Standard Contract NLI Dataset using Cross Entropy Loss

Optimizer [SGD, ADAM]	Batch Size $\mathbb{N}$ [64, 128, 256]	Learning Rate α [1e-2, 1e-3, 1e-4]	Regularization λ [1e-3, 1e-4, 1e-5]	Class Re-balance Factor β [0.75, 0.9, 0.999]	Modulating Factor γ [1,2]	Entailment F1 Score	Contradiction F1 Score	Neutral F1 Score	Accuracy
ADAM	256	1e-2	1e-3	0.9	2	0.46	0.22	0.86	0.84

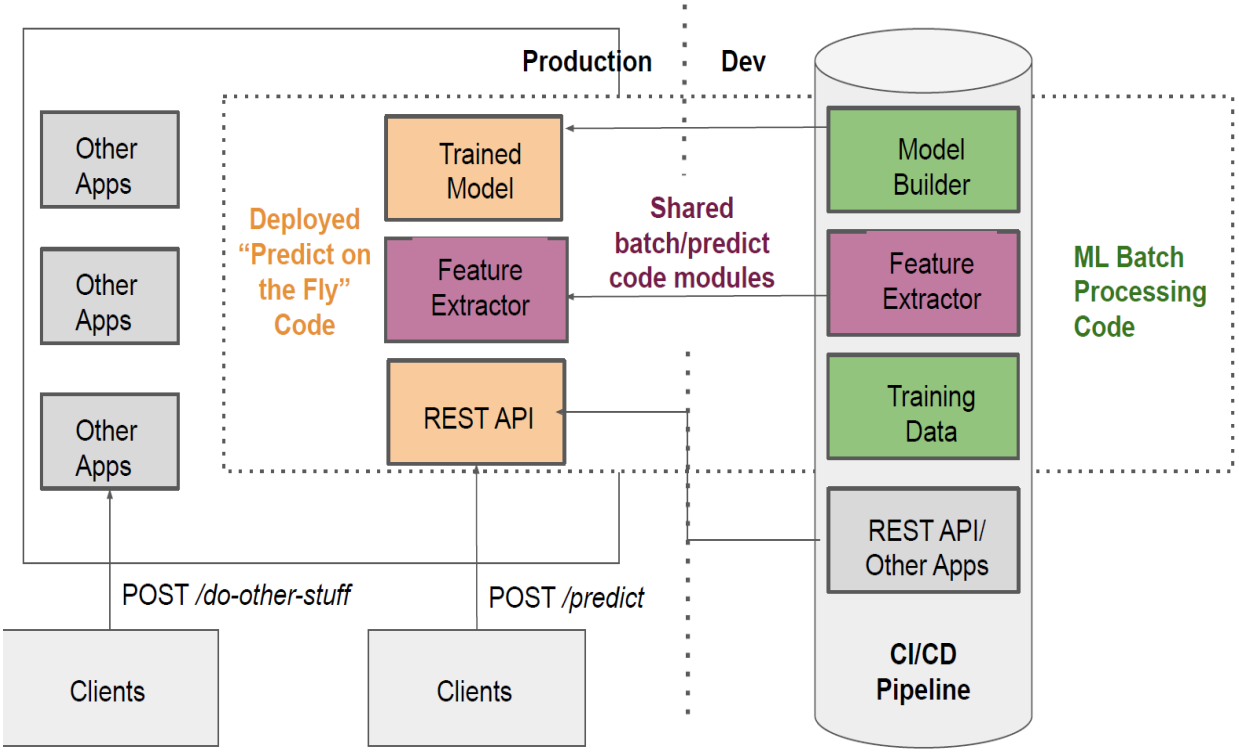
Table 2. Optimal Hyper-parameters and Validation Results for Standard Contract NLI Dataset using Focal Loss

using the above hyperparameter we got the best precision, recall, f1-score along with accuracy and loss. After selecting model and hyperparameter our next task was to productionizing code.

3. Architecture Component Breakdown

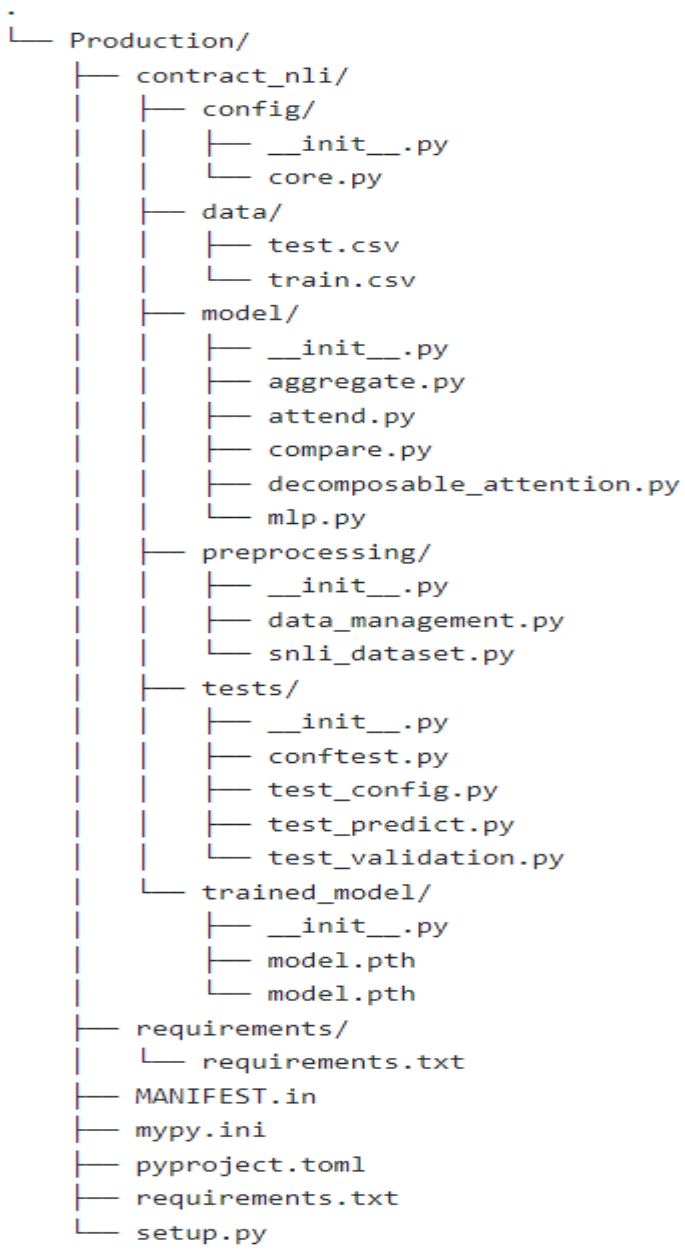
Next our goal is building the below architecture by creating model package, web api and CI/CD pipelines for package and api. Below is architecture breakdown

Diagram: Train by batch, predict on the fly, serve via REST API



4. Productionizing Model package

Continuing with the project, the next step was to write production code that would be deployed to end users. I kept my attention focused on testability, maintainability, scalability, performance, and reproducibility. The package structure created below breaks down research code into separation of concern components, implying that each module has a single responsibility in performing its function. As a result, the config package will only contain configuration modules. Only modules designed for testing will be included in the testing package. NOTE: The company code contains a lot more unit tests and an ensemble of models with different hyperparameters.



As you can see from the folder structure above, I created various model modules to build a deep learning package. The dependency injection pattern, which involves passing objects that objects require rather than creating them, aided in the creation of scalable and testable code. Below is an implementation of pydantic, a Python library that simplifies configuration code and compiles it into an object that can be passed into various parts of the application.

```python
from pathlib import Path
import typing as t
from pydantic import BaseModel, validator
from strictyaml import load, YAML
import os


# Project Directories
PWD = os.path.dirname(os.path.abspath(__file__))
PACKAGE_ROOT = Path(os.path.abspath(os.path.join(PWD, '..')))
ROOT = PACKAGE_ROOT.parent
CONFIG_FILE_PATH = PACKAGE_ROOT / "config.yaml"
TRAINED_MODEL_DIR = PACKAGE_ROOT / "trained_model"
DATA_DIR = PACKAGE_ROOT / "data"

class AppConfig(BaseModel):
    """
    Application-level config.
    """
    package_name: str
    train_path: str
    test_path: str
    vocab_path: str
    model_path: str


class ModelConfig(BaseModel):
    """
    All configuration relevant to model
    training and feature engineering.
    """
    num_step: int
    batch_size: int
    learning_rate: float
    embed_size: int
    num_hiddens: int
    epochs: int
    save_best: bool
    trainer: str
    loss: str

```

Below are some screenshots for Data Service class

29 lines (26 sloc) | 1.17 KB

Raw Blame

```python
import pandas as pd
import torch
import d2l
from .snli_dataset import SNLIDataset

class DataService():
    def __init__(self):
       pass

    #loads csv into pandas dataframe.
    def load_data(self, train_path, test_path):
        train = pd.read_csv(train_path)
        test = pd.read_csv(test_path)
        train["label"] = train["label"].astype(int)
        test["label"] = test["label"].astype(int)
        return train, test

    #load pandas dataframe to dataset.
    def create_snli_dataset(self, train, test, num_steps = 50, batch_size = 256, num_workers = 4):
        train_set = SNLIDataset(train, num_steps)
        test_set = SNLIDataset(test, num_steps, train_set.vocab)
        vocab = train_set.vocab
        train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                                 shuffle=False,
                                                 num_workers=num_workers)
        test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                                shuffle=False,
                                                num_workers=num_workers)
        return train_iter, test_iter, vocab
```

Below are some screenshots for testing code modules

main CSE-6748 / production / contract_nli / tests / test_predict.py / <> Jump to Go to file ...

sgudiduri Create/UpdateUnit tests ... Latest commit afc0a80 2 weeks ago History

1 contributor

19 lines (16 sloc) | 1.02 KB Raw Blame

```
'''
Here is where we want to test inputs, outputs and model quality
below are sample tests
'''

#Testing for entailment condition
def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_1):
    res = load_predict_class.make_single_prediction(sample_input_data_1["premise"].split(), sample_input_data_1["hypothesis"].split())
    assert res == sample_input_data_1["result"]

#Testing for Contradiction condition
def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_2):
    res = load_predict_class.make_single_prediction(sample_input_data_2["premise"].split(), sample_input_data_2["hypothesis"].split())
    assert res == sample_input_data_2["result"]

#Testing for neutral condition
def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_3):
    res = load_predict_class.make_single_prediction(sample_input_data_3["premise"].split(), sample_input_data_3["hypothesis"].split())
    assert res == sample_input_data_3["result"]
```

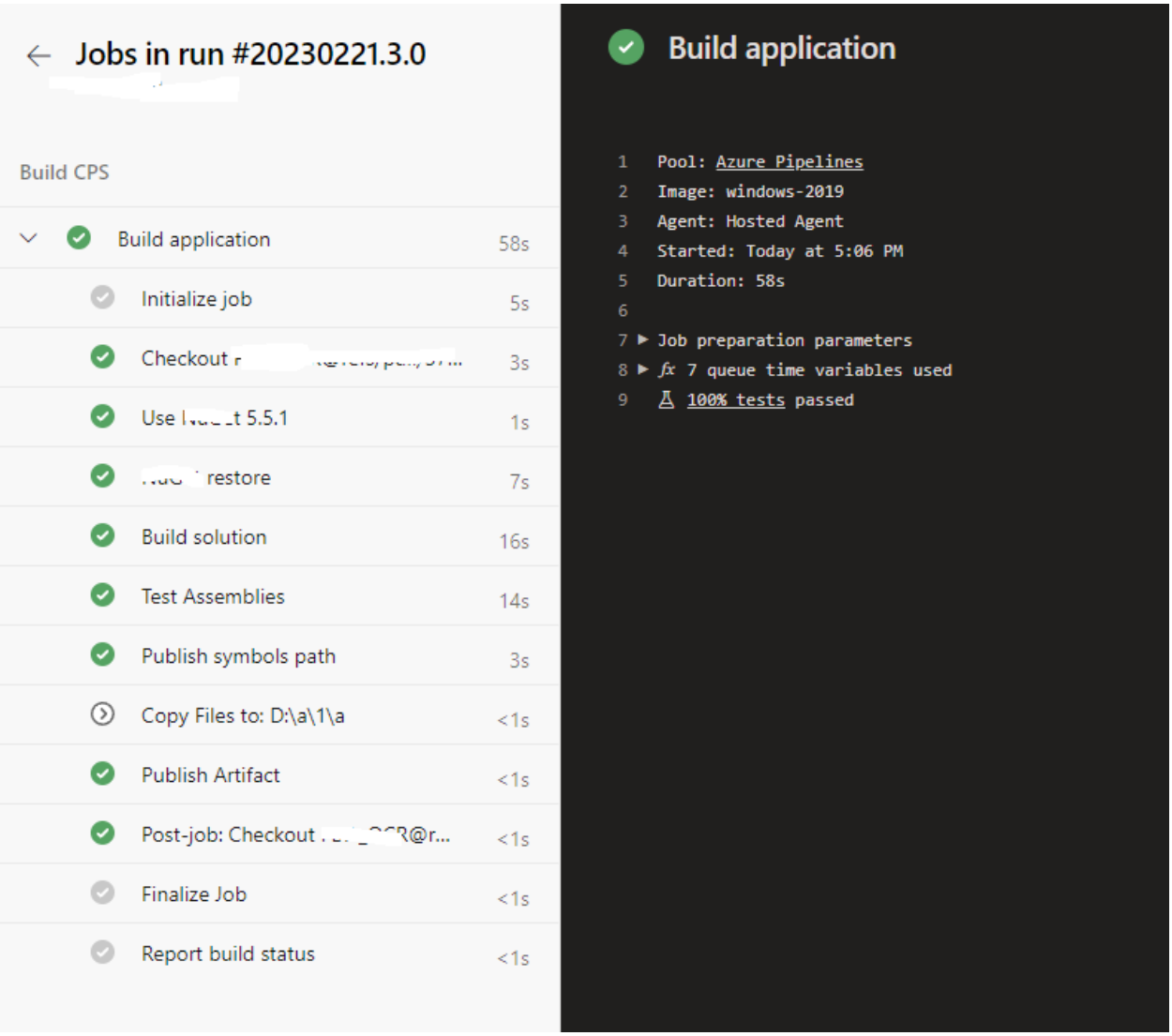
63 lines (53 sloc) | 1.96 KB

Raw Blame

```python
#!/usr/bin/python

'''
https://stackoverflow.com/questions/34466027/in-pytest-what-is-the-use-of-conftest-py-files
conftest.py is used to define
fixture used to define static data used by tests
External plugin
Hooks
Test root path
'''

import pytest
from contract_nli.config.core import config, TRAINED_MODEL_DIR
from contract_nli.predict import Predict

trained_model_dir_path = TRAINED_MODEL_DIR.as_posix()
model_config = config.model_config
embed_size=model_config.embed_size
num_hiddens=model_config.num_hiddens

@pytest.fixture()
def raw_app_config():
    #For larger datasets, here we would use a testing sub-sample.
    return config.app_config

@pytest.fixture()
def raw_model_config():
    return model_config

@pytest.fixture()
def load_predict_class():
    model_path =  f"{trained_model_dir_path}/{config.app_config.model_path}"
    vocab_path =  f"{trained_model_dir_path}/{config.app_config.vocab_path}"
    pr = Predict(model_config.embed_size,model_config.num_hiddens, model_path, vocab_path)
    return pr

@pytest.fixture()
def sample_input_data_1():
    row_1 = {
        "hypothesis":"Receiving Party shall destroy or return some Confidential Information upon the termination of Agreement",
        "premise": "1 the completion or termination of the dealings between the parties contemplated hereunder or",
        "result": "Entailment"
        }
    return row_1

@pytest.fixture()
def sample_input_data_2():
    row_2 = {
        "hypothesis":"All Confidential Information shall be expressly identified by the Disclosing Party",
        "premise": "i marked confidential or proprietary or",
        "result": "Contradiction"
        }
    return row_2

@pytest.fixture()
def sample_input_data_3():
    row_3 = {
        "hypothesis":"Receiving Party shall not reverse engineer any objects which embody Disclosing Party s Confidential Information",
        "premise": "6 Compelled Disclosure of Confidential Information",
        "result": "neutral"
        }
    return row_3
```

Give feedback

After finishing the packaging module, I began integrating Azure pipelines for CI/CD, which stands for continuous integration, continuous delivery, and continuous deployment. This means that when a developer, such as myself, submits code for review and checks in after approval, the code goes through a process of building, testing, and publishing files to a private server. Instead of creating a monolithic application, this is done so that the machine learning model can be integrated with a website or a web API. Here's an example from my company's pipeline after a feature for this project was checked in.



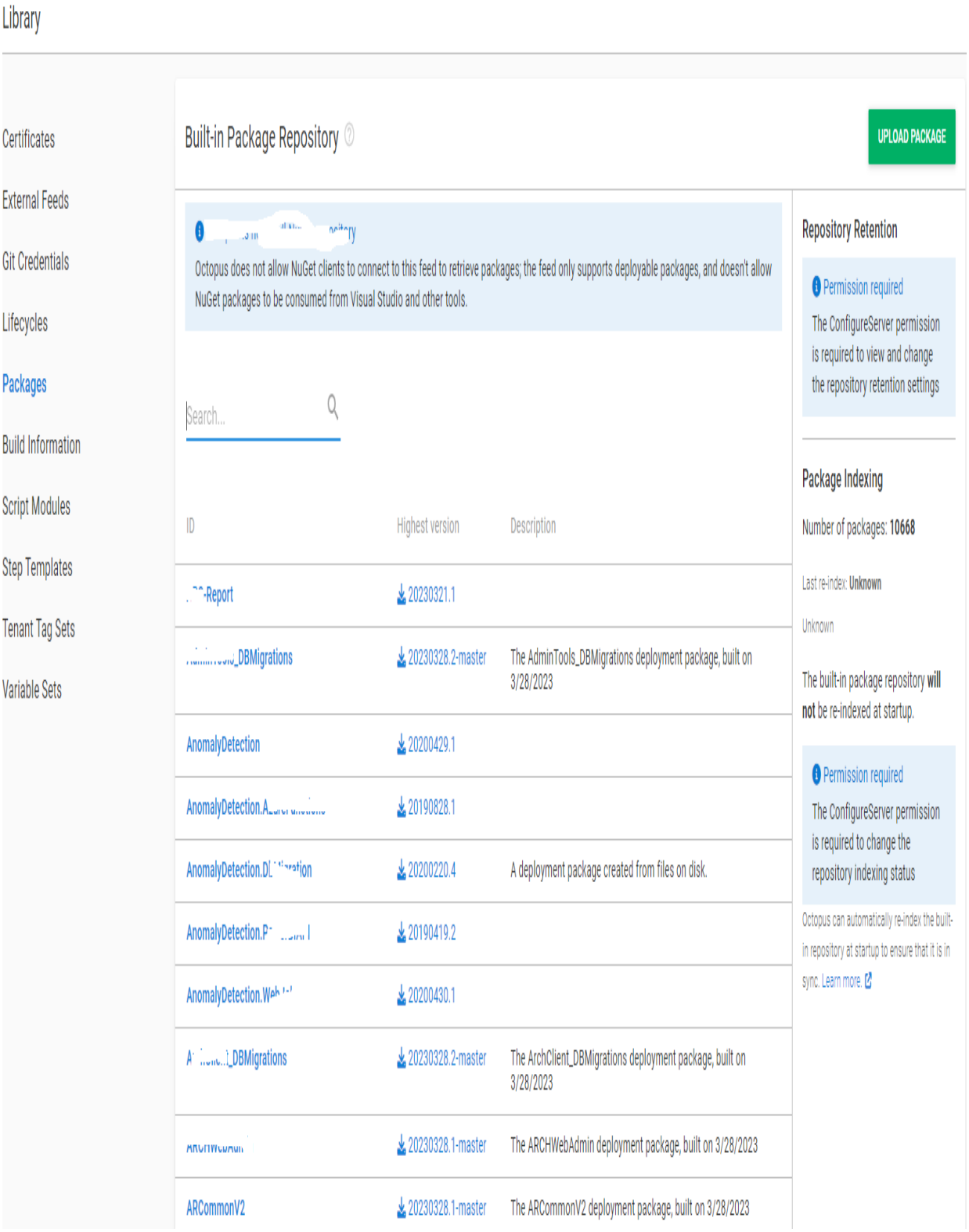
Note, I created a similar example for this class as a POC before integrating with company code. This deployed on pypi is an experimental version and not the model package used at my company. Link can be found here

5. Octopus Private model server

Octopus is a repository of software packages for the various programming language. It is a central location where developers can upload and share their software packages, making it easy for other developers to install and use them in their own projects. Octopus is not specifically a server for deploying machine learning models, but it is commonly used for hosting and distributing Python packages that contain machine learning models, as well as other types of software tools and libraries.

To deploy a machine learning model using Octopus, you would first need to create a Python package that includes the model and any necessary dependencies. You can then upload the package to Octopus using tools like VSTS build or the Octopus web interface. Once the package is uploaded, other developers can install it using package management tools.

Screenshot of library below



6. API to interact with Model

To create Web API to we imported libraries from the private server. Then we defined the input and the output formats for our model. This is a JSON request with specific keys. Next we defined end points using Fast API decorator. Below are the screenshot of the a) file structure b) main.py c) on the web.

a)

```
.
└── ContractNLI_API/
    ├── app/
    │   ├── __init__py
    │   ├── main.py
    │   └── contract_nli/
    │       └── app.py
    ├── tests/
    │   └── contract_nli/
    │       └── app.py
    ├── .dockerignore
    ├── Dockerfile
    ├── Procfile
    ├── requirements.txt
    ├── runtime.txt
    ├── test_requirements.txt
    └── tox.ini
```

b) partial implementation, real implementation is redacted.

```python
from typing import Any

from fastapi import APIRouter, FastAPI, Request, File, UploadFile
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import RedirectResponse, FileResponse
from loguru import logger
import numpy as np
from contract_nli.predict import Predict
from contract_nli.config.core import config,TRAINED_MODEL_DIR

favicon_path = 'favicon.ico'
api_router = APIRouter()

app = FastAPI(
    title="Contract NLI V1", openapi_url=f"/api/v1/openapi.json"
)

def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in set(['png', 'jpg', 'jpeg'])


root_router = APIRouter()


@app.get('/favicon.ico', include_in_schema=False)
async def favicon():
    return FileResponse(favicon_path)

@app.get("/")
async def docs_redirect():
    return RedirectResponse(url='/docs')

@root_router.get("/health", response_model=dict, status_code=200)
async def health() -> dict:
    """
    API Endpoint to get health check
    """
    health = dict(
        name="Contract NLI Language Inference API", api_version="1.0.0", model_version="1.0.7"
    )

    return health

```

```python
@root_router.get("/PredictNLI/{premise}/{hypothesis}", response_model=dict, status_code=200)
async def PredictNLI(premise: str, hypothesis: str ) -> dict:
    """
    Given a premise and hypothesis predict Entailment, Contradiction or neutral
    """
    model_config = config.model_config
    trained_model_dir_path = TRAINED_MODEL_DIR.as_posix()
    vocab_path = f"{trained_model_dir_path}/{config.app_config.vocab_path}"
    model_path = f"{trained_model_dir_path}/{config.app_config.model_path}"
    p = Predict(model_config.embed_size, model_config.num_hiddens, model_path, vocab_path)
    res = p.make_single_prediction(premise.split(), hypothesis.split())
    return dict(Prediction=res)


app.include_router(root_router)

# #Set all CORS enabled origins
if __name__ == "__main__":
    # Use this for debugging purposes only
    logger.warning("Running in development mode. Do not run like this in production.")
    import uvicorn

```

c) deployed version of contract NLI submissions and endpoint that returns prediction.

[redacted] 1.6.0-rc9 OAS3

/openapi.json

Authorize

default

POST	/submissions	Predict
GET	/submissions/{sub_id}	Get Result
DELETE	/submissions/{sub_id}	Delete Result
GET	/status/{sub_id}	Get Status
GET	/artifacts/{sub_id}/{artifact_name}	Get Persistent Artifacts
POST	/config/{document_type}/{config_name}	Update Config Map
GET	/config	Get Config Map
GET	/config/{sub_id}	Get Config Map Update Results

GET /submissions/{sub_id} Get Result

Parameters

Try it out

Name	Description
sub_id * required string($uuid) *(path)*	sub_id

Responses

Code	Description	Links
200	Successful Response	*No links*

Media type

application/json

Controls Accept header.

Example Value | Schema


```
{
  "sub_id": "string",
  "processingTime": 0,
  "usedOCR": true,
  "confidence": 0,
  "n_words": 0,
  "readingMethod": "string",
  "data": [
    {
      "[illegible]ctionName": "str[illegible]",
      [illegible]ing",
      [illegible]
      [illegible]ColumnIndex": 0,
      "output": [
        {
          [illegible]wNum[illegible]": 0,
          [illegible]string",
          [illegible]
          [illegible]
          [illegible]
          "Addi[illegible]": [
```

Next steps:

Now that Machine learning model Web API is in UAT we are going to work on several other tasks. These tasks include monitoring and evaluation of machine learning model. Maintenance, Security and Scaling and then I will be finish with documenting model and its API so others may understand how to work with it.