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.

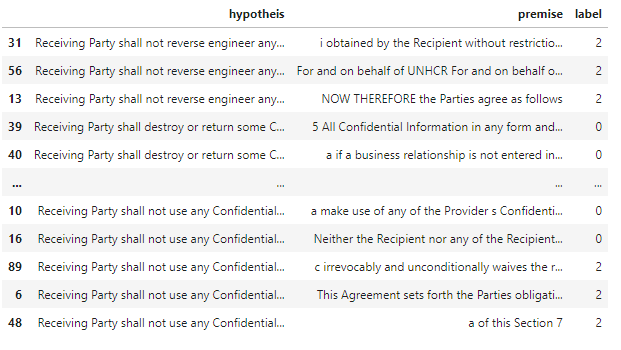
**1. Data Analysis**

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

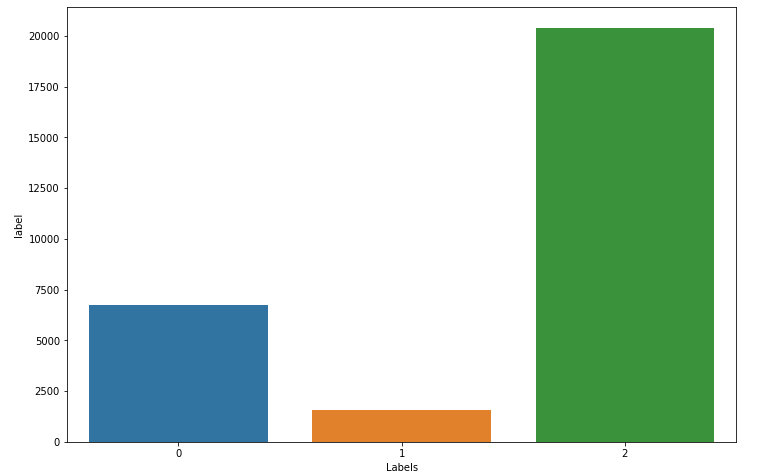


At the document level, we have the "text" key, which contains the entire document text, and the "spans" key, which divides the text into a list of premises. The key term "annotation sets" refers to a list of multiple annotations for a given document. At the annotation level, each key "nda-1," "nda-2," 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.

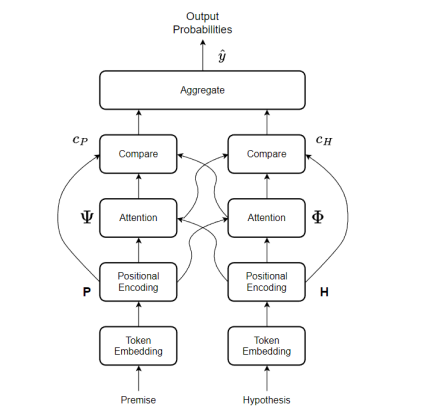


I discovered that the data was imbalanced, which will play a role in selecting hyperparameters for model building. The screenshot below depicts a higher frequency of neutral cases, followed by entitlement and contradiction. Here's a research code:



**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.

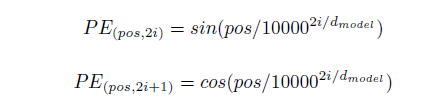


**Token Embedding**:

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.



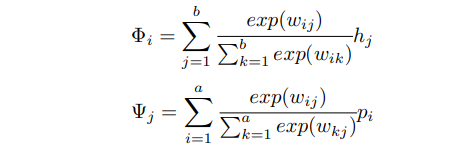
Here, d is the embedding dimension, pos is the position of the token in the sequence, and i maps to sin and cosine functions.

**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

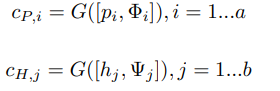


where F is the multi-layer perceptron with ReLU nonlinear activation that maps pi, hj to a hidden dimension space. This allows us to calculate the projection of the premise over the hypothesis. The intuition behind the alignment model is based on a bidirectional RNN used as an encoder and decoder



**Comparing**:

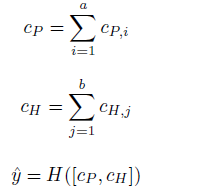
In the compare section, all the tokens from one sequence, with their corresponding weights are compared with a token in the other sequence.



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:



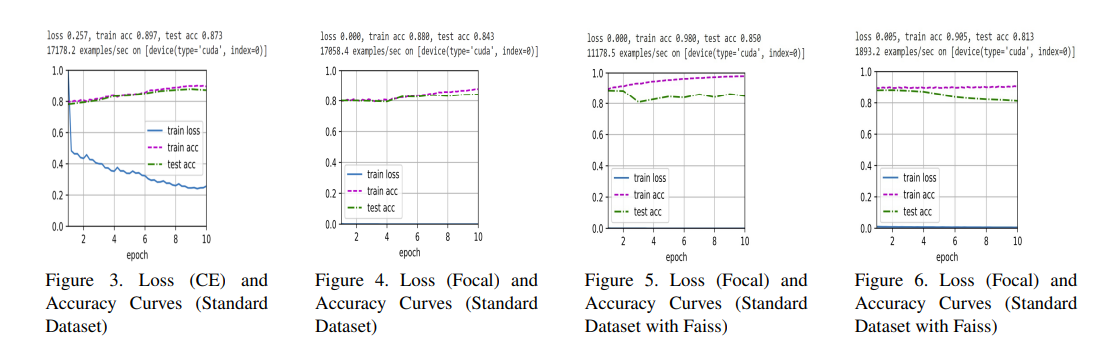
**Focal Loss**:

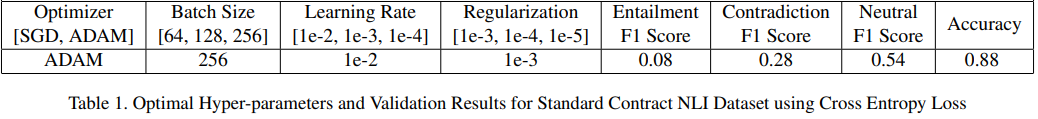
While training the decomposable attention model, we use focal loss as there exists class imbalance among the 3 classes - Entailment, Contradiction and Neutral.

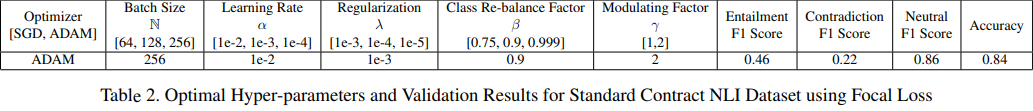


The β hyper-parameter can be tuned to perform reweighting. When pt is small and consequently, (1 − pt)γ is close to 1, then Focal loss becomes classic cross entropy, and would result in incorrect classification by the model. As the model adjusts its weights, Focal Loss scales down the contribution of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

**Results**:



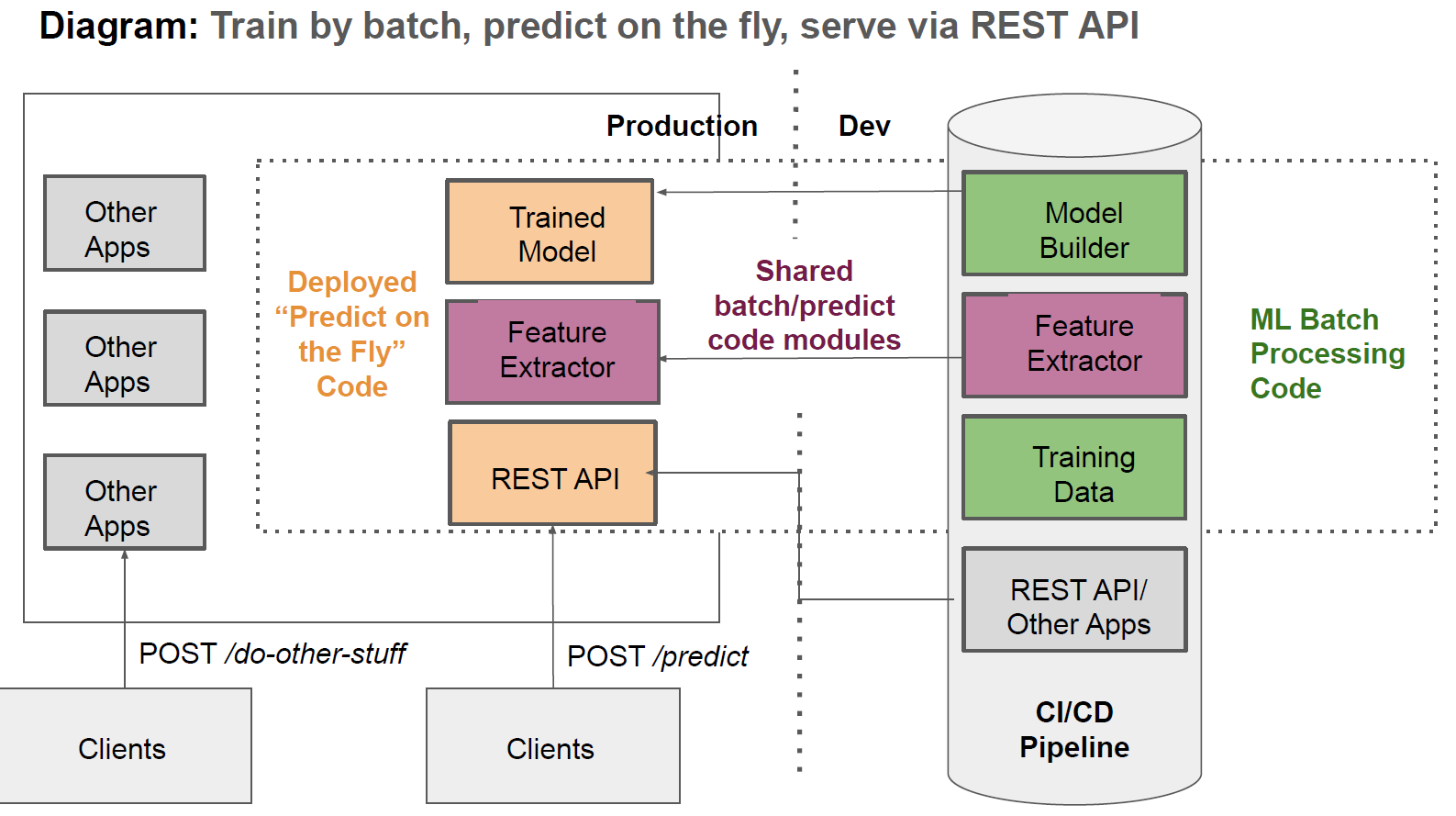




using the above hyperparameter we got the best precision, recall, f1-score along with accuracy and loss. After selecting model and hyperparameter our next task was to productionizing code.

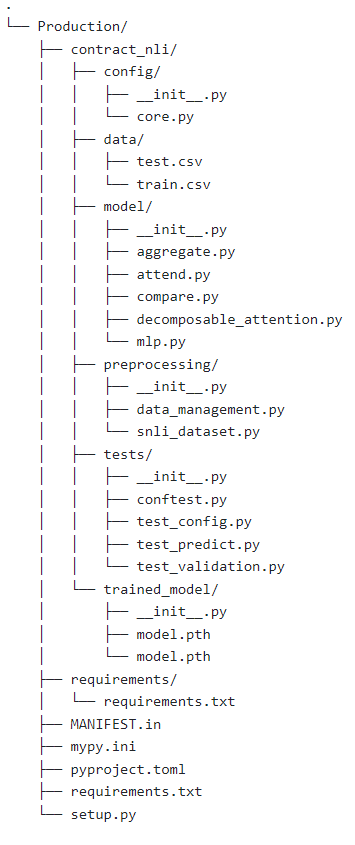
**3. Architecture Component Breakdown**

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

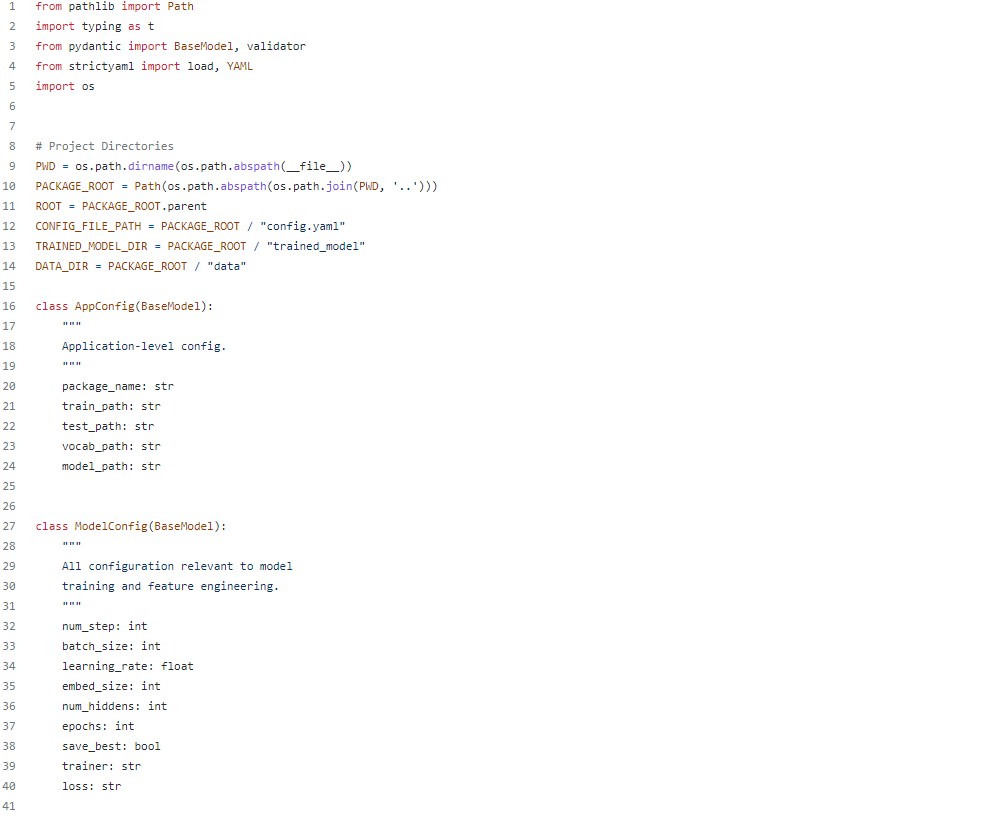


**4. Productionizing Model package**

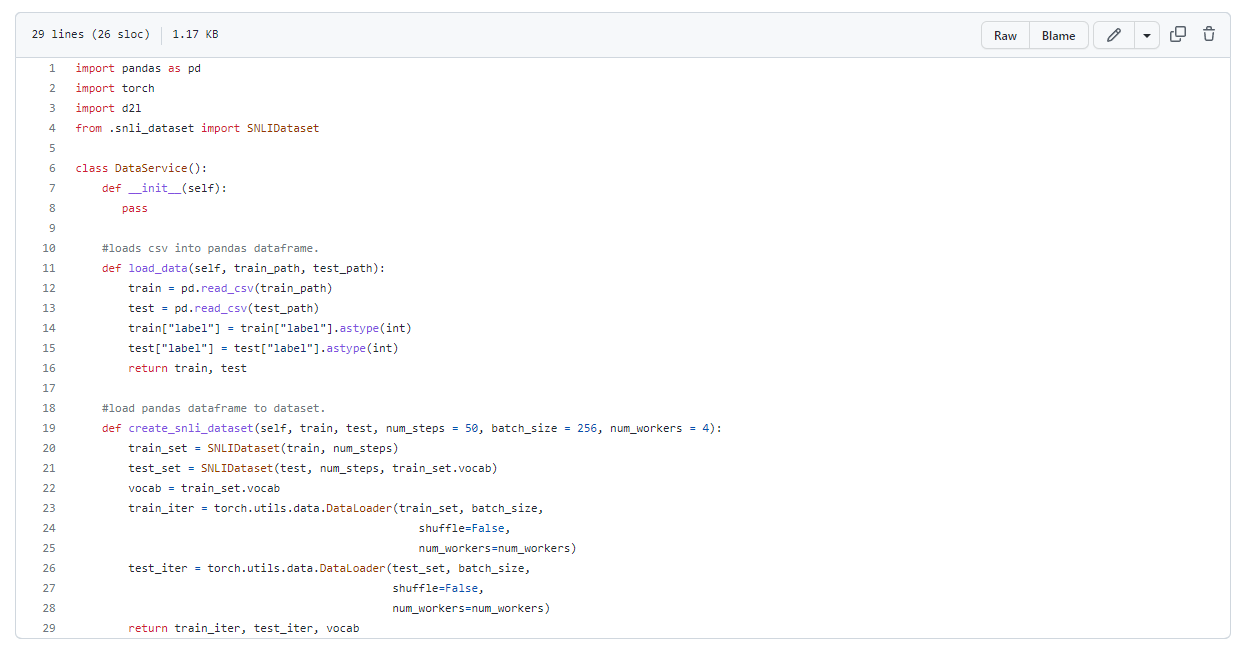
Continuing with the project, the next step was to write production code that would be deployed to end users. I kept my attention focused on testability, maintainability, scalability, performance, and reproducibility. The package structure created below breaks down research code into separation of concern 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.



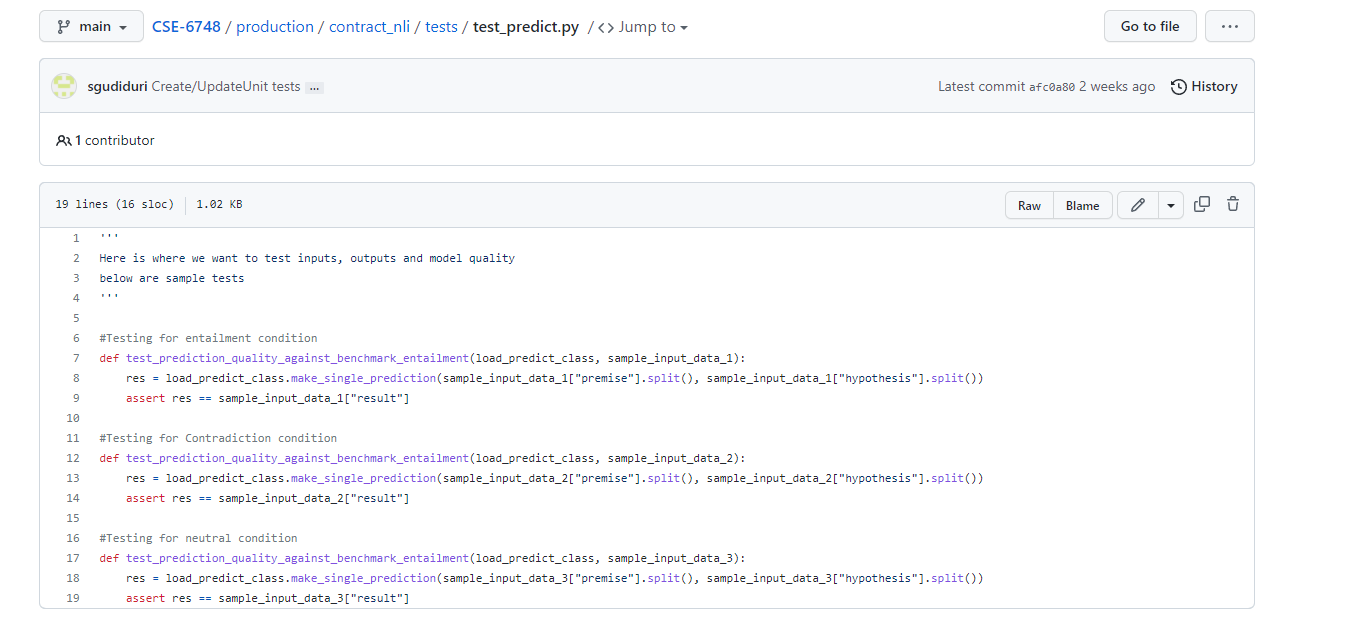
As you can see from the folder structure above, I created various model modules to build a deep learning package. The dependency injection pattern, which involves passing objects that objects require rather than creating them, aided in the creation of scalable and testable code. Below is an implementation 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.

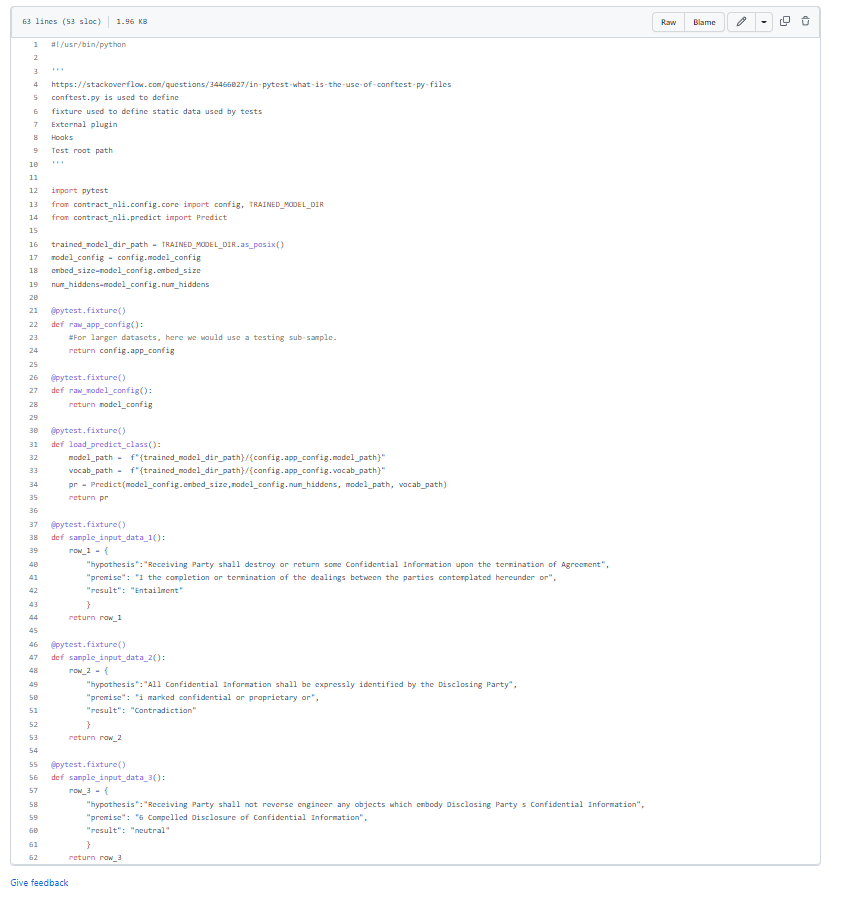


Below are some screenshots for Data Service class

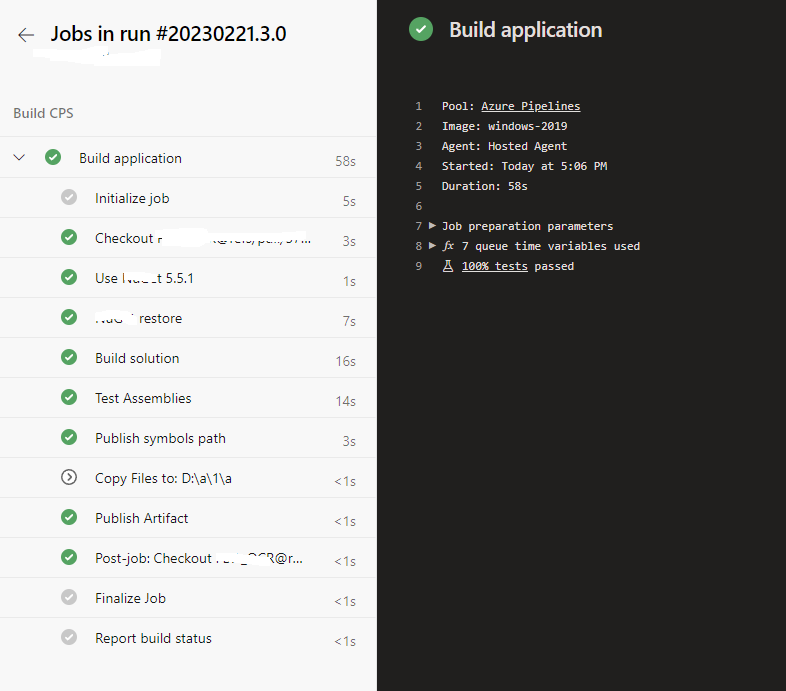


Below are some screenshots for testing code modules





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.



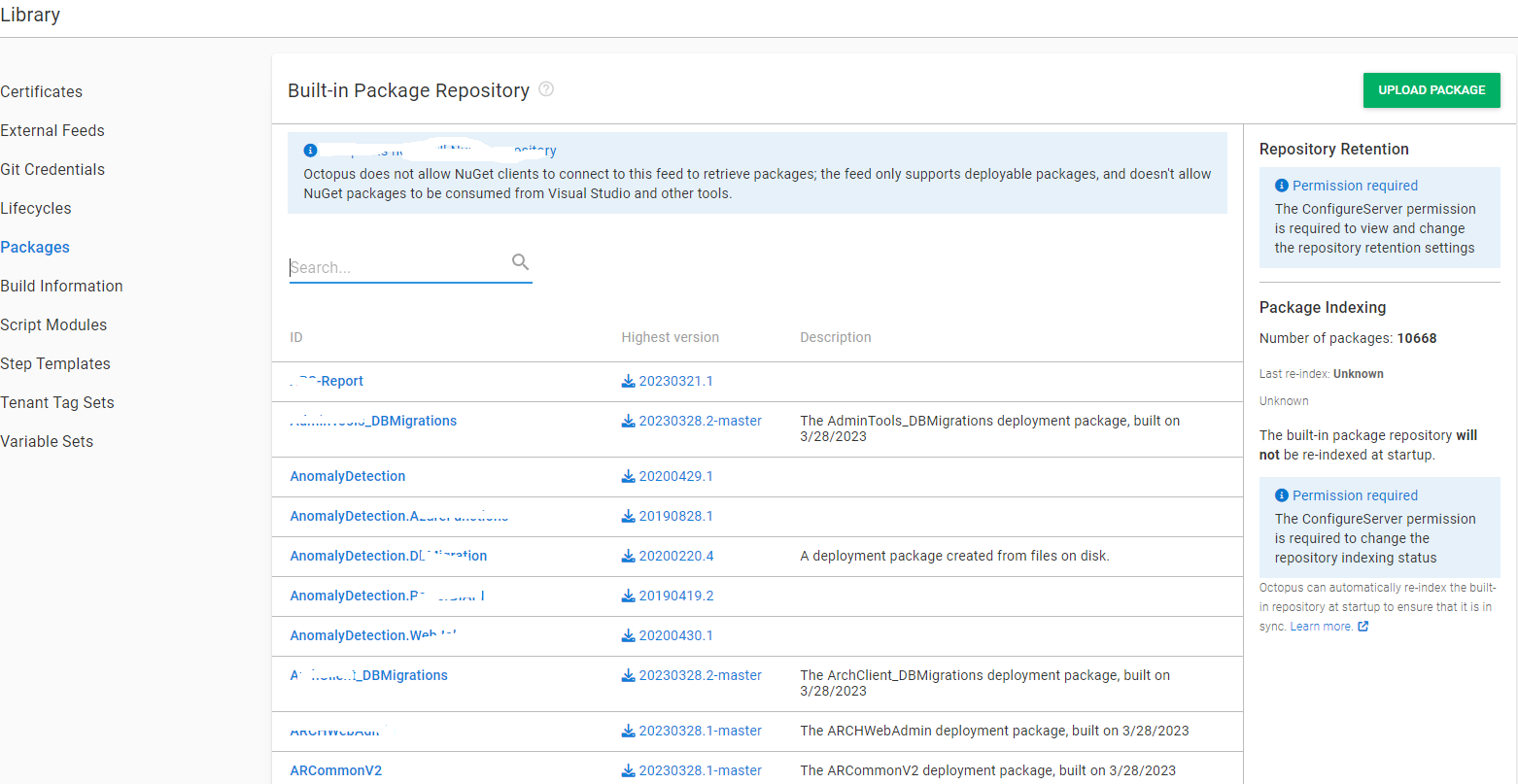
Note, I created a similar example for this class as a POC before integrating with company code. This deployed on pypi is an experimental version and not the model package used at my company. Link can be found [here](https://pypi.org/project/contract-nli/)

**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.

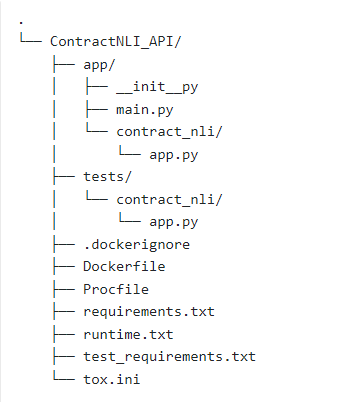
Screenshot of library below



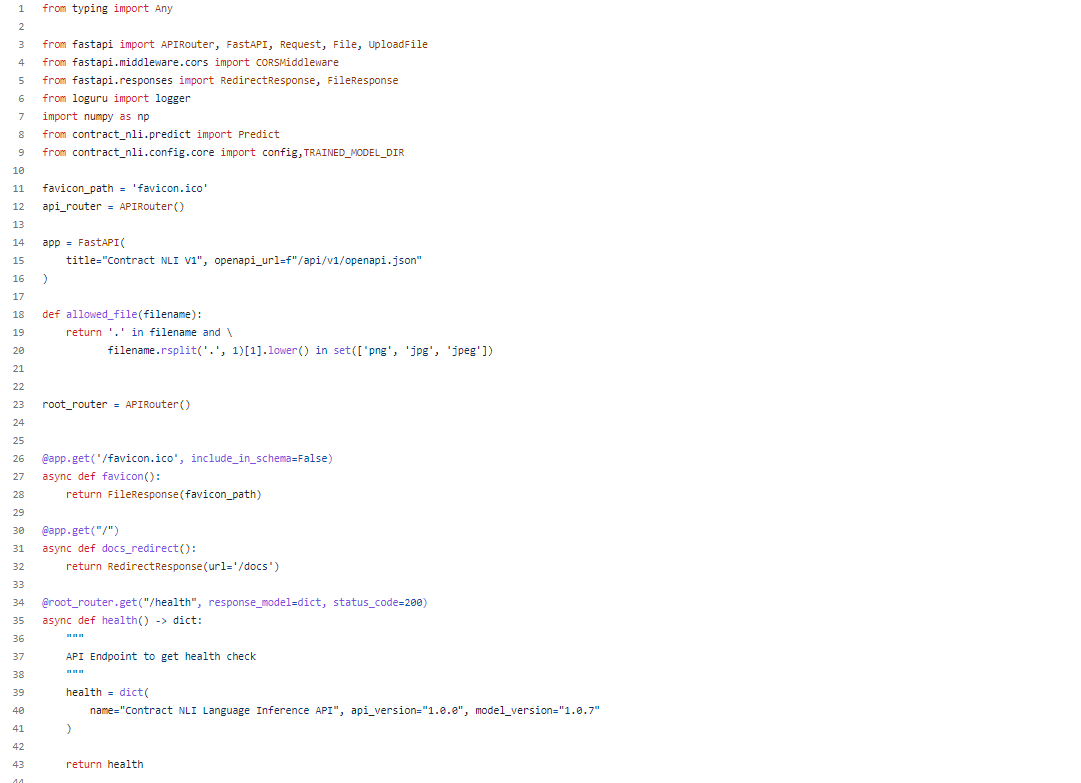
**6. API to interact with Model**

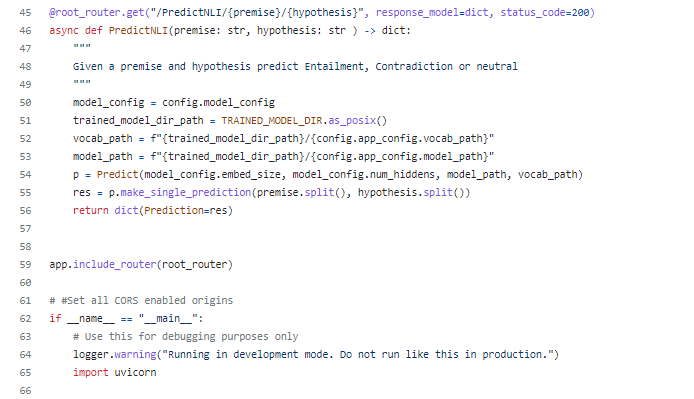
To create Web API to we imported libraries from the private server. Then we defined the input and the output formats for our model. This is a JSON request with specific keys. Next we defined end points using Fast API decorator. Below are the screenshot of the a) file structure b) main.py c) on the web.

a)

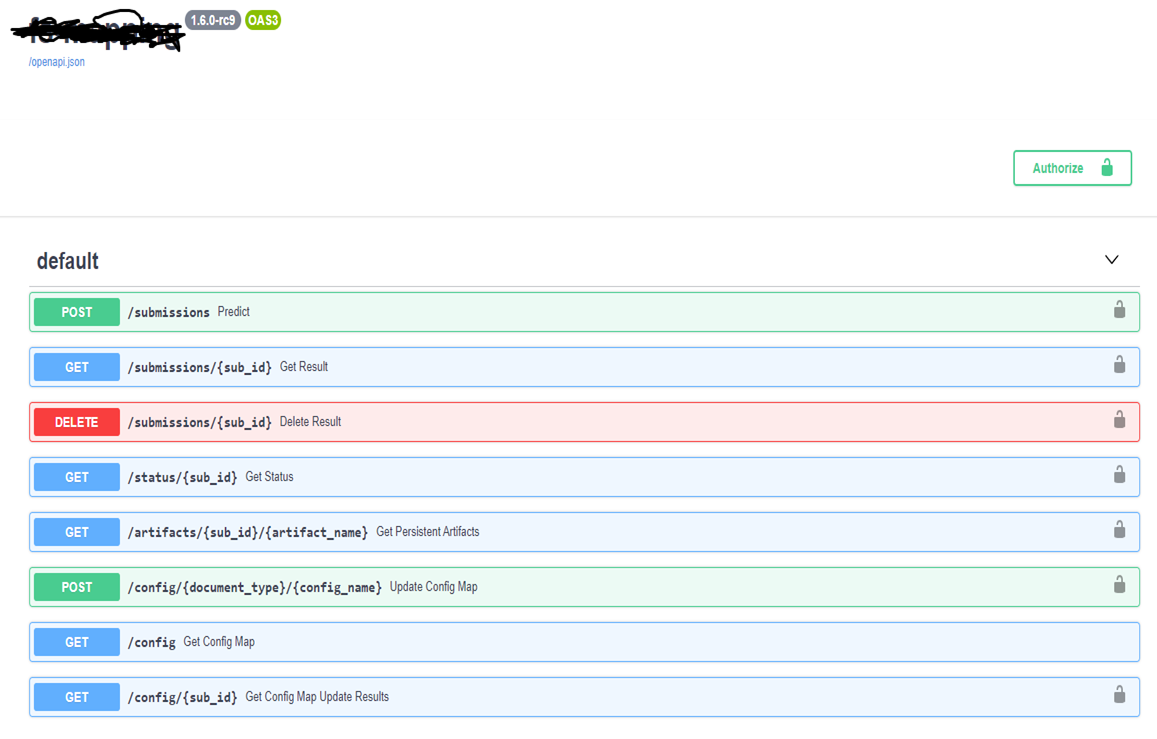


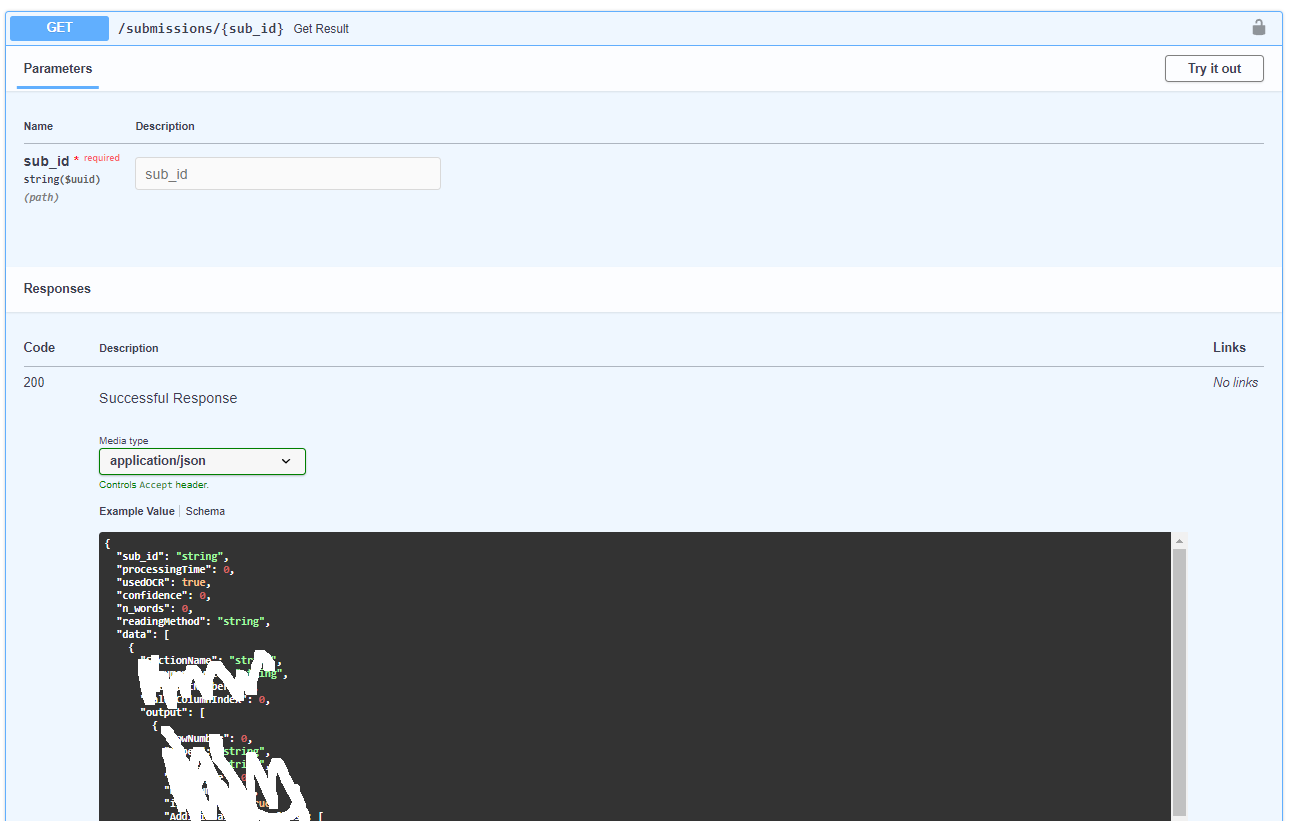
b) partial implementation, real implementation is redacted.





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





**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.