NANYANG POLYTECHNIC

2023/2024 SEMESTER 2 – PROJECT

Course : Diploma in AI and Data Engineering

Module : EGT216 - Natural Language Processing

Instructions to students:

- **Plagiarism**

  Nanyang Polytechnic (NYP) takes plagiarism seriously and regards it as a form of academic dishonesty. Students who violate the NYP Academic Integrity Policy of NYP by committing plagiarism will be subject to disciplinary action.

  Learners will face disciplinary action if they provide or receive unauthorized assistance, such as:

  - Copying another learner's answer
  - Providing answers for other learners
  - Falsifying data, information or quotations
  - Asking another person to do the work you are supposed to do
  - Taking and using ideas, words or work of others in whole or in part and passing it off as your own work without citing the original source (including AI resource)

  Please refer to the NYP Academic Integrity Policy on the NYP website

  Plagiarism is taking the work or ideas of another person, whether in whole or in part, and presenting it as one's own work without acknowledging the original source.

  Sharing your work with other learners is considered plagiarism. ALL those involved, including those who share their work, will be penalised in accordance with the NYP Academic Integrity Policy.

  Cite and reference ALL sources you have used in your report in APA style.

- **Copyright Infringement**

  It is equally important that you avoid copyright infringements and name the authors of the images you use. When searching for images, make sure that you have obtained the necessary permissions or licenses to use the images in your work. Using copyrighted images without proper permission is a direct violation of intellectual property rights. You must use copyright-free or Creative Commons licensed image databases, as this will allow you to obtain imagery in an ethical manner while adhering to the principles of academic integrity.

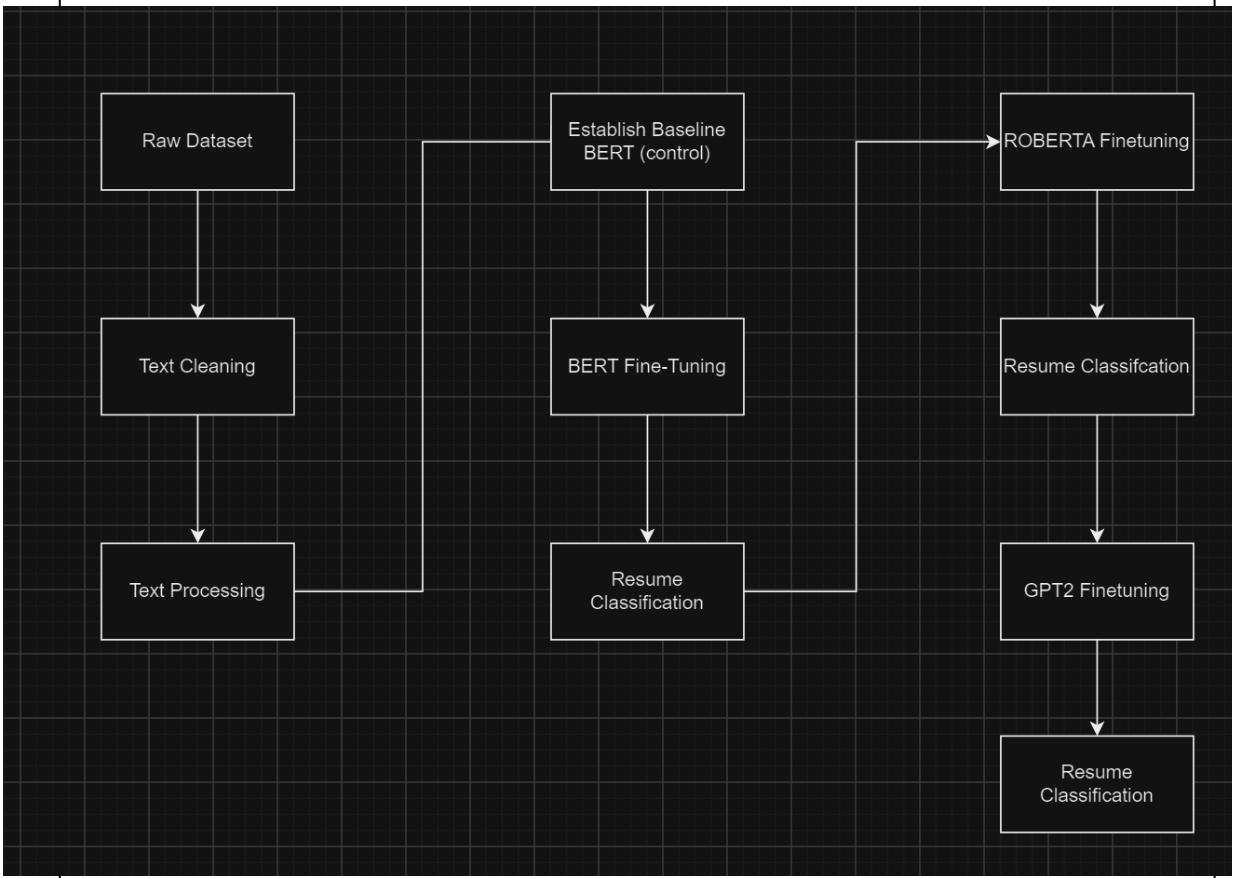
- **Late Submission**

  For each working day of late submission, a penalty of 5% of the basic grade will be deducted from the grade. Late submission after 5 working days will be awarded 0 points.

Project / EGT216

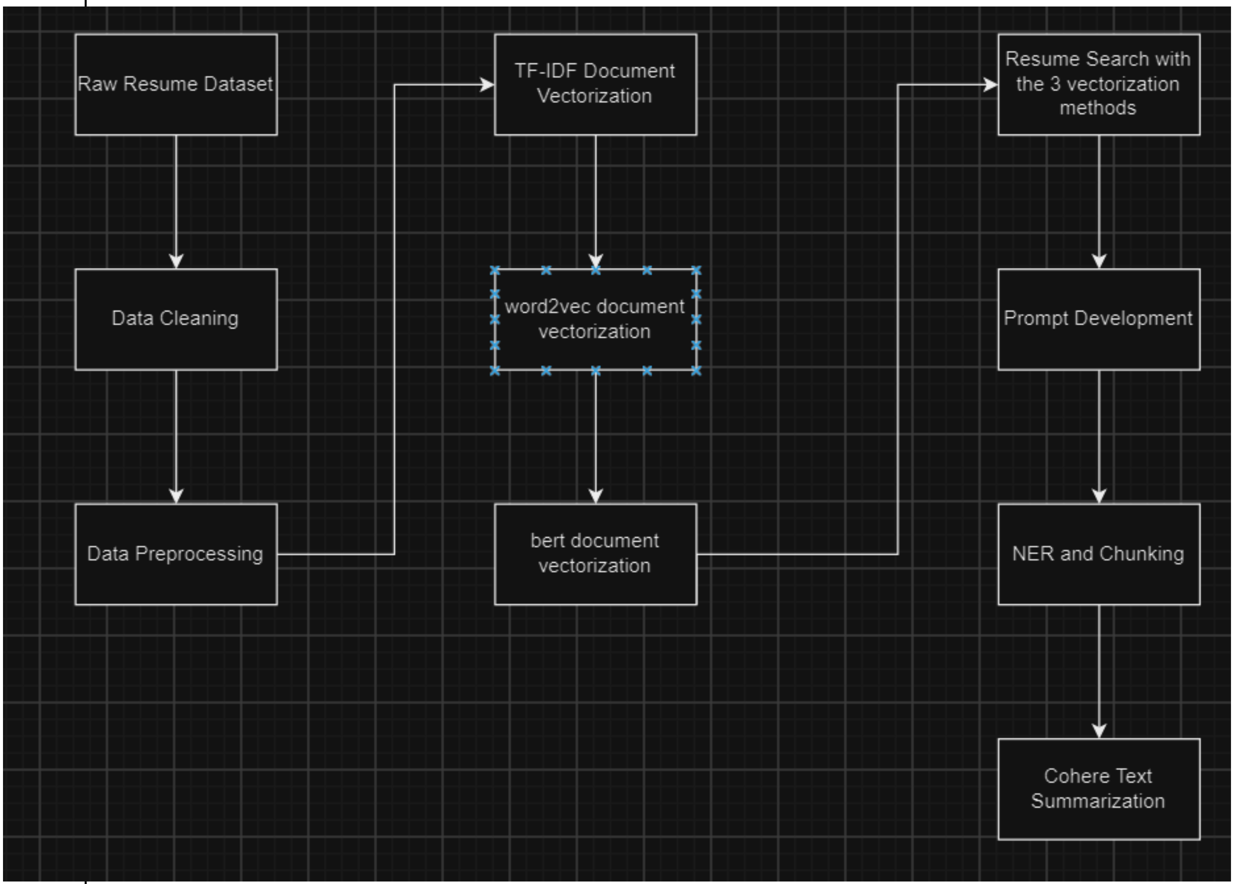
Admin No. 222142U

1. Process Flow Diagram - Resume Classification



Admin No. 222142U

2. Process Flow Diagram - Resume Search and Summarization



3. Pre-processing

- **Clean Duplicates**

```
def clean_duplicates(self, resumes_df, drop=True):
    # drop duplicates to remove the repeated data that may skew representation
    self.duplicate_rows = resumes_df[resumes_df.duplicated()]
    if drop==True:
        resumes_df = resumes_df.drop_duplicates(subset='Resume', keep='first')
    return resumes_df
```

From the data review, I observed that there were 9404 duplicate rows/resumes in the dataset. This accounts for 98.01% of the dataset. These duplicate rows do not provide additional information about the different categories and instead skews the representation of different categories in the dataset, particularly Java Developer category which has 825 duplicate rows. Hence, these duplicate rows need to be dropped.

- Clean Encoding

```
def clean_encoding(self, resumes):
    # ftfy fixes encoding mix ups (mojibake) through the detection of characters that were meant to be UTF-8
    # but was decoded as another form instead
    # this can be useful for Naive bayes whereby the "i" can be the unique variant, can also fix the issue of
    # any foreign language such as text with à or é
    # init a fixed_encoding resume list
    fixed_encoding_resumes = []
    for resume in resumes:
        fixed_encoding_resume = fix_encoding(resume)
        fixed_encoding_resumes.append(fixed_encoding_resume)
    return np.array(fixed_encoding_resumes)
```

From the data review, I observed that there were instances whereby non-english symbols were located in the resumes. After research, these were likely due to certain documents converting from a legacy encoding method to utf8. The conversion must have been incorrect or improperly established, causing these noise to appear in resumes. They then act as potential noise that might affect the performance of the BERT model, RoBerta or GPT2. Hence, we should recover them back to their original characters in order to recover any potential semantic meaning that may have been lost when their encoding was mixed up(mojibake).

- Clean Spaces

```
def clean_spaces(self, resumes):
    # remove additional spacing between words and linebreaks.
    # technically this step is done in processing pipeline,
    # this is a fallback if needed to remove additional spaces, if the removal of stop words/lemmatization/stemming was not used.
    cleaned_spaces_resumes = []
    for resume in resumes:
        new_resume = re.sub(r'\s+', ' ', resume)
        cleaned_spaces_resumes.append(new_resume)
    return np.array(cleaned_spaces_resumes)
```

This was to remove the spacing and linebreaks in the resume. This was needed because certain resumes have long lengths of spaces between words. This may be due to pdf file reading errors. The long space between the words causes the length of the resumes to be unnecessarily longer, which may contribute to longer computation time. Hence, removing the spaces may help reduce computation time. However, this operation is already conducted by various other processing such as removing stop words and stemming. It instead acts as a fallback to give an option to remove these spacings if the other processing methods are not used.

- Clean Spelling

```python
# initalise spell checker
sym_spell = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)
dictionary_path = pkg_resources.resource_filename(
    "symspellpy", "frequency_dictionary_en_82_765.txt"
)
# term_index is the column of the term and count_index is the
# column of the term frequency
sym_spell.load_dictionary(dictionary_path, term_index=0, count_index=1)
```

```python
def clean_spelling(self, resumes):
    # ensures that the text will not be Out of Vocabulary just because it is spelled wrong, ensuring the semantic meaning of the word can be captured
    # we can use symspellpy, Nk-Tree or Peter Novigs method

    #initialise corrected spelling resume list
    corrected_spelling_resumes = []

    for resume in resumes:
        # split the text into words
        words = resume.split()

        # initialise new resume string that will be modified
        new_resume = []

        # correct any misspelled words
        for word in words:
            # lookup top suggestion for misspelled word
            suggestion = sym_spell.lookup(word, Verbosity.CLOSEST, max_edit_distance=2, include_unknown=True)[0].term
            new_resume.append(suggestion)
        new_resume = ' '.join(new_resume)
        corrected_spelling_resumes.append(new_resume)

    return np.array(corrected_spelling_resumes)
```

From the data review, I observed that there were minor spelling errors in certain resumes. Hence, spelling needs to be corrected in the resumes to normalize the text and ensure that they are represented correctly by the BERT model, as well as reduce vocab size. A small max dictionary edit distance was used to ensure that named entities such as Sci-kit does not get changed to science and only minor spelling errors will get corrected.

- Clean Contractions

```python
def clean_contractions(self, resume, model): # this function is too intense to store it all in memory, so instead this function will be used iteratively over the resumes instead of as an entire batch
    # perform expansion of contractions where needed in order to reduce vocab size by splitting up contractions into their actual words
    # and also normalize the text to ensure that the semantic meaning of the  actual words in the contraction can be associated with their corresponding token and vectors which may improve performance of model

    # load the model to check distance between the possible variants of contractions
    word_vector_model = model
    # search for simple contractions and expand them
    for pattern, expansion in simple_contractions.items():
        expanded_resume = pattern.sub(expansion, resume)

    # search for contextual contractions and expand them
    for pattern, potential_expansions in contextual_contractions.items():
        # check if contextual contractions exist in the resume:
        if pattern.search(expanded_resume):
            word_mover_distance_scores = [] #contains the  wm distance scores of the resumes when applied with different potential expansions
            # iterate over potential expansions
            for expansion in potential_expansions:
                # replace the pattern with the current option
                temporary_expanded_resume = pattern.sub(expansion, expanded_resume)
                # calculate word mover's distance between resume after simple expansion and resume after contextual expansion with the current chosen expansion
                wmd_distance = word_vector_model.wmdistance(expanded_resume.split(), temporary_expanded_resume.split())
                word_mover_distance_scores.append(wmd_distance)

            index_ideal_pattern = np.argmax(np.array(word_mover_distance_scores))
            expanded_resume = pattern.sub(potential_expansions[index_ideal_pattern], expanded_resume)

    return expanded_resume
```

There may also exist contractions in the resumes. Contractions are just words that have been combined for spoken language. Hence, we may expand these contractions into individual words to normalize the contraction, reducing the vocab size, and ensuring that the models can represent them more accurately, making the model more stable.

- Clean Camel Case

```python
def clean_camel_case(self, resumes):
    # use regular expressions to find camel case patterns to identify instances whereby the linebreak was not captured,
    # resulting in camelCase situations. These are then split into proper words
    cleaned_camelcase_list = []
    for resume in resumes:

        words = re.findall(r'[A-Z]?[a-z]+|[A-Z]+(?=[A-Z]|$)', resume)
        # join the words with space
        cleaned_text = ' '.join(words)
        cleaned_camelcase_list.append(cleaned_text)
    return cleaned_camelcase_list
```

From the data observation, I observed that sometimes the line breaks are lost when the resumes were being read. This results in the a capatalized word to directly follow the word from the previous line resulting in a camelCase word form. This causes the 2 words to form a completely different non-existent word, making the semantic meaning of the 2 words to be lost. Hence, we need to expand them out.

- Clean Punctuations

```python
def clean_punctuation(self, resumes):
    # see if normalizing the resumes without punctuations removes semantic meaning, or reduces noise
    cleaned_punctuation_resumes = []

    for resume in resumes:
        token=RegexpTokenizer(r'\w+')
        resume = token.tokenize(resume)
        cleaned_punctuation_resume = " ".join(resume)
        cleaned_punctuation_resumes.append(cleaned_punctuation_resume)
    return np.array(cleaned_punctuation_resumes)
```

Punctuations may add semantic meaning that can improve the model, but it can lead to increased complexity that the model will need to learn which may reduce the performance of the model. Larger models may be able to capture semantics of punctuations. Hence, the option to remove or keep punctuation will be mostly decided by how well the BERT model will perform in fine tuning.

- Clean Non-ASCII

```python
def clean_non_ascii(self, resumes):
    # instead of cleaning and recovering the non ascii character,
    # see if removing them is better
    cleaned_non_ascii_resumes = []
    for resume in resumes:
        cleaned_non_ascii_resume = "".join(character for character in resume if ord(character)<128)
        cleaned_non_ascii_resumes.append(cleaned_non_ascii_resume)

    return np.array(cleaned_non_ascii_resumes)
```

Non-ASCII words may add to the complexity of the resumes as well or may contain information that is valuable for the Bert model to learn, such as positional information using point forms. Since the non-ASCII words may cause noise or poor representations of other words, these non-ASCII words may need to be removed. Furthermore, removing non-ascii words can also improve the normalization of the dataset. However, the choice will still mostly be decided by whether the BERT model performs well with it or not.

- Drop stop words

```python
def drop_stop_words(self, resumes):
    # init a list to contain the resumes that are clean of stop words
    dropped_stop_words_resumes = []
    # Get the English stop words list
    stop_words = set(stopwords.words('english'))

    #add custom words
    stop_words.update(('and','I','A', 'And','So','arnt','This','When','It','many','Many','so','cant','Yes','yes','No','no','These','these', 'regards', 'like', 'email'))
    for resume in resumes:
        resume_words = resume.split()
        # Remove stop words
        filtered_tokens = [word for word in resume_words if word not in stop_words]
        filtered_text = ' '.join(filtered_tokens)
        dropped_stop_words_resumes.append(filtered_text)
```

Stop words are words that are useful for language understanding, but do not carry much useful contextual information. Furthermore, stop words may cause the dataset vocab size to be much larger as there is a wide variety of stop words. Furthermore, stop words exist in text in large quantities, which may also negatively affect the performance of the resume search as it may fill skew the resume vector representations. Hence, stop words will likely need to be dropped, which will likely improve performance.

- Perform lower casing

```
def perform_lower_casing(self, resumes):
    # lower cases all the resumes
    lowercased_resumes = [resume.lower() for resume in resumes]
    return lowercased_resumes
```

Lower casing can help normalize the text to a non capitalized form. This can help the normalize the resumes and reduce vocab size. Furthermore, with a lower case resume, BERT will be more stable during training as there will be fewer variants of the same word that it will need to train to learn the similarities between capitalized and non-capitalized words. Hence, lower casing may be needed to improve the performance.

- Perform stemming

```
def perform_stemming(self, resumes): # perform stemming on the  resumes
    # init stemmed resumes list
    stemmed_resumes_list = []
    # Initialize the PorterStemmer
    stemmer = PorterStemmer()
    for resume in resumes:
        tokens = resume.split()
        # perform stemming on each token
        stemmed_tokens = [stemmer.stem(token) for token in tokens]
        stemmed_text = ' '.join(stemmed_tokens)
        stemmed_resumes_list.append(stemmed_text)
    return stemmed_resumes_list
```

Stemming can normalize the words by cutting off letters at the end of words. However, this may lead to words that are not actual words and may lead to poorer text normalization as some words have more complex spellings. This may be able to perform well if the vocabulary of the resumes are simple, however that may not be the case with the long resumes which can be up to 1900 words long. Hence, stemming may not be useful. However, it will be tested in fine tuning first before it gets decided.

- Perform lemmatization

Lemmatization converts words to their base forms (lemmas). This normalization method is more valuable than stemming as the words formed are actual English words. Hence, lemmatization will improve text normalization, resulting in smaller vocab sizes, stabler training and improved representation of words as they are more prevalent in the dataset, giving the bert model more opportunities to learn the semantic meaning of these words.

4. BERT Training and Validation

```python
class BERT_Model:
    def __init__(self, max_len, validation_size, batch_size, epochs, num_labels, lr=1.5e-5, model_name='bert-base-uncased', lower_case=True, bert_config=BertConfig(), weight_decay_1=0.01, weight_decay_2=0.0):
        self.max_len = max_len
        self.validation_size = validation_size
        self.batch_size = batch_size
        self.tokenizer = BertTokenizer.from_pretrained(model_name, do_lower_case=lower_case)
        self.num_labels = num_labels
        self.epochs = epochs
        self.lr = lr
        self.model_name = model_name
        self.lower_case = lower_case
        self.bert_config = bert_config
        self.weight_decay_1 = weight_decay_1
        self.weight_decay_2 = weight_decay_2

    def train_BERT(self, X_train, y_train):
        data_frame = pd.concat([X_train, y_train], axis=1)
        data_frame.columns = ['sentence', 'label']
        # Prepare sentences and labels
        sentences = data_frame.sentence.values
        sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
        labels = data_frame.label.values

        # Tokenize sentences using BERT tokenizer
        tokenized_texts = [self.tokenizer.tokenize(sent) for sent in sentences]

        # Pad sequences and create attention masks
        MAX_LEN = self.max_len
        input_ids = [self.tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]
        input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long", truncating="post", padding="post")

        attention_masks = []
        for sequence in input_ids:
            sequence_mask = [float(id > 0) for id in sequence]
            attention_masks.append(sequence_mask)
        attention_masks = np.array(attention_masks)
        # Split data into training and validation sets
        # training_inputs, validation_inputs, training_labels, validation_labels, training_masks, validation_masks = train_test_split(
        #     input_ids, labels, attention_masks,
        #     random_state=2018, test_size=self.validation_size, stratify=labels
        # )
        sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
        for train_index, val_index in sss.split(input_ids, labels):
            training_inputs, validation_inputs = input_ids[train_index], input_ids[val_index]
            training_masks, validation_masks = attention_masks[train_index], attention_masks[val_index]
            training_labels, validation_labels = labels[train_index], labels[val_index]
```

```python
        # Create DataLoader for training set
        batch_size = self.batch_size
        training_data = TensorDataset(torch.tensor(training_inputs), torch.tensor(training_masks), torch.tensor(training_labels))
        training_sampler = RandomSampler(training_data)
        training_dataloader = DataLoader(training_data, sampler=training_sampler, batch_size=batch_size)

        # Create DataLoader for validation set
        validation_data = TensorDataset(torch.tensor(validation_inputs), torch.tensor(validation_masks), torch.tensor(validation_labels))
        validation_sampler = SequentialSampler(validation_data)
        validation_dataloader = DataLoader(validation_data, sampler=validation_sampler, batch_size=batch_size)

        # Configure BERT model for sequence classification
        configuration = self.bert_config
        self.model = BertModel(configuration)
        configuration = self.model.config


        self.model = BertForSequenceClassification.from_pretrained(self.model_name, num_labels=self.num_labels)
        self.model = nn.DataParallel(self.model)
        self.model.to(device)

        param_optimizer = list(self.model.named_parameters())
        no_decay = ['bias', 'LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_1},
            {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_2}
        ]
        optimizer = AdamW(optimizer_grouped_parameters, lr=self.lr, correct_bias=False)

        def flat_accuracy(predicted_labels, labels):
            predicted_labels = np.argmax(predicted_labels.to('cpu').numpy(), axis=1).flatten()
            labels = labels.to('cpu').numpy().flatten()
            return np.sum(predicted_labels == labels) / len(labels)

        # Train the BERT model
        epochs = self.epochs
        training_losses = []

        for epoch in trange(epochs, desc="Epoch"):
            self.model.train()
            training_loss = 0
            training_steps = 0

            for step, batch in enumerate(training_dataloader):
                inputs = batch[0].to(device)
                attention_masks = batch[1].to(device)
                labels = batch[2].to(device)
```

```python
# Train the BERT model
epochs = self.epochs
training_losses = []

for epoch in trange(epochs, desc="Epoch"):
    self.model.train()
    training_loss = 0
    training_steps = 0

    for step, batch in enumerate(training_dataloader):
        inputs = batch[0].to(device)
        attention_masks = batch[1].to(device)
        labels = batch[2].to(device)

        optimizer.zero_grad()
        outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()

        training_loss += loss.item()
        training_steps += 1

        training_losses.append(loss.item())

    average_training_loss = training_loss/training_steps
    print("Epoch {}: Average Training Loss: {}".format(epoch+1, average_training_loss))

    self.model.eval()
    validation_accuracy = 0
    validation_steps = 0

    for batch in validation_dataloader:
        inputs = batch[0].to(device)
        attention_masks = batch[1].to(device)
        labels = batch[2].to(device)

        with torch.no_grad():
            outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)

        logits = outputs.logits

        temp_validation_accuracy = flat_accuracy(logits, labels)
        validation_accuracy += temp_validation_accuracy
        validation_steps += 1

    average_validation_accuracy = validation_accuracy/validation_steps
    print("Epoch {}: Validation Accuracy: {}".format(epoch+1, average_validation_accuracy))
```

```python
# Evaluate the BERT model on the out-of-domain dataset
self.model.eval()
logits_set = []
labels_set = []

for batch in prediction_dataloader:
    batch_input_ids, batch_attention_masks, batch_labels = batch
    batch_input_ids, batch_attention_masks, batch_labels = batch_input_ids.to(device), batch_attention_masks.to(device), batch_labels.to(device)

    with torch.no_grad():
        outputs = self.model(batch_input_ids, attention_mask=batch_attention_masks)
        logits = outputs.logits

    logits_set.append(logits.cpu().numpy())
    labels_set.append(batch_labels.cpu().numpy())

from sklearn.metrics import matthews_corrcoef, accuracy_score
matthews_set = []
accuracy_set = []
# Calculate Matthews correlation coefficient for each batch
for i in range(len(labels_set)):
    mcc = matthews_corrcoef(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
    acc = accuracy_score(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
    matthews_set.append(mcc)
    accuracy_set.append(acc)

for i, mcc in enumerate(matthews_set):
    print(f"Batch {i + 1}: MCC = {mcc}")
    print(f"Batch {i + 1}: Accuracy = {accuracy_set[i]}")

# Calculate the overall Matthews correlation coefficient
overall_mcc = np.mean(matthews_set)
overall_acc = np.mean(accuracy_set)
print(f"\nOverall MCC: {overall_mcc}")
print(f"\nOverall Accuracy: {overall_acc}")
```

I made the Bert model into a class for training and testing. I have set the default parameters to be:
Max_len = 128
Validation_size=0.2
Batch size = 16
Epochs = 50
Num labels = 25 (the number of unique classes)
Learning rate = 1.5e-5
Model_type = bert-base-uncased
Bert config will be default
Weight_decay will be 0.01 and 0.0 respectively for the 2 weight decays

Max_len will be default 128. This was arbitrarily defined as the starting value for what the length of the length of the input ids should be as it gives room to increase or decrease in fine tuning.

Validation size will be 0.2 as it ensures that the validation dataset will have at least 1 example in each category while also ensuring that the training dataset has more data to train the Bert model.

Batch size was set to be 16 as it is large enough for the model to train on accurately and produce good reduction in training loss while also ensuring that the memory of the computer does not overload.

Epochs was set to 50 as I aim to observe the validation accuracy stabilise and observe the model overfit first, then by observing the stabilised accuracy of the model, we can get a better understanding of what parameters increase the maximum accuracy of the model, and which ones decrease it. This assumes that parameters that improve the performance of the model, increases the maximum stabilised accuracy of the BERT model.

Learning rate was set to 1.5e-5 as I determined that to be fast enough for convergence, while also not being too large to cause the model to converge too early

The initial model used will be bert-base-uncased as I believe bert-large will overfit to the data very easily due to the small size of the dataset after dropping duplicates.

The bert configuration will be default and will be adjusted during fine tuning.

The weight decay will be used to reduce overfitting when needed. The default values are defined arbitrarily.

These parameters will be used in my control experiment when running my first BERT model before additional fine tuning.

In the testing stage, I added an accuracy metric to each batch to observe both MCC and accuracy during the testing of the BERT model

5. Resume Classification

```python
# optimal params for the BERT model class
optimal_max_len = 512
optimal_model_name = 'bert-base-uncased'

# optimal params in Bert config
optimal_vocab_size = 4000
optimal_attention_heads = 8
#start timer
start_time = time.time()
X_train, X_test, y_train, y_test, cleaning_pipeline, processing_pipeline  = clean_and_process(
    documents_df,
    # for cleaning
    clean_encoding=True,
    clean_spaces=False,
    clean_spelling=True,
    clean_contraction=True,
    clean_camel_case=False,
    # for processing
    lower_case=True,
    drop_stop_words=True,
    stemming=False,
    lemmatization=True,
    # for the stratified split
    n_splits=5,
    test_size=0.2,
    random_state=42
)

# end processing timer
end_processing_time = time.time()

# calculate preprocessing time
processing_time = end_processing_time - start_time
print("Processing time:", processing_time, "seconds")

#start timer for training
start_training_time = time.time()

bert_config = BertConfig(
    vocab_size = optimal_vocab_size,
    hidden_size = 768,
    num_hidden_layers = 12,
    num_attention_heads = optimal_attention_heads,
    intermediate_size = 3072,
    hidden_act = "gelu",
    hidden_dropout_prob = 0.1,
    attention_probs_dropout_prob = 0.1,
    max_position_embeddings = 512,
    type_vocab_size = 2,
    initializer_range = 0.02,
    layer_norm_eps = 1e-12,
)

bert_model = BERT_Model(max_len=optimal_max_len, validation_size=0.2, batch_size=16, epochs=50, num_labels=len(np.unique(y_train)), lr=1.5e-5, model_name=optimal_model_name, lower_case=True, bert_config=bert_config)

bert_model.train_BERT(X_train, y_train)

# end training timer
end_training_time = time.time()

# calculate training time
training_time = end_training_time - start_training_time
print("Training time:", training_time, "seconds")

bert_model.test_BERT(X_test, y_test)
```

```
Processing time: 69.00532674789429 seconds
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:411: FutureWarning: This implem
  warnings.warn(
Epoch:   0%|          | 0/50 [00:00<?, ?it/s]Epoch 1: Average Training Loss: 3.252115100622177
Epoch:   2%||         | 1/50 [00:12<09:52, 12.10s/it]Epoch 1: Validation Accuracy: 0.13125
Epoch 2: Average Training Loss: 3.121789425611496
```

```
Epoch:  78%|███████   | 39/50 [08:20<02:23, 13.03s/it]Epoch 39: Validation Accuracy: 0.8375
Epoch 40: Average Training Loss: 0.08816165011376143
Epoch:  80%|████████  | 40/50 [08:33<02:10, 13.03s/it]Epoch 40: Validation Accuracy: 0.8375
Epoch 41: Average Training Loss: 0.08743474539369345
Epoch:  82%|████████  | 41/50 [08:46<01:57, 13.03s/it]Epoch 41: Validation Accuracy: 0.80625
Epoch 42: Average Training Loss: 0.08514079637825489
Epoch:  84%|████████  | 42/50 [08:59<01:44, 13.04s/it]Epoch 42: Validation Accuracy: 0.80625
Epoch 43: Average Training Loss: 0.0812156330794096
Epoch:  86%|████████  | 43/50 [09:12<01:31, 13.05s/it]Epoch 43: Validation Accuracy: 0.80625
Epoch 44: Average Training Loss: 0.07818134129047394
Epoch:  88%|████████  | 44/50 [09:25<01:18, 13.05s/it]Epoch 44: Validation Accuracy: 0.80625
Epoch 45: Average Training Loss: 0.07365255942568183
Epoch:  90%|█████████ | 45/50 [09:39<01:05, 13.06s/it]Epoch 45: Validation Accuracy: 0.80625
Epoch 46: Average Training Loss: 0.07365631312131882
Epoch:  92%|█████████ | 46/50 [09:52<00:52, 13.06s/it]Epoch 46: Validation Accuracy: 0.80625
Epoch 47: Average Training Loss: 0.0726084541529417
Epoch:  94%|█████████ | 47/50 [10:05<00:39, 13.06s/it]Epoch 47: Validation Accuracy: 0.80625
Epoch 48: Average Training Loss: 0.06808700319379568
Epoch:  96%|█████████ | 48/50 [10:18<00:26, 13.06s/it]Epoch 48: Validation Accuracy: 0.80625
Epoch 49: Average Training Loss: 0.06857918854802847
Epoch:  98%|█████████ | 49/50 [10:31<00:13, 13.07s/it]Epoch 49: Validation Accuracy: 0.80625
Epoch 50: Average Training Loss: 0.06365389889106154
Epoch: 100%|██████████| 50/50 [10:44<00:00, 12.89s/it]Epoch 50: Validation Accuracy: 0.80625
Training time: 652.2947010993958 seconds

Batch 1: MCC = 0.7436169051971016
Batch 1: Accuracy = 0.75
Batch 2: MCC = 0.5387931034482759
Batch 2: Accuracy = 0.5625
Batch 3: MCC = 0.675
Batch 3: Accuracy = 0.7142857142857143

Overall MCC: 0.6524700028817925

Overall Accuracy: 0.6755952380952381
```

I observed that the most optimal method of training bert was using the parameters shown in the picture (please also refer to the code I used).

Clean Encoding: I observed that clean encoding helped improve the model because the semantic meaning of the words with broken encodings were recovered. This may have helped provide BERT with additional formatting and positional information between words as points forms are fixed for example. The clean encoding function also fixed broken encoding involving certain letters such as ï. This may help normalize words as now the broken encoding is fixed, resulting in proper words being formed. As a result, the model was trained more stable, words are represented more effectively, and the model was able to capture more dependencies between words and characters or symbols. Hence, I believe that was why clean encoding improved the accuracy of Bert

Clean spaces: I believe that clean spaces reduced the performance as the spacing in resumes provided semantic meaning, which now the BERT model was unable to capture. Hence, the performance was reduced

Clean Spelling; I observed this to improve the accuracy of BERT. This was likely since the spell checker was only allowed to correct words if only 2 modifications were needed. This ensured that the resumes had corrected spelling, which normalized the words, providing a greater representation of words in the resume. But, it also ensured that larger corrections were not made, ensuring that named entities such as sci-kit do not get lost as much as compared to other spelling models used such as peter nordvigs method and pyspellchecker. This allows for the normalization of text without the lost of named entities.

Clean Contractions: I observed this to have greatly improved the accuracy of BERT as compared to other preprocessing methods. I believe this method improved the model because it was able to expand contractions such as “it’s” to “it is”, which helped normalize the text. But in addition, it helped remove stop words as now the expanded word can now be identified removed as a stop word by the stop word process, allowing them to synergise well.

Clean Camel Case: I observed this to have decreased the accuracy of BERT substantially. This is likely because it also cleans out the spaces and the line breaks, resulting in the loss of the formatting information that may be crucial for the BERT model to learn. It may also be because the regular expression had unintended effects that caused the formatting or the words in the resumes to be wrongly formatted, such as making words that were supposed to be in camel case style separated. This may have caused the vocab size to increase and the text to diversify as the resulting words formed may not be actual words. Hence, it likely caused the performance of the model to decrease.

Clean Punctuation: I observed this to have increased the accuracy of BERT. This was likely because the punctuation added complexity to the resumes which bert-base was unable to capture effectively. By removing it, BERT was likely able to focus more on the words in the resumes, allowing BERT to perform better as it was able to capture the contextual information of the resumes more, instead of the punctuations.

Clean Non-ASCII: This reduced the accuracy of the BERT Model. This was likely because the non ascii words or characters held important information whether semantically or in terms of formatting and positioning. By removing them, this information was lost, which resulted in poorer performing BERT.

Lower Casing: This improved the model as the bert-base-uncased was already trained on lower case words. Hence, by normalizing the words to lower case, the model was able to capture the semantic meaning of these words more easily. In addition, the text will be normalized, resulting in smaller vocab size, stable training and better representation of words

Dropping stop words: This improved the model as it removed words which did not add semantic meaning to the resumes. This allowed the model to focus on words which matter more in the resumes, such as topics, skills etc. This allowed the BERT model to capture more semantic meaning and contextual dependencies in the resumes, which improves the performance of the bert model

Stemming: This reduced the accuracy of the model as it likely diversified the vocab size instead of normalizing it. By cutting of the ends of words, the resulting words do not actually exist , resulting in diversity in the vocabulary. This is especially the case because the resumes in the dataset are verbose, with resumes that have lengths of about 1900 words. Hence, stemming likely resulted in poorer text normalization and poorer representation of words. Hence, stemming reduced the accuracy of the model

Lemmatization. This normalized the text in the resumes by turning words into their base forms (lemmas) resulting in normalized text and allows the model to perform better on a smaller vocab size and with greater representations of words.

Bert config vocab size: Setting this value to be 4000 helped bert perform better on the dataset. This was likely because it helped reduce the vocab size of bert to only the top 4000 most frequent words in the resumes dataset. This likely resulted in the gibberish in the resumes, due to errors in the decryption and sending of data, to be OOV. This is useful as the BERT will then be able to focus on the semantic meaning and the contextual dependencies of the most important and prevalent words in the resumes dataset. Hence, reducing the vocab size improved the performance.

Bert config attention heads: reducing the attention heads to 8 improved the accuracy of the model. This was likely due to the fact that the fewer attention heads helped prevent the bert model from overfitting too much. It may also be because the intricate details in the resume are not as important in identifying the category of the resume, and capturing the general details of the resume is more important instead. Hence, fewer attention head (8) is more optimal.

Bert model type: bert-base-uncased seems to perform the best as compared to bert-base-cased, bert—large-uncased and bert-large-cased. This was likely because the capitalizations of the words in the resumes are not as important in distinguishing the category of different resumes in the resumes dataset. The bert-large, with its larger number of parameters, is likely overfitting to the small dataset of the training dataset. Hence, bert-large was unable to train and perform well as compared to bert-base.

6. Vectorization

- TF-IDF

```python
def get_tf_idf(self):
    # Calculate the TF of a value
    def calculate_tf(value):
        tf = np.log10(value + 1)
        return tf
    # Calculate the IDF of a value
    def calculate_idf(total_number_of_documents, value):
        idf = np.log10(total_number_of_documents/value)
        return idf

    # Create a CountVectorizer object
    self.tdm_vectorizer = CountVectorizer()
    # Create a term-document matrix
    X = self.tdm_vectorizer.fit_transform(self.resumes)
    tdm = pd.DataFrame(X.toarray(), index=self.resumes_tag, columns=self.tdm_vectorizer.get_feature_names_out())
    return tdm

    # Create the IDF array
    total_number_of_documents = len(self.resumes)
    total_number_of_documents_in_which_each_word_occurs = np.sum(tdm > 0, axis=0)
    idf_array = total_number_of_documents_in_which_each_word_occurs.apply(lambda value: calculate_idf(total_number_of_documents, value))

    # Create the TF-IDF matrix
    tf_matrix = tdm.apply(lambda value: calculate_tf(value))
    tf_idf_matrix = tf_matrix.T*idf_array.to_numpy()[:, np.newaxis]
    tf_idf_matrix = tf_idf_matrix.T # transpose the tf idf matrix to ensure that the unique document vecotrs are represented as rows

    return tf_idf_matrix
```

Get TF-IDF resume vectors

```python
# get tf-idf matrix as resume vectors for resume similarity
def create_tf_idf_resume_vectors(resume_vectorizer_object):
    tf_idf = resume_vectorizer_object.get_tf_idf()
    return tf_idf

tf_idf_resume_vectors = create_tf_idf_resume_vectors(resume_vectorizor)
tf_idf_resume_vectors
```

	000	01	017ydlarrfnns	03	04th	05	050education	07	07education	08	...	ymcaust	young	yr	yyyy	zambia	zenoss	zero	zhypility	zone	zookeeperbrbegnceae
Document 0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 3	0	0	0	0	0	0	0	0	0	0	...	1	0	0	0	0	0	0	0	0	0
Document 4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Document 871	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 872	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 873	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 874	0	0	0	0	0	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
Document 7135	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

191 rows × 7644 columns

TF-IDF Document vectorization was used as there could be a relationship between the frequencies of the words as well as the use of rare words in the resume. By using tf-idf document vectors, I will extract the top 10 most similar resumes based on this relationship.

- Word2Vec (GoogleNews-vectors-negative300.bin.gz)

```python
def get_w2v_resume_vectors(self):
    # init list for w2v resume vectors
    w2v_resume_vectors = []
    word2vec_model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz', binary=True) # word2vec model
    for resume in self.resumes:
        tokens = resume.lower().split()
        vectors = []

        for token in tokens:
            if token in word2vec_model:
                vectors.append(word2vec_model[token]) # append the vector for the token to the list

        if vectors:
            w2v_resume_vectors.append(sum(vectors) / len(vectors))
        else:
            # Handle the case where all words are out-of-vocabulary
            w2v_resume_vectors.append([0] * 300)
    w2v_document_vector_df = pd.DataFrame(w2v_resume_vectors, index=self.resumes_tag)
    return w2v_document_vector_df
```

Get Word2Vec resume vectors

```python
# use pretrained w2v model to get word embeddings, then aggregating them to represent the documents as resume vectors
def create_w2v_resume_vectors(resume_vectorizer_object):
    resume_vectors = resume_vectorizer_object.get_w2v_resume_vectors()
    return resume_vectors

w2v_resume_vectors_df = create_w2v_resume_vectors(resume_vectorizor)
w2v_resume_vectors_df
```

	0	1	2	3	4	5	6	7	8	9	...
Document 0	0.036967	0.084483	-0.024058	0.221029	-0.066040	-0.079427	0.133138	0.083506	-0.017314	0.008708	...
Document 1	0.007552	0.081964	0.003778	0.063619	-0.014320	0.064831	0.046822	-0.126431	0.083323	0.038452	...
Document 2	-0.009160	0.005853	-0.012335	0.028625	-0.075631	0.026635	0.055609	-0.072438	0.065193	0.017904	...
Document 3	-0.024649	0.061257	-0.010159	0.018992	0.022893	0.054801	0.097628	-0.159597	0.077424	0.015363	...
Document 4	0.001597	0.047217	0.017758	0.078117	-0.003512	0.022782	0.095858	-0.104076	0.047373	0.009634	...
...	...	...	...	...	...	...	...	...	...	...	...
Document 871	-0.060573	0.028913	0.017195	0.018556	-0.046230	-0.020992	0.043296	-0.095800	0.033689	0.032672	...
Document 872	-0.006490	-0.020571	0.006540	0.021436	-0.030863	0.013456	0.101242	-0.091641	0.025359	0.021968	...
Document 873	-0.013087	0.001890	0.001140	0.002284	-0.034119	0.046723	0.071164	-0.059569	0.064456	0.062079	...
Document 874	-0.042782	-0.012504	0.007297	0.028540	-0.042507	0.001484	0.076802	-0.098524	0.009501	0.042471	...
Document 7135	-0.028080	0.007825	-0.022238	0.000052	-0.009220	0.005478	0.037713	-0.048563	0.097795	-0.018298	...

	290	291	292	293	294	295	296	297	298	299
Document 0	-0.081319	0.004145	0.007324	-0.040446	-0.008362	0.047623	-0.021439	0.134725	-0.034912	0.012919
Document 1	-0.115987	0.003219	-0.089246	0.012662	-0.023070	-0.016394	0.014498	-0.017062	0.021132	-0.020109
Document 2	-0.097536	0.060173	-0.057288	0.015484	-0.047940	0.010749	0.021289	-0.023826	-0.003708	0.005645
Document 3	-0.105144	0.001250	-0.088572	0.034225	-0.057445	-0.025053	0.029345	-0.052178	0.011758	-0.039627
Document 4	-0.097507	0.016839	-0.128038	-0.016298	-0.016221	-0.037907	-0.002667	-0.019781	0.014034	0.011253
...	...	...	...	...	...	...	...	...	...	...
Document 871	-0.055669	0.036343	-0.074221	0.057324	-0.048289	-0.020312	-0.050574	-0.006789	-0.039624	-0.066933
Document 872	-0.126423	0.040666	-0.087931	0.005204	-0.002130	-0.023163	-0.022640	-0.034506	-0.020826	-0.028530
Document 873	-0.082955	0.054987	-0.056833	0.037414	0.015923	-0.003708	-0.004996	-0.039468	-0.034522	-0.030580
Document 874	-0.089859	0.042612	-0.112273	0.054253	-0.018781	-0.032906	-0.040647	-0.032560	-0.019426	-0.018774
Document 7135	-0.085184	0.047800	-0.068797	0.010702	-0.005321	-0.003034	0.028134	-0.020296	-0.000159	-0.058162

191 rows × 300 columns

Using the word2vec model I downloaded, I will make embedding vectors for each word in each resume, then aggregate them to form a document/resume vector for each resume. This is based on the idea that similar aggregated word vectors are similar documents/resumes. Hence, I will explore this method using word2vec.

- BERT document vectors

```python
def get_bert_resume_vectors(self, model='bert-base-uncased', lower_case=False):
    bert_tokenizer = BertTokenizer.from_pretrained(model, do_lower_case=lower_case)

    # Tokenize and obtain input IDs for the sentences
    inputs = bert_tokenizer(self.resumes, return_tensors="pt", padding=True, truncation=True)
    tokenized_sentence_ids = inputs["input_ids"]

    # Use the BERT model to obtain embeddings
    self.bert_model = BertModel.from_pretrained(model)

    with torch.no_grad():
        outputs = self.bert_model(**inputs)

    # Get the embeddings from the last hidden layer
    last_hidden_state = outputs.last_hidden_state

    resume_vectors = torch.mean(last_hidden_state, dim=1)

    bert_resume_vectors_df = pd.DataFrame(resume_vectors, index=self.resumes_tag)

    return bert_resume_vectors_df
```

Similar to word2vec, I will make embedding vectors for each word in each resume, then aggregate them to form a document/resume vector for each resume. This is based on the idea that similar aggregated word vectors are similar documents/resumes. Hence, I will explore this method using bert. This should theoretically perform better than word2vec due to the larger dimension of the document/resume vectors produced by bert (768) which is higher than word2vec's (300).

7. Resume Search

Please refer to my resume to know what are the contents of my_resume_text

- TF-IDF

Resume Similarity using TF-IDF

```python
# get the tf idf document vector for my resume
def get_my_resume_tf_idf(resume, resume_vectorizer_object):
    my_resume = [resume] # put the resume text into a list for the CountVectorizer

    # transform my resume using the existing count vectorizer object
    tdm_vectorizer = resume_vectorizer_object.tdm_vectorizer
    my_resume_word_count_vector = tdm_vectorizer.transform(my_resume)

    # Calculate TF for my resume
    my_resume_tf_vector = np.log10(my_resume_word_count_vector.toarray() + 1)

    # Calculate IDF for the my resume
    total_number_of_documents = len(resume_vectorizer_object.resumes)
    total_number_of_documents_in_which_each_word_occurs = np.sum(my_resume_tf_vector > 0)
    my_resume_idf_vector = np.log10(total_number_of_documents / total_number_of_documents_in_which_each_word_occurs)

    # Calculate TF-IDF for the new resume
    my_resume_tf_idf_vector = my_resume_tf_vector * my_resume_idf_vector

    return pd.DataFrame(my_resume_tf_idf_vector, index=['My Resume'], columns=tdm_vectorizer.get_feature_names_out())

my_resume_tf_idf = get_my_resume_tf_idf(my_resume_text, resume_vectorizor)
my_resume_tf_idf
```

	000	01	017ydlarrfnns	03	04th	05	050education	07	07education	08	...	young	yr	yyyy	zambia	zenoss	zero	zhypility	zone	zookeeperbrbegnceae	ã¾â
My Resume	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

1 rows × 7667 columns

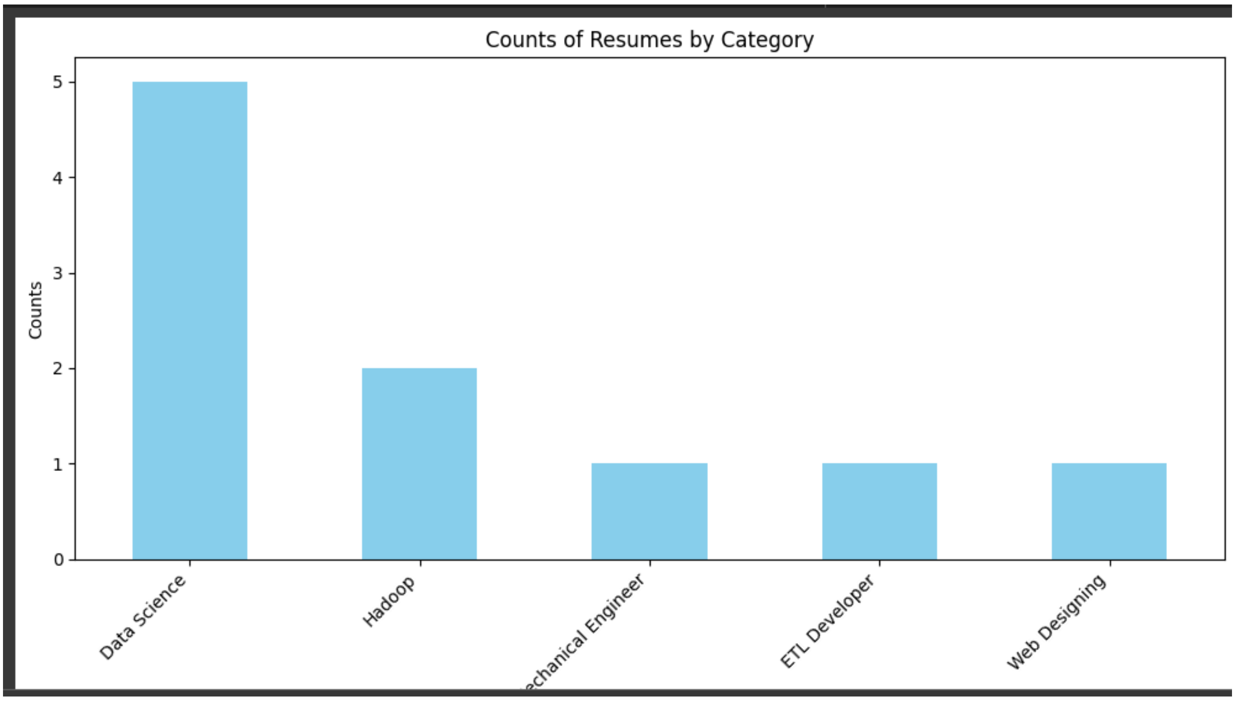
```python
# Calculate Cosine Similarity between my resume and all resumes
cosine_similarities_scores = []

i = 0 # used to iterate through the categories  variable to obtain the corresponding category information of the document tag
for resume_tag, resume_vector in tf_idf_resume_vectors.iterrows():
    similarity = calculate_cosine_similarity(np.array(resume_vector).reshape(1, -1), np.array(my_resume_tf_idf).reshape(1, -1))
    category = resume_vectorizor.categories[i]
    resume = resume_vectorizor.resumes[i]
    cosine_similarities_scores.append([resume_tag, category, similarity, resume])
    i += 1

my_resume_tf_idf_similarity_df = pd.DataFrame(cosine_similarities_scores, columns=['resume_tag', 'category', 'similarity', 'resume'])

my_resume_tf_idf_most_similar_resumes = get_n_most_similar(10, my_resume_tf_idf_similarity_df)
my_resume_tf_idf_most_similar_resumes
```

	resume_tag	category	similarity	resume
5	Document 5	Data Science	0.268152	skill python tableau data visualization studio...
8	Document 8	Data Science	0.267382	expertise data quantitative analysis decision ...
6	Document 6	Data Science	0.242921	education detail bbntgbqlmkktech maya para ins...
157	Document 724	Hadoop	0.207867	operating systems:-linux- ubuntu window 2007/0...
158	Document 725	Hadoop	0.205180	technical kill get programming language apache...
2	Document 2	Data Science	0.201203	skill python sap hana tableau sap hana sql sap...
7	Document 7	Data Science	0.197579	personal skill ability quickly grasp technical...
51	Document 184	Mechanical Engineer	0.196127	education detail may 1999 september 2002 diplo...
167	Document 767	ETL Developer	0.195067	technical summary knowledge informatics power ...
45	Document 139	Web Designing	0.195053	skill language (basic), java (basic) web techn...



My resume is for the job role of Product Manager (AI and Data). The job focuses on using AI and Data in the job to improve business opertaions and improve customer relationships and upsell opportunities, which was emphasised in the resume. Hence, it makes sense that the majority(5) of the top 10 most similar resumes is in data science category. ETL developer makes sense as the resume does talk about knowing how to sift through and use data advantageously for insights. However, hadoop and web designing may not be as relevant. Furthermore, the similarity scores of my resume to other resumes are very low, at about 0.25.

TF-IDF may not be as useful. The resume similarity between the resumes and my resume are so low that it becomes difficult to differentiate similar documents and dissimilar documents/resumes. Hence TF-IDF may not be very useful

- Word2Vec

Resume Similarity using Word2Vec

```python
# get the w2v document vector for my resume
def get_my_resume_w2v_vector(resume, resume_vectorizer_object):
    w2v_resume_vectors = []
    word2vec_model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin.gz', binary=True)
    tokens = resume.lower().split()
    vectors = []

    for token in tokens:
        if token in word2vec_model:
            vectors.append(word2vec_model[token])

    if vectors:
        w2v_resume_vectors.append(sum(vectors) / len(vectors))
    else:
        # Handle the case where all words are out-of-vocabulary
        w2v_resume_vectors.append([0] * 300)
    w2v_document_vector_df = pd.DataFrame(w2v_resume_vectors, index=['My Resume'])
    return w2v_document_vector_df

my_resume_w2v_vector = get_my_resume_w2v_vector(my_resume_text, resume_vectorizor)
my_resume_w2v_vector
```

	0	1	2	3	4	5	6	7	8	9	...	290	291	292	293	294	295	296	297	298	299
My Resume	-0.008947	0.020084	0.008825	0.062197	-0.05413	0.021215	0.039885	-0.069193	0.029237	0.013633	...	-0.06785	0.041891	-0.072088	0.040057	-0.021161	-0.022423	0.024432	-0.040705	-0.006532	0.02525

1 rows × 300 columns

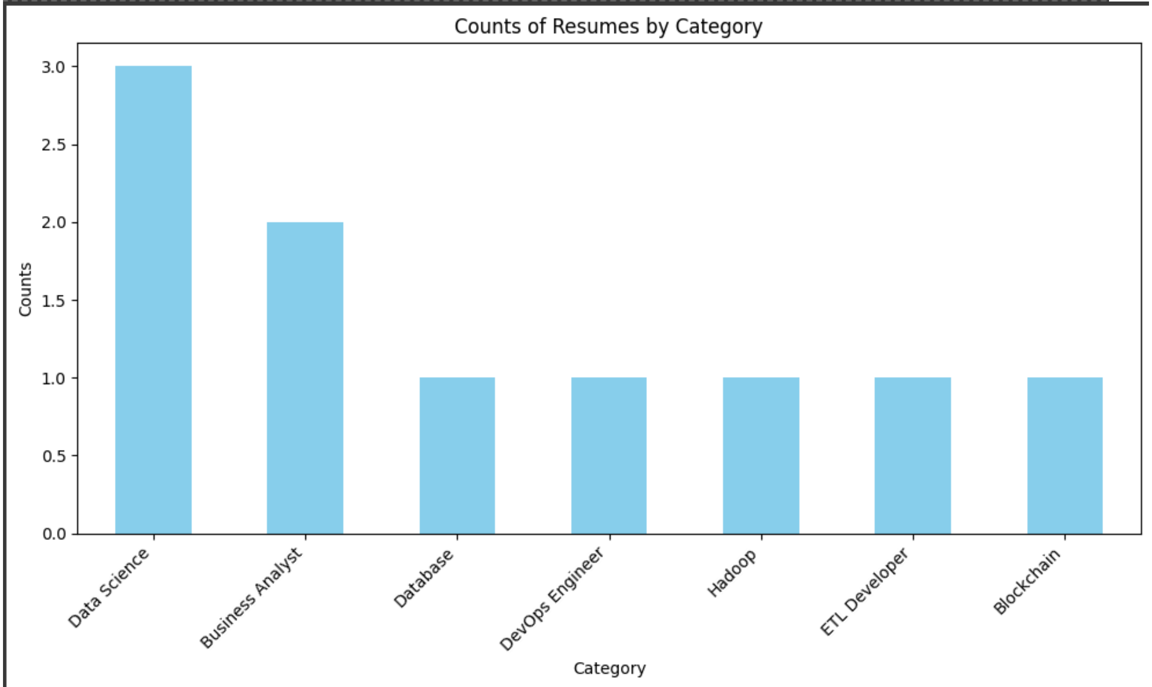
```python
# Calculate Cosine Similarity between my resume and all resumes
cosine_similarities_scores = []

i = 0 # used to iterate through the categories  variable to obtain the corresponding category information of the document tag
for resume_tag, resume_vector in w2v_resume_vectors_df.iterrows():
    similarity = calculate_cosine_similarity(np.array(resume_vector).reshape(1, -1), np.array(my_resume_w2v_vector).reshape(1, -1))
    category = resume_vectorizor.categories[i]
    resume = resume_vectorizor.resumes[i]
    cosine_similarities_scores.append([resume_tag, category, similarity, resume])
    i += 1

my_resume_w2v_similarity_df = pd.DataFrame(cosine_similarities_scores, columns=['resume_tag', 'category', 'similarity', 'resume'])

my_resume_w2v_most_similar_resumes = get_n_most_similar(10, my_resume_w2v_similarity_df)
my_resume_w2v_most_similar_resumes
```

	resume_tag	category	similarity	resume
2	Document 2	Data Science	0.894835	skill python sap hana tableau sap hana sql sap...
8	Document 8	Data Science	0.884474	expertise data quantitative analysis decision ...
152	Document 699	Database	0.881976	technical skill sql oracle v10, v11, via progr...
6	Document 6	Data Science	0.881449	education detail bbntgbqlmkktech maya para ins...
130	Document 588	DevOps Engineer	0.879029	technical skill alm, rtc jira as400 (iseries) ...
158	Document 725	Hadoop	0.875425	technical kill get programming language apache...
89	Document 395	Business Analyst	0.874430	key skill requirement gathering requirement an...
88	Document 394	Business Analyst	0.874184	technical skill application server ii 6bntgbql...
168	Document 768	ETL Developer	0.873582	technicalproficiencies db: oracle big domain i...
178	Document 830	Blockchain	0.872243	skill strong fundamental problem solving ether...



Data science is the main category(3) as my resume emphasised a lot on my experience in getting advantageous data insights.

Business Analyst is also valid because I did emphasis my ability to collect insightful data for businesses

DevOps Engineer and database is also valid because the resume did emphasise on abilities such as cloud, databases and managed service providers which devops and database engineers usually have experience in.

Web Designing is anomalous as the resume did not emphasise on my ability to design website. This is not valid

Hadoop and blockchain is not very accurate as I have not had any experience and did not talk about any of those in the resume.

ETL Developer is valid as i did emphasise on extracting information and insights from data in the resume.

Word2Vec was able to extract documents which have high ranges of similarity score, allowing it to more easily differentiate between dissimilar and similar documents. This is more useful than TF-IDF. However, the top 10 resumes come from diverse categories, some of which are not valid to my resume type. This suggests that the word2vec document vectors may not be as reliable at finding similar documents of similar classes.

- Bert Document/Resume Vectors

Resume Similarity using Bert Embeddings

```python
# get the bert resume vector for my resume
def get_my_resume_bert_vector(resume, resume_vectorizer_object):
    my_resume = [resume]

    bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)

    # Tokenize and obtain input IDs for the sentences
    inputs = bert_tokenizer(my_resume, return_tensors="pt", padding=True, truncation=True)
    tokenized_sentence_ids = inputs["input_ids"]

    # Use the BERT model to obtain embeddings
    bert_model = BertModel.from_pretrained('bert-base-uncased')

    with torch.no_grad():
        outputs = bert_model(**inputs)

    # Get the embeddings from the last hidden layer
    last_hidden_state = outputs.last_hidden_state
    resume_vectors = torch.mean(last_hidden_state, dim=1)
    bert_document_vector_df = pd.DataFrame(resume_vectors, index=['My Resume'])
    return bert_document_vector_df

my_resume_bert_vector = get_my_resume_bert_vector(my_resume_text, resume_vectorizor)
my_resume_bert_vector
```

	0	1	2	3	4	5	6	7	8	9	...	758	759	760	761	762	763	764	765	766	767
My Resume	-0.13721	0.277003	0.43285	-0.082165	0.667725	-0.093854	0.185473	0.335892	-0.172753	-0.388005	...	-0.186621	0.043721	0.209761	-0.295223	0.041073	-0.313661	-0.222223	-0.179762	-0.121265	-0.200716

1 rows × 768 columns

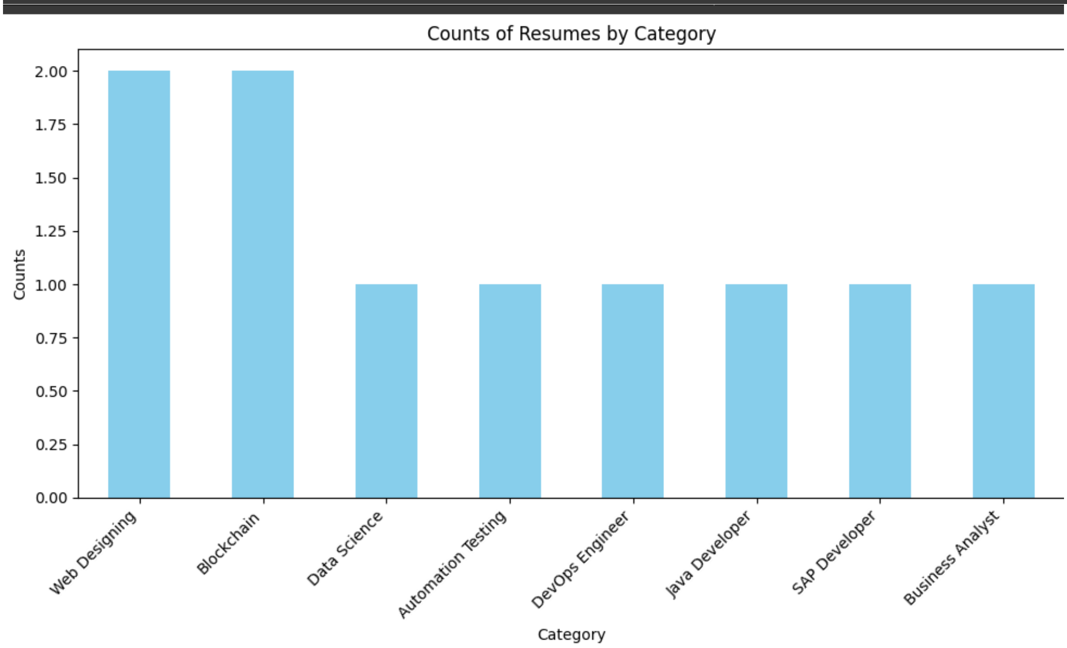
```python
# Calculate Cosine Similarity between my resume and all resumes
cosine_similarities_scores = []

i = 0 # used to iterate through the categories  variable to obtain the corresponding category information of the document tag
for resume_tag, resume_vector in bert_resume_vectors_df.iterrows():
    similarity = calculate_cosine_similarity(np.array(resume_vector).reshape(1, -1), np.array(my_resume_bert_vector).reshape(1, -1))
    category = resume_vectorizor.categories[i]
    resume = resume_vectorizor.resumes[i]
    cosine_similarities_scores.append([resume_tag, category, similarity, resume])
    i += 1

my_resume_bert_similarity_df = pd.DataFrame(cosine_similarities_scores, columns=['resume_tag', 'category', 'similarity', 'resume'])

my_resume_bert_most_similar_resumes = get_n_most_similar(10, my_resume_bert_similarity_df)
my_resume_bert_most_similar_resumes
```

	resume_tag	category	similarity	resume
45	Document 139	Web Designing	0.876968	skill language (basic), java (basic) web techn...
6	Document 6	Data Science	0.871144	education detail bbntgbqlmkktech maya para ins...
107	Document 455	Automation Testing	0.869507	excellent grasping power learning new concept ...
178	Document 830	Blockchain	0.864927	skill strong fundamental problem solving ether...
127	Document 585	DevOps Engineer	0.862021	technical skill key skill technology bntgbqlmk...
82	Document 320	Java Developer	0.860908	technical skill programming language java (ser...
43	Document 137	Web Designing	0.859637	education detail bbntgbqlmkkcckekjofvwqa bache...
182	Document 837	Blockchain	0.855121	skill bitcoin, ethereum solidity hyperledger, ...
98	Document 424	SAP Developer	0.853625	education detail sap technical architect sap t...
89	Document 395	Business Analyst	0.852954	key skill requirement gathering requirement an...



Web Designing, Blockchain and Java Developer are not valid as my resume does not emphasise on these areas.

Data science is valid as my resume emphasised programming capabilities

Automation testing is valid as my resume did include IP configuration abilities and world skills mechatronics experience.

SAP Developer is valid as my resume did emphasise on creating applications with my programming abilities

Business analyst is valid because my resume emphasised on my ability to create business value using data analysis.

The BERT resume vectors can extract documents which have high ranges of similarity score, allowing it to more easily differentiate between dissimilar and similar documents, similar to word2vec. However, the majority (5) of the top 10 most similar resumes should not be similar to my resume, meaning that the bert document vectors are not as reliable at finding similar documents of similar classes.

8. Resume Summarization

8. Text Summarization

```python
def get_important_details(resume):
    words = word_tokenize(resume)
    tags = pos_tag(words)
    ner_tags = nltk.ne_chunk(tags)
    ner_categories = set()

    for chunk in ner_tags:
        if isinstance(chunk, nltk.Tree):
            ner_label = chunk.label()
            token = " ".join(word for word, tag in chunk.leaves())
            ner_categories.add(ner_label)

    # Define the chunk grammar and attempt to get NP and VP in order to capture the tasks the person has done,
    # such as "I enhanced the performance of the company".
    # It is also to capture specific subjects from noun phrases, which may be exceptional such as a robust AI model, or a skilled analyst
    chunk_grammar = r"""
        NP: {<DT>?<JJ>*<NN>}
        VP: {<VB.*><.*>*<RB>?}
    """
    # Create a chunk parser
    chunk_parser = nltk.RegexpParser(chunk_grammar)

    # Chunking
    tree = chunk_parser.parse(tags)

    # Initialize an empty list to store phrases
    phrases = []

    # Iterate through subtrees and append phrases to the list
    for subtree in tree.subtrees():
        if subtree.label() in ['NP']:
            phrases.append(subtree)

    cohere_string_appendage = f"These are some key named entities, you should consider in your text summarisation to identify unique educational institutes, companies or locations: {ner_categories}\nThese are
    return cohere_string_appendage

# Read my own resume by defining a function to read the pdf file
def extract_text_from_pdf(pdf_path):
    with pdfplumber.open(pdf_path) as pdf:
        text = ''
        for page in pdf.pages:
            text += page.extract_text()
    return text

pdf_path = '222142U_Gawain_Resume.pdf'

my_resume_text = extract_text_from_pdf(pdf_path)
```

```
# initialise the prompt with one shot learning and prepare it to receive Named entities and Verb phrases and noun phrases
co = cohere.Client('ik52RHHMaq7EpXT2GPUOiRMXKS71oRdhhzenzrZ7')

prompt = f"You are tasked to summarize a given resume in order to identify key attributes of the potential employee to give the company a concise and unbiased overview of the key attrib

response = co.chat(
    prompt,
    model="command",
    temperature=0.9
)
answer = response.text
answer
```

```
'I understand the task and requirements well. I'll make sure to evaluate the provided resume and create a clear and concise summary of the potential employee's key attributes for the company
\n\nPlease provide the resume document, along with any specific formatting or instructions, and let me know if there's anything else I can help you with!'
```

```
[36]
# send information regarding my resume for resume summarization

important_details = get_important_details(my_resume_text)

message = f"{important_details}\n\n{my_resume_text}"

response = co.chat(
    message,
    model="command",
    temperature=0.9
)
answer = response.text
answer
```

```
'Gawain Yeo-Koh is a skillful and ambitious AI and Data Engineering enthusiast with a knack for pioneering cutting-edge solutions using AI models. With experience in Artificial Intelligence E
expertise in Data Preparation and Visualization enables him to analyze customer behavior and identify upsell opportunities, making him a valuable asset in the AI and Data field.\n\nIn 2023, C
ed upsell opportunities using Customer Relationship Management. He also explored AI tools such as Deepface, Keras, and YOLO in his Artificial Intelligence Engineering Project. Additionally, H
-sales.\n\nWith a Diploma in AI & Data Engineering from Nanyang Polytechnic (2022-2025), Gawain Yeo-Koh has demonstrated his commitment to the field. He has a solid educational background, wi
in Yeo-Koh is an ambitious and dedicated individual who is always pushing the envelope in search of novel and cutting-edge AI solutions. His expertise in AI and Data Engineering, coupled with
ower of AI and data. \n\nHere is a brief summary of the key points:\n\nGawain Yeo-Koh is a talented individual with expertise in AI and data engineering, possessing a range of skills such as
re and Robocup, showcasing his proficiency in mechatronics and robotics. His experience in the engineering product lifecycle management and artificial intelligence engineering projects has fu
ration and visualization through his analysis of customer behavior and market trends. With his determination and track record of success, Gawain Yeo-Koh is poised to make significant contribu
g operational efficiency and identifying opportunities to leverage CRM product offerings. He has also demonstrated his innovative mindset through his exploration and development of AI applica
solutions makes him a valuable asset to any organization looking to advance industries through the power of AI and data.<'
```

To ensure that cohere is able to recognise its purpose at the start, I started the prompt with:

“You are tasked to summarize a given resume in order to identify key attributes of the potential employee to give the company a concise and unbiased overview of the key attributes of a potential employee”

This was aimed to ensure that cohere understands its purpose

I then informed cohere of the inputs to expect, the processing it needs to conduct, as well as the output desired

Input:
“You will be provided a resume text, as well as named entities and phrases as input.”

Processing:
“You are to analyze the resume text to identify information such as education, skills, langauges, achievements, work experience and objectives. You are to then condense the information into a clear and concise summary for the company to read and have an overview of the potential of the person as an employee of the company”

Output:
“The output will be the summarized version of the resume”

I also gave cohere an example of a small portion of a resume that it might need to summarise. I also gave a target output by providing it information of what I expect from the summary.

One-shot learning:
“For example, a small portion of the resume may contain information such as: Contributed to the development of Large Language Models in AI Singapore, which increased performance by 50%.”

“The sample output should include information such as: He has substantially improved performance by 50% in AI Singapore with his contributions in enterprise-level AI initiatives in AI Singapore.”

However, cohere believed my experiences with modules in NYP were actual companies. Hence, I appended additional information to ensure it does not make the same mistake.
“Take note that experiences may include modules or courses, and not necessarily positions at companies"

To improve coheres ability to fully capture the important information from the resume, I gave it additional information regarding the phrase chunks and named entities in my resume, in order for it to fully capture those important information.

important_details:
"These are some key named entities, you should consider in your text summarisation to identify unique educational institutes, companies or locations: {ner_categories}\nThese are some key phrases you should consider in order to understand what the person did, and what were his key abilities: {phrases}"

9. Optimization

- BERT Fine-Tuning

```python
# try again with 35 epochs
# optimal for the BERT model class i created
optimal_max_len = 512
optimal_model_name = 'bert-base-uncased'

# optimal in Bert config
optimal_vocab_size = 4000
optimal_attention_heads = 12
#start timer
start_time = time.time()
X_train, X_test, y_train, y_test, cleaning_pipeline, processing_pipeline  = clean_and_process(
    documents_df,
    # for cleaning
    clean_encoding=True,
    clean_spaces=False,
    clean_spelling=True,
    clean_contraction=False,
    clean_camel_case=False,
    # for processing
    lower_case=True,
    drop_stop_words=True,
    stemming=False,
    lemmatization=True,
    # for the stratified split
    n_splits=5,
    test_size=0.2,
    random_state=42
)

# end processing timer
end_processing_time = time.time()

# calculate preprocessing time
processing_time = end_processing_time - start_time
print("Processing time:", processing_time, "seconds")

#start timer for training
start_training_time = time.time()

bert_config = BertConfig(
    vocab_size = optimal_vocab_size,
    hidden_size = 768,
    num_hidden_layers = 12,
    num_attention_heads = optimal_attention_heads,
    intermediate_size = 3072,
    hidden_act = "gelu",
    hidden_dropout_prob = 0.1,
    attention_probs_dropout_prob = 0.1,
    max_position_embeddings = 512,
    type_vocab_size = 2,
    initializer_range = 0.02,
    layer_norm_eps = 1e-12,
)

bert_model = BERT_Model(max_len=optimal_max_len, validation_size=0.2, batch_size=16, epochs=35, num_labels=len(np.unique(y_train)), lr=1.5e-5, model_name=optimal_model_name, lower_case=True, bert_config=bert_config, weight_decay_1=0.02, weight_decay_2=0.0)

bert_model.train_BERT(X_train, y_train)

# end training timer
end_training_time = time.time()

# calculate training time
training_time = end_training_time - start_training_time
print("Training time:", training_time, "seconds")

bert_model.test_BERT(X_test, y_test)
```

```
Processing time: 80.19900965690613 seconds
```

This is the optimized code for the BERT-Fine-Tuning (but not the best one).

The clean contractions function was disabled in order to reduce the processing time of the code by about 90s. This was because clean contractions was the longest function to run, taking about 1.5minutes to complete. Hence, it was disabled.

The epochs used to train BERT was changed from 50 to 35. This was to reduce the overfitting that may occur, as observed in the previous iterations of BERT fine tuning.

The weight decay 1 parameter for one of the weight decay parameters for adamW was increased from 0.01 to 0.02 as a measure to reduce the chances of overfitting.

- Roberta

```python
class ROBERTA_Model:
    def __init__(self, max_len, validation_size, batch_size, epochs, num_labels, lr=2e-5, model_name='roberta-base', lower_case=True, roberta_config=RobertaConfig(), weight_decay_1=0.01, weight_decay_2=0.0):
        self.max_len = max_len
        self.validation_size = validation_size
        self.batch_size = batch_size
        self.tokenizer = RobertaTokenizer.from_pretrained(model_name, do_lower_case=lower_case)
        self.num_labels = num_labels
        self.epochs = epochs
        self.lr = lr
        self.model_name = model_name
        self.lower_case = lower_case
        self.roberta_config = roberta_config
        self.weight_decay_1 = weight_decay_1
        self.weight_decay_2 = weight_decay_2

    def train_ROBERTA(self, X_train, y_train):
        data_frame = pd.concat([X_train, y_train], axis=1)
        data_frame.columns = ['sentence', 'label']
        # Prepare sentences and labels
        sentences = data_frame.sentence.values
        sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
        labels = data_frame.label.values

        # Tokenize sentences using roberta tokenizer
        tokenized_texts = [self.tokenizer.tokenize(sent) for sent in sentences]

        # Pad sequences and create attention masks
        MAX_LEN = self.max_len
        input_ids = [self.tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]
        input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long", truncating="post", padding="post")

        attention_masks = []
        for sequence in input_ids:
            sequence_mask = [float(id > 0) for id in sequence]
            attention_masks.append(sequence_mask)
        attention_masks = np.array(attention_masks)
        # Split data into training and validation sets
        sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
        for train_index, val_index in sss.split(input_ids, labels):
            training_inputs, validation_inputs = input_ids[train_index], input_ids[val_index]
            training_masks, validation_masks = attention_masks[train_index], attention_masks[val_index]
            training_labels, validation_labels = labels[train_index], labels[val_index]

        # Create DataLoader for training set
        batch_size = self.batch_size
        training_data = TensorDataset(torch.tensor(training_inputs), torch.tensor(training_masks), torch.tensor(training_labels))
        training_sampler = RandomSampler(training_data)
        training_dataloader = DataLoader(training_data, sampler=training_sampler, batch_size=batch_size)

        # Create DataLoader for validation set
        validation_data = TensorDataset(torch.tensor(validation_inputs), torch.tensor(validation_masks), torch.tensor(validation_labels))
        validation_sampler = SequentialSampler(validation_data)
        validation_dataloader = DataLoader(validation_data, sampler=validation_sampler, batch_size=batch_size)

        # Configure roberta model for sequence classification
        configuration = self.roberta_config
        self.model = RobertaModel(configuration)
        configuration = self.model.config

        self.model = RobertaForSequenceClassification.from_pretrained(self.model_name, num_labels=self.num_labels)
        self.model = nn.DataParallel(self.model)
        self.model.to(device)

        param_optimizer = list(self.model.named_parameters())
        no_decay = ['bias', 'LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_1},
            {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_2}
        ]
        optimizer = AdamW(optimizer_grouped_parameters, lr=self.lr, correct_bias=False)

        def flat_accuracy(predicted_labels, labels):
            predicted_labels = np.argmax(predicted_labels.to('cpu').numpy(), axis=1).flatten()
            labels = labels.to('cpu').numpy().flatten()
            return np.sum(predicted_labels == labels) / len(labels)

        # Train the roberta model
        epochs = self.epochs
        training_losses = []

        for epoch in trange(epochs, desc="Epoch"):
            self.model.train()
            training_loss = 0
            training_steps = 0

            for step, batch in enumerate(training_dataloader):
                inputs = batch[0].to(device)
                attention_masks = batch[1].to(device)
                labels = batch[2].to(device)

                optimizer.zero_grad()
                outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)
                loss = outputs.loss
                loss.backward()
                optimizer.step()

                training_loss += loss.item()
                training_steps += 1

                training_losses.append(loss.item())

            average_training_loss = training_loss/training_steps
            print("Epoch {}: Average Training Loss: {}".format(epoch+1, average_training_loss))

            self.model.eval()
            validation_accuracy = 0
            validation_steps = 0

            for batch in validation_dataloader:
                inputs = batch[0].to(device)
                attention_masks = batch[1].to(device)
                labels = batch[2].to(device)

                with torch.no_grad():
                    outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)

                logits = outputs.logits

                temp_validation_accuracy = flat_accuracy(logits, labels)
                validation_accuracy += temp_validation_accuracy
                validation_steps += 1

            average_validation_accuracy = validation_accuracy/validation_steps
            print("Epoch {}: Validation Accuracy: {}".format(epoch+1, average_validation_accuracy))
```

```python
def test_ROBERTA(self, X_test, y_test):
    data_frame = pd.concat([X_test, y_test], axis=1)
    data_frame.columns = ['sentence', 'label']
    sentences = data_frame.sentence.values
    sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
    labels = data_frame.label.values

    tokenized_texts = [self.tokenizer.tokenize(sent) for sent in sentences]

    input_ids = [self.tokenizer.convert_tokens_to_ids(tokens) for tokens in tokenized_texts]
    input_ids = pad_sequences(input_ids, maxlen=self.max_len, dtype="long", truncating="post", padding="post")

    attention_masks = [[float(i > 0) for i in seq] for seq in input_ids]

    input_ids = torch.tensor(input_ids)
    attention_masks = torch.tensor(attention_masks)
    labels = torch.tensor(labels)

    prediction_data = TensorDataset(input_ids, attention_masks, labels)
    prediction_dataloader = DataLoader(prediction_data, batch_size=self.batch_size)

    # Evaluate the roberta model on the out-of-domain dataset
    self.model.eval()
    logits_set = []
    labels_set = []

    for batch in prediction_dataloader:
        batch_input_ids, batch_attention_masks, batch_labels = batch
        batch_input_ids, batch_attention_masks, batch_labels = batch_input_ids.to(device), batch_attention_masks.to(device), batch_labels.to(device)

        with torch.no_grad():
            outputs = self.model(batch_input_ids, attention_mask=batch_attention_masks)
            logits = outputs.logits

        logits_set.append(logits.cpu().numpy())
        labels_set.append(batch_labels.cpu().numpy())

    from sklearn.metrics import matthews_corrcoef, accuracy_score
    matthews_set = []
    accuracy_set = []
    # Calculate Matthews correlation coefficient for each batch
    for i in range(len(labels_set)):
        mcc = matthews_corrcoef(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
        acc = accuracy_score(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
        matthews_set.append(mcc)
        accuracy_set.append(acc)

    for i, mcc in enumerate(matthews_set):
        print(f"Batch {i + 1}: MCC = {mcc}")
        print(f"Batch {i + 1}: Accuracy = {accuracy_set[i]}")

    # Calculate the overall Matthews correlation coefficient
    overall_mcc = np.mean(matthews_set)
    overall_acc = np.mean(accuracy_set)
    print(f"\nOverall MCC: {overall_mcc}")
    print(f"\nOverall Accuracy: {overall_acc}")
```

```python
# use similar params for roberta model

# optimal params for the roberta model class
optimal_max_len = 512
optimal_model_name = 'roberta-base'

# optimal params for Roberta config
optimal_vocab_size = 4000
optimal_attention_heads = 8

#start timer
start_time = time.time()
X_train, X_test, y_train, y_test, cleaning_pipeline, processing_pipeline  = clean_and_process(
    documents_df,
    # for cleaning
    clean_encoding=True,
    clean_spaces=False,
    clean_spelling=True,
    clean_contraction=True,
    clean_camel_case=False,
    # for processing
    lower_case=True,
    drop_stop_words=True,
    stemming=False,
    lemmatization=True,
    # for the stratified split
    n_splits=5,
    test_size=0.2,
    random_state=42
)

# end processing timer
end_processing_time = time.time()

# calculate preprocessing time
processing_time = end_processing_time - start_time
print("Processing time:", processing_time, "seconds")

#start timer for training
start_training_time = time.time()

roberta_config = RobertaConfig(
    vocab_size = optimal_vocab_size,
    hidden_size = 768,
    num_hidden_layers = 12,
    num_attention_heads = optimal_attention_heads,
    intermediate_size = 3072,
    hidden_act = "gelu",
    hidden_dropout_prob = 0.1,
    attention_probs_dropout_prob = 0.1,
    max_position_embeddings = 512,
    type_vocab_size = 2,
    initializer_range = 0.02,
    layer_norm_eps = 1e-12,
)

roberta_model = ROBERTA_Model(max_len=optimal_max_len, validation_size=0.2, batch_size=16, epochs=50, num_labels=len(np.unique(y_train)), lr=1.5e-5, model_name=optimal_model_name, lower_case=True, roberta_config=roberta_config)

roberta_model.train_ROBERTA(X_train, y_train)

# end training timer
end_training_time = time.time()

# calculate training time
training_time = end_training_time - start_training_time
print("Training time:", training_time, "seconds")

roberta_model.test_ROBERTA(X_test, y_test)
```

```
Epoch:  96%|██████████ | 23/24 [04:52<00:12, 12.69s/it]Epoch 23: Validation Accuracy: 0.8354166666666667
Epoch 24: Average Training Loss: 0.12620799709111452
Epoch: 100%|██████████| 24/24 [05:05<00:00, 12.72s/it]Epoch 24: Validation Accuracy: 0.8354166666666667
Training time: 314.5214624404907 seconds

Batch 1: MCC = 0.7426226443941474
Batch 1: Accuracy = 0.75
Batch 2: MCC = 0.3608832094529943
Batch 2: Accuracy = 0.375
Batch 3: MCC = 0.675
Batch 3: Accuracy = 0.7142857142857143

Overall MCC: 0.5928352846157139

Overall Accuracy: 0.6130952380952381
```

I will try to optimize the resume classification solution using Roberta. Roberta was trained on a larger, more diverse corpus of data, so it should be able capture a larger spectrum of language nuance. It also trains with dynamic masking, so it will be more adaptable to different contexts because of varied word mask positions. Roberta also was trained with longer sequences, making it more capable of getting contextual dependencies in the data. Lastly, it has a streamlined training process due to it not having Next Sentence Prediction, which makes it more optimized for learning contextual information from individual sentences. Because of these reasons, I tried to optimize the resume classification solution by using and fine tuning Roberta.

- GPT2

```python
class GPT2_Model:
    def __init__(self, model_name='gpt2', max_length=60, label_ids=None, epochs=4, batch_size=16, lr=2e-5, weight_decay_1=0.01, weight_decay_2=0):
        self.model_name = model_name
        self.max_length = max_length
        self.label_ids = label_ids
        self.epochs = epochs
        self.batch_size = batch_size
        self.lr = lr
        self.weight_decay_1 = weight_decay_1
        self.weight_decay_2 = weight_decay_2
        self.tokenizer = GPT2Tokenizer.from_pretrained(pretrained_model_name_or_path=model_name)
        self.tokenizer.padding_side = "left"
        self.tokenizer.pad_token = self.tokenizer.eos_token
        self.device = device

    def train_gpt2(self, X_train, y_train):

        self.model_configuration = GPT2Config.from_pretrained(pretrained_model_name_or_path=model_name, num_labels=len(label_ids))
        self.model = GPT2ForSequenceClassification.from_pretrained(pretrained_model_name_or_path=model_name, config=self.model_configuration)
        self.model.resize_token_embeddings(len(self.tokenizer))
        self.model.config.pad_token_id = self.model.config.eos_token_id
        self.model.to(self.device)

        # Prepare sentences and labels
        sentences = ["[CLS] " + sentence + " [SEP]" for sentence in X_train]
        labels = y_train.values

        # Tokenize sentences using the gpt2 tokenizer
        tokenized_texts = [self.tokenizer.tokenize(sent) for sent in sentences]

        # Pad sequences and create attention masks
        MAX_LEN = self.max_length
        input_ids = [self.tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]
        input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long", truncating="post", padding="post")

        attention_masks = []
        for sequence in input_ids:
            sequence_mask = [float(id > 0) for id in sequence]
            attention_masks.append(sequence_mask)
        attention_masks = np.array(attention_masks)

        # split the data into training and validation sets
        sss = StratifiedShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
        for train_index, val_index in sss.split(input_ids, labels):
            training_inputs, validation_inputs = input_ids[train_index], input_ids[val_index]
            training_masks, validation_masks = attention_masks[train_index], attention_masks[val_index]
            training_labels, validation_labels = labels[train_index], labels[val_index]

        # gpt2_classification_collator = Gpt2ClassificationCollator(tokenizer=self.tokenizer, label_ids=self.label_ids, max_sequence_length=self.max_length)

        training_dataset = TensorDataset(torch.tensor(training_inputs), torch.tensor(training_masks), torch.tensor(training_labels))
        training_sampler = RandomSampler(training_dataset)
        training_dataloader = DataLoader(training_dataset, sampler=training_sampler, batch_size=batch_size)

        validation_dataset = TensorDataset(torch.tensor(validation_inputs), torch.tensor(validation_masks), torch.tensor(validation_labels))
        validation_sampler = SequentialSampler(validation_dataset)
        validation_dataloader = DataLoader(validation_dataset, sampler=validation_sampler, batch_size=batch_size)

        param_optimizer = list(self.model.named_parameters())
        no_decay = ['bias', 'LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in param_optimizer if not any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_1},
            {'params': [p for n, p in param_optimizer if any(nd in n for nd in no_decay)], 'weight_decay': self.weight_decay_2}
        ]
        optimizer = AdamW(optimizer_grouped_parameters, lr=self.lr, correct_bias=False)


        def flat_accuracy(predicted_labels, labels):
            predicted_labels = np.argmax(predicted_labels.to('cpu').numpy(), axis=1).flatten()
            labels = labels.to('cpu').numpy().flatten()
            return np.sum(predicted_labels == labels) / len(labels)

        epochs = self.epochs
        training_losses = []
```

```python
        for epoch in trange(epochs, desc="Epoch"):
            self.model.train()
            training_loss = 0
            training_steps = 0

            for step, batch in enumerate(training_dataloader):
                inputs = batch[0].to(device)
                attention_masks = batch[1].to(device)
                labels = batch[2].to(device)

                optimizer.zero_grad()
                outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)
                loss = outputs.loss
                loss.backward()
                optimizer.step()

                training_loss += loss.item()
                training_steps += 1

                training_losses.append(loss.item())

            average_training_loss = training_loss/training_steps
            print("Epoch {}: Average Training Loss: {}".format(epoch+1, average_training_loss))

            self.model.eval()
            validation_accuracy = 0
            validation_steps = 0

            for batch in validation_dataloader:
                inputs = batch[0].to(device)
                attention_masks = batch[1].to(device)
                labels = batch[2].to(device)

                with torch.no_grad():
                    outputs = self.model(inputs, attention_mask=attention_masks, labels=labels)

                logits = outputs.logits

                temp_validation_accuracy = flat_accuracy(logits, labels)
                validation_accuracy += temp_validation_accuracy
                validation_steps += 1

            average_validation_accuracy = validation_accuracy/validation_steps
            print("Epoch {}: Validation Accuracy: {}".format(epoch+1, average_validation_accuracy))
```

```python
def test_gpt2(self, X_test, y_test):
    # Prepare sentences and labels
    sentences = ["[CLS] " + sentence + " [SEP]" for sentence in X_test]
    labels = y_test.values

    tokenized_texts = [self.tokenizer.tokenize(sent) for sent in sentences]

    input_ids = [self.tokenizer.convert_tokens_to_ids(tokens) for tokens in tokenized_texts]
    input_ids = pad_sequences(input_ids, maxlen=max_length, dtype="long", truncating="post", padding="post")

    attention_masks = [[float(i > 0) for i in seq] for seq in input_ids]

    input_ids = torch.tensor(input_ids)
    attention_masks = torch.tensor(attention_masks)
    labels = torch.tensor(labels)

    prediction_data = TensorDataset(input_ids, attention_masks, labels)
    prediction_dataloader = DataLoader(prediction_data, batch_size=self.batch_size)

    # Evaluate the gpt2 model on the out-of-domain dataset
    self.model.eval()
    logits_set = []
    labels_set = []

    for batch in prediction_dataloader:
        batch_input_ids, batch_attention_masks, batch_labels = batch
        batch_input_ids, batch_attention_masks, batch_labels = batch_input_ids.to(device), batch_attention_masks.to(device), batch_labels.to(device)

        with torch.no_grad():
            outputs = self.model(batch_input_ids, attention_mask=batch_attention_masks)
            logits = outputs.logits

        logits_set.append(logits.cpu().numpy())
        labels_set.append(batch_labels.cpu().numpy())

    from sklearn.metrics import matthews_corrcoef, accuracy_score
    matthews_set = []
    accuracy_set = []
    # Calculate Matthews correlation coefficient for each batch
    for i in range(len(labels_set)):
        mcc = matthews_corrcoef(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
        acc = accuracy_score(labels_set[i], np.argmax(logits_set[i], axis=1).flatten())
        matthews_set.append(mcc)
        accuracy_set.append(acc)

    for i, mcc in enumerate(matthews_set):
        print(f"Batch {i + 1}: MCC = {mcc}")
        print(f"Batch {i + 1}: Accuracy = {accuracy_set[i]}")

    # Calculate the overall Matthews correlation coefficient
    overall_mcc = np.mean(matthews_set)
    overall_acc = np.mean(accuracy_set)
    print(f"\nOverall MCC: {overall_mcc}")
    print(f"\nOverall Accuracy: {overall_acc}")
```

```python
# conduct similar preprocessing that was deemed most optimal

#start timer
start_time = time.time()
X_train, X_test, y_train, y_test, cleaning_pipeline, processing_pipeline  = clean_and_process(
    documents_df,
    # for cleaning
    clean_encoding=True,
    clean_spaces=False,
    clean_spelling=True,
    clean_contraction=True,
    clean_camel_case=False,
    # for processing
    lower_case=True,
    drop_stop_words=True,
    stemming=False,
    lemmatization=True,
    # for the stratified split
    n_splits=5,
    test_size=0.2,
    random_state=42
)

# end processing timer
end_processing_time = time.time()

# calculate preprocessing time
processing_time = end_processing_time - start_time
print("Processing time:", processing_time, "seconds")
```

```
Processing time: 95.46746277889143 seconds
```

```python
categories = processing_pipeline.categories
category_encodings = processing_pipeline.category_encodings

set_seed(123)
batch_size = 16
epochs = 50

model_name = 'gpt2'
max_length = 256 # reduced max len as it takes up too much memory
label_ids = [{category: encoding} for category, encoding in zip(categories, category_encodings)]

#start timer
start_time = time.time()

gpt2_model = GPT2_Model(model_name=model_name, max_length=max_length, label_ids=label_ids, epochs=epochs, batch_size=batch_size, lr=1.5e-5)

gpt2_model.train_gpt2(X_train, y_train)

# end training timer
end_training_time = time.time()

gpt2_model.test_gpt2(X_test, y_test)

# calculate training time
training_time = end_training_time - start_time
print("Processing time:", training_time, "seconds")
```

```
Epoch 49: Average Training Loss: 0.01134278968675062
Epoch:  98%|██████████ | 49/50 [05:29<00:06,  6.87s/it]Epoch 49: Validation Accuracy: 0.32291666666666663
Epoch 50: Average Training Loss: 0.01049671252258122
Epoch: 100%|██████████| 50/50 [05:36<00:00,  6.73s/it]Epoch 50: Validation Accuracy: 0.35625

Batch 1: MCC = 0.06495028154709068
Batch 1: Accuracy = 0.125
Batch 2: MCC = 0.2230593437233266
Batch 2: Accuracy = 0.25
Batch 3: MCC = 0.45
Batch 3: Accuracy = 0.42857142857142855

Overall MCC: 0.24600320842347245

Overall Accuracy: 0.26785714285714285
Processing time: 340.61499667167664 seconds
```

GPT2 has 117M parameters whereas BERT Base has 110M parameters. GPT2 may be able to perform better than bert base, while also not overfit like BERT Large. I will use it to try and observe if it can perform better than, or faster than bert base. From the observed accuracy and time, GPT2 did not surpass bert base despite having more parameters. GPT2 had a lower testing MCC and accuracy of 0.246 and 0.268 respectively. Bert base had a higher testing MCC and accuracy of 0.652 and 0.675 respectively. However, GPT2 did finish training at a faster time(340s) than bert base(652s) with similar configurations.

- Resume Search Preprocessing

```python
# preprocess the original documents df first in order to remove the dupes as well as conduct preprocessing in a way that may improve the resume vectorization and sea
# clean_contraction wont be used here as it takes too long to perform and will likely not improve performance as much (OPTIMIZATION)

#start timer
start_time = time.time()
_, _, _, _, _, processing_pipeline  = clean_and_process(
    documents_df,
    # for cleaning
    clean_encoding=True,
    clean_spaces=False,
    clean_spelling=True,
    clean_contraction=False,
    clean_camel_case=False,
    clean_punctuation=False,
    clean_non_ascii=False,
    # for processing
    lower_case=True,
    drop_stop_words=True,
    stemming=False,
    lemmatization=True,
    # for the stratified split
    n_splits=5,
    test_size=0.2,
    random_state=42
)

# end timer
end_time = time.time()

# calculate preprocessing time
processing_time = end_time - start_time

print("\nPreprocessing time:", processing_time, "seconds\n")

processed_resume_df = processing_pipeline.processed_resumes_df
category_mapping = processing_pipeline.categories # get the category mapping in order to revert the label encoding that was done on the category column

resume_vectorizor = Vectorization(processed_resume_df, category_mapping)
```

```
Preprocessing time: 7.196032285690308 seconds
```

Disabled clean contractions to reduce processing time by about 90s

Included the use of clean encoding and clean spelling to normalize the text further before actually vectorizing them into document vectors.