

# **CSE 6748**

# **Applied Analytics**

# **Practicum**

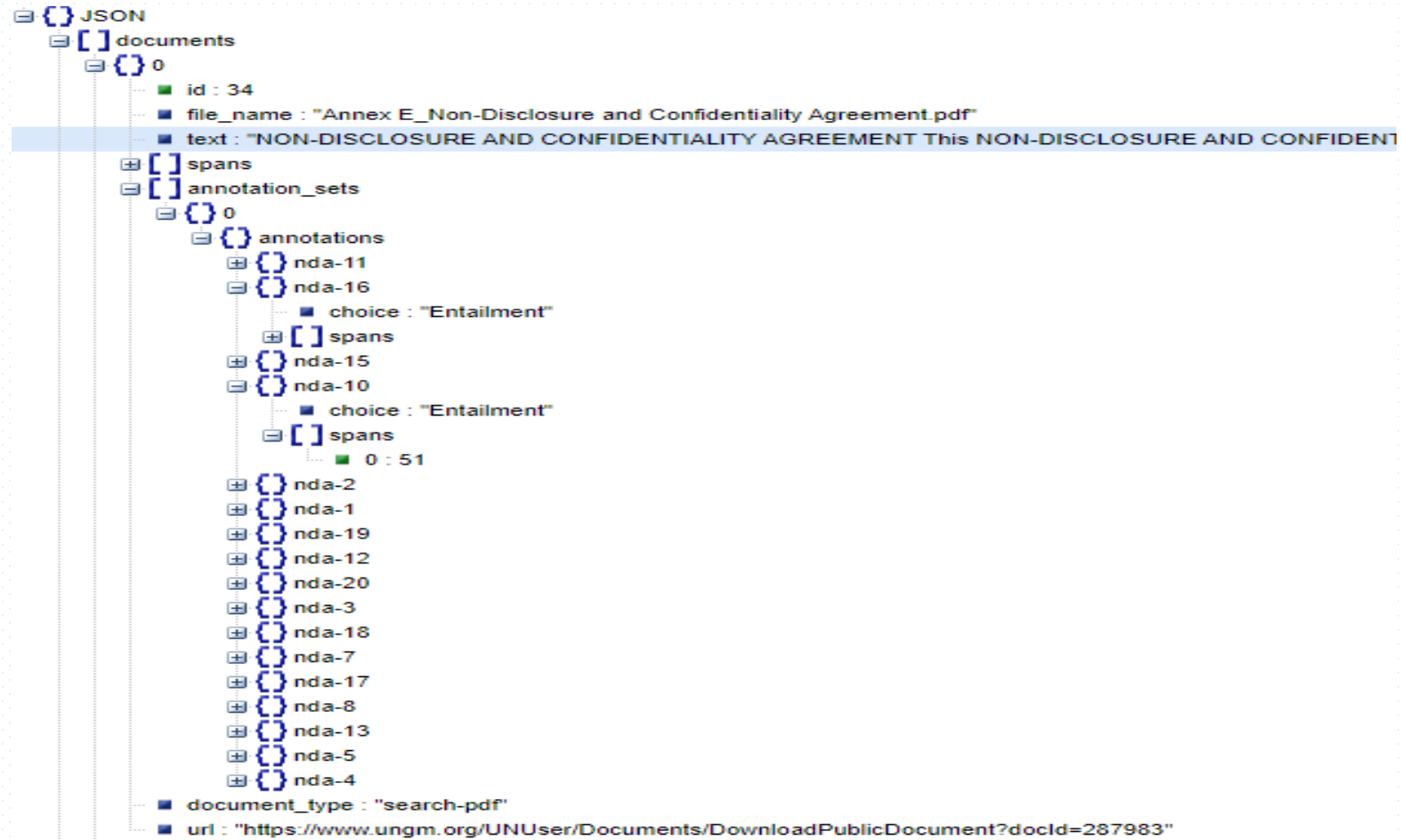
Siddharth Gudiduri, GT ID: 903464761

# Overview

For CSE-6748 project, I was tasked to build an end-to-end process that will automate contract reviews. Our goal was to develop a process mimicking an experienced attorney, but with enhanced speed and accuracy. Such a service will not only maximize document's review but will also minimize risk involved with human error mitigating unacceptable or missing clause. Proof of concept Code for this project can be found at the following location: <https://github.com/sgudiduri/CSE-6748> . Note this code is just for your preview and not company's active repository check-in as there are many details within the algorithm and check-in omitted. Contract Review Automation (CRA) is broken down into required parts and three bonus parts. Stages of CRA involves around Data Analysis, Model development, productionizing code, Creating Model API, Deploy to PaaS, Testing. Bonus tasks involve implementing minikube, 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



## 2. Model Building

Comparing:

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

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

The representation for premise token  $p_i$  and the softly aligned weight  $\Phi_i$  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.

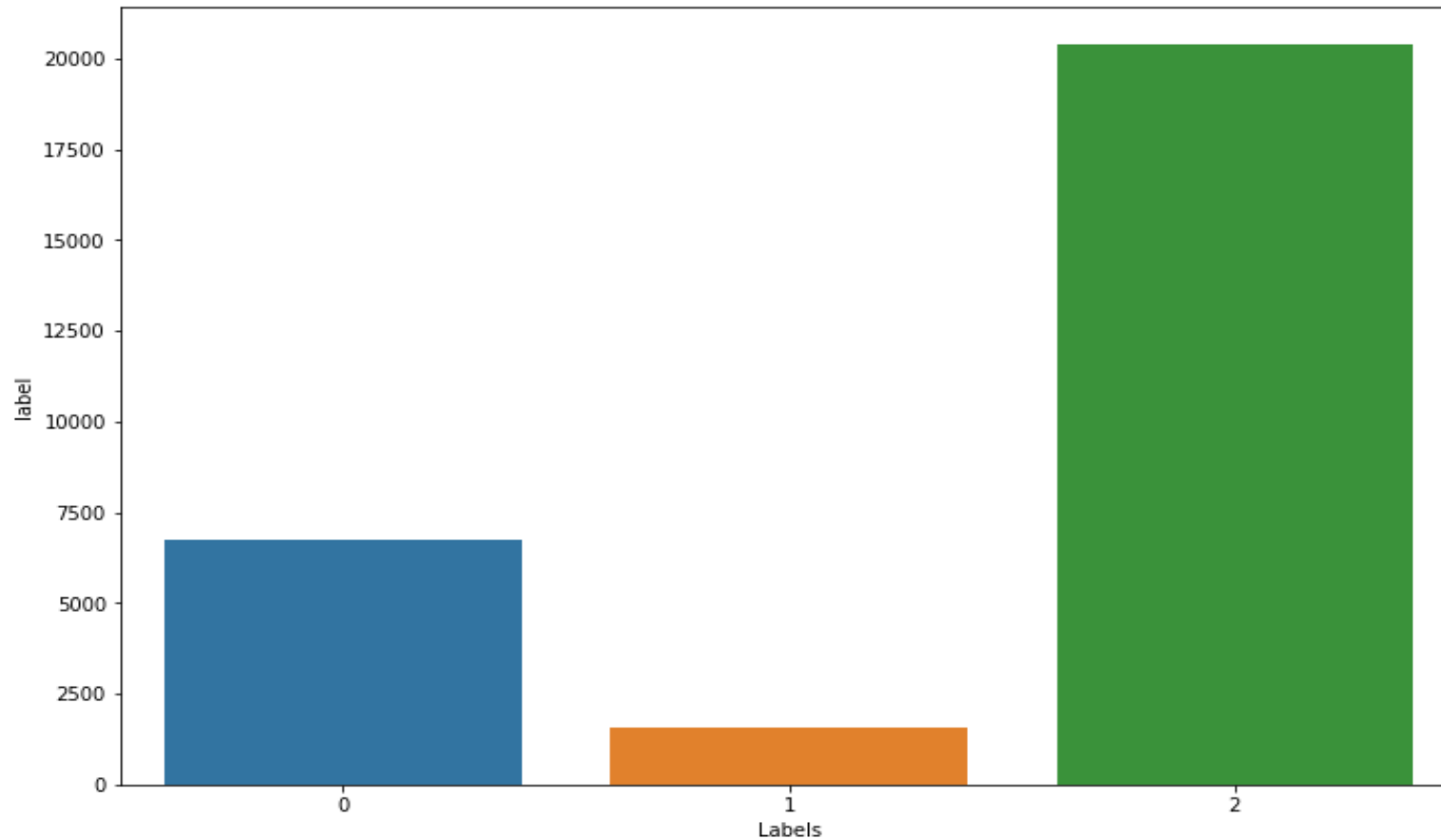
# 1. Data Analysis

At the document level, we have “text” key that contains full document text, and “spans” key splitting text into a list of premises. The key “annotation sets” is a list containing multiple annotations for a given documents. At the annotation level, every key “nda-1”, “nda-2”, etc. is a hypothesis labeled either entails, contradicts, or is neutral to the given document. The “spans” key under each hypothesis is indexed the “spans” key at the document level. Example – “nda-1” entails the spans 1, 13, and 91. Here, span 1 at the document level corresponds to sentence text indexed between characters [25, 89]. The “labels” key describes the text sequence for each hypothesis. Next step as part of data analysis is to process data and extract features, I will need for building. This feature engineer step was recorded and will be used in machine learning pipeline at a later step to process incoming data. Below is the extracted Tibble used for Model building.

	hypothesis	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

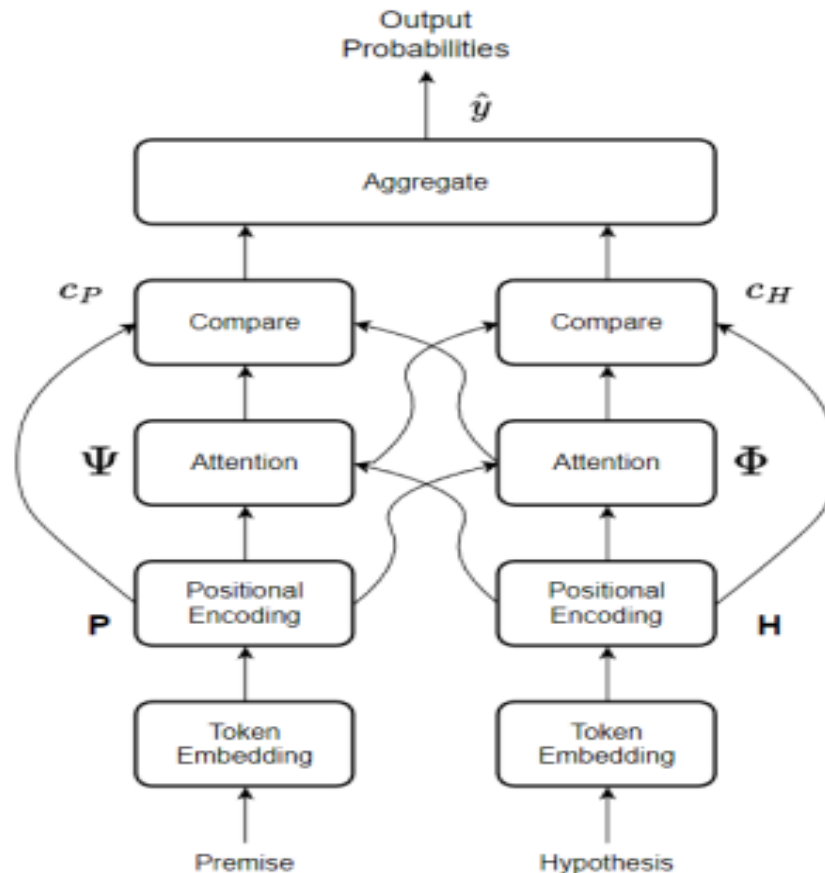
# 1. Data Analysis

I found data to be imbalanced and this will play a role in selecting hyperparameters for model building. Below is screenshot which shows higher frequency of neutral cases followed by entitlement and contradiction. Code for research here:



## 2. Model Building

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



## 2. Model Building

Token Embedding:

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

Positional Encoding:

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

Here,  $d$  is the embedding dimension,  $pos$  is the position of the token in the sequence, and  $i$  maps to the dimension of the embedding. The positional encoding is calculated using sine and cosine functions.

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

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



## 2. Model Building

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

where  $F$  is the perceptron with ReLU nonlinear activation that maps  $p_i$ ,  $h_j$  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

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

## 2. Model Building

Comparing:

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

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

The representation for premise token  $p_i$  and the softly aligned weight  $\Phi_i$  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.

## 2. Model Building

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:

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

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

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

## 2. Model Building

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.

The  $\beta$  hyper-  $CB_{focal}(z, y) = -\frac{1-\beta}{1-\beta^{n_y}} \sum_{i=1}^C (1-p_i^t)^\gamma \log(p_i^t)$  weighting. When  $p_i^t$  is small and consequently,  $(1-p_i^t)^\gamma$  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.

# 3. Model Evaluation

loss 0.257, train acc 0.897, test acc 0.873  
17178.2 examples/sec on [device(type='cuda', index=0)]

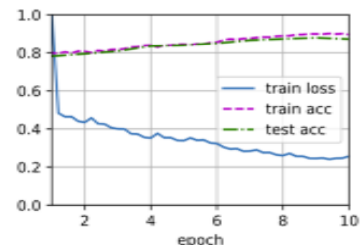


Figure 3. Loss (CE) and Accuracy Curves (Standard Dataset)

loss 0.000, train acc 0.880, test acc 0.843  
17058.4 examples/sec on [device(type='cuda', index=0)]

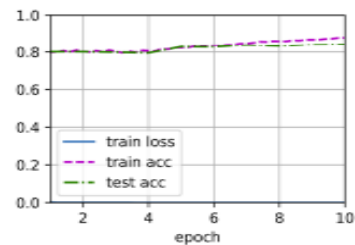


Figure 4. Loss (Focal) and Accuracy Curves (Standard Dataset)

loss 0.000, train acc 0.980, test acc 0.850  
11178.5 examples/sec on [device(type='cuda', index=0)]

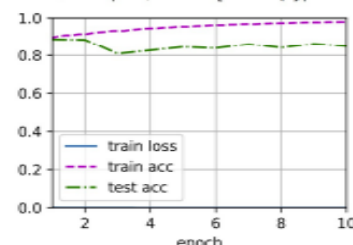


Figure 5. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

loss 0.005, train acc 0.905, test acc 0.813  
1893.2 examples/sec on [device(type='cuda', index=0)]

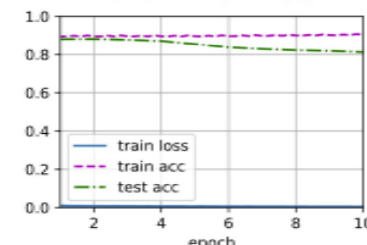


Figure 6. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

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 $N$ [64, 128, 256]	Learning Rate $\alpha$ [1e-2, 1e-3, 1e-4]	Regularization $\lambda$ [1e-3, 1e-4, 1e-5]	Class Re-balance Factor $\beta$ [0.75, 0.9, 0.999]	Modulating Factor $\gamma$ [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

# 3. Evaluation and Hyper-parameters selection

loss 0.257, train acc 0.897, test acc 0.873  
17178.2 examples/sec on [device(type='cuda', index=0)]

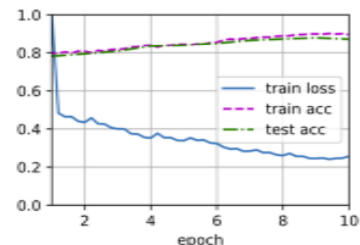


Figure 3. Loss (CE) and Accuracy Curves (Standard Dataset)

loss 0.000, train acc 0.880, test acc 0.843  
17058.4 examples/sec on [device(type='cuda', index=0)]

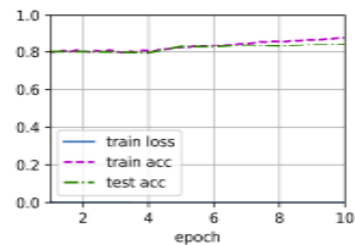


Figure 4. Loss (Focal) and Accuracy Curves (Standard Dataset)

loss 0.000, train acc 0.980, test acc 0.850  
11178.5 examples/sec on [device(type='cuda', index=0)]

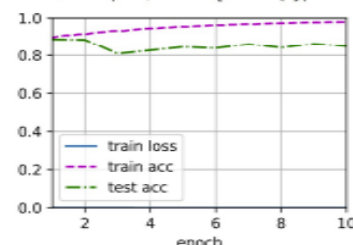


Figure 5. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

loss 0.005, train acc 0.905, test acc 0.813  
1893.2 examples/sec on [device(type='cuda', index=0)]

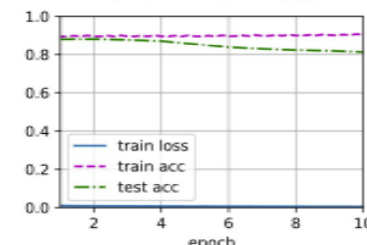


Figure 6. Loss (Focal) and Accuracy Curves (Standard Dataset with Faiss)

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

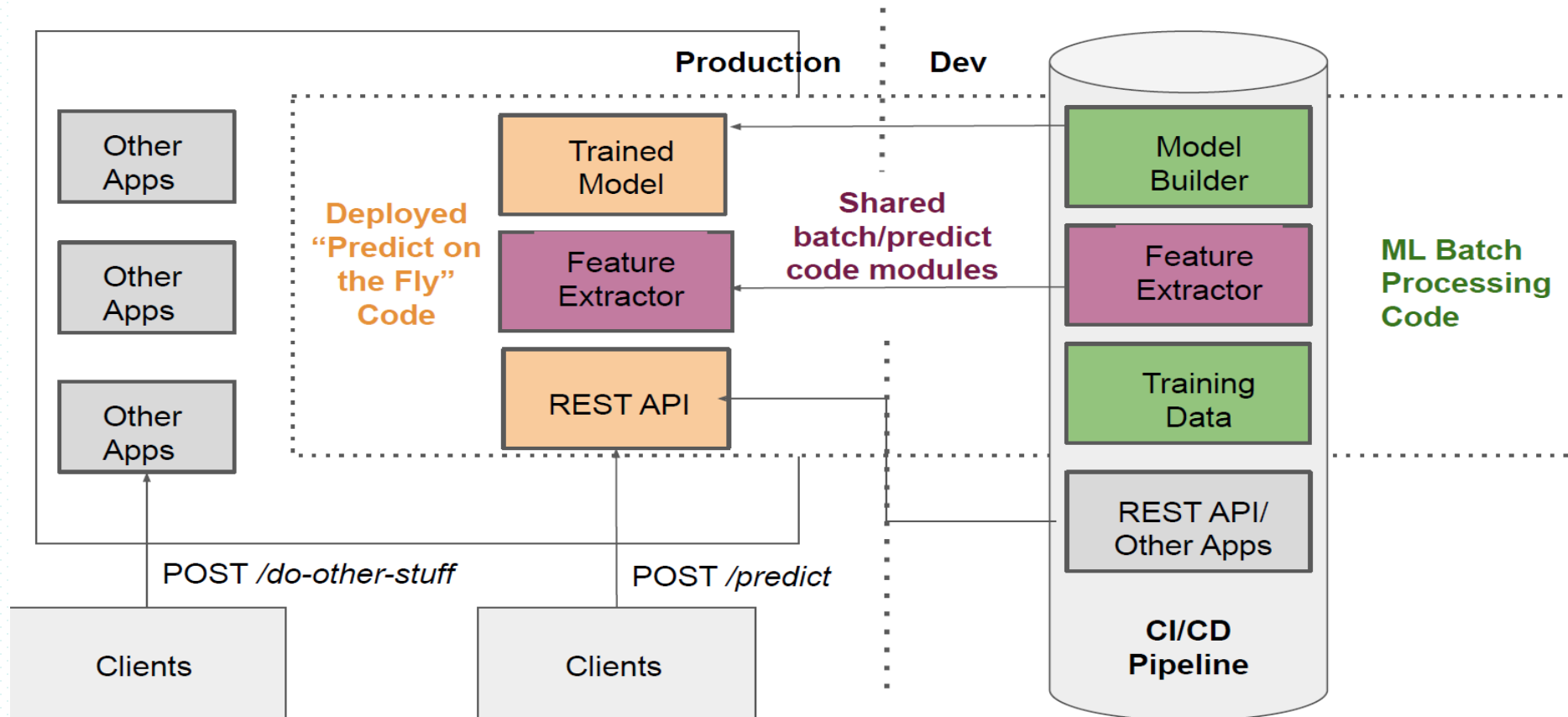
Optimizer [SGD, ADAM]	Batch Size $N$ [64, 128, 256]	Learning Rate $\alpha$ [1e-2, 1e-3, 1e-4]	Regularization $\lambda$ [1e-3, 1e-4, 1e-5]	Class Re-balance Factor $\beta$ [0.75, 0.9, 0.999]	Modulating Factor $\gamma$ [1, 2]	Entailment F1 Score	Contradiction F1 Score	Neutral F1 Score	Accuracy
ADAM	256	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

# 4. Architecture Component Breakdown (Productionizing model)

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. Production Model Package

Continuing with project, next steps were to write production code designed to be deployed to end user. I continued to focus on Testability, Maintainability, Scalability, Performance and Reproducibility. Below is the package structure created breaking down research code into separation of concern components, meaning each module has single responsibility in doing its job. So config package, will only contain modules used for configuration. Testing package will only contain modules designed for testing. NOTE, company code has lot more unit tests and ensemble of models with various hyperparameters.



## 4. Production Model Package Structure

```
.
├── Production/
│   ├── contract_nli/
│   │   ├── config/
│   │   │   ├── __init__.py
│   │   │   └── core.py
│   │   ├── data/
│   │   │   ├── test.csv
│   │   │   └── train.csv
│   │   ├── model/
│   │   │   ├── __init__.py
│   │   │   ├── aggregate.py
│   │   │   ├── attend.py
│   │   │   ├── compare.py
│   │   │   ├── decomposable_attention.py
│   │   │   └── mlp.py
│   │   ├── preprocessing/
│   │   │   ├── __init__.py
│   │   │   ├── data_management.py
│   │   │   └── snli_dataset.py
│   │   ├── tests/
│   │   │   ├── __init__.py
│   │   │   ├── conftest.py
│   │   │   ├── test_config.py
│   │   │   ├── test_predict.py
│   │   │   └── test_validation.py
│   │   └── trained_model/
│   │       ├── __init__.py
│   │       ├── model.pth
│   │       └── model.pth
│   ├── requirements/
│   │   └── requirements.txt
│   ├── MANIFEST.in
│   ├── mypy.ini
│   ├── pyproject.toml
│   ├── requirements.txt
│   └── setup.py
```

## 4. Production Model Module vs Package

I created various model modules to build a deep learning package as you can see from the above folder structure. I have used dependency injection pattern, i.e. passing objects that objects need instead of creating them, helped in creating scalable and testable code. Below is an implementation of a python library called pydantic which makes configuration code easy and compiled into an object that can be passed into various parts of the application.

## 4. Production Model configuration module

```
1  from pathlib import Path
2  import typing as t
3  from pydantic import BaseModel, validator
4  from strictyaml import load, YAML
5  import os
6
7
8  # Project Directories
9  PWD = os.path.dirname(os.path.abspath(__file__))
10 PACKAGE_ROOT = Path(os.path.abspath(os.path.join(PWD, '..')))
11 ROOT = PACKAGE_ROOT.parent
12 CONFIG_FILE_PATH = PACKAGE_ROOT / "config.yaml"
13 TRAINED_MODEL_DIR = PACKAGE_ROOT / "trained_model"
14 DATA_DIR = PACKAGE_ROOT / "data"
15
16 class AppConfig(BaseModel):
17     """
18     Application-level config.
19     """
20     package_name: str
21     train_path: str
22     test_path: str
23     vocab_path: str
24     model_path: str
25
26
27 class ModelConfig(BaseModel):
28     """
29     All configuration relevant to model
30     training and feature engineering.
31     """
32     num_step: int
33     batch_size: int
34     learning_rate: float
35     embed_size: int
36     num_hiddens: int
37     epochs: int
38     save_best: bool
39     trainer: str
40     loss: str
41
```

## 4. Production Model Data Service

29 lines (26 sloc) | 1.17 KB


Raw

Blame





```
1 import pandas as pd
2 import torch
3 import d2l
4 from .snli_dataset import SNLIDataset
5
6 class DataService():
7     def __init__(self):
8         pass
9
10    #loads csv into pandas dataframe.
11    def load_data(self, train_path, test_path):
12        train = pd.read_csv(train_path)
13        test = pd.read_csv(test_path)
14        train["label"] = train["label"].astype(int)
15        test["label"] = test["label"].astype(int)
16        return train, test
17
18    #load pandas dataframe to dataset.
19    def create_snli_dataset(self, train, test, num_steps = 50, batch_size = 256, num_workers = 4):
20        train_set = SNLIDataset(train, num_steps)
21        test_set = SNLIDataset(test, num_steps, train_set.vocab)
22        vocab = train_set.vocab
23        train_iter = torch.utils.data.DataLoader(train_set, batch_size,
24                                                  shuffle=False,
25                                                  num_workers=num_workers)
26        test_iter = torch.utils.data.DataLoader(test_set, batch_size,
27                                                  shuffle=False,
28                                                  num_workers=num_workers)
29        return train_iter, test_iter, vocab
```

# 4. Production Model testing 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   

```
1 '''
2 Here is where we want to test inputs, outputs and model quality
3 below are sample tests
4 '''
5
6 #Testing for entailment condition
7 def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_1):
8     res = load_predict_class.make_single_prediction(sample_input_data_1["premise"].split(), sample_input_data_1["hypothesis"].split())
9     assert res == sample_input_data_1["result"]
10
11 #Testing for Contradiction condition
12 def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_2):
13     res = load_predict_class.make_single_prediction(sample_input_data_2["premise"].split(), sample_input_data_2["hypothesis"].split())
14     assert res == sample_input_data_2["result"]
15
16 #Testing for neutral condition
17 def test_prediction_quality_against_benchmark_entailment(load_predict_class, sample_input_data_3):
18     res = load_predict_class.make_single_prediction(sample_input_data_3["premise"].split(), sample_input_data_3["hypothesis"].split())
19     assert res == sample_input_data_3["result"]
```

# 4. Production Model more testing modules

```
63 lines (53 sloc) | 1.96 KB
1  #!/usr/bin/python
2
3  '''
4  https://stackoverflow.com/questions/34466027/in-pytest-what-is-the-use-of-confest-py-files
5  confest.py is used to define
6  fixture used to define static data used by tests
7  External plugin
8  Hooks
9  Test root path
10 '''
11
12 import pytest
13 from contract_nli.config.core import config, TRAINED_MODEL_DIR
14 from contract_nli.predict import Predict
15
16 trained_model_dir_path = TRAINED_MODEL_DIR.as_posix()
17 model_config = config.model_config
18 embed_size=model_config.embed_size
19 num_hidden=model_config.num_hidden
20
21 @pytest.fixture()
22 def raw_app_config():
23     #For larger datasets, here we would use a testing sub-sample.
24     return config.app_config
25
26 @pytest.fixture()
27 def raw_model_config():
28     return model_config
29
30 @pytest.fixture()
31 def load_predict_class():
32     model_path = f"{trained_model_dir_path}/{config.app_config.model_path}"
33     vocab_path = f"{trained_model_dir_path}/{config.app_config.vocab_path}"
34     pr = Predict(model_config.embed_size,model_config.num_hidden, model_path, vocab_path)
35     return pr
36
37 @pytest.fixture()
38 def sample_input_data_1():
39     row_1 = {
40         "hypothesis": "Receiving Party shall destroy or return some Confidential Information upon the termination of Agreement",
41         "premise": "I the completion or termination of the dealings between the parties contemplated hereunder or",
42         "result": "Entailment"
43     }
44     return row_1
45
46 @pytest.fixture()
47 def sample_input_data_2():
48     row_2 = {
49         "hypothesis": "All Confidential Information shall be expressly identified by the Disclosing Party",
50         "premise": "I marked confidential or proprietary or",
51         "result": "Contradiction"
52     }
53     return row_2
54
55 @pytest.fixture()
56 def sample_input_data_3():
57     row_3 = {
58         "hypothesis": "Receiving Party shall not reverse engineer any objects which embody Disclosing Party's Confidential Information",
59         "premise": "Compelled Disclosure of Confidential Information",
60         "result": "neutral"
61     }
62     return row_3
```

[Give feedback](#)

## 4. Model implementation CI/CD pipeline

Once I completed packaging production module, then started integrating Azure pipelines for CI/CD which stands for continuous integration, continuous delivery and continuous deployment. What this means is when a developer like me submits code for review and check's in after approval, code goes through a process of building, testing, and publishing files to the private server. This is done so machine learning model can be integrated with a website or a web api, instead of creating monolithic application. Here is an example from my company pipeline when a feature has been checked in for this project.

## 4. Model implementation CI/CD pipeline

← Jobs in run #20230221.3.0

Build CPS

✓	Build application	58s
✓	Initialize job	5s
✓	Checkout r...	3s
✓	Use Visual Studio 5.5.1	1s
✓	... restore	7s
✓	Build solution	16s
✓	Test Assemblies	14s
✓	Publish symbols path	3s
⌕	Copy Files to: D:\a\1\1\...	<1s
✓	Publish Artifact	<1s
✓	Post-job: Checkout ...	<1s
✓	Finalize Job	<1s
✓	Report build status	<1s

✓ Build application

```
1 Pool: Azure Pipelines
2 Image: windows-2019
3 Agent: Hosted Agent
4 Started: Today at 5:06 PM
5 Duration: 58s
6
7 ▶ Job preparation parameters
8 ▶ fx 7 queue time variables used
9 ⚠ 100% tests passed
```



## 4. Model implementation CI/CD pipeline

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. Next Steps

To complete my project, I will need to implement Fast API to serve contract\_nli model in test. I will need to implement strategy to make single prediction and save a json file to make multiple predictions. Then I will containerize fast api and deploy as PaaS to company's private server and so testing can begin in shadow mode and promote to clients. Given I have more time, I will work on implementing MiniKube(Kubernetes), Redis and Dynamic.