



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

COMPUTER SCIENCE FOR AUTONOMOUS SYSTEMS

Call center data classification using NLP algorithm with blockchain-based rewarding smart contract

Author:

Letiushev Nikita

Computer Science for Autonomous Systems MSc

Internal supervisor:

Németh Zsolt

Associate professor, PhD

External supervisor:

Brunner Tibor

Software Developer

Budapest, 2023

Contents

1	Introduction	2
2	Development documentation	3
2.1	Data pre-processing	3
2.1.1	Simple techniques	3
2.1.2	Advanced techniques	4
2.2	Machine Learning	8
2.2.1	Linear SVM with TF-IDF Vectorizer	8
2.2.2	BERT model	10
2.3	Lottery	25
2.3.1	Lottery implementation	25
2.3.2	Tests conducting	27
2.3.3	Deployment script	28
3	User documentation	30
3.1	Category classification	31
3.2	Lottery execution	35
4	Conclusion	40
	Acknowledgements	41
A	Development workflow	42
	Bibliography	43
	List of Figures	44
	List of Codes	45

Chapter 1

Introduction

This project's main idea is to automatize a company's call center. The customer writes his concern to the hotline and describes the issue, this information is passed through our algorithm, and the output of this algorithm is the potential category of the particular case, this case will be forwarded to the corresponding company department (e.g., to the IT or Marketing department). Example of a simple demo: input text about some advertising offer to the company should be properly classified and sent to the Marketing department.

As training data is used the "190k Medium Articles" dataset from Kaggle which is a scraping process from the Medium website, looking for published articles.

The general structure of this paper is the following: Development documentation and User documentation

In the Development documentation, there is a description of the research process: data pre-processing part[1], Machine Learning algorithm implementation (Linear SVM with TF-IDF Vectorizer), and BERT language model implementation[2]. In addition, smart contract lottery implementation[3] is also described here.

User documentation has an end-user guide and description of how the sample demo works. However, the user documentation defines some helping functions, which implies that the end user should be familiar with basic Python code and patterns.

Chapter 2

Development documentation

In this chapter there is documentation about steps done which are used to create a stable and maintainable language model classifier, the development workflow can be seen in Figure A.1.

2.1 Data pre-processing

2.1.1 Simple techniques

For data manipulation Python csv and pandas libraries are used. The input dataset is loaded as a Python variable using csv library. The issue with the dataset is the following: it has around 20 000 unique elements in column "tags" therefore we need to decrease its number to 6 main categories. For this, we will convert all tags into a list, and from this list, we will get 100 most common tags. In the next step, we will select the 6 main ones by hand and with common sense that they are the most separated. As a result, we have a list of main tags which look like this: *['programming', 'business', 'health', 'marketing', 'politics', 'sports']*.

Now the idea is to run a simple Bayes Classifier from the sklearn Python library on the "tags" column and predict the most common tag for each element. For that, we are creating training and prediction datasets. From the dataset we select records which has any of the main tags which were selected previously and add these records to training data. If the record doesn't have main tags in it we add it to the prediction dataset. As a result, we have 21835 records that are selected as training data, and 170533 records that are selected as prediction data.

Next, we will run the *Naive Bayes algorithm* on the training data with *TfidfVectorizer* as a vectorizer, to increase the accuracy of the approach Code 3.8:

```
1 X_train = [' '.join(row['tags']) for row in training_data]
2 Y_train = [list(set(row['tags']) & set(main_tags))[0] for row in
              training_data]
3
4 vectorizer = TfidfVectorizer()
5 X_train_tfidf = vectorizer.fit_transform(X_train)
6
7 clf = MultinomialNB().fit(X_train_tfidf, Y_train)
```

Code 2.1: Naive Bayes Classifier

Now we are loading the saved classifier and running it to the whole dataset. The idea is the following: every element has a "tags" column, now we run Bayes Classifier on this column. In the input, it has a list of tags, in the output it predicts the most possible main tag from the main tags list. We are saving the result into a new dataset. Now we have a main tag assigned for each element, we will remove unnecessary columns ['title', 'url', 'authors', 'timestamp', 'tags'] as they are not needed for Machine Learning experiments, switch places columns "main tags" and "text" and rename "main tags" into "category" as this is going to be our target category for Machine Learning

2.1.2 Advanced techniques

From the previous steps we have the dataset ready for conducting Machine Learning experiments, but the issue is that ML models do not work with words, they are converting words into their numerical representation, therefore we need to apply some advanced pre-processing techniques before we convert words in the dataset into numbers.

A good start will be checking for empty values in both columns, after implementation the result "False" indicates that there are no missing or empty values exist in the dataset. The histogram of character length distribution will show us how big elements in column "text" are: Figure 2.1:

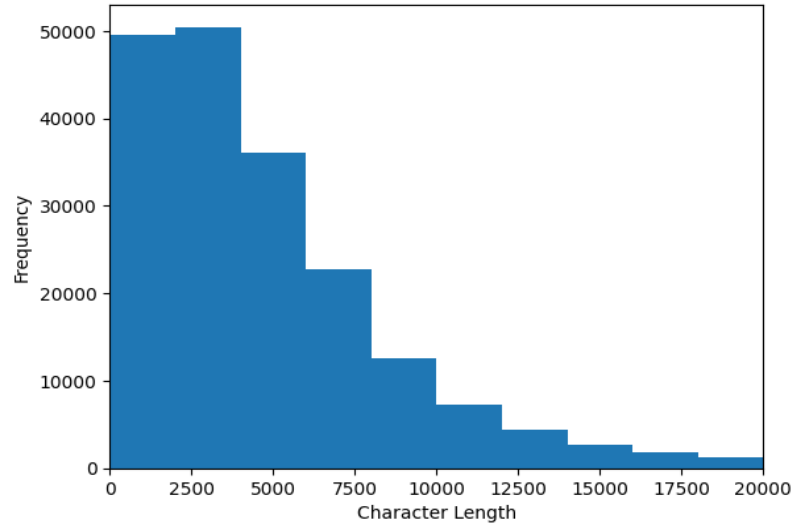


Figure 2.1: Character length distribution

This diagram shows us that a lot of elements have from 0 to 5000-7500 characters.

Also, it will be good to study the distribution of items among categories in Figure 2.2:

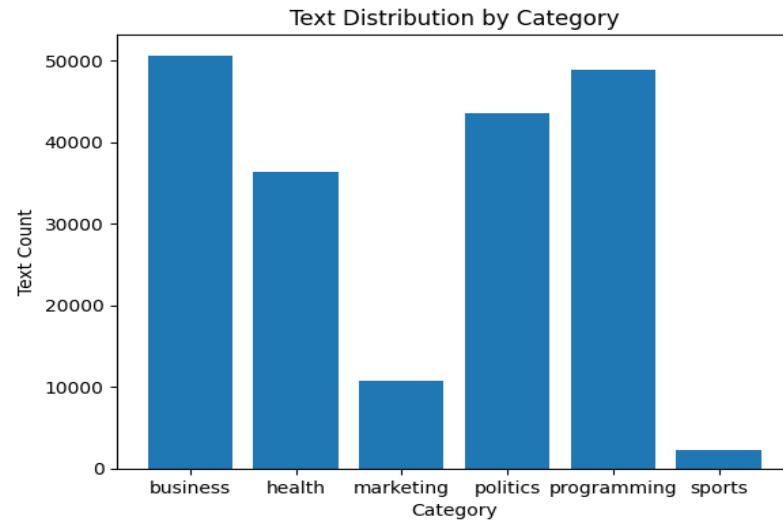


Figure 2.2: Text distribution by category

It shows us that each category is represented unequally, this is a bad situation, consequences of which we could see later, therefore we will sample the dataset in a way to have an almost equal representation of each category. In addition, we will convert categories into their numerical representation as a position in the list of main tags. The result of these steps we can see in Figure 2.3:

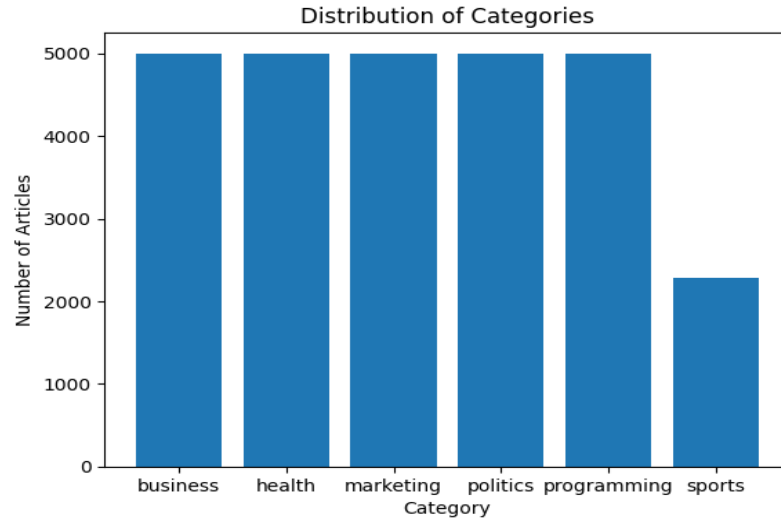


Figure 2.3: Text distribution by category after sampling

Now we should tokenize the "text" column. Tokenize means breaking down a sentence or a paragraph into individual words, phrases, or other meaningful elements, which are referred to as "tokens". For that we load the csv dataset into a list of dictionaries, where each dictionary represents a row in the CSV file and loop through each row in the data list, convert the text column to lowercase, tokenize the text, and save the changes. As a result, we will have a new column with tokenized text, in addition, we should study the length distribution of words in this dataset to get the complexity of the task: Figure 2.4:

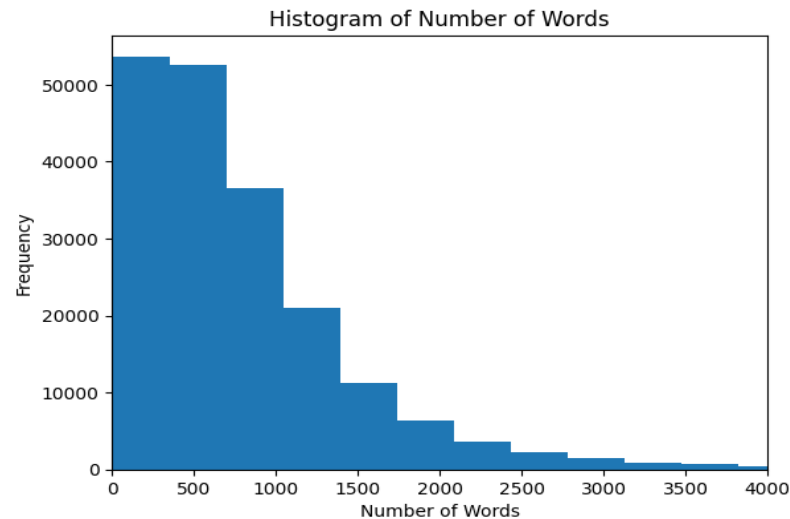


Figure 2.4: Histogram of number of words

This histogram shows us that a lot of elements have from 0 to 1000 words.

It seems like the most of items have lengths from 0 to 1500 words, which is good, as there are no very big items.

Now we would like to study the distribution of unique values of tokens, to see if there are additional pre-processing steps needed, Figure 2.5:

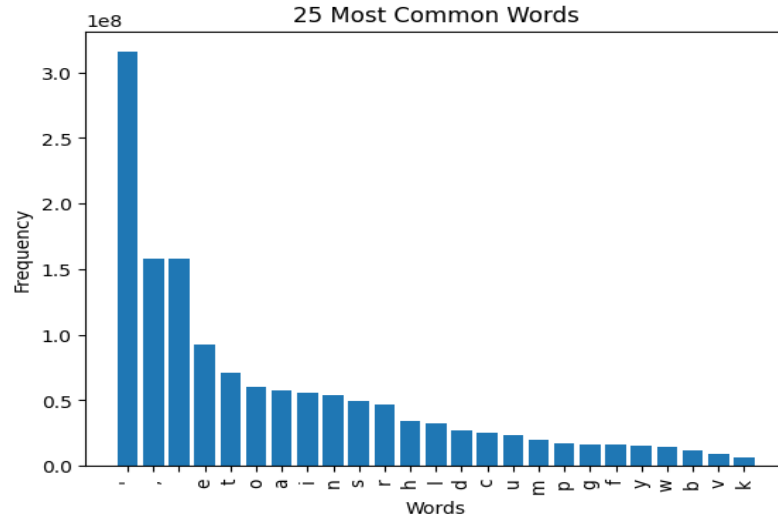


Figure 2.5: 25 most common words

shows us that additional pre-processing steps are needed, because the most common words are meaningless in understanding the context of the item. Therefore, we want to remove meaningless words like commas, single characters, etc. from our tokenized text, we can use NLTK's built-in stopwords list and a list comprehension to filter out the unwanted tokens. The result of this step is shown in Figure 2.6:

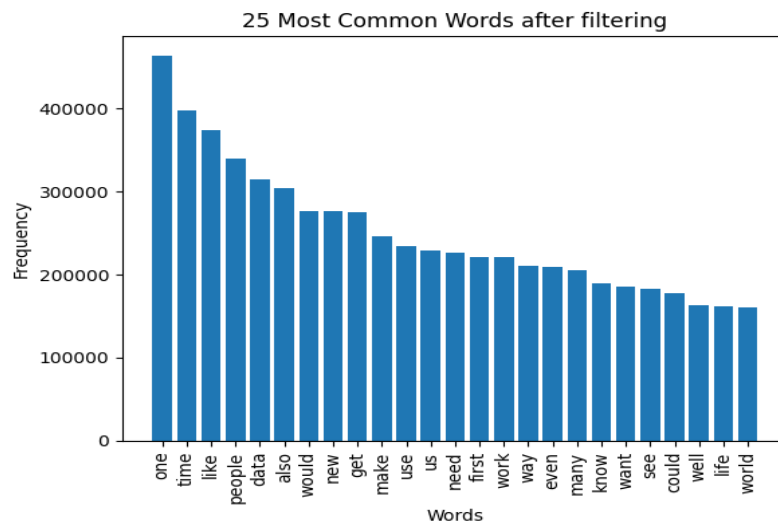


Figure 2.6: 25 most common words after filtering

In addition, we should apply the *porter/stemmer* tool from the NLTK library. Porter/Stemmer tool is a tool used in natural language processing to transform words into their root form. It removes suffixes and prefixes from a word, leaving only the root, or base of the word. This tool helps reduce the number of unique words in a dataset, making it easier to analyze and identify patterns. After implementation, the example of newly stemmed data is the following: *['mind', 'nose', 'smell', 'train', 'chang', 'brain']*, as we can see most of the words are converged to their root form. The final step here is to remove unnecessary columns "text" and "tokenized text" and save the changes into a final dataset which we will use for Machine Learning experiments

2.2 Machine Learning

Machine Learning involves training algorithms to learn patterns and relationships from data, so that they can make predictions or decisions based on new, unseen data. Since this is a classification problem we will consider two techniques to learn our model to make predictions: *Linear SVM* and *BERT language model*.

2.2.1 Linear SVM with TF-IDF Vectorizer

Linear SVM with TF-IDF Vectorizer is a machine learning model that uses Support Vector Machine (SVM) algorithm to classify text documents based on their content. The input data is represented as a matrix of TF-IDF values, which is a statistical measure of how important a word is to a document in a collection or corpus of documents.

The TF-IDF Vectorizer transforms the raw text data into a matrix of TF-IDF features. The SVM algorithm then trains on this matrix to learn the patterns and relationships between the input features and output labels. The Linear SVM is a variant of SVM that uses a linear decision boundary to separate the different classes.

At this step, we split input data using the *train_test_split* function from the scikit-learn library. The "stemmed text" column is the input data and the "category" column is the target variable that needs to be predicted. We also specify the fraction of the data that should be used for testing and apply a random number generator to ensure reproducibility. We vectorize the text data using the Term Frequency-Inverse

Document Frequency (TF-IDF) algorithm with the `TfidfVectorizer` function. This algorithm converts the raw text into a numerical representation. Next, we fit the `TfidfVectorizer` on the training data using the `fit_transform` method to transform the training text data into a matrix of TF-IDF features. Then we use the `transform` method to transform the test data into a matrix of TF-IDF features. We train a Linear Support Vector Machine (SVM) classifier model on the transformed training data using the `LinearSVC` function from `scikit-learn`. The model is fitted to the target variable `y_train` and the matrix of TF-IDF features `X_train`. Finally, the trained model is used to predict the categories of the test data (`X_test`) using the `predict` method, and the accuracy of the model is evaluated using the `accuracy_score` function from `scikit-learn`, Code 3.8:

```
1 X_train, X_test, y_train, y_test = train_test_split(df['  
    stemmed_text'], df['category'], test_size=0.2, random_state=42)  
2  
3 vectorizer = TfidfVectorizer()  
4 X_train = vectorizer.fit_transform(X_train)  
5 X_test = vectorizer.transform(X_test)  
6  
7 svm = LinearSVC()  
8 svm.fit(X_train, y_train)  
9  
10 y_pred = svm.predict(X_test)
```

Code 2.2: Linear SMV with `TfidfVectorizer`

Now we have an accuracy of around 0.69 which is not bad, but still, it will be possible to increase the accuracy using more advanced models.

The confusion matrix will show us more understandable results. A confusion matrix is a table used to evaluate the performance of a classification model, which shows the number of true positives, true negatives, false positives, and false negatives predicted by the model. It allows us to see how well the model is performing in terms of correctly or incorrectly predicting the class labels of a dataset. The matrix has two dimensions, where the rows represent the actual values and the columns represent the predicted values. The matrix for Linear SVM is shown in Figure 2.7:

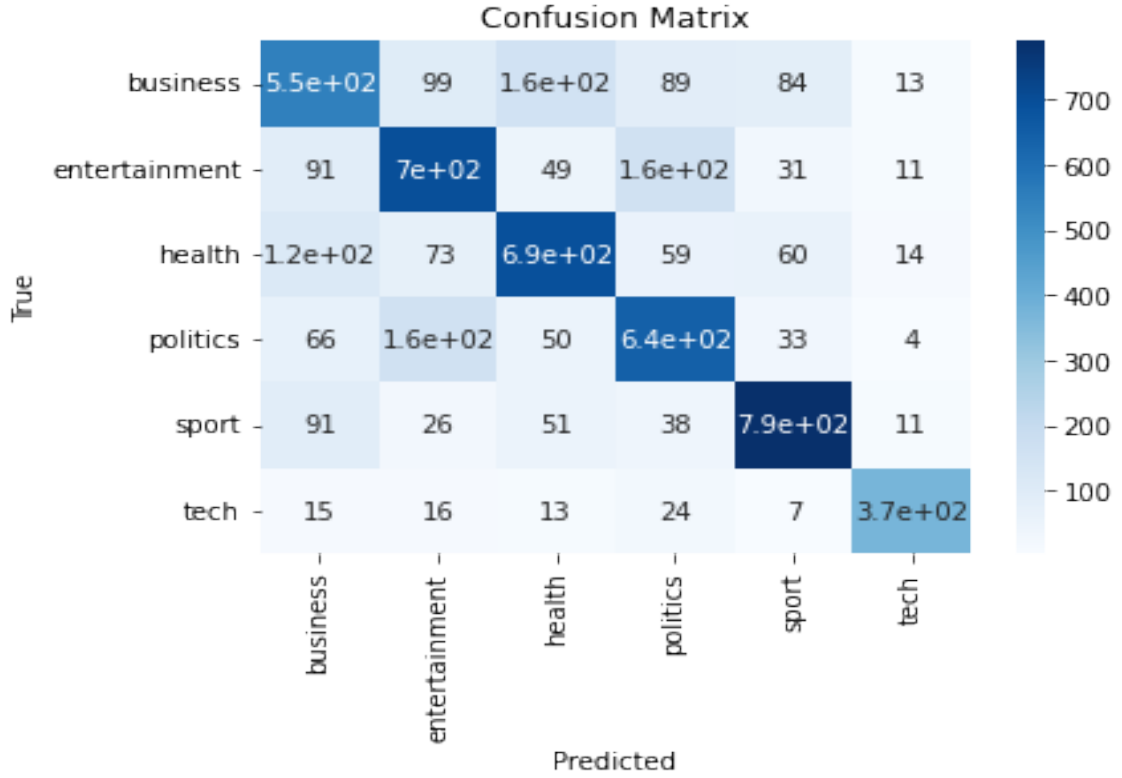


Figure 2.7: Confusion matrix for Linear SVM

As we can see the results for the tech category seems not as intensive as for other categories, however, they are quite accurate if we study distribution among tech => other classes relation. On the other hand, there are some difficulties for our model to distinguish between business, entertainment, and politics categories. And it is understandable because these categories are close to each other in terms of context.

2.2.2 BERT model

BERT (Bidirectional Encoder Representations from Transformers) is a language model developed by Google that uses a deep neural network architecture known as Transformer, which can learn from the context of words in a sentence by processing them in both directions. This bidirectional processing allows BERT to capture the meaning of words in a more accurate and context-specific way. It has been pre-trained on massive amounts of text data, and its pre-trained weights can be fine-tuned on specific downstream tasks with comparatively small amounts of labeled data.

As a good start, we convert the stemmed text column which contains strings represented as lists (e.g. "['word1', 'word2', 'word3']"), into space-separated strings (e.g. "word1 word2 word3").

As a learning template, we will use the pre-trained BERT model *"bert-base-uncased"* with its tokenizer. As the BERT model works with fixed item length we will set the maximal length to 300 words. The next global variable is the batch size. Batch size refers to the number of training examples utilized in one iteration of the gradient descent algorithm. In other words, it is the number of samples processed in a single forward/backward pass of a neural network during training.

A smaller batch size means that the model is trained on fewer samples at once, which can result in slower training but can be beneficial for fine-tuning the model. A larger batch size means that the model is trained on more samples at once, which can result in faster training but may also lead to overfitting.

The appropriate batch size for a specific problem depends on various factors, such as the size of the dataset, the complexity of the model, the available computing resources, and the desired accuracy.

We will set the batch size to 16 as it is suggested in BERT documentation.

The GPReviewDataset class, Code 3.8:

```
1 class GPReviewDataset(Dataset):
2
3     def __init__(self, reviews, targets, tokenizer, max_len):
4         self.reviews = reviews
5         self.targets = targets
6         self.tokenizer = tokenizer
7         self.max_len = max_len
8
9     def __len__(self):
10         return len(self.reviews)
11
12     def __getitem__(self, item):
13         review = str(self.reviews[item])
14         target = self.targets[item]
15
16         encoding = self.tokenizer.encode_plus(
17             review,
18             add_special_tokens=True,
19             max_length=self.max_len,
20             truncation=True,
21             return_token_type_ids=False,
22             padding='max_length',
23             return_attention_mask=True,
```

```
24     return_tensors='pt',
25 )
26
27 return {
28     'review_text': review,
29     'input_ids': encoding['input_ids'].flatten(),
30     'attention_mask': encoding['attention_mask'].flatten(),
31     'targets': torch.tensor(target, dtype=torch.long)
32 }
```

Code 2.3: GPReviewDataset class

defines a PyTorch dataset class for category analysis of text reviews. It takes in four parameters:

- *reviews*: a list of text reviews.
- *targets*: a list of labels indicating the class of each review.
- *tokenizer*: a tokenization function from the Hugging Face Transformers library
- *max_len*: the maximum length of a review (in tokens) to be considered

The *len* method returns the number of reviews in the dataset, and the *getitem* method returns a dictionary containing the review text, the input IDs (token IDs) of the encoded review, the attention mask (indicating which tokens to pay attention to), and the class label (converted to a PyTorch tensor).

The tokenizer is used to encode each review into a sequence of tokens that can be fed into a neural network model. The resulting encoded sequence is then padded or truncated to max len tokens and returned as a PyTorch tensor. The attention mask is also generated to indicate which tokens should be attended to by the model, and the sentiment label is converted to a PyTorch tensor for use in training the model.

Next, we split the input data frame into three smaller data frames: train, validation, and test.

The *train_test_split* function from the Scikit-learn library is used to split the df data frame into two smaller data frames: *df_train* and *df_test*. The *test_size* parameter is set to 0.1, which means that 10% of the original data will be used for testing and the remaining 90% will be used for training.

The *train_test_split* function is used again, this time on the *df_test* data frame created in the previous step. It splits *df_test* into two smaller data frames: *df_val* and *df_test*, with each of them containing 50% of the data that was not included in the *df_train* data frame.

A function *create_data_loader* takes a Pandas DataFrame *df*, a tokenizer, a maximum length *max_len*, and a batch size *batch_size* as input parameters. The function first creates a *GPReviewDataset* instance *ds* using the *stemmed_text* and *category_id* columns of the DataFrame, the tokenizer, and the maximum length. Then it returns a PyTorch *DataLoader* that loads data from the *GPReviewDataset* instance in batches of size *batch_size*, using four worker processes for parallelization.

In the next steps three data loaders (*train_data_loader*, *val_data_loader*, and *test_data_loader*) are created which are using the *create_data_loader* function, which takes the data, tokenizer, *max_len*, and *batch_size* as inputs. The *create_data_loader* function creates a *GPReviewDataset* object with the *stemmed_text* and *category_id* columns from the input data, and the tokenizer, *max_len*, and *batch_size*. It then returns a *DataLoader* object with the *GPReviewDataset* object, *batch_size*, and *num_workers*.

Later on, the *next()* function is used to get the first batch of data from the *train_data_loader* and stores it in the data variable. The data variable is a dictionary with four keys: *'review_text'*, *'input_ids'*, *'attention_mask'*, and *'targets'*. These keys correspond to the four attributes of the *GPReviewDataset* object: the raw review text, the tokenized and encoded input sequence, the attention mask, and the target category ID, respectively. The *keys()* function is used to print out the keys of the data dictionary.

We are checking if the matrices have the correct size (16 by 300 for *input_ids* and *attention_mask* and 16 by 1 for *targets*)

Finally, we load the BERT model with a pre-trained model name, which was "bert-base-uncased" in our case.

The *SentimentClassifier* class inherits from the *nn.Module* class in PyTorch consists of several layers. The *BertModel* layer loads the pre-trained BERT model specified by the *PRE_TRAINED_MODEL_NAME* variable. The drop layer applies dropout regularization to the outputs of the BERT layer. Finally, the out layer is a linear layer that maps the output of the dropout layer to the number of classes in the classification task.

The *forward* method defines the forward pass through the layers of the network. It takes `input_ids` and `attention_mask` as input, which are inputs to the pre-trained BERT model. The `BertModel` layer returns various outputs, but here we use the `pooler_output` tensor. The drop layer applies dropout regularization to the `pooler_output`, and the resulting tensor is passed through the out layer to produce the final output of the network.

Also, we create an instance of the `SentimentClassifier` class with 6 output classes and then move the model to the specified device (GPU in our case) to perform computation.

We move the `'input_ids'` and `'attention_mask'` tensors from the CPU to the specified device (GPU) for faster computation during the model training or prediction. The `'input_ids'` tensor contains the encoded review texts in a sequence of numerical token IDs, and the `'attention_mask'` tensor is used to tell the model which tokens to attend to and which ones to ignore.

The next part applies the model to the `input_ids` and `attention_mask` tensors, which represent a batch of input sequences encoded by a BERT tokenizer. The resulting output is passed through a softmax activation function, Code 3.8:

```
1 tensor([[0.1696, 0.0989, 0.1550, 0.1328, 0.3374, 0.1063],
2         [0.1855, 0.1006, 0.2009, 0.1225, 0.2554, 0.1352],
3         [0.1269, 0.1481, 0.1480, 0.1405, 0.3318, 0.1048],
4         [0.0835, 0.0862, 0.1300, 0.1382, 0.4833, 0.0788],
5         [0.1913, 0.1499, 0.1847, 0.1369, 0.2380, 0.0994],
6         [0.1751, 0.1332, 0.1584, 0.1116, 0.3238, 0.0979],
7         [0.1202, 0.1067, 0.1689, 0.1373, 0.3872, 0.0796],
8         [0.1438, 0.0970, 0.2588, 0.1319, 0.2402, 0.1283],
9         [0.1699, 0.1303, 0.1483, 0.1654, 0.2579, 0.1282],
10        [0.1566, 0.1489, 0.1586, 0.1074, 0.3299, 0.0985],
11        [0.1498, 0.1230, 0.2095, 0.1665, 0.2623, 0.0890],
12        [0.2051, 0.0850, 0.1786, 0.1565, 0.2383, 0.1365],
13        [0.1202, 0.1067, 0.1794, 0.1601, 0.3373, 0.0963],
14        [0.1626, 0.1081, 0.2217, 0.1537, 0.2720, 0.0818],
15        [0.1316, 0.1293, 0.1784, 0.1245, 0.2845, 0.1517],
16        [0.1535, 0.0888, 0.1095, 0.1488, 0.3875, 0.1120]], device='
      cuda:0',
17      grad_fn=<SoftmaxBackward0>)
```

Code 2.4: Softmax activation function

along the second dimension (dim=1) to get a probability distribution over the six possible categories. The output is a tensor with shape (batch_size, num_classes).

Finally, we set up the hyper-parameters for the model:

- *EPOCHS*: specifies the number of times the model will iterate through the entire training dataset during training.
- *optimizer*: sets the optimization algorithm to be used during training. Here, AdamW is used with a learning rate of 2e-5 and correct_bias set to False.
- *total_steps*: calculates the total number of training steps by multiplying the number of batches per epoch with the total number of epochs.
- *scheduler*: sets up the learning rate scheduler that will decrease the learning rate as training progresses, get_linear_schedule_with_warmup is used with 0 warmup steps.
- *loss_fn*: sets up the loss function that will be used during training, in our case CrossEntropyLoss is used, which is a commonly used loss function for multi-class classification problems.

We define a function *train_epoch*, Code 3.8:

```
1 def train_epoch(  
2     model ,  
3     data_loader ,  
4     loss_fn ,  
5     optimizer ,  
6     device ,  
7     scheduler ,  
8     n_examples  
9 ):  
10     model = model.train()  
11  
12     losses = []  
13     correct_predictions = 0  
14  
15     for d in data_loader:  
16         input_ids = d["input_ids"].to(device)  
17         attention_mask = d["attention_mask"].to(device)  
18         targets = d["targets"].to(device)
```



```
19
20     outputs = model(
21         input_ids=input_ids,
22         attention_mask=attention_mask
23     )
24
25     _, preds = torch.max(outputs, dim=1)
26     loss = loss_fn(outputs, targets)
27
28     correct_predictions += torch.sum(preds == targets)
29     losses.append(loss.item())
30
31     loss.backward()
32     nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
33     optimizer.step()
34     scheduler.step()
35     optimizer.zero_grad()
36
37     return correct_predictions.double() / n_examples, np.mean(losses)
```

Code 2.5: Train epoch function

which performs one epoch (i.e., one iteration over the entire training set) of training on the model, using the `data_loader` to load the data in batches. The function also takes as input the loss function (`loss_fn`), optimizer (`optimizer`), a device to perform the computation on (`device`), learning rate scheduler (`scheduler`), and the number of training examples (`n_examples`).

Within the function, the model is set to training mode using the `model.train()`, and the losses and correct predictions are initialized to empty lists and zero, respectively. Then, for each batch in the `data_loader`, the input IDs, attention masks, and targets are loaded onto the device. The model is then used to compute outputs for the batch using the input IDs and attention masks, and the predicted labels are obtained by taking the argmax of the outputs along the second dimension (i.e., along the classes). The loss is then computed using the outputs and the targets, and the correct predictions are updated by counting the number of predictions that match the targets.

Next, the gradients of the loss concerning the model parameters are computed using `loss.backward()`, and the gradients are clipped to a maximum norm of 1.0 using

`nn.utils.clip_grad_norm_()`. The optimizer is then used to update the model parameters using the `optimizer.step()`, and the learning rate scheduler is updated using the `scheduler.step()`. Finally, the gradients are zeroed using `optimizer.zero_grad()`.

At the end of the function, the proportion of correct predictions and the average loss over the entire epoch is returned.

eval_model function, Code 3.8:

```
1 def eval_model(model, data_loader, loss_fn, device, n_examples):
2     model = model.eval()
3
4     losses = []
5     correct_predictions = 0
6
7     with torch.no_grad():
8         for d in data_loader:
9             input_ids = d["input_ids"].to(device)
10            attention_mask = d["attention_mask"].to(device)
11            targets = d["targets"].to(device)
12
13            outputs = model(
14                input_ids=input_ids,
15                attention_mask=attention_mask
16            )
17            _, preds = torch.max(outputs, dim=1)
18
19            loss = loss_fn(outputs, targets)
20
21            correct_predictions += torch.sum(preds == targets)
22            losses.append(loss.item())
23
24     return correct_predictions.double() / n_examples, np.mean(losses)
```

Code 2.6: Evaluation of model function

evaluates the performance of a trained model on a given data loader by calculating the accuracy and average loss on the data. The function takes as input the trained model, the data loader, the loss function, the device to run the model on, and the number of examples in the data loader.

Inside the function, the model is set to evaluation mode using `model.eval()`. Then, for each batch of data in the data loader, the inputs and targets are extracted and

converted to the appropriate format for the model and sent to the device using `to(device)`. The model is then called with these inputs to generate predictions for the batch, which are compared with the targets to calculate the number of correct predictions using a `torch.sum(preds == targets)`. The loss for the batch is also calculated using the loss function.

At the end of the evaluation, the function returns the accuracy and average loss over all the batches in the data loader. We also define empty lists for correct visualization of results and starting the training.

The model is trained on the training set using the `train_epoch` function, which computes the forward pass of the model, computes the loss, and performs back-propagation to update the weights. The `eval_model` function is used to evaluate the model on the validation set after each epoch.

The optimizer used is AdamW and a linear scheduler with warmup is used to adjust the learning rate during training.

The training history is stored in a dictionary called history, which contains the training and validation accuracy and loss for each epoch.

If the validation accuracy of the model improves during training, the model's state is saved as the best model so far.

The result of training is the following, Code 3.8:

```

1 Epoch 1/3
2 -----
3 Train loss 1.0114015067924507 accuracy 0.6239869680309509
4 Val loss 0.9178728353838588 accuracy 0.6693548387096774
5
6 Epoch 2/3
7 -----
8 Train loss 0.753837581372028 accuracy 0.7344329057218489
9 Val loss 0.8933253097672795 accuracy 0.6759530791788856
10
11 Epoch 3/3
12 -----
13 Train loss 0.6311768608052489 accuracy 0.7820403176542455
14 Val loss 0.9119356021631596 accuracy 0.6803519061583577
15
16 CPU times: user 2h 17min 5s, sys: 27 s, total: 2h 17min 32s
17 Wall time: 2h 17min 57s

```

Code 2.7: Results of training

The plotting of train and validation accuracy is shown in Figure 2.8:

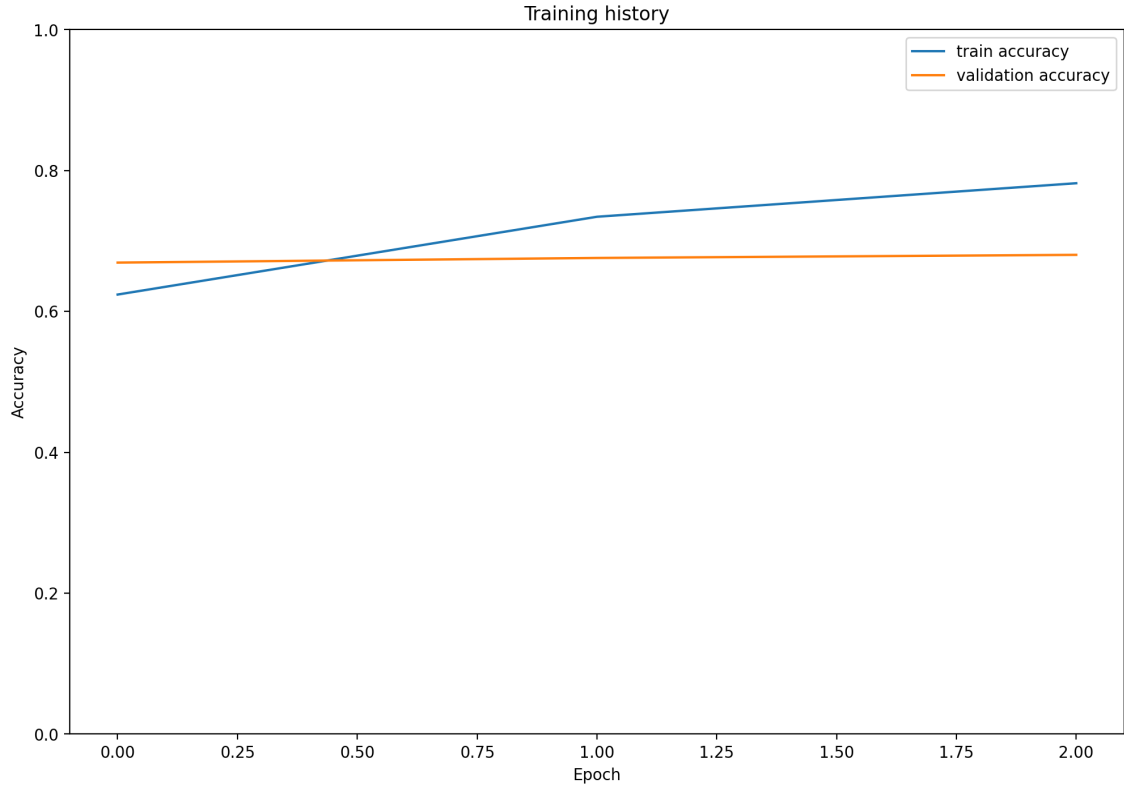


Figure 2.8: Training history

In the end, we evaluate the performance of the trained model on the test set, which is data that the model has not seen during training or validation. It calls the `eval_model` function with the test data loader, the loss function, and the device. Then we retrieve the test accuracy from the output of the `eval_model` function, which is the number of correct predictions divided by the total number of test examples. The `item()` method is called to convert the test accuracy from a PyTorch tensor to a Python float. In the Code 3.8:

```
1 0.7025641025641025
```

Code 2.8: Accuracy of trained model

we can see that the accuracy was good and acceptable.

We define a function `get_predictions` that takes a trained model and a data loader as inputs. It evaluates the model on the data in the data loader, returning the review texts, predicted labels, predicted probabilities, and real labels for each item in

the dataset. It is done by iterating over each batch in the data loader and running the input through the model, obtaining the predicted probabilities and predicted labels. We append these values to lists along with the review texts and real labels, which are later stacked into tensors and returned as the output of the function. We generate predictions for the test data and save the review texts, predicted labels, predicted probabilities, and actual labels in separate variables `y_review_texts`, `y_pred`, `y_pred_probs`, and `y_test`. The function `get_predictions` takes the trained model and the test data loader as input and returns the aforementioned variables. The `y_review_texts` variable contains the review texts for each example in the test set, the `y_pred` variable contains the predicted labels (as integers), the `y_pred_probs` variable contains the predicted probabilities for each label, and the `y_test` variable contains the actual labels (as integers) for each example in the test set. The list of category names is defined which will be used later as target names. The Code 3.8:

1		precision	recall	f1-score	support
2					
3	programming	0.76	0.78	0.77	258
4	business	0.63	0.58	0.60	245
5	health	0.69	0.68	0.68	258
6	marketing	0.69	0.72	0.71	260
7	politics	0.68	0.69	0.68	240
8	sports	0.83	0.83	0.83	104
9					
10	accuracy			0.70	1365
11	macro avg	0.71	0.71	0.71	1365
12	weighted avg	0.70	0.70	0.70	1365

Code 2.9: Classification report

shows us the classification report for the predicted labels (`y_pred`) and the actual labels (`y_test`) of the test dataset. The `target_names` argument is used to specify the names of the categories in the report. The classification report includes precision, recall, F1-score, and support for each category, as well as the accuracy, macro-average F1-score, and weighted-average F1-score for the whole dataset. The report helps to evaluate the performance of the model for each category and overall.

The visualization of the confusion matrix on Figure 2.9:

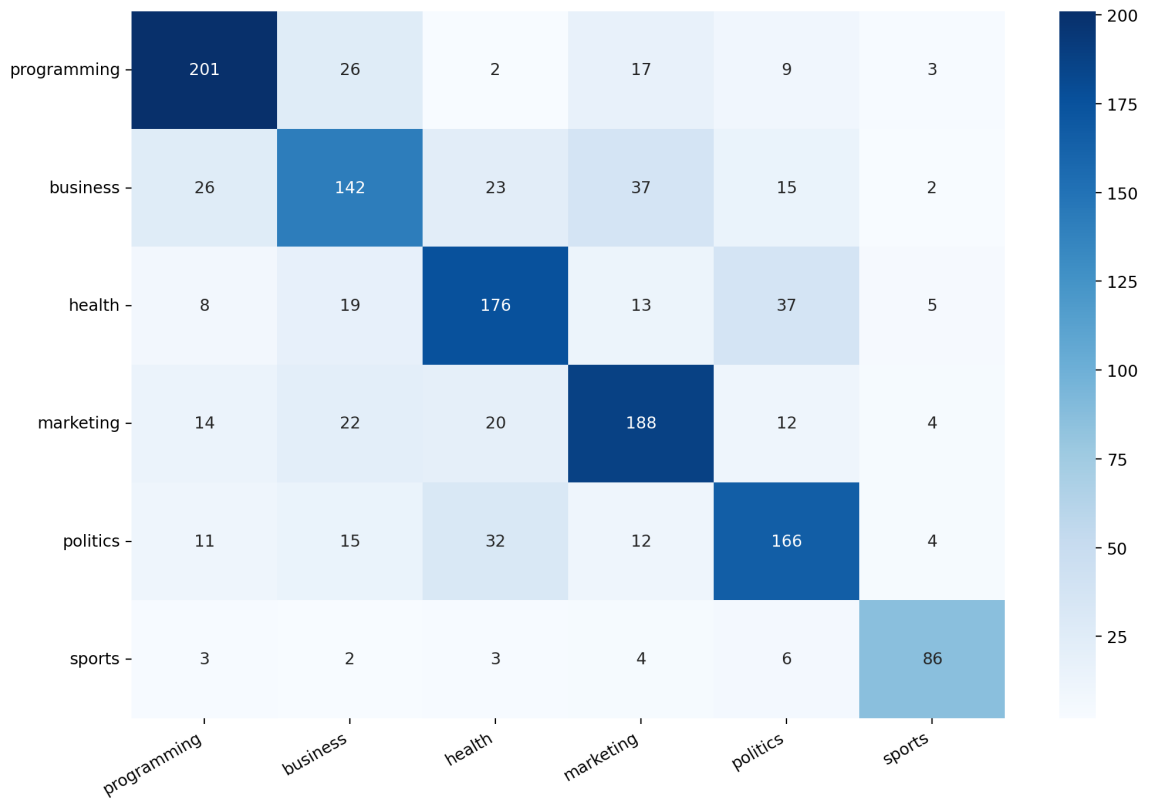


Figure 2.9: Confusion matrix for BERT model

tells us that the results are good and accurate

In addition if we study the performance on real example, Code 3.8:

```

1 mani peopl attempt lose weight succeed other fail howev biggest
  battl
2 peopl abl reduc weight often keep ideal weight mani peopl find soon
3 return weight went diet even actual fatter cours depress result
  lose
4 lot self esteem need perman solut weight problem obviou rout take
5 battl lose weight would includ increas amount exercis reduc amount
  eat
6 eat issu hardest control reduc temptat often get better us opinion
7 need make hous fat free zone becom hungri start look cupboard notic
8 exampl packet crisp often difficult eat desir instant food becom
  great
9 inner demon tri convinc us one packet hurt packet crisp cupboard
  would
10 put posit temptat would cours abl eat number year ago went lose
  excess
11 weight decid remov food cupboard awar need stop eat also remov
  certain

```

```
12 drink alcohol drink also someth contribut weight problem put
    dustbin
13 takeaway menu basic attempt make hard possibl eat drink anyth
    determin
14 keep diet tempt buy item shop etc easi somebodi love fatti type
    food
15 weekli food shop bought far fruit veget surpris quickli tast bud
    start
16 chang soon look forward eat appl exampl weight slowli sure start
    reduc
17 number month reach weight happi wife state abl start eat item dri
18 roast peanut particular favourit mine possibl true could easili
    result
19 return old bad habit cours weight problem decid stick fruit
    cupboard
20 still free food love eat good weight worldweight loss product
21
22 True category: health
```

Code 2.10: Example of prediction process

it is seen that the model was trained successfully. This is clearly understandable if we take a look on the confidence (probability) value that the above-mentioned example will have category "health", Figure 2.10:

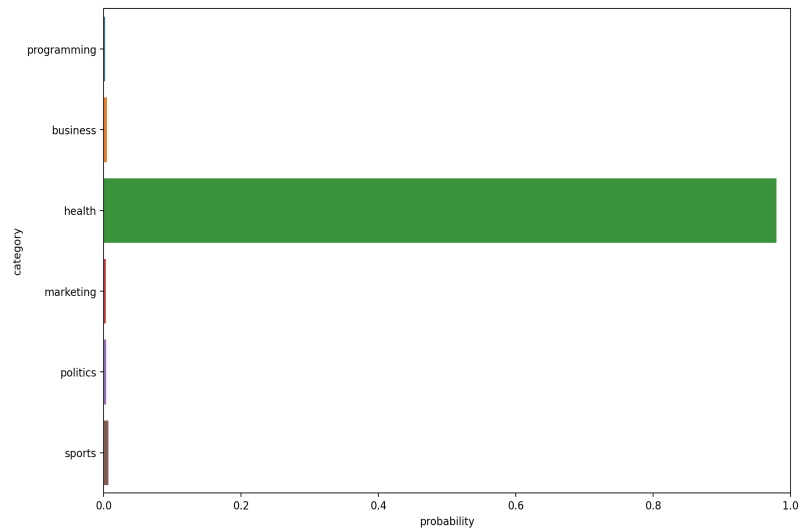


Figure 2.10: Probability of class distribution

This figure shows us that the category of the previously discussed text is health with probability 0.98.

It is also necessary to create a template for the production which implies the generation of the general algorithm for category classification on some real example (*bertEval.ipynb*). For this, we should first define some global variables like `PRE_TRAINED_MODEL_NAME = "bert-base-uncased"` which we will use for tokenizer loading, `MAX_LEN = 300` which we will use as the max length of input text and `category_names = ['programming', 'business', 'health', 'marketing', 'politics', 'sports']` which will define the target categories. Next, we will define a PyTorch nn.Module subclass called *CategoryClassifier*, which is a sentiment classification model based on the pre-trained BERT model. The *MyDataset* class is a custom dataset class that represents the input data for the model. This dataset takes in the reviews, targets, tokenizer, and max_len as input, tokenizes the reviews using the tokenizer, and returns a dictionary with the review text, input_ids, attention mask, and targets. Finally, a function called *create_data_loader* is defined, which takes in a data frame, tokenizer, max_len, and batch_size, creates an instance of the *MyDataset* class with the given inputs, and returns a PyTorch DataLoader object that can be used to iterate over the data in batches during evaluation.

For this demo template we create *input_text* example which is quite similar to the possible input text and create an instance of the *CategoryClassifier* class with 6 output classes, load the state dictionary saved in the file *'best_model_state.bin'* into the model, and move the model to the device. We also load the *BertTokenizer* using the *from_pretrained()* method, which will be used to encode the text data.

We encode a given input text using the BERT tokenizer. It performs the following steps:

- *tokenizer.encode_plus()*: This method of the tokenizer takes the input text as an argument and returns a dictionary containing the tokenized representation of the text along with additional information.
- *add_special_tokens=True*: This argument tells the tokenizer to add special tokens like [CLS] and [SEP] to the beginning and end of the input text respectively.
- *max_length=MAX_LEN*: This argument sets the maximum length of the tokenized text. If the tokenized text exceeds this length, it will be truncated.

- *truncation=True*: This argument tells the tokenizer to truncate the tokenized text if it exceeds the maximum length.
- *return_token_type_ids=False*: This argument tells the tokenizer to not return the token type IDs. Token-type IDs are used in BERT to differentiate between different segments of input text (e.g., between the question and the answer in a question-answering task).
- *padding='max_length'*: This argument tells the tokenizer to pad the tokenized text with zeros so that it has the same length as MAX_LEN.
- *return_attention_mask=True*: This argument tells the tokenizer to return an attention mask that has the same length as the tokenized text. The attention mask is used by BERT to indicate which tokens should be attended to and which tokens should be ignored.
- *return_tensors='pt'*: This argument tells the tokenizer to return PyTorch tensors instead of Python lists.

In the end, we perform the implementation of the fine-tuned BERT model to predict the category of a given input text.

First, the input text is tokenized using the BERT tokenizer with specified settings like max_length, truncation, etc. The resulting encoding is stored in encoded_text.

The input_ids and attention_mask tensors are extracted from encoded_text and moved to the device (e.g. GPU) specified during training.

Then, the model is used to predict the category of the input text by passing the input_ids and attention_mask tensors to it. The prediction is the class index with the highest probability.

Finally, the predicted category is printed along with the input text, Code 3.8:

```
1 Input text: As the global economy continues to recover from the
    impact of the COVID-19 pandemic, businesses are looking for new
    ways to grow and thrive in the post-pandemic world. With the
    rise of digital technologies and changing consumer preferences,
    companies must be agile and innovative to stay competitive. From
    leveraging data analytics to exploring new markets and
    partnerships, businesses must adapt quickly to meet the evolving
    needs of their customers. Those that can successfully navigate
    these challenges and seize new opportunities will be well-
    positioned for success in the years ahead.
```

2 Category : business

Code 2.11: Example of demo

2.3 Lottery

The section is describing the implementation of lottery smart-contract on local development blockchain. However, before we start it is important what we imply about smart contracts and blockchain.

The reasons we are using blockchain smart contracts as an environment for lottery execution are the following: it is a great marketing approach (as this is a young and fancy technology) with it we can attract more users to become our company's customers; in addition, the cost of running lottery via blockchain is much cheaper than organizing the lottery using 3rd party members; from the previous statement the last one comes - we don't need any traditional legal agreement to use blockchain which eases organization and faster the deployment of the lottery.

A smart contract is a self-executing program that runs on a blockchain. In it, we define the set of rules and regulations that are encoded into computer code, which can automatically enforce the terms of an agreement between parties. The contract can be programmed to execute automatically when certain conditions are met, without the need for human intervention or third-party intermediaries. With smart contracts, we can provide an efficient way to exchange value and assets without relying on traditional legal agreements.

A blockchain is a digital ledger of records that is decentralized and secure. It is essentially a database that is managed by a network of computers rather than a single central authority. Each block in the chain contains several transactions, and once a block is added to the chain, it cannot be altered or deleted.

2.3.1 Lottery implementation

The lottery folder consists of several parts:

- *build/contracts* contain built ready-for execution contracts, they are created automatically.

- *contracts* folder contains the main logic of smart contracts we will describe it later.
- *migration* folder containing the deployment part, we will use this folder to launch our contract to the blockchain.
- *test* folder contains tests of the smart contract before deploying it to the blockchain

The *Lottery.sol* inside the *contracts* folder contains the core logic of our lottery implementation. The contract allows participants to enter the lottery by sending ether, and the contract owner can request a random number to determine the winner. Here's a breakdown of the functionalities:

- *ticketPrice*: A public variable storing the ticket price for the lottery.
- *participants*: An array that holds the addresses of participants in the lottery.
- *requestId*: A private variable storing a unique ID for each random number request.
- *randomNumber*: A mapping that links *requestId* to the corresponding random number.

Events:

- *RequestedRandomNumber*: Emitted when a random number is requested.
- *LotteryResult*: Emitted when the lottery winner is determined.

Functions:

- *constructor*: Initializes the ticket price for the lottery.
- *enter*: Allows participants to enter the lottery by sending the correct ticket price as ether.
- *getParticipants*: Returns the list of participants in the lottery.
- *requestRandomNumber*: Allows the contract owner to request a random number, which is derived from *requestId* (using *block.timestamp*, *msg.sender*, *block.number*, and contract address). Emits *RequestedRandomNumber* event.

- *setRandomNumber*: Sets the random number for the corresponding request and calls the *pickWinner* function.
- *pickWinner*: Determines the lottery winner based on the random number and sends the entire contract balance to the winner. Emits the *LotteryResult* event and resets the lottery.
- *resetLottery*: Resets the participant's array for the next round of the lottery.
- *withdraw*: Allows the contract owner to withdraw the remaining ether from the contract.

In other words, this contract allows users to participate in a lottery game by sending ether, and the contract owner can determine a winner based on a random number. The winner receives the entire ether balance stored in the contract.

2.3.2 Tests conducting

The *lottery.test.js* in the test folder check the functionality of the Lottery contract and ensures it works as expected. It uses the Truffle framework, OpenZeppelin Test Environment, and Chai assertion library.

The code initializes variables and sets up the test environment by deploying a new Lottery contract and adding two participants (*addr1* and *addr2*) to the lottery in the *beforeEach* hook.

The test suite consists of four main test sections:

- *Deployment*: This section tests if the ticket price is set correctly during the deployment of the Lottery contract.
- *Participation*: This section tests if the contract allows users to enter the lottery and if the contract rejects entries with incorrect ticket prices.
- *Random Number Request and Setting*: This section checks if only the contract owner can request and set a random number.
- *Lottery Execution*: This section tests if the contract correctly picks a winner, resets the lottery, and ensures only the owner can withdraw the balance.

In summary, the test suite ensures the Lottery contract behaves as expected by checking various conditions, such as correct ticket prices, valid user participation, proper access control for random number generation, and correct lottery execution.

To test the contract we need to run the following commands, Code 3.8:

```
1 truffle compile
2 truffle test
```

Code 2.12: Run tests

The results of tests are presented on the Code 3.8:

```
1 Contract: Lottery
2
3 Deployment
4     should set the ticket price correctly
5 Participation
6     should allow users to enter the lottery (115ms)
7     should reject entries with incorrect ticket price (178ms)
8 Random Number Request and Setting
9     should allow only owner to request a random number (165ms)
10    should allow only owner to set a random number (91ms)
11 Lottery Execution
12    should pick a winner and reset the lottery (151ms)
13    should allow only owner to withdraw the balance (104ms)
14
15 7 passing (2s)
```

Code 2.13: Tests results

2.3.3 Deployment script

The code inside the migrations folder called *2_deploy_lottery.js* contains the deployment script for the Lottery smart contract. The script specifies how the Lottery contract should be deployed to the blockchain.

In the script, the "Lottery" artifact is imported, which represents the compiled Lottery smart contract. The *module.exports* function is the main deployment function, taking a deployer object as an argument.

Inside the deployment function, the *deployer.deploy()* method is called with two arguments:

- *Lottery*: The compiled Lottery smart contract.
- *web3.utils.toWei("0.01", "ether")*: This converts 0.01 Ether to its equivalent in Wei (the smallest unit of Ether). This value is passed as an argument to the Lottery contract's constructor to set the ticket price for participating in the lottery.

When this script is executed using Truffle, it will deploy the Lottery smart contract to the blockchain with the specified ticket price.

Chapter 3

User documentation

This part describes step by step end-user guide about how to use the created software (*CategoryClassifier.ipynb*), the general workflow of production is shown in Figure 3.1:

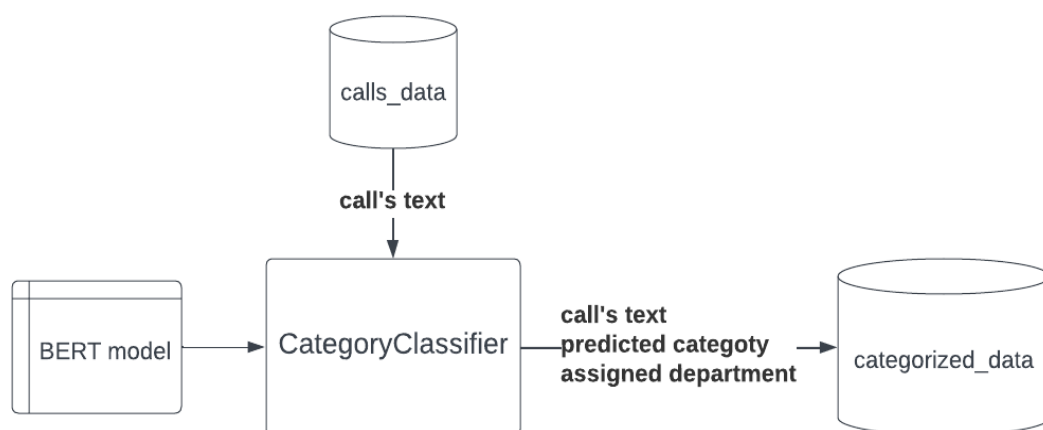


Figure 3.1: Production workflow

The general idea is the following: as input, we have PostgreSQL database with call recording, the *CategoryClassifier* loads input data and applies trained BERT model on it. As a result for each input call recording, we have predicted the category as well as assigned to it department. This result is saved and pushed into output *categorized_data* PostgreSQL database, which can be used later.

3.1 Category classification

First, we define some global parameters:

- *PRE_TRAINED_MODEL_NAME*: name of the model from which we will load tokenizer in our case it is "bert-base-uncased"
- *MAX_LEN*: max length of input call recording, in our case it is 128 words
- *category_names*: the list of target names, which we are going to predict for each call recording
- *conn_string*: connection string to the PostgreSQL database
- *text_list*, *category_list*, *department_list*: lists in which we are going to store call recordings, predicted category and assigned company department

CategoryClassifier, Code 3.8:

```
1 class CategoryClassifier(nn.Module):
2
3     def __init__(self, n_classes):
4         super(CategoryClassifier, self).__init__()
5         self.bert = BertModel.from_pretrained(PRE_TRAINED_MODEL_NAME)
6         self.drop = nn.Dropout(p=0.3)
7         self.out = nn.Linear(self.bert.config.hidden_size, n_classes)
8
9     def forward(self, input_ids, attention_mask):
10         outputs = self.bert(
11             input_ids=input_ids,
12             attention_mask=attention_mask
13         )
14         pooled_output = outputs.pooler_output
15         output = self.drop(pooled_output)
16         return self.out(output)
```

Code 3.1: CategoryClassifier class

is a PyTorch module that defines a BERT-based classifier. It takes as input the number of classes to predict and returns the logits for each class.

MyDataset, Code 3.8:


```
1 class MyDataset(Dataset):
2
3     def __init__(self, reviews, targets, tokenizer, max_len):
4         self.reviews = reviews
5         self.targets = targets
6         self.tokenizer = tokenizer
7         self.max_len = max_len
8
9     def __len__(self):
10         return len(self.reviews)
11
12     def __getitem__(self, item):
13         review = str(self.reviews[item])
14         target = self.targets[item]
15
16         encoding = self.tokenizer.encode_plus(
17             review,
18             add_special_tokens=True,
19             max_length=self.max_len,
20             truncation=True,
21             return_token_type_ids=False,
22             padding='max_length',
23             return_attention_mask=True,
24             return_tensors='pt',
25         )
26
27         return {
28             'review_text': review,
29             'input_ids': encoding['input_ids'].flatten(),
30             'attention_mask': encoding['attention_mask'].flatten(),
31             'targets': torch.tensor(target, dtype=torch.long)
32         }
```

Code 3.2: MyDataset class

is a PyTorch Dataset that prepares the data for the model. It takes as input the reviews, targets, tokenizer, and maximum length of the tokenized sequences, and returns a dictionary containing the tokenized review, attention mask, and target category.

create_data_loader, Code 3.8:

```
1 def create_data_loader(df, tokenizer, max_len, batch_size):
2     ds = MyDataset(
3         reviews=df.stemmed_text.to_numpy(),
4         targets=df.category_id.to_numpy(),
5         tokenizer=tokenizer,
6         max_len=max_len
7     )
8
9     return DataLoader(
10         ds,
11         batch_size=batch_size,
12         num_workers=4
13     )
```

Code 3.3: create_data_loader function

is a function that creates a PyTorch DataLoader from a DataFrame of reviews and their associated target categories. It takes as input the DataFrame, tokenizer, maximum length, and batch size, and returns a DataLoader that shuffles and batches the data for training.

department_name, Code 3.8:

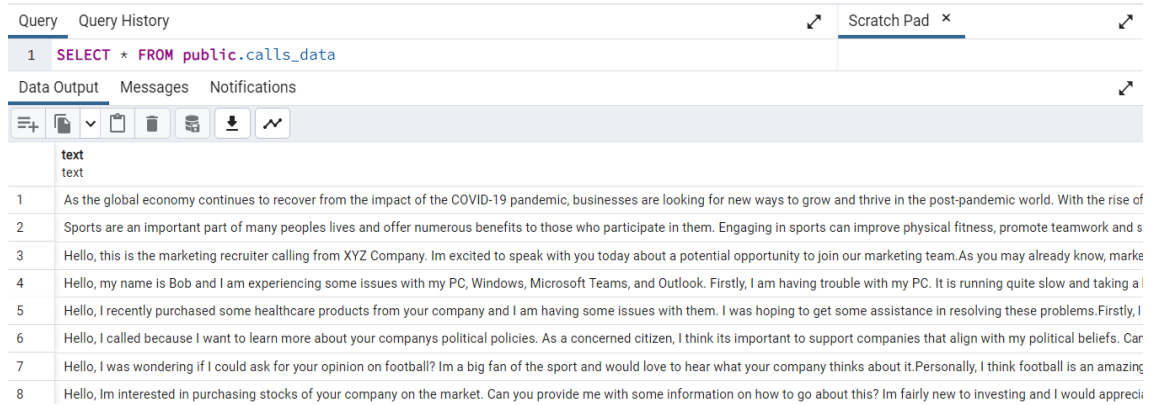
```
1 def department_name(category):
2     if category == 'programming':
3         return 'Information Technology'
4     if category == 'business' or category == 'politics':
5         return 'Business and Public Affairs'
6     if category == 'health':
7         return 'Healthcare Services'
8     if category == 'marketing':
9         return 'Marketing'
10    if category == 'sports':
11        return 'Sports Division'
```

Code 3.4: department_name function

is a function that maps each category label to a department name.

Then we load a trained BERT-based text classification model from a saved state dictionary and move it to the available device (GPU if available, CPU otherwise). The state dictionary contains the trained weights and biases of the model. We also initialize a tokenizer object that was used during training to tokenize the text inputs.

Later on, we connect to a PostgreSQL database using the `psycopg2` library and execute a SQL query to fetch the text column from a table named `calls_data`, an example of `calls_data` database is shown on Figure 3.2:



	text	text
1	As the global economy continues to recover from the impact of the COVID-19 pandemic, businesses are looking for new ways to grow and thrive in the post-pandemic world. With the rise of	
2	Sports are an important part of many peoples lives and offer numerous benefits to those who participate in them. Engaging in sports can improve physical fitness, promote teamwork and s	
3	Hello, this is the marketing recruiter calling from XYZ Company. Im excited to speak with you today about a potential opportunity to join our marketing team.As you may already know, marke	
4	Hello, my name is Bob and I am experiencing some issues with my PC, Windows, Microsoft Teams, and Outlook. Firstly, I am having trouble with my PC. It is running quite slow and taking a	
5	Hello, I recently purchased some healthcare products from your company and I am having some issues with them. I was hoping to get some assistance in resolving these problems.Firstly, I	
6	Hello, I called because I want to learn more about your companys political policies. As a concerned citizen, I think its important to support companies that align with my political beliefs. Car	
7	Hello, I was wondering if I could ask for your opinion on football? Im a big fan of the sport and would love to hear what your company thinks about it.Personally, I think football is an amazing	
8	Hello, Im interested in purchasing stocks of your company on the market. Can you provide me with some information on how to go about this? Im fairly new to investing and I would appreci	

Figure 3.2: Input database

Then we fetch all the rows returned by the query using the `fetchall()` method and store the text values in a list named `text_list`. Finally, it closes the cursor and connection to the database.

Finally, we fetch text data from a PostgreSQL database using the `psycopg2` library and then make predictions for the category of each text using a pre-trained BERT model. For each text, we encode the text using the BERT tokenizer, feed the encoded text through the BERT model, and obtain a predicted category. Then append the predicted category to a list called `category_list` and the corresponding department to another list called `department_list`.

In the last step, we connect to a PostgreSQL database using the connection string, create a cursor object to execute SQL statements, and insert categorized data into the database. More precisely: we loop through the list of text values and predict the category and department of each text using the trained model. The SQL INSERT statement is used to insert the text, category, and department values into the `categorized_data` table in the database. Finally, the code commits the changes to the database and closes the cursor and connection objects. As a result, we have the predicted information, example in Code 3.8:

```
1 Text: Hello, my name is Bob and I am experiencing some issues with
    my PC, Windows, Microsoft Teams, and Outlook. Firstly, I am
    having trouble with my PC. It is running quite slowly and taking
    a long time to start up. I have tried running various anti-
    virus and anti-malware software but it hasnt helped. Im not sure
```

```

    what else I can do to improve its performance.Secondly, I am
    having trouble with my Windows operating system. Some of my
    applications are crashing unexpectedly, and I am unable to
    update my system. I have tried troubleshooting the issue, but I
    havent been successful in resolving it. Thirdly, I am
    experiencing some problems with Microsoft Teams. I am unable to
    join some of my scheduled meetings and I am getting error
    messages when I try to do so. It is affecting my productivity
    and I am not sure what the issue is.Lastly, I am having issues
    with Outlook. I am unable to receive emails on my account and I
    have tried checking my settings but nothing seems to be working.
    It is causing me a lot of inconvenience as I am unable to keep
    up with my work emails.I would greatly appreciate it if you
    could assist me with these issues. Thank you.
2 Predicted category: programming
3 Assigned department: Information Technology

```

Code 3.5: Example of output

This information is pushed to the output database, Figure 3.3:

Query

Query History









1

SELECT * FROM public.categorized_data

Data Output

Messages

Notifications



	text text	category text	department text
1	As the global economy continues to recover from the impact of t...	business	Business and Public Affairs
2	Sports are an important part of many peoples lives and offer num...	sports	Sports Division
3	Hello, this is the marketing recruiter calling from XYZ Company. I...	marketing	Marketing
4	Hello, my name is Bob and I am experiencing some issues with m...	programming	Information Technology
5	Hello, I recently purchased some healthcare products from your c...	health	Healthcare Services
6	Hello, I called because I want to learn more about your companys...	business	Business and Public Affairs
7	Hello, I was wondering if I could ask for your opinion on football? ...	sports	Sports Division
8	Hello, Im interested in purchasing stocks of your company on the ...	business	Business and Public Affairs

Figure 3.3: Output database

3.2 Lottery execution

Every software product should contain not only code implementation but also advertising and marketing take to make the product attractive to users and investors.

The lottery smart contract deployed on the Ethereum blockchain is a perfect choice. It is a great marketing approach (as this is a young and fancy technology) with it we can attract more users to become our company's customers; in addition, the cost of running the lottery via blockchain is much cheaper than organizing the lottery using 3rd party members; from previous statements, the next one comes - we don't need any traditional legal agreement to use blockchain which eases organization and fast the deployment of the lottery. The content and logic of the lottery contract will be visible to everyone on the web, as this is a part of smart contract rules, this will be the proof of an objective and fair lottery.

This section contains end-user step-by-step execution of lottery using local blockchain.

For the execution, the following steps should be done:

- Download and install Ganache from <https://www.trufflesuite.com/ganache>.
- Launch Ganache and create a new workspace.
- Install Node.js and npm (Node Package Manager) if you haven't already. Run `npm install -g truffle` to install Truffle globally.
- Run `truffle init` to initialize the Truffle project.
- Run `npm init -y` to create a package.json file.
- Run `npm install @openzeppelin/contracts` to install the OpenZeppelin Contracts library.
- Replace the content of a created folder with the content of the "lottery" folder
- Run `truffle compile` to compile the smart contract.
- Run `truffle migrate --network development` to deploy the smart contract to the local Ganache network.
- Run `truffle console --network development` to interact with the contract through the console.
- Run these commands one by one to execute the lottery: Code 3.8:

```
1 let instance = await Lottery.deployed();
2 let accounts = await web3.eth.getAccounts();
3 await instance.enter({from: accounts[1], value: web3.utils.
  toWei('0.01', 'ether')});
4 await instance.enter({from: accounts[2], value: web3.utils.
  toWei('0.01', 'ether')});
5 await instance.enter({from: accounts[3], value: web3.utils.
  toWei('0.01', 'ether')});
6 await instance.enter({from: accounts[4], value: web3.utils.
  toWei('0.01', 'ether')});
7 await instance.enter({from: accounts[5], value: web3.utils.
  toWei('0.01', 'ether')});
8 let result = await instance.requestRandomNumber({from: accounts
  [0]});
9 let winnerAddress = result.logs[0].args.winner;
10 console.log("The winner's address is:", winnerAddress);
```

Code 3.6: Lottery commands

We will deploy sample lottery for 5 accounts and during the execution of these commands we will receive this output, Code 3.8:

```
1 2_deploy_lottery.js
2 =====
3
4 Replacing 'Lottery'
5 -----
6 > transaction hash:      0
   x012bcb1cba39a1f5bd49b1f18af97012dbcb6532bacd1b5328ab89ebb8e11506
7
8 > Blocks: 0              Seconds: 0
9 > contract address:      0
   xccBd740AC8eB18d07B452F29F9f2C76EB3648005
10 > block number:          1
11 > block timestamp:       1683559995
12 > account:               0
   x1bB9e0BF792701B3A31c5577ff659b5d9864Cb35
13 > balance:               0.99666029575
14 > gas used:              989542 (0xf1966)
15 > gas price:             3.375 gwei
   > value sent:           0 ETH
```

```

16 > total cost:          0.00333970425 ETH
17
18 > Saving artifacts
19 -----
20 > Total cost:          0.00333970425 ETH
21
22 Summary
23 =====
24 > Total deployments:    1
25 > Final cost:           0.00333970425 ETH

```

Code 3.7: Contract deployment

which tells us that the smart contract with lottery is deployed successfully, as you can notice the price of deploying the lottery is about 2100 HUF which is low and incomparable to the price of deploying the lottery from 3rd party organizations. Finally, the result of the lottery is the following, Code 3.8:

```

1 The winner's address is: 0x1B785C12980760EB08cbcabdf0a391bC207408cE

```

Code 3.8: Lottery results

according to this result the winner is account number 5, Figure 3.4:

ADDRESS	BALANCE	TX COUNT	INDEX
0xeAe2eA5058959deC2161c2B5c1edCc273ee58bEC	0.99 ETH	1	1
0x1f8993D674C0a95Cf051143A9D65aC28bD5d5E95	0.99 ETH	1	2
0x09f18a19Bc9535d626b7267f6553d42e8c5B48CF	0.99 ETH	1	3
0xc96C1e4A764e85E5BE2960dF3f7c8BA2F034d426	0.99 ETH	1	4
0x1B785C12980760EB08cbcabdf0a391bC207408cE	1.04 ETH	1	5

Figure 3.4: Winner account

As was mentioned previously the results will be also available for anyone via public source, Figure 3.5:

3. User documentation

CONTRACT NAME Lottery		CONTRACT ADDRESS 0×ccBd740AC8eB18d07B452F29F9f2C76EB3648005	
SIGNATURE (DECODED) LotteryResult(randomNumber: uint256, winner: address)			
TX HASH 0×b3fb7e252b578989b9a088f718c046b0e2cdf445b63c7ca64992a5c3eadaab29		LOG INDEX 0	BLOCK TIME 2023-05-08 17:34:53
RETURN VALUES			
RANDOMNUMBER 594909			
WINNER 0×1b785c12980760eb08cbcabdf0a391bc207408ce			

Figure 3.5: Proof of fairness

Chapter 4

Conclusion

In conclusion, we can say that all of the requirements for the project were satisfied and the results completely fulfilled the desired planned output.

The optimal language model was acquired using scientific research in the natural language processing area. The language model was trained on the appropriate and well-pre-processed data with equal representations of categories, and their boundaries were well-defined. The obtained model was put into an easy-to-use final algorithm, which does not require any intervention or set-up preparation work.

The lottery was implemented using the latest documentation available, therefore its approaches and algorithms satisfy modern security and functionality requirements.

Acknowledgements

First of all, I would like to thank my family and Tempus Public Foundation, because of them I had an opportunity to study at ELTE. In addition, without teachers at my University, it wouldn't be possible to create this work.

Finally, I would like to express my gratitude to my supervisors: Németh Zsolt and Brunner Tibor who were helping me with this thesis. Special thanks to my friend - Kuknyó Dániel, who was consulting me with troubleshooting.

Appendix A

Development workflow

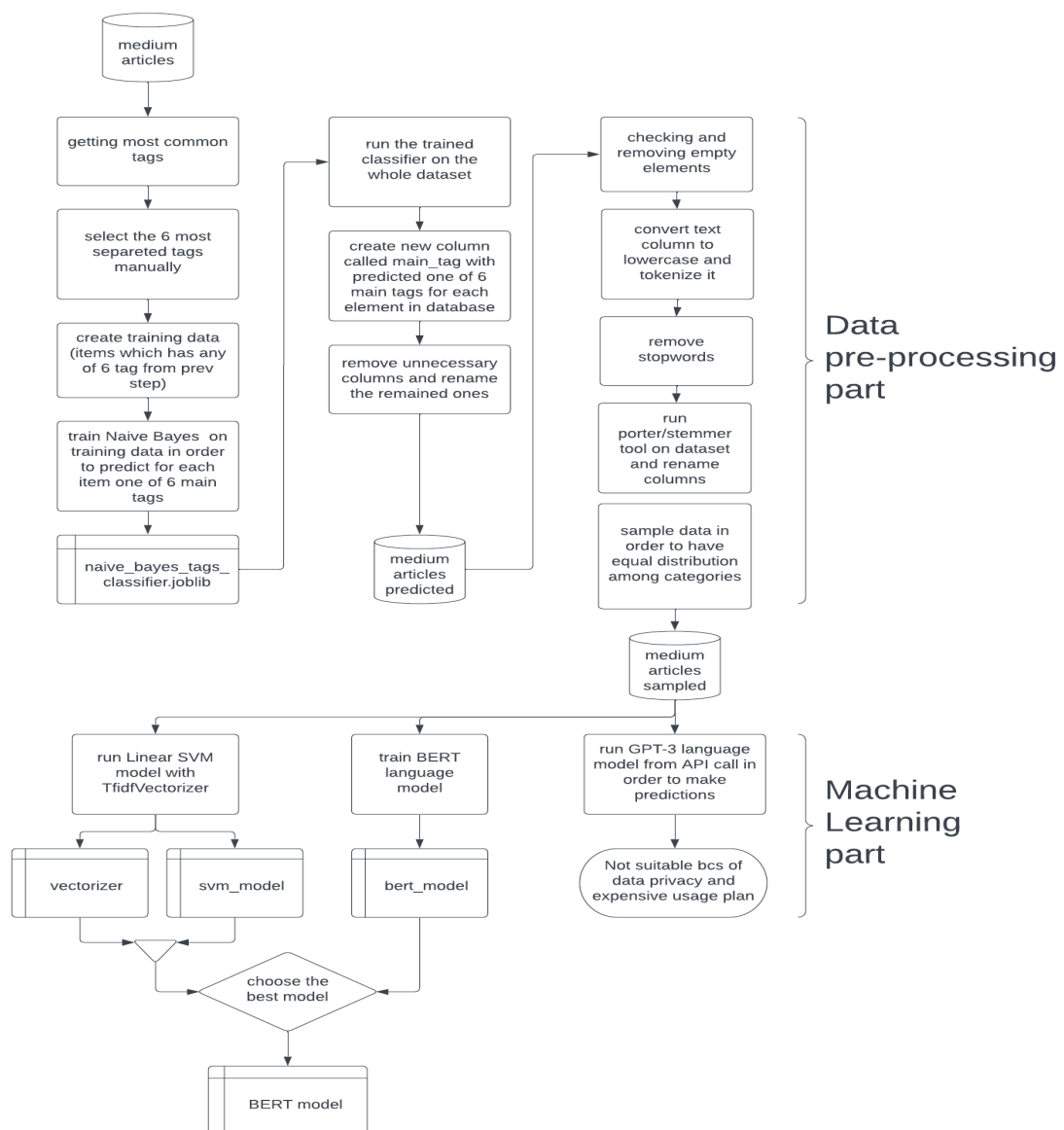


Figure A.1: Development workflow

Bibliography

- [1] J. Moreira, A. Carvalho, and T. Horvath. *A General Introduction to Data Analytics*. Wiley, 2018. ISBN: 9781119296249. URL: <https://books.google.at/books?id=UoZfDwAAQBAJ>.
- [2] John Doe. *Get SH*T Done with PyTorch: Solve Real-World Machine Learning Problems with Deep Neural Networks in Python*. Kindle edition. 2023. URL: <https://www.amazon.com/dp/B0895YQYFC>.
- [3] freeCodeCamp.org. *Solidity, Blockchain, and Smart Contract Course – Beginner to Expert Python Tutorial*. <https://www.youtube.com/watch?v=M576WGiDBdQ>. YouTube video. 2021.

List of Figures

2.1	Character length distribution	5
2.2	Text distribution by category	5
2.3	Text distribution by category after sampling	6
2.4	Histogram of number of words	6
2.5	25 most common words	7
2.6	25 most common words after filtering	7
2.7	Confusion matrix for Linear SVM	10
2.8	Training history	19
2.9	Confusion matrix for BERT model	21
2.10	Probability of class distribution	22
3.1	Production workflow	30
3.2	Input database	34
3.3	Output database	35
3.4	Winner account	38
3.5	Proof of fairness	39
A.1	Development workflow	42

List of Codes

2.1	Naive Bayes Classifier	4
2.2	Linear SMV with TfidVectorizer	9
2.3	GPReviewDataset class	11
2.4	Softmax activation function	14
2.5	Train epoch function	15
2.6	Evaluation of model function	17
2.7	Results of training	18
2.8	Accuracy of trained model	19
2.9	Classification report	20
2.10	Example of prediction process	21
2.11	Example of demo	24
2.12	Run tests	28
2.13	Tests results	28
3.1	CategoryClassifier class	31
3.2	MyDataset class	32
3.3	create_data_loader function	33
3.4	department_name function	33
3.5	Example of output	34
3.6	Lottery commands	37
3.7	Contract deployment	37
3.8	Lottery results	38