**JobQuail**

*A job title classification tool.*

*​​*

**Problem**

Job seekers may have a limited idea of job titles they may be qualified to apply for due to the various names of job titles that companies use with an overlap of skills and qualifications. At the same time, companies often use very different or specific job titles for roles where it would have another name in a general job search.

After speaking with many other Data Analytics students at Miami Dade College, the majority have stated they are looking specifically for a Data Analyst role without knowing there are other roles they can apply to where there is an overlap of skills to expand their job search.

**Goal**

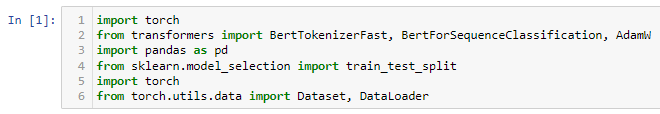
Our goal was to create an application that is simple and easy to use where job seekers or companies can input text, be it skills, qualifications, or job description. Then, receive an output of possible job titles associated with the input.

**Methodology**

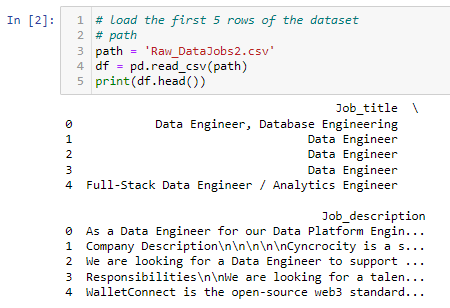
Data was scraped using Octoparse to obtain job data from LinkedIn for a variety of job titles such as Data Analyst, Data Engineer, Software Engineer, Full Stack Developer, and Front End Developer amongst others. Running Octoparse to scrape data for each job title took anywhere for 2 to 8 hours, and provided the option to export to CSV which was very useful. The raw data obtained was enormous, so we performed quite a bit of pre-processing - removing excess data to focus on the attributes we plan to use at the moment and combining all the separate job data into one file. It is important to point out that the job title which resulted in the most number of jobs is Data Engineering then Database Engineering.

The data was stored on GitHub as a repository available for public use. A Jupyter notebook was used to create the code for the model and StreamLit was used to create the application code as well as connect it to the notebook code.

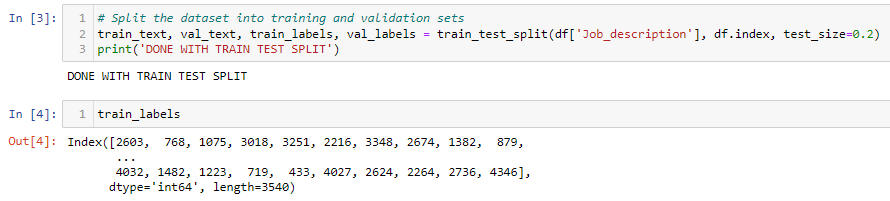
Below we started with importing libraries and tools such as pandas, torch to transform the model, Bert Tokenizer to tokenize the data.



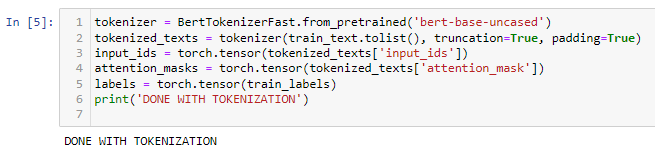
The dataset was then imported.



Train Test Split was performed on the dataset.



In this section below, we employ the 'bert-base-uncased' tokenizer to break down our training text into tokens suitable for the BERT model. Next, we will ensure the input sequences are of uniform length through truncation and padding. To help the model discern real tokens from padding tokens, we will generate attention masks. We will conclude this stage by converting our training labels into PyTorch tensors, and confirming completion with a 'DONE WITH TOKENIZATION' print statement.

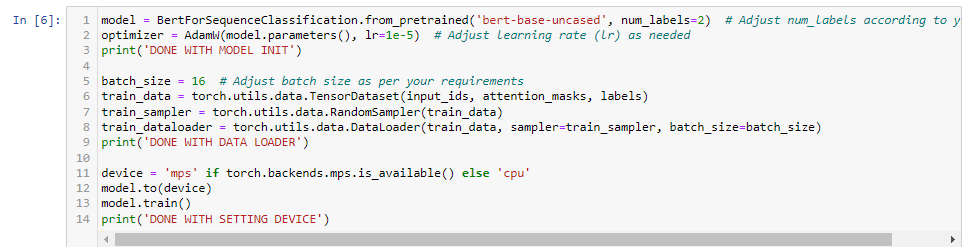


In the following code, steps were performed to train a BERT-based sequence classification model.

We will initialize the BERT model for sequence classification (BertForSequenceClassification) with the base uncased version. An optimizer (AdamW) will also be initialized with a specified learning rate.

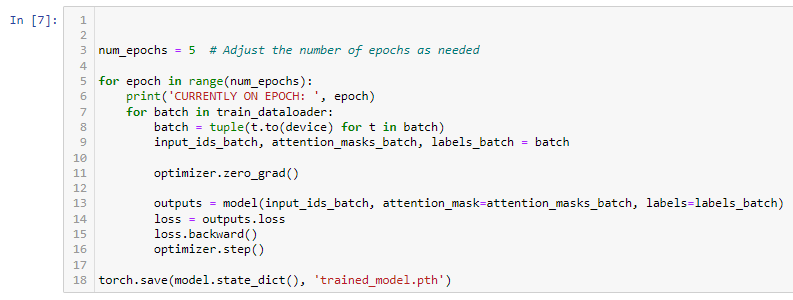
A data loader will be created (train\_dataloader) using the TensorDataset, RandomSampler, and DataLoader classes from torch.utils.data. The input tensors, attention masks, and labels will be combined into a dataset, and the data loader will handle batching and shuffling during training.

We will set the device to either 'mps' (multi-process service) if available or 'cpu' otherwise. The model will be moved to the specified device using the to method, and the model's training mode will be enabled using model.train().



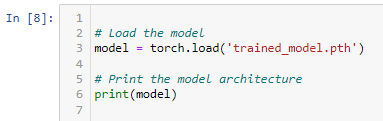
Lastly we will enter a loop over a specified number of epochs. For each epoch, it will iterate over the batches in the training data loader. The batches will be moved to the device, and the optimizer's gradients will be reset with optimizer.zero\_grad(). The model will be called with the input tensors and attention masks to obtain the outputs, and the loss will be calculated based on the predicted labels and the provided labels. The loss will then be backpropagated, and the optimizer's step will be performed to update the model's parameters.

Finishing, with saving the trained model's state dictionary to a file named 'trained\_model.pth' using torch.save.

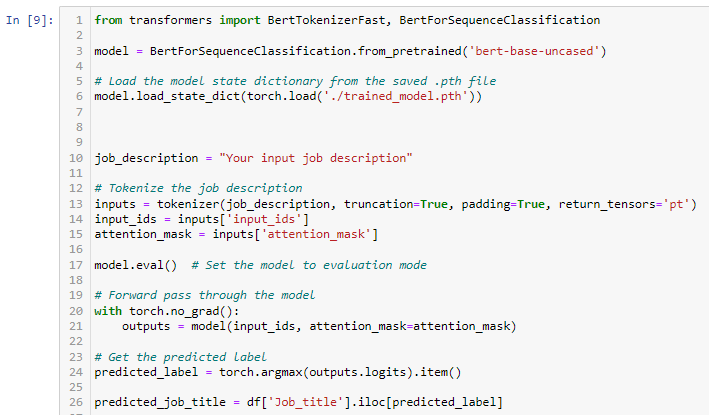


*Note: The model is over 25mb so it cant be supported on GitHub, therefore, it’s saved locally.*

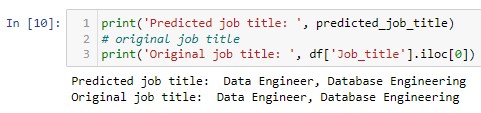
Now we will inspect how the model was saved.



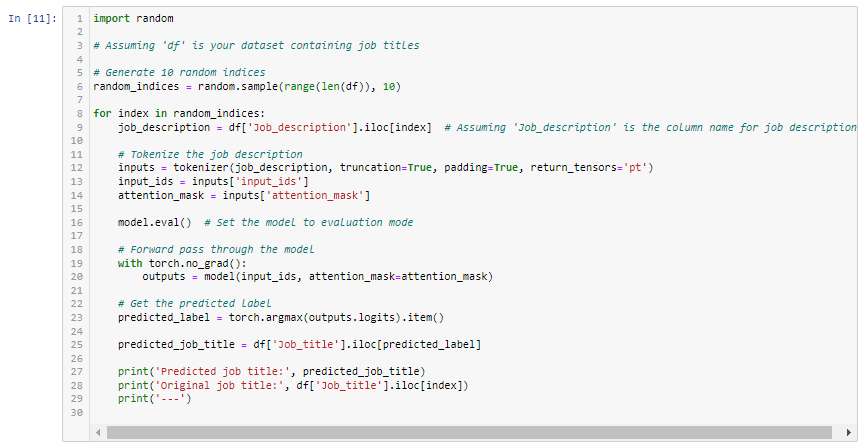
Here, we will evaluate the model predictions.



We ran through the model.



Now, we ran through an additional set at random.

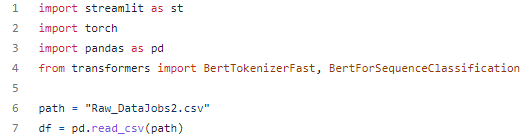


All outputs were Data Engineer.

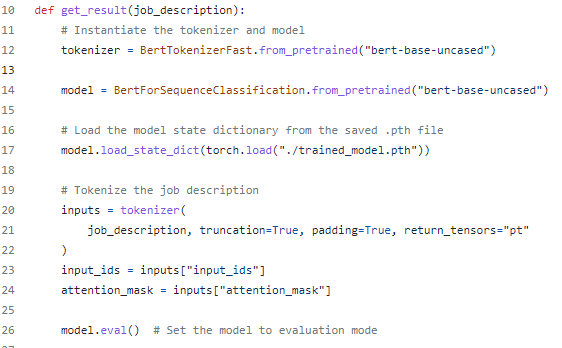


Once the model was developed, we moved forward to building the app via StreamLit.

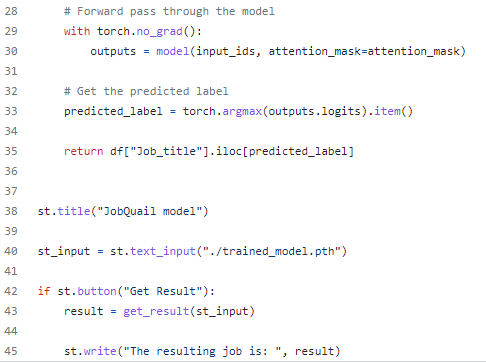
First, we will start with importing necessary libraries such as Streamlit, PyTorch, Pandas, and Transformers. We'll then read a CSV file, "Raw\_DataJobs2.csv", into a Pandas DataFrame for further processing.



Then, we will define a function, `get\_result`, which takes a job description as input. In this function, we will initiate the 'bert-base-uncased' tokenizer and the BERT model for sequence classification. The pre-trained model will then be updated with our specific training outcomes by loading a previously saved state dictionary from the ".pth" file. The job description will be tokenized and inputted into the model for evaluation. The model will then predict a label corresponding to a specific job title, which is extracted from our DataFrame.



Finally, we'll construct a Streamlit app interface with a title and a text input field. Upon entering a job description and clicking the "Get Result" button, the job description will be evaluated by the function `get\_result` and the predicted job title will be displayed on the Streamlit interface.



**Results**

After training the model with the processed data, we first tested as a company looking for a candidate by inputting a job description which resulted in our first output of Data Engineer. We then posed as a job seeker by inputting specific skills such as “python, r” and received a result of Data Engineer and Database Engineer. We tried several more tests with different job descriptions and skills, and noticed that Data Engineer kept resulting. We realized that our dataset is heavily populated with Data Engineer job data versus the other job titles since that’s how much was scraped from LinkedIn based on the parameters of my search.

We do intend to further develop this tool which means we will also focus on data quality such as having a more balanced dataset. The highly uneven amount of job data is likely the reason for the constant output of Data Engineer.

**Implications**

The application, JobQuail, could be of great value with helping potential employees and employers on their search for employment or a candidate alike. As mentioned previously, the application would often result in an output of Data Engineer. We definitely plan to further build out this application to have a more balanced dataset. We also want to include a list of salaries via feature engineering as an added attribute.

**Github Link**

<https://github.com/Tonycall/JobQuail>

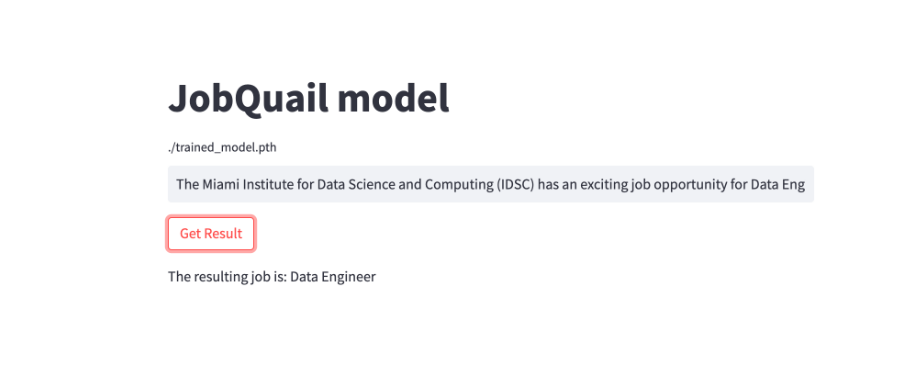
**App User Interface**

Upon navigating to the local web server where your Streamlit app is running (usually http://localhost:8501), you'll see a simple user interface with the title "JobQuail model". Below the title, you'll find a text input box. It accepts nearly any length of a job description as the input.

There is a "Get Result" button which when clicked, will allow the app to run the get\_result function. This function tokenizes the entered job description and feeds it into a pre-trained BERT model for sequence classification. The model then produces a prediction, represented by an ID.

The app uses this ID to look up the corresponding job title in the original DataFrame (df) loaded from "Raw\_DataJobs2.csv". It returns and displays this job title, suggesting that the entered job description most closely aligns with this predicted role.

In essence, the Streamlit app allows users to input a job description and, using a trained BERT model, it provides a prediction for the most suitable job title based on the inputted description.



We can see that the model is biased to assume that the highest correlated job title is either Data Engineer or Database Engineer which we believe is due to the unbalanced dataset we created, which is largely composed of records pertaining to this specific label class(Data Engineer/Database Engineer).

