**Final Capstone Submission**

**Problem Description**

With all the text documents and information out there, locating answers to questions quickly is an important task. Simple methods and algorithms such as “ctrl f”, tfidf, or BM25 may be able to find words in your question to lead to an answer. However, these searches are very dependent on the words you are searching for, and one may still need to read several pages to find the answer. They do not use word semantics for searching. The latest transformer models use semantic relations and attention layers to add further semantic capability into searching. This capstone was conducted on the SQuAD dataset to learn about the latest transformer models and libraries to leverage this improved question and answering capability.

**Dataset**

The SQuAD dataset v2 can be downloaded at the following site: [The Stanford Question Answering Dataset (rajpurkar.github.io)](https://rajpurkar.github.io/SQuAD-explorer/). This set of data is a common test set for question and answering extraction type models. It contains 100,000 questions with answers and 50,000 without answers. The data set is in JSON format. Each element in the JSON has a “title” and “paragraphs” key. Each element in “paragraphs” has a “context” key and a list of question and answers referenced by the “qas” key. The context is the paragraph that contains the answers to the questions in the question list associated with the context paragraph. The questions and answers pairs contain the following information: the question as text, answer as text, starting character of the answer text within the context paragraph, and if the question is impossible. Impossible questions are adversarial examples meant for teaching the absence of an answer. If impossible the answer text will be an empty string and the answer start character will be 0. Below is a sample of SQuAD data taken from the dataset provided in the link above.

{‘title’: ‘Beyoncé’,

‘paragraphs’: [{‘qas’: [{‘question’: ‘When did Beyonce start becoming popular?’,

‘id’: ‘56be85543aeaaa14008c9063’,

‘answers’: [{‘text’: ‘in the late 1990s’, ‘answer\_start’: 269}],

‘is\_impossible’: False},

{‘question’: ‘What areas did Beyonce compete in when she was growing up?’,

‘id’: ‘56be85543aeaaa14008c9065’,

‘answers’: [{‘text’: ‘singing and dancing’, ‘answer\_start’: 207}],

‘is\_impossible’: False}],

‘context’: ‘Beyoncé Giselle Knowles-Carter (/biːˈjɒnseɪ/ bee-YON-say) (born September 4, 1981) is an American singer, songwriter, record producer and actress. Born and raised in Houston, Texas, she performed in various singing and dancing competitions as a child, and rose to fame in the late 1990s as lead singer of R&B girl-group Destiny\’s Child. Managed by her father, Mathew Knowles, the group became one of the world\’s best-selling girl groups of all time. Their hiatus saw the release of Beyoncé\’s debut album, Dangerously in Love (2003), which established her as a solo artist worldwide, earned five Grammy Awards and featured the Billboard Hot 100 number-one singles “Crazy in Love” and “Baby Boy”.’}]

**Data Preprocessing**

To build a question and answering extraction model the questions and context must be encoded into numbers. [CLS] and [SEP] or <s> and </s> tokens are added to separate the start and end of the question from the context. The context must be split into word chucks with the question appended to the front of each chunk. These chunks are of length equal to a max\_length parameter, with an over lapping stride of length equal to the stride parameter. The chunks are padded with 0s to make each is the same size. The stride parameter guarantees the entire answer is present in at least one of the chunks. To further complicate this, one must generate an attention mask, an overflow mapping, an offset mapping, and sequence ids. The attention mask is used to disregard unimportant tokens during the attention step of training. The overflow mapping maps the chunk back to its original context and question pair. The offset mapping is used to trace tokens back to their original characters in the context in order to provide the text of string with the answer. Sequence ids separate the question from the context in each chunk. The overflow mapping, offset mapping, and sequence ids can be used to label the start and end character of the answer in each chunk if it exists. (0,0) is used to label non-existent answers.

This is a lot of preprocessing. Luckily there is a tokenizers library from hugging face that can do all these steps. To utilizer this library one needs to load their own data into an Apache data arrow object using the hugging face datasets library. The SQuAD data set downloaded from [The Stanford Question Answering Dataset (rajpurkar.github.io)](https://rajpurkar.github.io/SQuAD-explorer/) was transformed to JSON with each single question paired with its context. The JSON structure used for each question and context pair is like the example below.

{'answers': [{'text': 'political science', 'answer\_start': 155}],

'context': 'Kerry was born in Aurora, Colorado and attended boarding school in Massachusetts and New Hampshire. He graduated from Yale University class of 1966 with a political science major. Kerry enlisted in the Naval Reserve in 1966, and during 1968–1969 served an abbreviated four-month tour of duty in South Vietnam as officer-in-charge (OIC) of a Swift Boat. For that service, he was awarded combat medals that include the Silver Star Medal, Bronze Star Medal, and three Purple Heart Medals. Securing an early return to the United States, Kerry joined the Vietnam Veterans Against the War organization in which he served as a nationally recognized spokesman and as an outspoken opponent of the Vietnam War. He appeared in the Fulbright Hearings before the Senate Committee on Foreign Affairs where he deemed United States war policy in Vietnam to be the cause of war crimes.',

'id': '572a982b34ae481900deaba3',

'question': "What was Kerry's major?",

'title': 'John\_Kerry'}

A function was created using the hugging face tokenizer library with additional code to label the chunks. This function gets passed to a hugging face dataset.map function. Based on the model tokenizer type you choose (distilbert, Bert, Roberta, or ect.) the text is tokenized as if you were going to use that named model for training.

**Model Training**

Hugging face has many popular pretrained base models for anyone to use as a starting check point. These models include DistilBert, Bert and Roberta. This is ideal for transfer learning and allows for better models with less data and less training time. When these models are loaded using Tensorflow they are Keras models and all the Keras methods for training work.

For model training a grid search was created and tested on the DistilBert Base, Bert Base, and Roberta Base checkpoints. Hyper parameters in the Grid search included max chunk length, overlapping chunk stride, batch size, number of epochs, and the learning rate. Learning rate had the biggest influence on model performance. Roberta preformed the best, followed by Bert, then DistilBert. The best Roberta model had the following hyperparameter values displayed below, with start and end validation logit accuracy of 76.25% and 74.35% respectively.

max\_length = 384

stride = 128

batch\_size = 16

num\_epochs = 3

learning\_rate = 0.000064

Hyper-parameter values

In order to train these models, computing units were purchase through Google Collab to use a premium GPU. Training time went from 15 hours for DistilBert to 30 minutes tops. There was a huge decrease in training time using the premium GPU. 500 computing units were purchase for $50. About 150 computing units were used for all the training during this project.

**Deployment Solution**

The solution chosen to deploy the application was a containerized flask API. There are many reasons for this.  First, learning FLASK is simple.  One can build a simple FLASK application and test it using postman in an hour or two.   Containerizing takes a little time to understand but is very powerful.  Once containerized the container can be run almost anywhere. It will not suffer from the “it worked on my machine” library dependency problem.  The container can be deployed on an on-premises server or through cloud services.  This makes it very flexible.  Anyone that wanted to deploy it, could use their setup.  AWS was chosen because it is popular, but the application was also tested and deployed through AZURE. The diagram below could switch AWS EC2 for Microsoft AZURE Container Services and it would have been another equivalent solution.

For deploying on AWS a t2.micro EC2 instance was tried, but there were memory issues building the docker file in AWS.  Tensorflow caused a memory crash when it was installed in the docker build process.  The instance was upgraded to a t2.2xlarge to fix this issue.  These cost about $.37 an hour for a Linux instance.  On Azure an equivalent solution used 4 cpu and 16 GB of memory.  The Azure process has one push their built image from your machine to an image registry. Conversely, the AWS solution builds the image inside the EC2 instance.  It is possible that the deployment could use less cores and memory in AWS to save money.

A nice thing about using a cloud provider is the scalability and redundancy.  The app could be deployed in several availability zones and as demand for the application goes up, more instances could be running.   When there are outages user traffic could be routed to another location.

Overall, this is a very simple and flexible solution that many applications or users could leverage.  Different front ends could be built for different needs and still leverage this API on the backend.  Other uses could be simple scripts from languages such as Python to send HTTP requests to the API.  Furthermore, testing is simple and can be done with something like Postman.  Below is a diagram of the deployment.

Graphical user interface, application

Description automatically generated