Genetic Mutation Classification based on Clinical Evidence to Enable Personalized Medicine for Cancer Treatment

By

LIU Kai Yang, Louis
(1830004016)

A Final Year Project thesis (STAT4004)
submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Science (Honours)
in Statistics

at

BNU-HKBU
UNITED INTERNATIONAL COLLEGE

December, 2021

DECLARATION

I hereby declare that all the work done in this Project is of my independent effort. I also certify that I have never submitted the idea and product of this Project for academic or employment credits.

LIU Kai Yang, Louis
(1830004016)

Date:

Genetic Mutation Classification based on Clinical Evidence to Enable Personalized Medicine for Cancer Treatment

LIU Kai Yang, Louis

Science and Technology Division

Abstract

Traditionally, interpretation of the clinical documents to diagnosis the genetic variants type for patients with cancer is very time-consuming, it will cost lots of human efforts. In recent years, the emergence of Machine Learning and Deep Learning classification models that combined with Natural Language processing techniques greatly facilitate the diagnosis process based on the clinical texts. In the future, personalized medicine for cancer treatment may become true by using advanced gene classification models. In this project, our goals is to classify 9 genetic mutation classes based on the clinical evidence. There are two proposed models. For the Word2Vec + LightGBM models, its computational strength with the balance of accuracy help it reach a 60% of accuracy and got a desirable 2.52648 private score in Kaggle. For the Word Embedding + Bidirectional GRU with Attention model, due to the strength of gate structure in GRU, full picture of text information retrieved by bidirectional layers, and also the contribution of attention mechanism in focusing more important part of texts, we got a great improvement and get an 85% accuracy result and finally reach 2.36962 private score with rank 88 out of 1386 in Kaggle.

Keywords: LightGBM, GRU, Bidirectional, Attention, Word2Vec

Preface

A Genetic Mutation Classification Project based on Clinical evidence. Processed Natural Language Processing and use Machine Learning methods LightGBM as well as Deep Learning models Bidirectional GRU with Attention.

Acknowledgment

This project would not have been possible without the support of many people. Thanks to my advisor Dr. Yuhui Deng for holding so many seminars, providing great support, and giving instructions during my final year project. Also, I will give many thanks to my friends in the same group who offer guidance and supports.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background information

According to International Agency for Research on Cancer, nearly ten million death in 2020 make cancer continually become the disease that dominates death around the world.[1] Cancer is caused by genetic mutation and normal cells thus transform into tumor cells. Since tumors always contain many types of genetic mutation and due to the mutational heterogeneity, different subgroups have different sensitivity to chemotherapy drugs. Although the use of chemotherapy drugs may temporarily suppress tumors, the selective pressure formed by them makes the drug-sensitive subgroups gradually disappear and become insensitive. The subgroup reproduces and causes tumor recurrence or metastasis. Therefore, more attention should be paid to heterogeneity to enable personalized cancer treatment.[2] That is also to say, identification and classification of the particular type of gene mutation from the clinical documents that cause cancer are essential. However, this process suffers from multiple drawbacks.[3] First, the identification of genetic mutation requires professional knowledge in gene and medicine science, and these constraints shut many ordinary labors out. Also, the interpretation of those clinical texts is based on the subjective perspective from person to person even though they are professional, and it always depends on the situation due to the huge difference between patients. Moreover, plenty of works are required for genetic mutation classification if the doctor is dealing with daunting denotation work manually since it is complicated and extremely time-consuming.

1.2 Motivation and Project Objectives

Tumor diagnosis and treatment always require precisely clinical documents. Also, distinguishing between mutations that promote tumor growth (drivers) and neutral genetic mutations (passengers) remains a challenge.[4] As a result, most interpretations of clinical literature still need to be handled manually.[5] Since this process is less efficient and subjective, new techniques, thanks to the development of NLP (Natural Language Processing) techniques, have emerged to address the gene classification problems among clinical evidence.[6] Generally, clinical document research mainly focuses on the words used (medical concepts, notations), sentence connections, and semantic attributes (describe the condition of patients). Although some of the studies try to put more effort to evaluate patients or population level, the evaluation approaches are often inconsistent due to the difference of objectives and methodology priorities.[7] Therefore, to face all the challenges mentioned before, applying different NLP techniques to the clinical document, and finding out some appropriate methodologies to automatically classify the genetic variations become critical and valuable. In the future, with the development of the application of machine learning and deep learning methods in clinical text, personalized medicine will be facilitated and benefit cancer treatment.

The ultimate purpose of this project is to develop some classification models to give the classified result of the gene mutation classes and realized personalized medicine for cancer treatment. Based on the clinical evidence, different genetic mutation types are asked to be classified by referring to the information given in the gene variation document. To achieve this goal, a powerful with high accuracy machine learning or deep learning model is required.

1.3 Related Work

Many studies have done in-depth analysis on the classification of genetic mutation texts in medical texts. Nowadays, the most popular classification model can be divided into machine learning method and deep learning method. Since the twentieth century, many new models are developed so that they can handle the classification of natural language better.

1.3.1 Machine Learning in Classification

With the development of the computation, instead of dealing with the clinical documents manually, the emergence of the traditional machine learning method can be seen as a great improvement.[8] Mark Singh et al. used a naive Bayesian classifier, which was widely used in the early 1960s, to accurately screen clinical reports and got detect abnormal radiology results. They conclude that it can help doctors review radiology reports more effectively.[9] In the 1990s, a powerful mode SVM was developed quickly and is still popular as a classification model by now. Adam Wright et al. in 2013 successfully get a good result by using a Support Vector Machine-based classifier by identifying EHR (Electronic Health Record) progress documents that are related to diabetes.[10]. In 1995, tree structure base classification methods came out, where random decision forests proposed by Tin Kam Ho increase the generalization accuracy for tree-based classifiers.[11]. Then, followed by the development of the Gradient Boosting Decision tree that was proposed by Jerome H. Friedman from Stanford university in 2001, the tree-based method greatly advanced.[12]. However, the cons for GBDT is that the computation cost is too high when facing modern industry data. Hopefully, an optimized GBDT based method XGboost was proposed in 2016, it greatly increases the speed and performance due to its improvement on loss function, avoiding overfitting, and computing.[13] Gupta et al. did a great work by using several machine learning models like random forest and XGboost to do the gene classification and reach a good results.[5]

However, in some situations, the traditional machine learning methods also get low accuracy. More advanced machine learning techniques have been proposed to face those challenges. The emergence of LightGBM (Light Gradient Boosting Machine) in 2017 proposed by Microsoft has great improvements in dealing with clinical documents.[14] It is intended to design for satisfying the needs of the industry to pursue a higher accuracy and shorten the calculation time of the model. It can also ensemble multiple learning classifiers to deal with more complex literature. Xuan Qin et al. proposed an ensemble Light GBM learning method, which integrates multiple BERT (Bidirectional Encoder Representations from Transformers) models to improve the classification efficiency of titles and abstracts filter.[15] The drawbacks of machine learning techniques is also obvious. Feature extraction done by experts in the medical field is needed and thus it will take plenty of labor cost.[16]

1.3.2 Deep Learning in Classification

Deep learning can be seen as an essential branch of machine learning. Instead of extracting features from the data set, which is part of the procedure in machine learning, deep learning can automatically learn those features and classify them to get appropriate results and save a huge amount of time.

Based on the original multiple layer perceptron, RNN (Recurrent Neural Network) was proposed in the 1980s that intend to solve the data with time and sequence. Meenu Gupta et al. also applied the Word2Vec text transformation model and use RNN (Recurrent Neural Network) model and finally get a well-performed result than other proposed classifiers.[5] Later in 1997, a modified type of RNN, LSTM (Long Short-Term Memory) was designed for combining the long and short-term memory by using the gate structure, and also solving the vanishing gradient problems.[17] Tang et al implement a modified Bi-LSTM-CRF model to do de-identification of the clinical documents and reach a great result. In 2014, GRU (Gate Recurrent Unit), a new type of RNN model that makes some modifications based on LSTM, was proposed to improve the computation performance and keep a relatively good result. In the same year, Attention context techniques were used in RNN in dealing with the images to make attention popular.[18] It enables the neural network to focus more on the important information while lowering the weight with the meaningless part. Later, many recurrent models combined with attention were proposed in solving the text classification.

For the development of CNN, though it is originally applied in images, some research based on medical documents is also finished. When Yujia Bao et al. study the Medical documents that are related to cancer susceptibility genes, CNN (Convolutional Neural Network), as one of the important deep learning methods are implemented. They conclude that it can help doctors and researchers get to acquire the most recent updated knowledge among medical literature in the field of gene mutation.[19]. Also, in Yanshan Wang et al. works, CNN (Convolutional Neural Network) express more advantages in proposing a clinical text classification paradigm. It shows a higher accuracy rate compared to Random Forest, SVM (Support Vector Machine), and MLPNN (Multilayer Perceptron Neural Networks).[20]

In 2018, a new deep learning method called BERT (Bidirectional Encoder Representations from Transformers) was proposed by Google. The unique feature of BERT is its deep bidirectional architecture, which only requires an additional output layer to fine-tune the pre-training BERT to meet various tasks, and there is no need to modify the model for specific tasks. [21] Later, in Yuhan Su et al. research, they applied a modified BERT model on genetic mutation classification and come up with the conclusion that BERT is applicable in dealing with clinical evidence and can contribute a lot in speeding up the diagnosis and treatment of cancer. What is more, the more precise and personalized medicine treatment will be facilitated.[22] However, in Stephen Wu et al. methodical review of the application of deep learning techniques among clinical evidence, they point out that there is still a long way for deep learning to go in the future.[23]

1.3.3 Discussion

From the related work above, we can see that there are some models with good performance in both machine learning and deep learning method. For example, LightGBM shows a high performance when dealing with the clinical texts, the RNN model is good at solving the text with sequential information, BERT achieves a high accuracy when classifying the gene variants. All of the models have its strength and weakness, therefore, how to appropriately use different techniques to achieve better results become critical.

1.4 Models Overview Diagram

In this project, two models will be proposed to solve the gene variants classification problems. For the first machine learning model, we will use word2Vec + LightGBM. For the second deep learning model, we will use word embedding + Bidirectional GRU with attention context. The whole process of this project is listed below in Figure 1.1.

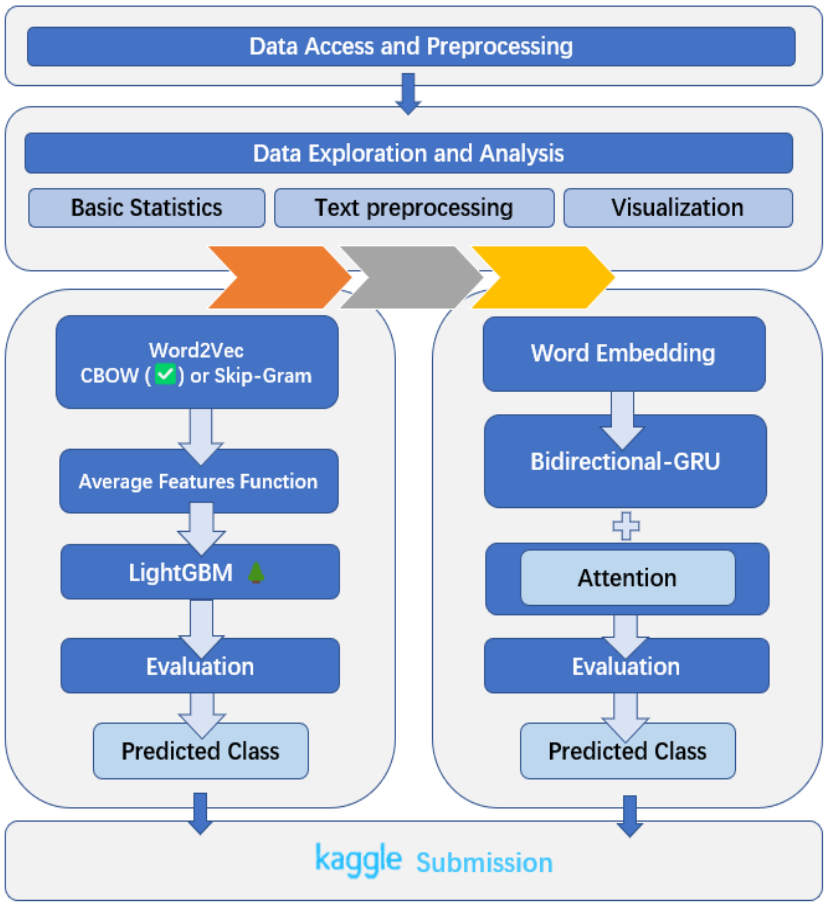


Figure 1.1: Project Overview Diagram

Chapter 2

Data Exploration and Analysis

2.1 Dataset Descriptions

In 2017, Memorial Sloan Kettering Cancer Center (MSKCC), which is the most renowned hospital in the tumor treatment area, released a competition that focus on genetic mutation classification to achieve personalized medicine treatment. The dataset they provide is all manually annotated by the doctor with proficient knowledge in cancer treatment. There are two different kinds of documents, clinical text and variation type of different genes. Also, according to the different combinations of genes and variations, the annotated classes are given. The texts are the clinical records or evidence are written by the doctor when diagnosing their patients and the gene variations are marked by those doctors based on their expertise skills.

There are two types of CSV files in this project. One of them contains clinical texts and the other one contains the genes and variation information.

For the training set, the file that contains the genetic mutation classification has four features in total. "ID" work as the identity of information for each corresponding text in another file, "Gene" gives the information of which specific genetic mutation is located, "Variation" provide the different type of mutation, and "Class" are the label of the final classification of this genes mutation. Also, Nine classes in total are provided, and our goal is to predict the class result by using our models. For the file containing the clinical texts, there are two features in total. The first one is "ID", which is the same as the first file. Another feature is the clinical evidence for each ID, which contains 3321 observations in total. For us, we need to do natural language processing to deal with those texts and train our model to get the final classification results.

For the testing dataset, there are 5668 observations in total and all the information is the same as the training dataset except for the unlabeled classes. This dataset will be used as the validation set in this project.

For the Stage 2 testing dataset, though all the information is the same as the testing set, which does not contain the classification labels, there are only 986 observations, and this data set will be used to predict the final prediction results. And we can submit the final classification result on Kaggle to check the score and loss.

2.2 Data Analysis

Before we go through our text data, let us do some data exploration so that we can have an overview of our dataset. Since there are Nine classes of genes variation, we would like to see the frequency of each class by plotting the distribution of genetic mutation classes in a histogram. From

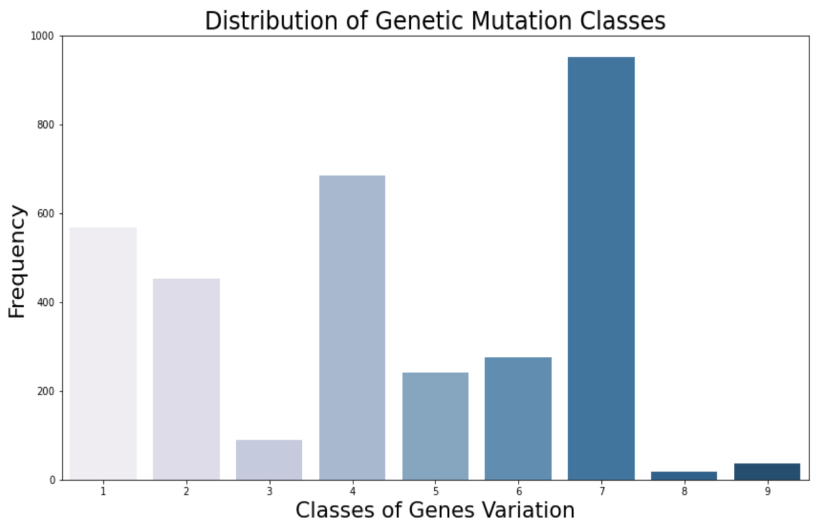
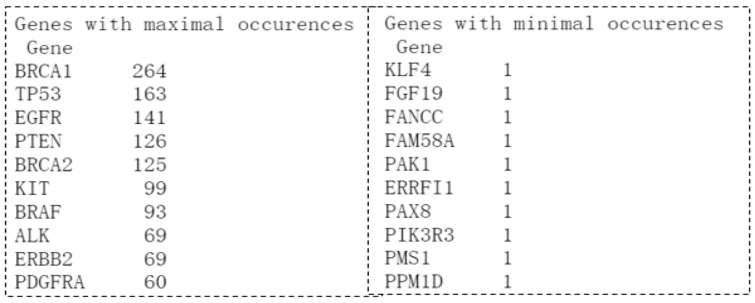


Figure 2.1: Distribution of Genetic Mutation Classes

Figure 2.1, we can see that Nine classes are shown, Class 7 have the highest frequency and class 3, 8, 9 have a relatively lower frequency.



```
Genes with maximal occurences    Genes with minimal occurences
 Gene                             Gene
BRCA1      264                   KLF4      1
TP53       163                   FGF19     1
EGFR       141                   FANCC     1
PTEN       126                   FAM58A    1
BRCA2      125                   PAK1      1
KIT         99                   ERRFI1    1
BRAF        93                   PAX8      1
ALK         69                   PIK3R3    1
ERBB2       69                   PMS1      1
PDGFRA      60                   PPM1D     1
```

Figure 2.2: Top 10 genes with maximal and minimal occurrences

There are not only nine kinds of gene mutation classes, different genes and mutation types combine to decide which specific 1 out of 9 classes there are belong to. After statistics, the number of unique genes is 1522,

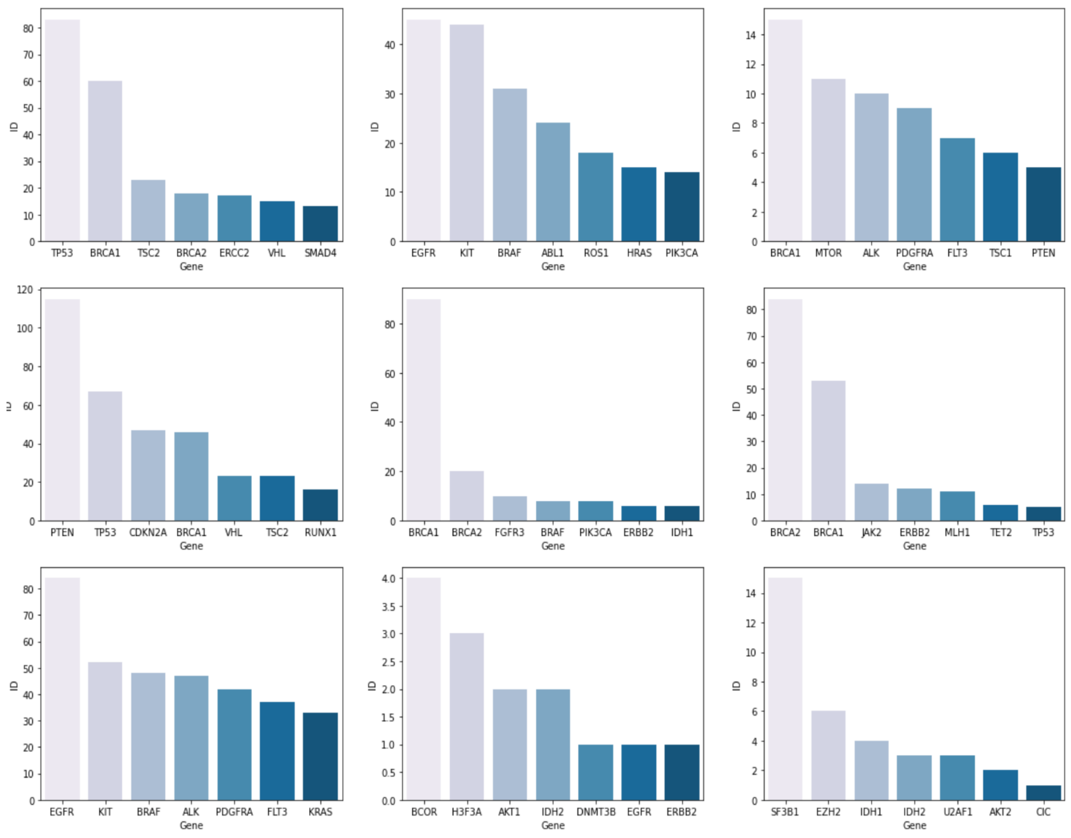


Figure 2.3: The distribution of gene in different classes

the number of unique variations is 347. Figure 2.2 shows the top 10 genes with maximal occurrences as well as minimal occurrences so that we can have a view of the number of the type of the most significant gene. And figure 2.3 shows the distribution of genes in 9 different classes. We can get the conclusion that some of the genes are highly dominating in their class. For example, Gene BRCA1 has the highest frequency and dominates in class 5.

2.3 Text Preprocessing

When dealing with text data, text preprocessing is always the most important part that we should consider. Firstly, tokenization is applied to separate the sentence from words. Then, the removal of punctuation and lower casting have proceeded. Also, some relatively meaningless stop words are removed. For example, "a", "and", "but" and so on.

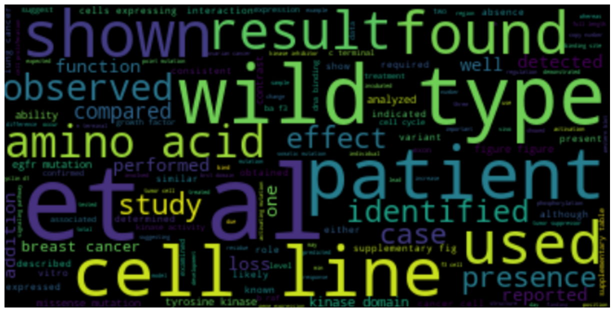
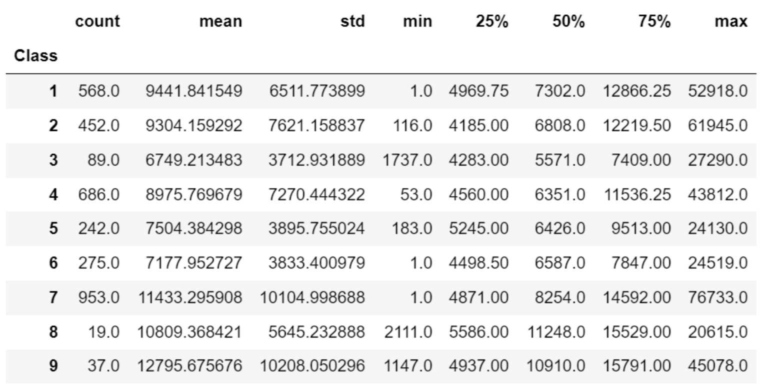


Figure 2.4: Word Cloud

Some word statistics are also implemented, including word count, text length, and word-cloud plot to show the frequency of the words. From figure 2.4, we can find that words like "patient", "cell", "line", "amino", "acid" are frequently used in our clinical documents. Besides, text length distribution is also visualized to see the text count for each class. Also, figure 2.5 shows the statistics description for Word count. From figure 2.6, we can see that some texts in class 7 have a huge amount of words compared to the classes like 3,4,5,6, and 8. The average word count for class 7 is about 11000 words, and the highest one reaches 76733-word counts. Besides, most of the text in class 3 is around 7000 words are also observed.



	count	mean	std	min	25%	50%	75%	max
Class								
1	568.0	9441.841549	6511.773899	1.0	4969.75	7302.0	12866.25	52918.0
2	452.0	9304.159292	7621.158837	116.0	4185.00	6808.0	12219.50	61945.0
3	89.0	6749.213483	3712.931889	1737.0	4283.00	5571.0	7409.00	27290.0
4	686.0	8975.769679	7270.444322	53.0	4560.00	6351.0	11536.25	43812.0
5	242.0	7504.384298	3895.755024	183.0	5245.00	6426.0	9513.00	24130.0
6	275.0	7177.952727	3833.400979	1.0	4498.50	6587.0	7847.00	24519.0
7	953.0	11433.295908	10104.998688	1.0	4871.00	8254.0	14592.00	76733.0
8	19.0	10809.368421	5645.232888	2111.0	5586.00	11248.0	15529.00	20615.0
9	37.0	12795.675676	10208.050296	1147.0	4937.00	10910.0	15791.00	45078.0

Figure 2.5: Statistics description for Word count

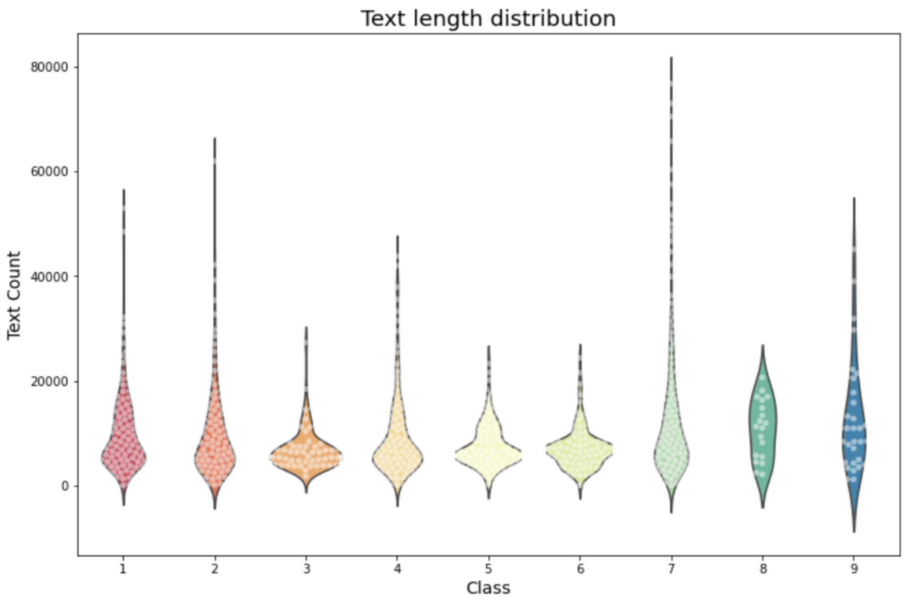


Figure 2.6: Text length distribution

Chapter 3

Methodology

3.1 Overview

There is two classification model that will be proposed in this project. The first one is the machine learning method LightGBM (Light Gradient Boosting Machine), which is an improved model of GBDT (Gradient Boosting Decision Tree model). The second one is the deep learning method RNN (Recurrent neural network). We will choose GRU (Gate Recurrent Unit) cells with a bidirectional layer and add attention context techniques as our final model. In this section, we will discuss these two models from the very beginning in a more understandable way, and show how these two models are applied to our classification.

3.2 Word Embedding

Since we cannot simply use the natural language as the input of the model, some transformation is needed so that we can let the computer understand the data that we are going to use before the classification.

In this project, for Model 1 - LightGBM, the Word2Vec Model is implemented and we use the average feature vector function to get the matrix representation of the words from word2vec and use it as the input of LightGBM. For Model 2 - Bi-GRU + Attention Model, we use the word embedding model from Tensorflow, where the input of model 2 is a tensor with multiple layers of the matrix representation of words.

3.2.1 Word2Vec + Average Feature

Work2Vec is a prevalent model with many applications. For example, it can be applied in a classification problem, clustering problem, calculation of word similarity, and so on. In this section, we will briefly introduce how the CBOW (Continues Bag of Words) model and Skip-gram (Continuous Skip-Gram) model works in generating the word vector that will be used in the classification model.

CBOW and Skip-gram have different ideas when predicting the word vector. For a CBOW model, the input is a continuous bag of words from a sentence except for the word that we want to predict. We use the vector form of the words as the input and get the predicted output (also in vector form) through a neutral network. Similarly, for a skip-gram model, the input is a central word after encoding, and through the neutron, network to get the predicted words in the sentence except for the central word. For the structure of the neural network, we will explain it further in the

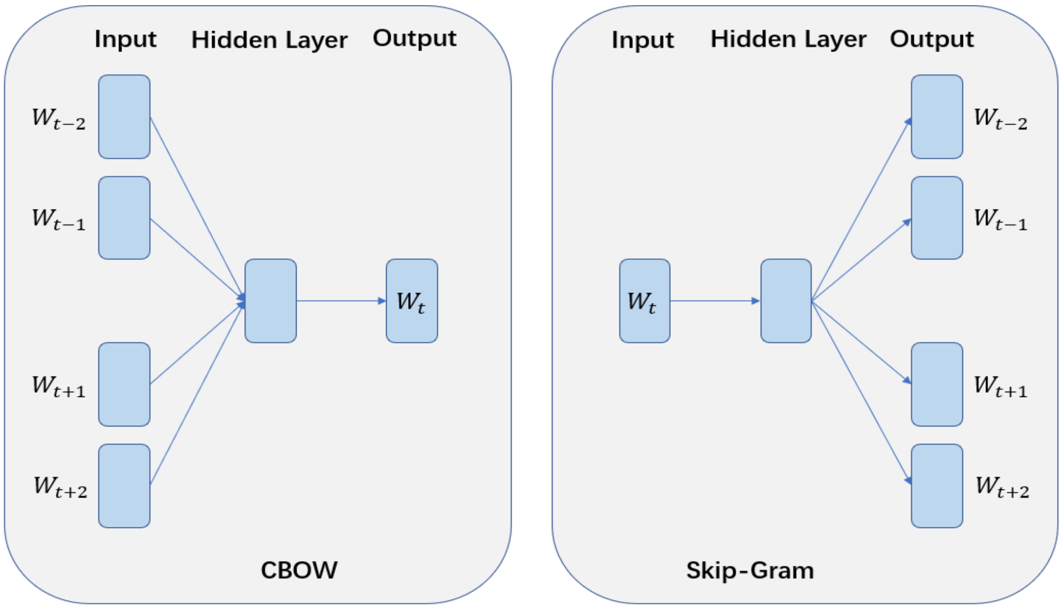


Figure 3.1: CBOW and Skip-Gram Structure

following chapter when we talk about the multiple layer perceptron.

In a classification problem, the input is our texts, which contain many sentences and paragraphs, we use those texts as the input of the Word2vec model and get the word vector from the model parameter is the vector representation of the texts. Here, we need to know that the word vector we get from this Word2Vec model is the weight matrix of the network. When we get the vector representation of the text, we find that if we would like to use the LightGBM as the model, we need to make sure that the input size of the word-represented matrix should equal to (text length x embedding dimension) so that it fits the classification label in the training set. Therefore the average feature function is used so that we can generate an input matrix with the correct size by lowering the number of word vectors in the weight matrix to the length of the text. For example, we set the word embedding dimension, which is the dimension of the hidden layer to be 200, and the shape of the weight matrix we get from the input side is 5852 times 200. Since there are 3321 rows of text data in our training set and there is 3321 corresponding label that we need to train. So What the average feature function do is to shorten the 5852 rows in the weight matrix to 3321 rows by calculating the average of the word vector to make sure the input size of the LightGBM model is correct.

3.2.2 Word Embedding from Keras

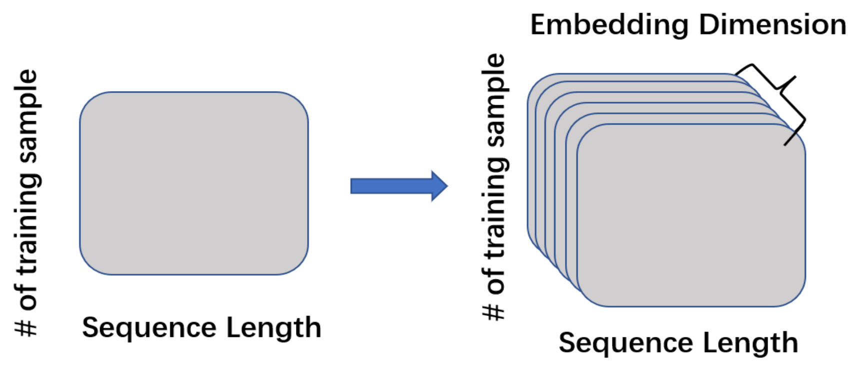


Figure 3.2: Word Embedding

The representation matrix after the average feature function has some weaknesses since it ignores so much important information from the text. When we are using a deep learning method that we propose in model 2, we generate a vector matrix with multiple layers to keep more information from the text.

To explain it further, there are two steps to get the result. First, we should use the Tokenizer to change texts to sequences, which means for each training sample, we will get a corresponding sequence. Since each texts sample has different words, the sequence size for each of them may not be equal. Therefore, we need to do the second steps, which is padding. Based on the sequence length that we would like to set, we will pad those sequences to be the same length. Then, based on the total vocabulary size, pre-set embedding dimension, and the sequence length for each training sample, we can finish the word embedding. Notice that the pre-set embedding dimension is the dimension that we would like to give for each word in the sequence. Therefore, the size of the vector representation of a text should become cubic in three dimensions, where the shape should be the number of observations in the training sample times the sequence length times the embedding dimension. Figure 3.2 shows the process more clearly.

3.3 Model 1 - LightGBM

3.3.1 Background information

LightGBM is a kind of GBDT (Gradient Boosting Decision Tree) model. Therefore, to have a better understanding of LightGBM, we can briefly introduce the concept of Gradient Boosting Decision Tree first.

To begin with, we need to know that GBDT belongs to the decision tree model. For a classification problem, the decision tree, which has a top to a button tree structure, has many criteria in their nodes, and those criteria in the parents' node can split the data into left child node and

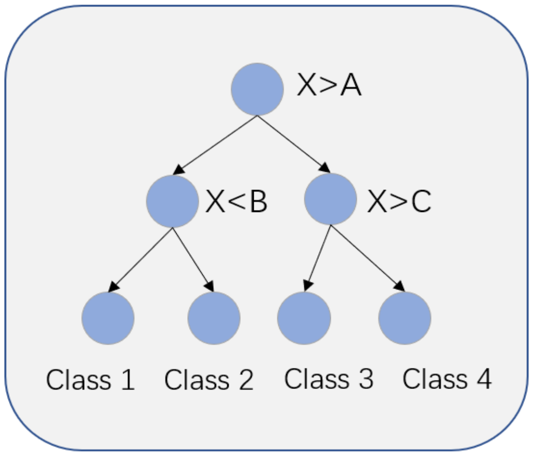


Figure 3.3: A Simple Decision Tree Model

right child node to achieve the classification purpose. While the gradient boosting decision tree is an Ensemble Learning method in machine learning. Generally speaking, for a classification problem, we need to calculate the log-likelihood for each observation as the first step, generating tree models that fit the data and calculating the residuals for each of the observations. Then, update the predicted result and reiterate the above procedures. The basic structure of the model is shown in figure 3.4. We will write down the GBDT algorithm for multiple classification problems to explain it further.

After talking about the GBDT model, we can go further to introduce the LightGBM model. A common machine learning method, or deep learning method, has less restriction when training the huge amount of data from the industry by using the mini-batch method. However, for a GBDT model, since it applies a level-wise tree growth algorithm, it is required to go through the whole data set for each iteration, which will

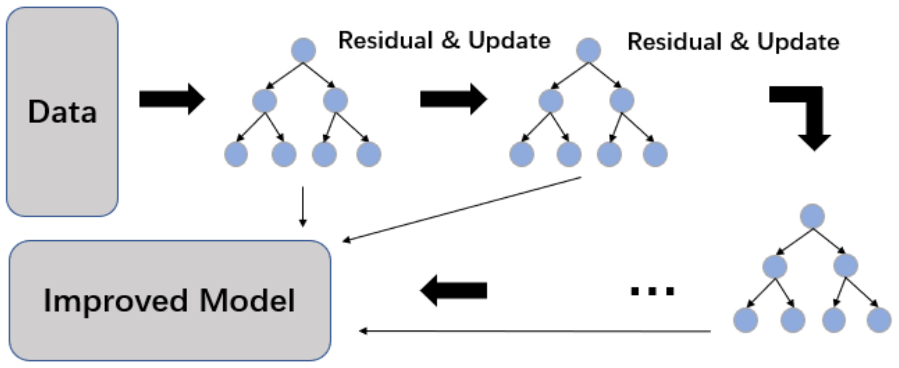


Figure 3.4: Gradient Boosting Decision Tree Model

lead to resource exhausted problems for the computer. Also, it is time-consuming to split the data for several training steps in the industrial application. Therefore, a highly efficient model LightGBM that applies a leaf-wise tree growth algorithm came out aiming to solve those problems.

3.3.2 Basic concept and Algorithm

In this section, we will go through the main idea of the Four important algorithm proposed in the LightGBM to see why it is a good model compared to the GBDT. However, since our project is focused on the application of LightGBM as a real classification problem, we will talk more about the understanding of the algorithm instead of how the coding works inside LightGBM.

Histogram-based Algorithm

Generally speaking, the design of this histogram-based algorithm is to make the storage of data easier and let the computation more efficient.

Also, the model will become more robust. The first step of the histogram algorithm is to determine how many bins are required for each feature and assign an integer number to each box. Then, we need to divide the range of floating points into several intervals, where the number of intervals should equal the number of the bin. Third, update the sample value in the bin to the new value corresponding to each bin. Finally, we have done the simpled of large-scale data to a histogram-based dataset. At this time, the features have become discrete and easy to store and have a low cost of computation. For the weakness of this algorithm, the splitting point of the decision tree is not as easy to find as GBDT. However, in the structure of gradient boosting, the precise of the splitting point is not so important, even in this case we can avoid over-fitting to some degree.

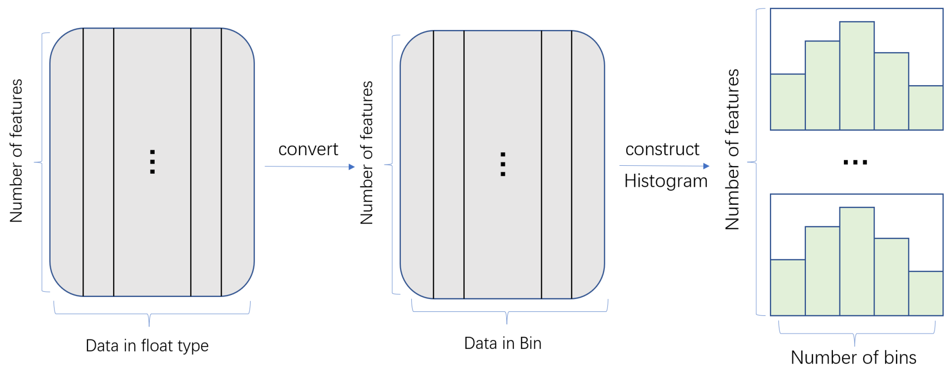


Figure 3.5: Histogram-Based Algorithm

Gradient-based One-Side Sampling

The motivation for using the Gradient-based One-Side Sampling is to have a compromise between the learning accuracy for the decision trees and reducing the size of the data set.[14] To be more specific, we would like to reduce the samples with the smaller gradients and use the remaining part of the samples to calculate the information gain in the LightGBM. Here is the problem we may face, samples with small gradients have relatively small training errors, indicating that the data has been learned by the model very well. However, if we discard this part of the data with small gradients directly, we will change the distribution of the data and will affect the accuracy of the training model. To avoid this problem, the GOSS algorithm is proposed.

When we are doing sampling, it is expected to discard those data that contribute less to the information gain. Since samples with large gradients have more impacts on the information gain, the GOSS Algorithm first sorts all the values of the features that are to be split in descending order of the absolute values. Then, select the top a*100% samples. After that, b*100% samples with a lower gradient will be randomly selected and will be amplified by multiplying with a constant (1-a)/b in the calculation of information gain. As a result, the distribution of the original sample data with a lower gradient will be discarded with less cost and the learning accuracy will not be affected so much at the same time.

Greedy Bundling

Greedy bundling aims to reduce the number of features to increase computation efficiency. The main idea of greedy bundling is to shrink the dimension of the features by bundling them together. In the high-dimensional data, especially when we are doing the encoding, sparse matrix is a very common form. However, we know the reason that we call it a sparse matrix is that it has great size but with less information. It is easy to find some of the features that are mutually exclusive so that we can bundle them together to reduce the dimension and optimize the time cost for the algorithm. The greedy bundling algorithm solves the exact problem about which kind of features can be bundled.

Merge Exclusive features

For merge exclusive features algorithm, it completes the remaining jobs of the greedy bundling algorithm. When we figure out what features should be bundled together, we need to construct the bundling part by using a new algorithm. What needs to be mentioned after merging is that we need to make sure the data in the bundled features can still be recognized. For the first algorithm, since we have already transformed the float data into different bins, it is a good idea for us to make the merged features set in the bins. To achieve this goal, we can add some offset on the original data to ensure that the range of two exclusive features is merged safer.

3.4 Model 2 - Bi-GRU + Attention

3.4.1 Basic Concepts in Neural network

In this section, we will talk about some basic concepts in a neural network so that we can have the prerequisite knowledge to understand Recurrent Neural networks and their improved version.

Perceptrons and Sigmoid

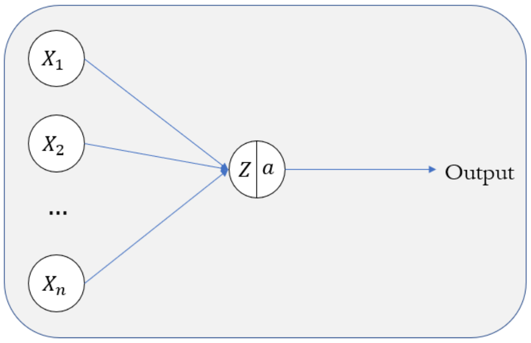


Figure 3.6: A Simple neutral network

To begin with, perceptrons can be seen as the fundamental of a neural network. The basic structure of perceptrons can be described as a transformer from the input to the output. Let's say a linear model, $X_1, X_2, X_3, ..., X_n$ is the input of the perceptron, b is the bias, and Z is the output. What needs to be mentioned is that we can give the input with different weights to decide which of them are more important that contribute to the result Z. Therefore, we can write a simple linear equation 3.1. We can go further by setting a threshold. If the output Z is larger than a threshold that we set, we label them with 1, otherwise, label it with 0. At this time, we finish a simple classification model.

$$Z = \sum_{i=0}^{n} W_i \cdot X_i + b \tag{3.1}$$

For the sigmoid, it makes some small changes on the output Z from equation 3.1 by using the sigmoid function 3.2. The sigmoid function is continuous, monotonic increasing, and differentiable. It enables us to get a range from 0 to 1 to represent the probability so that we can make some connection to some distribution. Also, a great propriety show in equation 3.3 that the derivative of a sigmoid function is equal to itself multiplied by one minus itself, which can help us calculate its derivative more easily. In the neutral network, we always need an activation function for each neuron, which can introduce some kind of nonlinear factor to face the data in the real world. And the sigmoid function is one of the most common use ones.

$$a = \sigma(Z) = \frac{1}{(1 + e^{-Z})} \tag{3.2}$$

$$\sigma^{'}(Z) = \sigma(Z) \cdot (1 - \sigma(Z)) \tag{3.3}$$

The basic structure for Multiple Layer Perceptrons

Let's move further to see the basic structure for a multiple-layer perceptron neural network as shown in figure 3.7. We will explain the structure and introduce the forward-propagation as well as the main idea of back-propagation. We can see that this network contains 3 layers in total, the input layer, the hidden layer l, and the output layer L. In reality, there may be many layers l in the hidden layer, but we will use only one for explanation purposes.

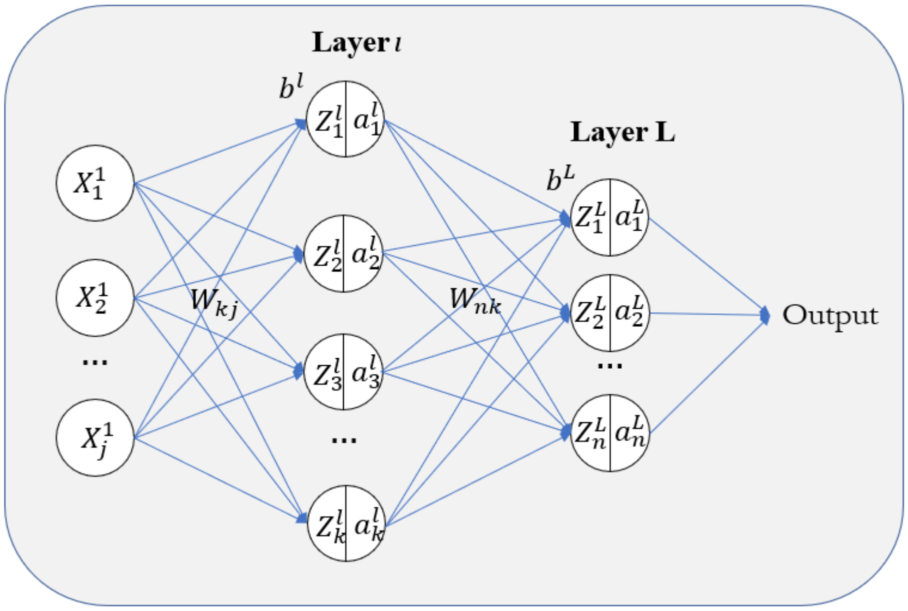


Figure 3.7: A Simple MLP network

For forward propagation, our aim is to get the output of the network and calculate the cost. The most left part $X_1^1, X_2^1, ..., X_j^1$ are the input layer of this network. W_{kj} are the weights from the first input layer to the next layer. Different from the last section, this time we write it in matrix form for simplicity and noticed that the shape for a weight matrix should contain the number of the neutron in hidden layers in a row and the number of the neutron in input in columns. The hidden layer in a neural network is always very flexible since we can change the number of layers and their dimension. But again we need to meet the dimension requirement for matrix multiplication. In the hidden layer l in figure 3.7, we can see that $Z_1^l, Z_2^l, ..., Z_k^l$ are the input in this layer, and Z^l in vector form should equal to ${W_{kj}}^l$·X^1 + b^l, where b_l is the error in the layer. After activation we can get the output $a_1^l, a_2^l, ..., a_k^l$. Then, the same procedure as the last steps, we multiply the output a from the hidden layer and the weighted matrix W_{nk} to get the input for the output layer L $Z_1^L, Z_2^L, ..., Z_n^L$, and their vector form $Z^L = {W_{nk}}^L$·a^l+b_L. After activation, we can get $a_1^L, a_2^L, ..., a_n^L$. In the final step, we can calculate the output C by using the cost function defined by $\frac{1}{2}\|y - a^L\|$, where y is the real labeled output.

For the backward propagation, the main idea is to figure out how the changing of the weights and biases in our network can change the cost function, which is $\frac{\partial C}{\partial W^l}$ and $\frac{\partial C}{\partial b^l}$. Based on that, we can update the Weights and biases by multiplying the learning rate to achieve better results. To calculate the partial derivative in a convenient way, we define an intermediate error $\delta_j^L = \frac{\partial C}{\partial Z_j^l}$, which is the output error from the layer l. Based on the chain rule step by step, we have the four fundamental equations for backward propagation.

$$\delta^L = \nabla_a C \odot \sigma^{'}(Z^L) \tag{3.4}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(Z^l) \tag{3.5}$$

$$\frac{\partial C}{\partial W_{jk}^l} = a_k^{l-1} \delta_j^l \tag{3.6}$$

$$\frac{\partial C}{\partial b^l} = \delta_j^l \tag{3.7}$$

In this section, we will not discuss the proof of those equations, but we need to get familiar with those equations so that we know how the rate of change of weights and biases with respect to cost works, which will provide us with the fundamental knowledge to understand the Recurrent Neural Network.

3.4.2 Recurrent Neural Network

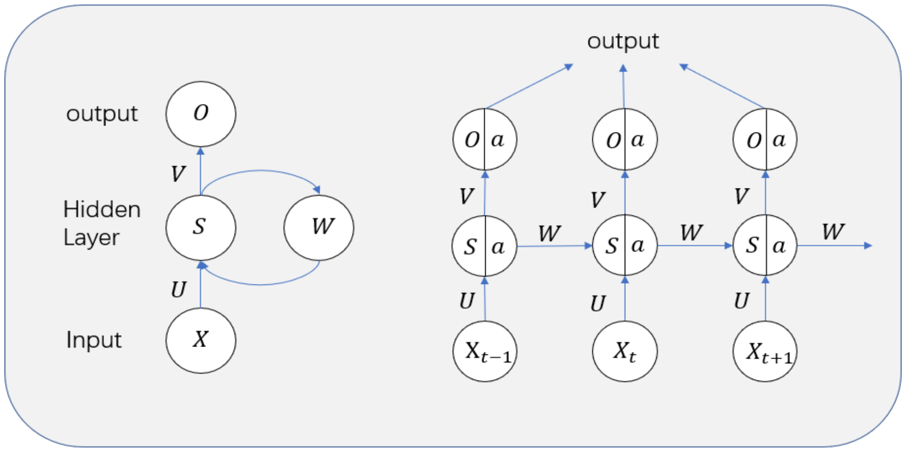


Figure 3.8: A Simple RNN structure

Recurrent Neural Network is one of the most important networks in this project, it provides us with the basic idea to build a better model. Recall from the last section, we can do some classification by using the MLP. In the real world, there are some data with sequence information that triggers our interest. For example, the stock price is affected by time, we can generate a sequence of the model on different days and the previous price will influence the price in the latter days. Or, we can think about the text, there is also some sequential pattern in our human language.

For instance, a simple sentence in the clinical evidence "The patients will take one dose of drag". From this sentence, we can see the first meaningful word is "patients", and the second one is "take". From the first two words, we can think about the possible word for the next one, "drag" is a reason that appears in a clinical document. Therefore, it is meaningful to use some model that can remember the sequence in the text to learn this text better.

From figure 3.8, we can see that there are two kinds of networks. The left one is a simpler model to show how an RNN works. From the input X to the hidden layer S. Then, recurrently calculate by weight matrix W and finally get the output. The right one is a flatter version of the previous one, it shows a sequence for t-1 to t+1, but the idea is exactly the same as the previous one. For the forward propagation and the backward propagation procedures, we can refer to the steps in MLP. The main idea is the same, except for an important note that we need to consider. Since the latest one is always based on the previous one, we need to notice the sequence effect when doing the calculation.

3.4.3 Gate Recurrent Unit

Gate Recurrent Unit is a kind of Recurrent Neutron Network. We will choose GRU as part of our model in the classification process. The purpose of the GRU is to low down the computation cost but keep a relatively good result. The basic structure of GRU is shown in the left part of Figure 3.9. The h_{t-1} and h_t are the hidden state in t-1 and t respectively. X_t and Y_t are the input and output at time t.

If we look inside the GRU model, we can have a better understanding

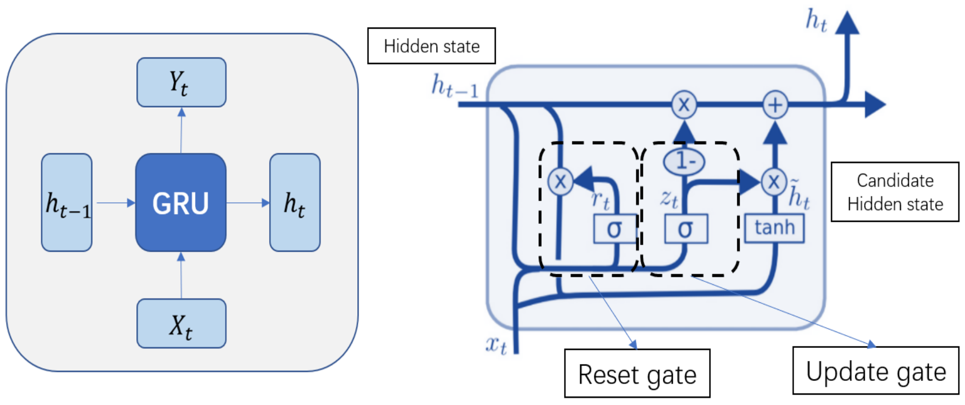


Figure 3.9: The structure of GRU

of how it works. For the right part of the figure, we can see that there are two gates called reset gate and update gate in the structure. We define the r to be the reset gate, where r = $\sigma(W_r \cdot [h_{t-1}, x_t])$, and z to be the update gate, where z = $\sigma(W_r \cdot [h_{t-1}, x_t])$. To reset the h_{t-1}, we need to use the equation $h'_{t-1} = h_{t-1} \odot r$, and then use the activation function tanh(W'_h·[r$_t$·h$_{t-1}$,x_t]) to get h', where it contains the information from the last hidden state and contain the information form the new input X_t. For the update steps, the updated h_t should equal to $(1 - z) \odot h_{t-1} + z \odot h'$. To understand this equation, we need to know that the range of z is from 0 to 1, which means the updated h_t is actually keep the $1 - z$ proportion from the last hidden state h_{t-1} and add the z proportion form the h'. To explain it further, it is some kind of balance between the "forget" and "remember", where we will remember all the things if z is 1. The strength of this kind of mechanism is that we can use only one update gate z to remember and forget the information at the same time, and the information proportion is under our control.

For the advantages, the proposed GRU will fix the vanishing gradient problem due to its gate structure. In the traditional RNN model, there may exist exploding gradient problem or vanishing gradient problem if there are too many hidden states in the structure. Gradient exploding and vanishing means the gradient will become too large or even equal to zero respectively. In this case, the model will no longer be in a learning status. In the structure of GRU, since the gates structure can help the model to have a memory of the previous hidden state, the sequence dependency problems that appeared in RNN will be fixed.

3.4.4 Bidirectional Layers

The existence of bidirectional layers is to apply the original model twice in different directions and combine them together. Traditionally, an RNN model will be applied to follow the exact sequence of the text, while a bidirectional layer adds a path from the end to the beginning of the text so that we can dive deeper into the text by providing a whole picture for the context. The basic structure is shown in figure 3.10.

3.4.5 Attention context

Attention context is another technique that will be added to our project. In natural language processing, there are many kinds of attention models. In this section, we will give a brief introduction by giving a general structure of attention that is used that combines the GRU model. The ultimate purpose of this mechanism is to help us focus more on valuable information and ignore some useless parts by giving different weights to the data. Generally speaking, there are three steps in total. First, we need

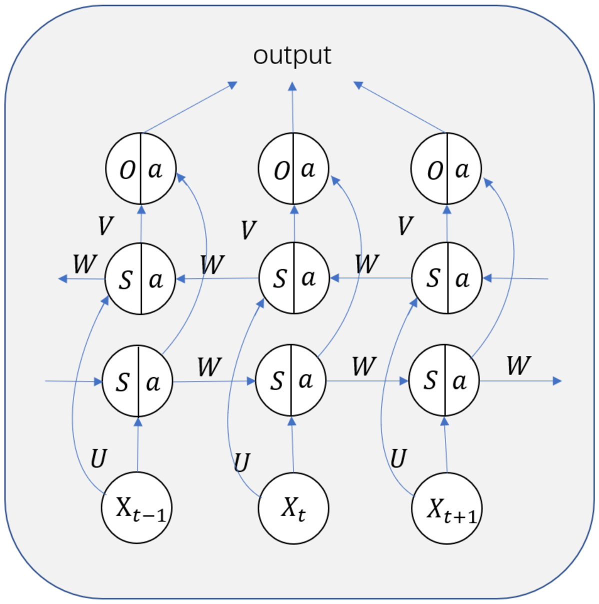


Figure 3.10: RNN with bidirectional layer

to calculate the similarity of query and keys. Then, the similarity output should be normalized by using the softmax function. Finally, calculate the summation of the multiplication of values and the similarity after softmax to get the attention value.

$$Attention(Query, Source) = \sum_{i=1}^{L} {}_x Similarity(Query, Key_i) \cdot Values_i \tag{3.8}$$

For the advantages, when we add the attention layer on our GRU model, it will give weight to different hidden states, which is very useful for focusing on the more important part of the text and will increase our model performance. On the contrary, if we only use the GRU model to do

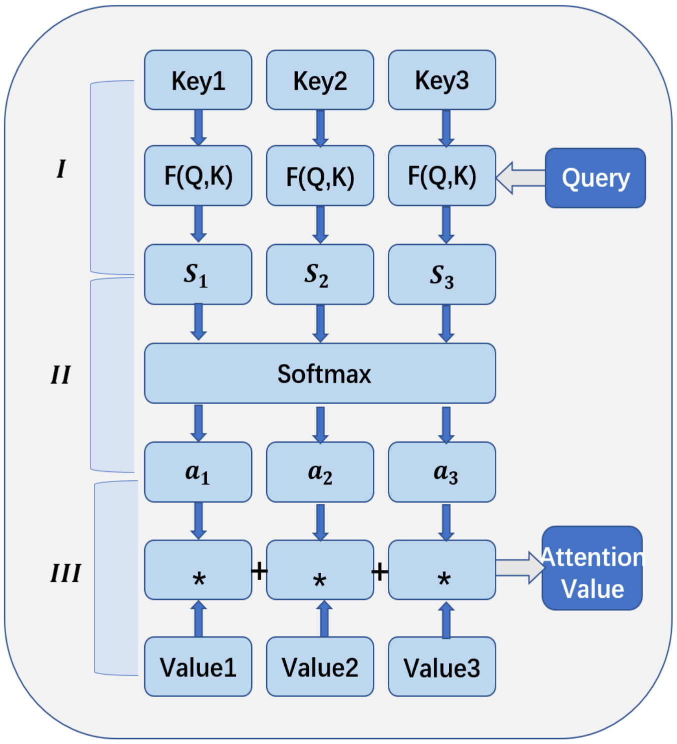


Figure 3.11: The structure of Attention Context

the classification, some of the information will be lost when we consider all the hidden states with equivalent importance.

3.4.6 Final Proposed Model

All the introduced theoretical concepts of the model provide us with great basics to construct a better model. Therefore, a GRU with a Bidirectional layer and attention layer added model was finally proposed. The overview structure is shown in figure 3.12, and we will apply this model to the classification process in the later chapter.

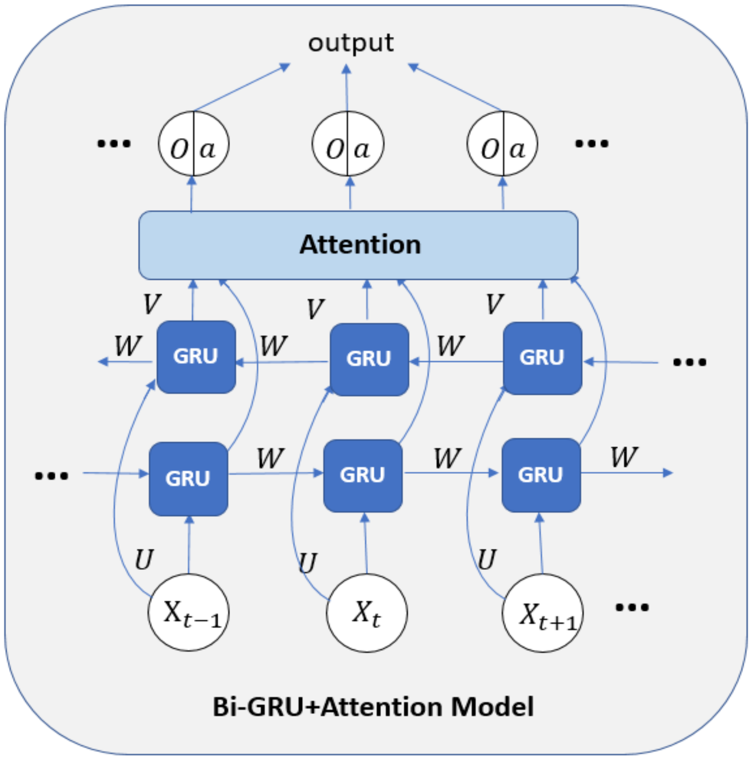


Figure 3.12: Structure of Bi-GRU+Attention Model

Chapter 4

Experiment and Results

4.1 Word2Vec + LightGBM

For our proposed Model 1, first of all, we use Word2Vec Model (we choose CBOW) from Gensim to get the vector representation of the words. Then, by applying the average feature function to correct the input size of the classification model. Then, after setting the parameters, we use LightGBM as the classification model to get the predicted results.

The parameter setting of the word2vec model, average features function, and LightGBM are listed in Table 4.1 and Table 4.2. Also, the training steps and model evaluation are shown in Table 4.3 and Table 4.4. Finally, we got a 60% accuracy for the predicted in the label of training data. After submission on Kaggle, we got a private score: 3.59873, a public score: 1.46629, and the final rank of the private score is 302 out of 1386.

Model Parameters - Word2Vec			
Model	CBOW	Model	Skip-Gram
min_count	1	min_count	1
vector size	200	vector size	200
window	5	window	5
		sg	1
Model Parameters - Average Features			
sentence	train	num_features	200
model	word2vec.wv		

Table 4.1: Parameters setting for Word2Vec and Average Features

Model Parameters - LightGBM			
boosting_type	gbdt	feature_fraction	0.9
objective	multiclass	bagging_fraction	0.8
num_class	9	bagging_freq	5
metric	multi_error	lambda_l1	0.4
num_leaves	500	lambda_l2	0.5
min_data_in_leaf	100	min_gain_to_split	0.2
learning_rate	0.1	verbose	-1

Table 4.2: Parameters setting for LightGBM

Training
Model1: Train data length: 2324
Model1: Test data length: 997
Training until validation scores don't improve for 300 rounds
[50] valid_0's multi_error: 0.437312
[100] valid_0's multi_error: 0.418255
[150] valid_0's multi_error: 0.417252
[200] valid_0's multi_error: 0.418255
[250] valid_0's multi_error: 0.412237
[300] valid_0's multi_error: 0.417252
[350] valid_0's multi_error: 0.415246
[400] valid_0's multi_error: 0.416249
[450] valid_0's multi_error: 0.406219
[500] valid_0's multi_error: 0.406219
[550] valid_0's multi_error: 0.402207
[600] valid_0's multi_error: 0.408225
[650] valid_0's multi_error: 0.402207
[700] valid_0's multi_error: 0.410231
[750] valid_0's multi_error: 0.408225
Early stopping, best iteration is:
[491] valid_0's multi_error: 0.398195
all tasks done. total time used:3.629003 s.
auc 0.4502928863833586

Table 4.3: Training for LightGBM

Model Evaluation				
	precision	recall	f1-score	support
0	0.53	0.58	0.55	170
1	0.55	0.40	0.47	146
2	0.44	0.29	0.35	28
3	0.65	0.64	0.64	208
4	0.43	0.36	0.39	72
5	0.82	0.67	0.74	73
6	0.62	0.77	0.69	287
7	0.00	0.00	0.00	3
8	1.00	0.50	0.67	10
accuracy			0.60	997
macro avg	0.56	0.47	0.50	997
weighted avg	0.60	0.60	0.59	997

Table 4.4: Model Evaluation for LightGBM

4.2 Word Embedding + Bi-GRU,Attention

For our proposed Model 2, first of all, we use Tokenizer from Kears to pad the texts to sequence for each observation. And do word embedding to make the input become a three-dimension tenor with matrix representation. Then, training with our proposed bi-directional GRU models and add attention layers to get the predicted output.

The parameter setting of the word embedding and Bidirectional GRU with Attention Model are listed in Table 4.5. Also, the model structure and training steps are shown in Table 4.6 and Table 4.7. Finally, we got an 84.52% accuracy for the predicted in the label of training data. After submission on Kaggle, we get a private score: 2.52648, public score: 1.76996, and the final rank of the private score is 91 out of 1386 when the length of the sequence parameter is 1000. Later, we found that by

Model Paraneters - Bi-GRU with Attention Model			
NUM_CLASS	9	Epoch	10
VOCABULARY_SIZE	10000	Batch size	32
SEQUENCE_LENGTH	1000/2000	Val_Epoch	3
Embedding_dim	200	Val_Batch size	32
lstm_out	64	recurrent_dropout	0.2
dense	32	dropout	0.2

Table 4.5: Parameters for Bi-GRU with Attention Model

increasing the sequence length, we will get a little bit better result in Kaggle. When increase the sequence length from 1000 to 2000, we got the private score: 2.36962, public score: 1.88202, and the final private rank is 88 out of 1386.

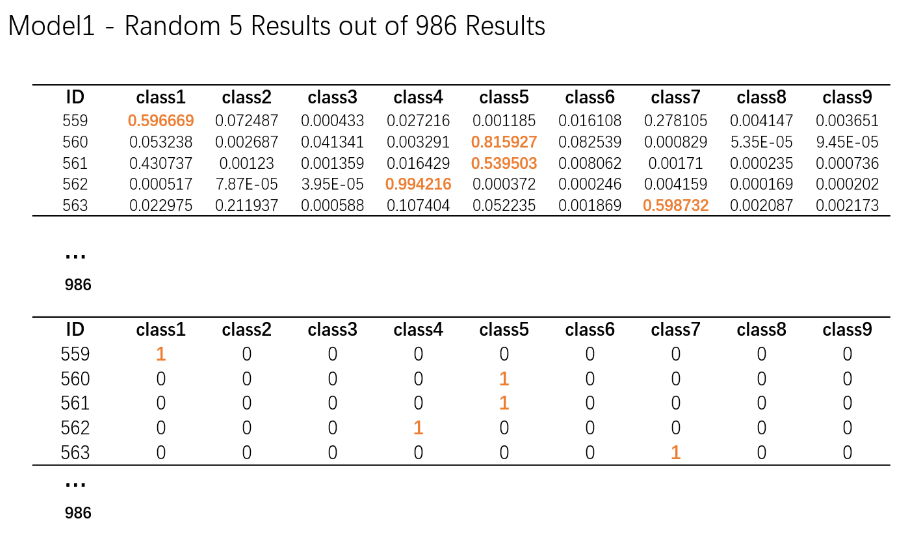
Bidirectional GRU with Attention Model			
Layer (type)	**Output Shape**	**Param #**	**Connected to**
input_1 (InputLayer)	[(None,1000)]	0	[]
input_2 (InputLayer)	[(None,1000)]	0	[]
embedding (Embedding)	(None, 1000,200)	2000000	['input_1 [0][0]']
embedding_1 (Embedding)	(None, 1000,200)	2000000	['input_2 [0][0]']
input_3 (InputLayer)	[(None,1522)]	0	[]
input_4 (InputLayer)	[(None,347)]	0	[]
bidirectional (Bidirectional)	(None, 1000, 128)	102144	['embedding [0][0]']
bidirectional_1 (Bidirectional)	(None, 1000, 128)	102144	['embedding_1 [0][0]']
concatenate (Concatenate)	(None,1869)	0	['input_3[0][0]', 'input_4[0][0]']
attention_with_context (AttentionWithContext)	(None,128)	16640	['bidirectional [0][0]']
attention_with_context_1 (AttentionWithContext)	(None,128)	16640	['bidirectional_1 [0][0]']
dense (Dense)	(None,32)	59840	['concatenate [0][0]']
concatenate_1 (Concatenate)	(None,288)	0	['attention_with_ context[0][0]', 'attention_with_ context_1[0][0]', 'dense[0][0]']
dense_1 (Dense)	(None,9)	2601	['concatenate_1[0][0]']
Total params: 4,300,009			
Trainable params: 4,300,009			
Non-trainable params: 0			

Table 4.6: Bi-GRU with Attention Model

Training for Bi-GRU with Attention Model
Epoch 1/10
- loss: 1.6689 - accuracy: 0.3797
Epoch 00001: val_loss improved from inf to 0.08144, saving model to keras_model
-loss: 1.6689 - accuracy: 0.3797 - val_loss: 0.0814 - val_accuracy: 0.0625
Epoch 2/10
- loss: 1.1092 - accuracy: 0.5929
Epoch 00002: val_loss improved from 0.08144 to 0.06775
- loss: 1.1092 - accuracy: 0.5929 - val_loss: 0.0678 - val_accuracy: 0.1637
Epoch 3/10
- loss: 0.8239 - accuracy: 0.7025
Epoch 00003: val_loss improved from 0.06775 to 0.06573
- loss: 0.8239 - accuracy: 0.7025 - val_loss: 0.0657 - val_accuracy: 0.3670
Epoch 4/10
- loss: 0.6516 - accuracy: 0.7712
Epoch 00004: val_loss improved from 0.06573 to 0.06354
- loss: 0.6516 - accuracy: 0.7712 - val_loss: 0.0635 - val_accuracy: 0.2945
Epoch 5/10
- loss: 0.5466 - accuracy: 0.7983
Epoch 00005: val_loss did not improve from 0.06354
- loss: 0.5466 - accuracy: 0.7983 - val_loss: 0.0696 - val_accuracy: 0.2096
Epoch 6/10
- loss: 0.4893 - accuracy: 0.8118
Epoch 00006: val_loss did not improve from 0.06354
- loss: 0.4893 - accuracy: 0.8118 - val_loss: 0.0704 - val_accuracy: 0.1978
Epoch 7/10
- loss: 0.4372 - accuracy: 0.8223
Epoch 00007: val_loss did not improve from 0.06354
- loss: 0.4372 - accuracy: 0.8223 - val_loss: 0.0731 - val_accuracy: 0.2971
Epoch 8/10
- ETA: 0s - loss: 0.4073 - accuracy: 0.8383
Epoch 00008: val_loss did not improve from 0.06354
- loss: 0.4073 - accuracy: 0.8383 - val_loss: 0.0720 - val_accuracy: 0.1971
Epoch 9/10
- ETA: 0s - loss: 0.3917 - accuracy: 0.8398
Epoch 00009: val_loss did not improve from 0.06354
- loss: 0.3917 - accuracy: 0.8398 - val_loss: 0.0759 - val_accuracy: 0.2041
Epoch 10/10
- ETA: 0s - loss: 0.3667 - accuracy: 0.8452
Epoch 00010: val_loss did not improve from 0.06354
- loss: 0.3667 - accuracy: 0.8452 - val_loss: 0.0765 - val_accuracy: 0.2101

Table 4.7: Training for Bi-GRU with Attention Model

4.3 Results


Model1 - Random 5 Results out of 986 Results

ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	0.596669	0.072487	0.000433	0.027216	0.001185	0.016108	0.278105	0.004147	0.003651
560	0.053238	0.002687	0.041341	0.003291	0.815927	0.082539	0.000829	5.35E-05	9.45E-05
561	0.430737	0.00123	0.001359	0.016429	0.539503	0.008062	0.00171	0.000235	0.000736
562	0.000517	7.87E-05	3.95E-05	0.994216	0.000372	0.000246	0.004159	0.000169	0.000202
563	0.022975	0.211937	0.000588	0.107404	0.052235	0.001869	0.598732	0.002087	0.002173

...

986

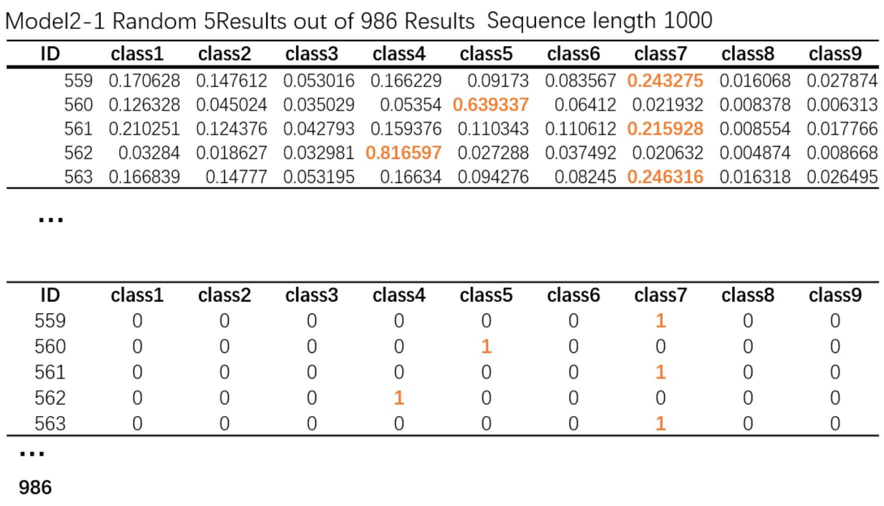
ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	1	0	0	0	0	0	0	0	0
560	0	0	0	0	1	0	0	0	0
561	0	0	0	0	1	0	0	0	0
562	0	0	0	1	0	0	0	0	0
563	0	0	0	0	0	0	1	0	0

...

986

Figure 4.1: Classification Results-1

For the classification Results, we can refer to Figure 4.1, 4.2, and 4.3. For the submission results, we make three submissions on Kaggle. For model 1 with the Lightgbm model, we get a private score: 3.59873, a public score: 1.46629, and the final rank of the private score is 302 out of 1386. For model 2 with Bi-GRU and Attention model, we get a private score: 2.52648, public score: 1.76996, and the final rank of the private score is 91 out of 1386 when the length of the sequence parameter is 1000. We get the best model with Bi-GRU and Attention model when the length of the sequence is 2000, where the private score: 2.36962, public score: 1.88202, and the final private rank is 88 out of 1386. (Note: The public score is for testing purposes in the Kaggle competition. The Private score will be considered as the useful score in the competition.)



Model2-1 Random 5Results out of 986 Results Sequence length 1000

ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	0.170628	0.147612	0.053016	0.166229	0.09173	0.083567	**0.243275**	0.016068	0.027874
560	0.126328	0.045024	0.035029	0.05354	**0.639337**	0.06412	0.021932	0.008378	0.006313
561	0.210251	0.124376	0.042793	0.159376	0.110343	0.110612	**0.215928**	0.008554	0.017766
562	0.03284	0.018627	0.032981	**0.816597**	0.027288	0.037492	0.020632	0.004874	0.008668
563	0.166839	0.14777	0.053195	0.16634	0.094276	0.08245	**0.246316**	0.016318	0.026495

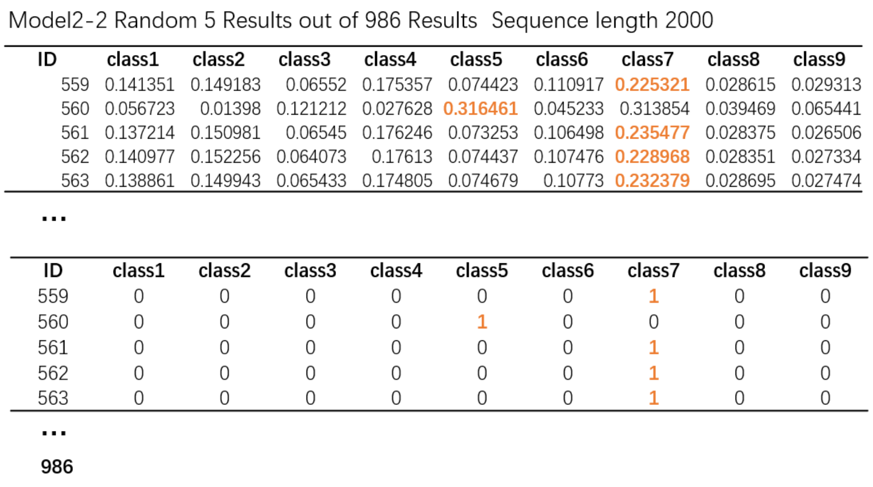
...

ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	0	0	0	0	0	0	**1**	0	0
560	0	0	0	0	**1**	0	0	0	0
561	0	0	0	0	0	0	**1**	0	0
562	0	0	0	**1**	0	0	0	0	0
563	0	0	0	0	0	0	**1**	0	0

...

986

Figure 4.2: Classification Results-2



Model2-2 Random 5 Results out of 986 Results Sequence length 2000

ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	0.141351	0.149183	0.06552	0.175357	0.074423	0.110917	**0.225321**	0.028615	0.029313
560	0.056723	0.01398	0.121212	0.027628	**0.316461**	0.045233	0.313854	0.039469	0.065441
561	0.137214	0.150981	0.06545	0.176246	0.073253	0.106498	**0.235477**	0.028375	0.026506
562	0.140977	0.152256	0.064073	0.17613	0.074437	0.107476	**0.228968**	0.028351	0.027334
563	0.138861	0.149943	0.065433	0.174805	0.074679	0.10773	**0.232379**	0.028695	0.027474

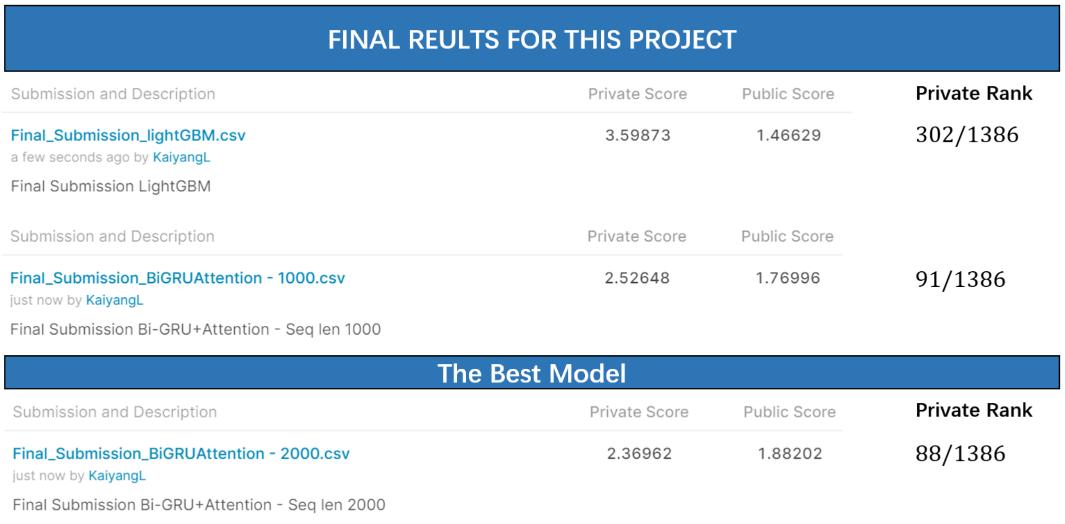
...

ID	class1	class2	class3	class4	class5	class6	class7	class8	class9
559	0	0	0	0	0	0	**1**	0	0
560	0	0	0	0	**1**	0	0	0	0
561	0	0	0	0	0	0	**1**	0	0
562	0	0	0	0	0	0	**1**	0	0
563	0	0	0	0	0	0	**1**	0	0

...

986

Figure 4.3: Classification Results-3



FINAL REULTS FOR THIS PROJECT

Submission and Description	Private Score	Public Score	Private Rank
Final_Submission_lightGBM.csv a few seconds ago by KaiyangL Final Submission LightGBM	3.59873	1.46629	302/1386

Submission and Description	Private Score	Public Score	
Final_Submission_BiGRUAttention - 1000.csv just now by KaiyangL Final Submission Bi-GRU+Attention - Seq len 1000	2.52648	1.76996	91/1386

The Best Model

Submission and Description	Private Score	Public Score	Private Rank
Final_Submission_BiGRUAttention - 2000.csv just now by KaiyangL Final Submission Bi-GRU+Attention - Seq len 2000	2.36962	1.88202	88/1386

Figure 4.4: Final Results

Chapter 5

Conclusions

In conclusion, two classification models are involved in our project. For the LightGBM model, we use the word vector from the Word2Vec model and do the feature average as the input, we successfully predict the class label. The strength of this model is very obvious, only a few seconds are needed when training the LightGBM model and at the same time get a result with relatively high accuracy. For the Bidirectional GRU model with an attention layer, we use the three-dimensional tensor as the input. Looking deeper into the model itself, thanks to the advantages of our model in many ways, we got a high accuracy up to around 85%. The gate structure from GRU will make the balance of the memory and new information mode easier and use fewer computation resources. Also, the bidirectional layer enables us to get the full picture of the sequence from two directions of the texts. Moreover, the adding of attention mechanism greatly improves the performance of our model by paying more attention to the useful information in the texts.

For the limitation, we may find an interesting phenomenon in the result given by Kaggle. The LightGBM model has the highest public score among the three submissions but gets the lowest score in the private score. To explain it further, the test data sets for public and private scores are different. The result in LightGBM shows that it can learn the text in a public dataset very well due to the potential over-fitting problems, and it cannot handle the text in a private dataset since the lack of generalization ability. For the Bi-GRU with attentions model, we make full use of the validation set and set the appropriate drop-out parameter to avoid over-fitting. However, for each run of this model, we will need a few hours to get the results. Therefore, those complex models will be greatly restrained by the computation limitation of the computer.

Nowadays, with the great help of machine learning and deep learning model in the classification of clinical documents, especially in dealing with the text that is related to the gene mutation, lots of human efforts will be avoided. In the future, some high-performance model with a lower computational cost is desirable to develop in dealing with the classification of the clinical text so that patients may receive a personalized treatment that benefits from NLP techniques and classification models.

Appendix A

Python Code

```python
# **FYP Final Version - LightGBM + (Bi-GRU and Attention)**

# # Packages Loading
import re
import time
import os
import math
import pandas as pd
import numpy as np

# Preprocess
import spacy
from scipy.stats import entropy
import string
import nltk
nltk.download('stopwords')
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk import FreqDist
nltk.download('punkt')
stop_words = set(stopwords.words('english'))
nltk.download('vader_lexicon')
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# Gensim
import gensim
from gensim.models import LdaModel
from gensim import models, corpora, similarities
from gensim.models.word2vec import Word2Vec

# Sklearn
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer, TfidfTransformer
from sklearn.cluster import MiniBatchKMeans, KMeans
from sklearn.decomposition import PCA
from sklearn.metrics import homogeneity_score
from sklearn.metrics import silhouette_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss, accuracy_score
from sklearn.svm import SVC
from sklearn.decomposition import TruncatedSVD
from sklearn.ensemble import RandomForestClassifier
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder
import scikitplot.plotters as skplt

# Modeling
import lightgbm as lgb
from datetime import datetime

# Viz
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
import seaborn as sns

# Sklearn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score

# Kears
from keras.preprocessing.sequence import pad_sequences
from keras.preprocessing.text import Tokenizer
from keras.utils import np_utils
from keras.layers.merge import concatenate
from keras.utils.np_utils import to_categorical
from keras.callbacks import ModelCheckpoint
from keras.models import load_model
from keras.optimizers import adam_v2

from keras import backend as K
import tensorflow.python.keras.engine
from tensorflow.python.keras.layers import Layer, InputSpec
from tensorflow.keras.layers import Layer, InputSpec
from keras import initializers, regularizers, constraints

# 搭建模型
from keras.models import Sequential, Model
from keras.layers import Dense, Embedding, Activation, Input
from keras.layers import Convolution1D, Flatten, Dropout, MaxPool1D
from keras.layers import  BatchNormalization
from keras.layers import Convolution1D, Conv1D,MaxPooling1D
from keras.layers import Dense, Embedding, Input, Lambda, Reshape
from keras.layers import Convolution1D, Flatten, Dropout, MaxPool1D, GlobalAveragePooling1D
from keras.layers import LSTM, GRU, TimeDistributed, Bidirectional
from keras.layers import Dense, Embedding, LSTM, GRU, Bidirectional, merge, Input, concatenate
from keras.layers.merge import Concatenate


# # Data Exploration

# ## Loading Data

train_variants_df = pd.read_csv("training_variants.csv")
train_text_df = pd.read_csv("training_text.zip",sep="\|\|",engine="python",names=["ID","Text"],skiprows=1)
test_variants_df = pd.read_csv("stage2_test_variants.csv")
test_text_df = pd.read_csv("stage2_test_text.csv",sep="\|\|",engine="python",names=["ID","Text"],skiprows=1)

val_variants_df = pd.read_csv("test_variants.csv")
val_text_df = pd.read_csv("test_text.zip",sep="\|\|",engine="python",names=["ID","Text"],skiprows=1)
val_labels_df = pd.read_csv("stage1_solution_filtered.csv")
val_labels_df['Class'] = pd.to_numeric(val_labels_df.drop('ID', axis=1).idxmax(axis=1).str[5:])
val_labels_df = val_labels_df[['ID', 'Class']]
val_text_df = pd.merge(val_text_df, val_labels_df, how='left', on='ID')

print("Train Variant".ljust(15), train_variants_df.shape)
print("Train Text".ljust(15), train_text_df.shape)
print("Test Variant".ljust(15), test_variants_df.shape)
print("Test Text".ljust(15), test_text_df.shape)
print("Validation Variant".ljust(15), val_variants_df.shape)
print("Validation Text".ljust(15), val_text_df.shape)

train_variants_df.head()
test_variants_df.head()
val_variants_df.head()
```

```
train_variants_df['Class'].value_counts()
train_text_df
test_text_df
val_text_df

# ## Distribution of genetic mutation classes

plt.figure(figsize=(14,8))
sns.countplot(x="Class", data=train_variants_df, palette="PuBu")
plt.ylabel('Frequency', fontsize=22)
plt.xlabel('Classes of Genes Variation', fontsize=22)
plt.title("Distribution of Genetic Mutation Classes", fontsize=24)
plt.show()

gene_group = train_variants_df.groupby("Gene")['Gene'].count()
minimal_occ_genes = gene_group.sort_values(ascending=True)[:10]
print("Genes with maximal occurences\n", gene_group.sort_values(ascending=False)[:10])
print("\nGenes with minimal occurences\n", minimal_occ_genes)


# ## Distribution of Gene in Different Classes

fig, axs = plt.subplots(ncols=3, nrows=3, figsize=(20,15))

for i in range(3):
    for j in range(3):
        gene_count_grp = train_variants_df[train_variants_df["Class"]==((i*3+j)+1)]
        .groupby('Gene')["ID"].count().reset_index()
        sorted_gene_group = gene_count_grp.sort_values('ID', ascending=False)
        sorted_gene_group_top_7 = sorted_gene_group[:7]
        sns.barplot(x="Gene", y="ID", data=sorted_gene_group_top_7, ax=axs[i][j],palette="PuBu")


# Some points we can conclude from these graphs:
#
# BRCA1 is highly dominating Class 5\
# SF3B1 is highly dominating Class 9\
# BRCA1 and BRCA2 are dominating Class 6

# # Data Preprocessing

# ## Text Preprocessing

# **Steps**\
# **1. Tokenization**\
# **2. Removal of punctuations**\
# **3. Lemmatization**\
# **4. Removal of stop words**\
# **5. Lower casting**\
# **6. Special consideration for clinical text***

# Remove punctuation
train_text_df['Text_processed'] = train_text_df['Text'].map(lambda x: re.sub('[\',.!?*]', '', str(x)))
# Convert the titles to lowercase
train_text_df['Text_processed'] = train_text_df['Text_processed'].replace(r'\n',' ', regex=True)
train_text_df['Text_processed'] = train_text_df['Text_processed'].map(lambda x: x.lower())
# Print out the first rows of papers
train_text_df['Text_processed'] = train_text_df['Text_processed'].apply(lambda x: x.strip())
train_text_df

```

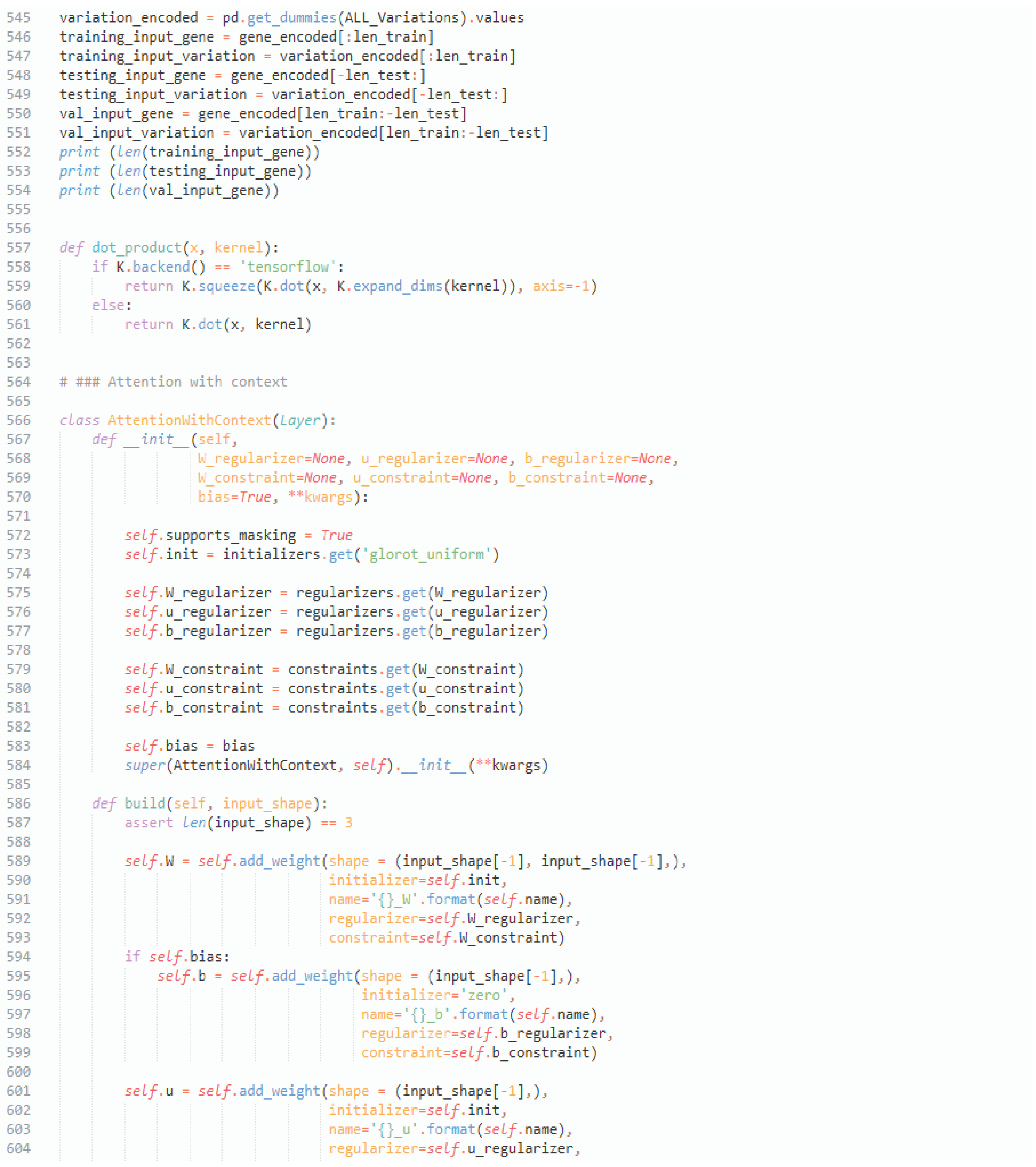
```python
# Remove punctuation
test_text_df['Text_processed'] = test_text_df['Text'].map(lambda x: re.sub('[\',.!?*]', '', str(x)))
# Convert the titles to lowercase
test_text_df['Text_processed'] = test_text_df['Text_processed'].replace(r'\n',' ', regex=True)
test_text_df['Text_processed'] = test_text_df['Text_processed'].map(lambda x: x.lower())
# Print out the first rows of papers
test_text_df['Text_processed'] = test_text_df['Text_processed'].apply(lambda x: x.strip())
test_text_df

# Remove punctuation
val_text_df['Text_processed'] = val_text_df['Text'].map(lambda x: re.sub('[\',.!?*]', '', str(x)))
# Convert the titles to lowercase
val_text_df['Text_processed'] = val_text_df['Text_processed'].replace(r'\n',' ', regex=True)
val_text_df['Text_processed'] = val_text_df['Text_processed'].map(lambda x: x.lower())
# Print out the first rows of papers
val_text_df['Text_processed'] = val_text_df['Text_processed'].apply(lambda x: x.strip())
val_text_df

train_full = train_variants_df.merge(train_text_df, how="inner", left_on="ID", right_on="ID")
train_full = train_full.drop("Text", axis=1)
train_full.head()

test_full = test_variants_df.merge(test_text_df, how="inner", left_on="ID", right_on="ID")
test_full = test_full.drop("Text", axis=1)
test_full.head()

val_full = val_variants_df.merge(val_text_df, how="inner", left_on="ID", right_on="ID")
val_full = val_full.drop("Text", axis=1)
val_full.head()

import nltk
nltk.download('words')
words = set(nltk.corpus.words.words())
train_full['Text_processed'] = train_full['Text_processed'].apply(lambda x: " "
    .join(w for w in nltk.wordpunct_tokenize(x) if w.lower() in words ))
test_full['Text_processed'] = test_full['Text_processed'].apply(lambda x: " "
    .join(w for w in nltk.wordpunct_tokenize(x) if w.lower() in words ))
val_full['Text_processed'] = val_full['Text_processed'].apply(lambda x: " "
    .join(w for w in nltk.wordpunct_tokenize(x) if w.lower() in words ))

# ## Word Statistics

train_text_df.loc[:, 'Text_count']  = train_text_df["Text_processed"].apply(lambda x: len(x.split()))
train_text_df.head()
test_text_df.loc[:, 'Text_count']  = test_text_df["Text_processed"].apply(lambda x: len(x.split()))
test_text_df.head()
val_text_df.loc[:, 'Text_count']  = val_text_df["Text_processed"].apply(lambda x: len(x.split()))
val_text_df.head()
train_full = train_variants_df.merge(train_text_df, how="inner", left_on="ID", right_on="ID")
train_full.head()
test_full = test_variants_df.merge(test_text_df, how="inner", left_on="ID", right_on="ID")
test_full.head()
val_full = val_variants_df.merge(val_text_df, how="inner", left_on="ID", right_on="ID")
val_full.head()

train_full = train_full.drop("Text", axis=1)
test_full = test_full.drop("Text", axis=1)
val_full = val_full.drop("Text", axis=1)

print(sum(train_full["Text_count"]))
print(sum(test_full["Text_count"]))
print(sum(val_full["Text_count"]))


from wordcloud import WordCloud
# Join the different processed titles together.
long_string = ','.join(list(train_text_df['Text_processed'].values))
# Create a WordCloud object
wordcloud = WordCloud(background_color="black", max_words=5000, contour_width=5, contour_color='steelblue')
# Generate a word cloud
wordcloud.generate(long_string)
# Visualize the word cloud
wordcloud.to_image()

count_grp = train_full.groupby('Class')["Text_count"]
count_grp.describe()

plt.figure(figsize=(12,8))
gene_count_grp = train_full.groupby('Gene')["Text_count"].sum().reset_index()
sns.violinplot(x="Class", y="Text_count", data=train_full, inner=None,palette="Spectral")
sns.swarmplot(x="Class", y="Text_count", data=train_full, color="w", alpha=.5);
plt.ylabel('Text Count', fontsize=14)
plt.xlabel('Class', fontsize=14)
plt.title("Text length distribution", fontsize=18)
plt.show()


fig, axs = plt.subplots(ncols=3, nrows=3, figsize=(20,16))

for i in range(3):
    for j in range(3):
        gene_count_grp = train_full[train_full["Class"]==((i*3+j)+1)].groupby('Gene')["Text_count"].mean().reset_index()
        sorted_gene_group = gene_count_grp.sort_values('Text_count', ascending=False)
        sorted_gene_group_top_7 = sorted_gene_group[:7]
        sns.barplot(x="Gene", y="Text_count", data=sorted_gene_group_top_7, ax=axs[i][j],palette="Spectral")


# # Vector Representation

train_full["Text_processed"]
test_full["Text_processed"]
val_full["Text_processed"]

# ## Word2Vec

# ### CBOW and Skip Gram


# Python program to generate word vectors using Word2Vec

# importing all necessary modules
from nltk.tokenize import sent_tokenize, word_tokenize
import warnings

warnings.filterwarnings(action = 'ignore')

import gensim
from gensim.models import Word2Vec

Word2Vec_dim = 200
train = train_full["Text_processed"].to_string()
```

```
test = test_full["Text_processed"].to_string()
val = val_full["Text_processed"].to_string()

# Replaces escape character with space
f = train.replace("\n", " ")

data = []

# iterate through each sentence in the file
for i in sent_tokenize(f):
    temp = []
    # tokenize the sentence into words
    for j in word_tokenize(i):
        temp.append(j.lower())

    data.append(temp)


# Replaces escape character with space
f2 = test.replace("\n", " ")

data2 = []

# iterate through each sentence in the file
for i in sent_tokenize(f2):
    temp2 = []
    # tokenize the sentence into words
    for j in word_tokenize(i):
        temp2.append(j.lower())

    data2.append(temp2)


# Replaces escape character with space
f3 = val.replace("\n", " ")

data3 = []

# iterate through each sentence in the file
for i in sent_tokenize(f3):
    temp3 = []
    # tokenize the sentence into words
    for j in word_tokenize(i):
        temp3.append(j.lower())

    data3.append(temp3)

# Create CBOW model
model1 = gensim.models.Word2Vec(data, min_count = 1,vector_size = Word2Vec_dim, window = 5)
model2 = gensim.models.Word2Vec(data2, min_count = 1,vector_size = Word2Vec_dim, window = 5)
model3 = gensim.models.Word2Vec(data3, min_count = 1,vector_size = Word2Vec_dim, window = 5)


#Create Skip Gram model
model4 = gensim.models.Word2Vec(data, min_count = 1, vector_size = Word2Vec_dim, window = 5, sg = 1)
model5 = gensim.models.Word2Vec(data2, min_count = 1, vector_size = Word2Vec_dim, window = 5, sg = 1)
# model6 = gensim.models.Word2Vec(data2, min_count = 1, vector_size = Word2Vec_dim, window = 5, sg = 1)

# model1.save("word2vec.model1_CBOW")
# model2.save("word2vec.model2_CBOW")
```

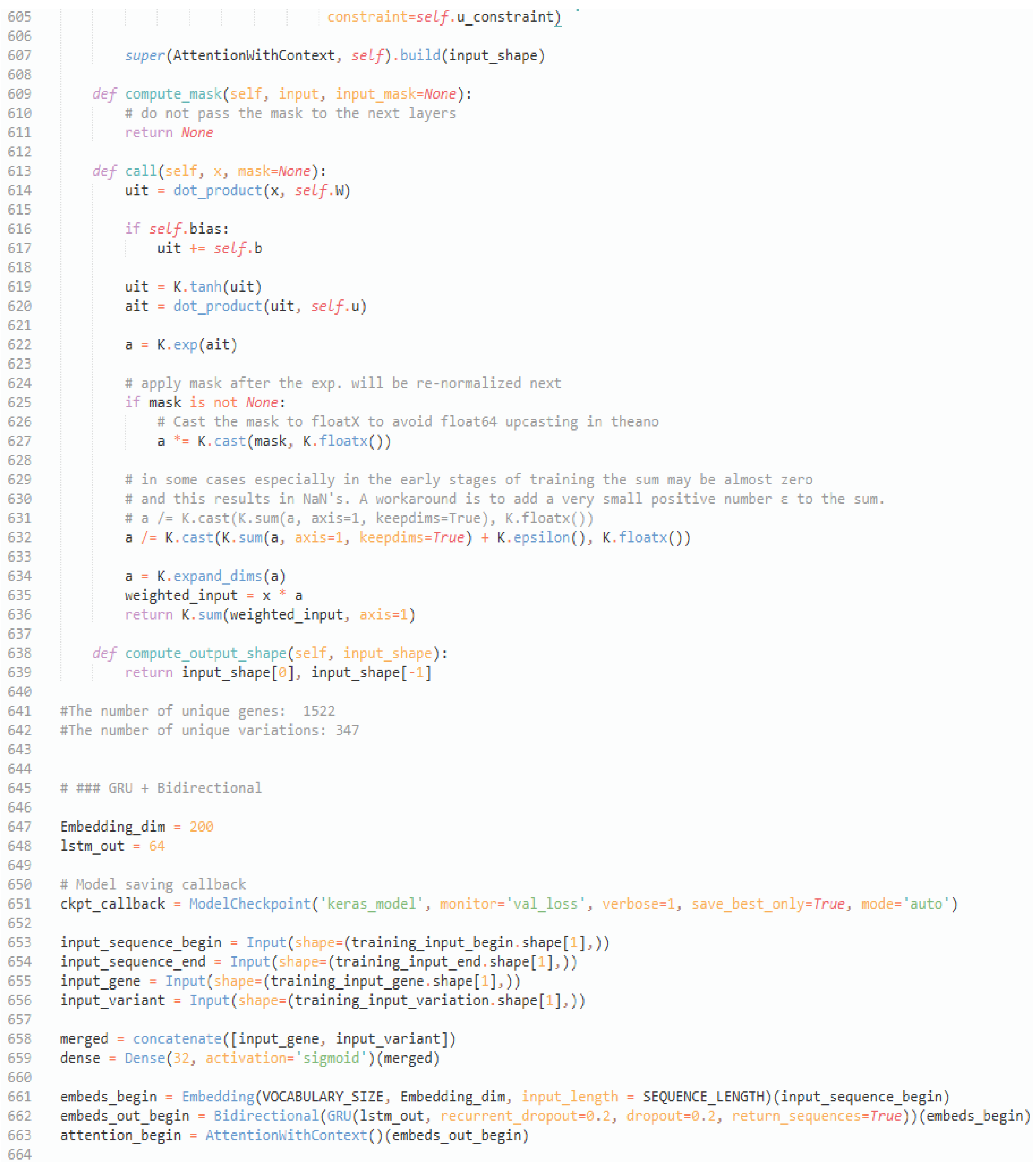
```python
# model3.save("word2vec.model3_CBOW")
# model4.save("word2vec.model4_SkipGram")
# model5.save("word2vec.model5_SkipGram")
# model6.save("word2vec.model6_SkipGram")


# ### Average feature vector


def avg_feature_vector(sentence, model, num_features):
    words = sentence.replace('\n'," ").replace(',',' ').replace('.'," ").split()
    feature_vec = np.zeros((num_features,),dtype="float32")
    i=0
    for word in words:
        try:
            feature_vec = np.add(feature_vec, model[word])
        except KeyError as error:
            feature_vec
            i = i + 1
    if len(words) > 0:
        feature_vec = np.divide(feature_vec, len(words)- i)
    return feature_vec


train_word2vec1 = np.zeros((len(train_full),Word2Vec_dim),dtype="float32")
test_word2vec1 = np.zeros((len(test_full),Word2Vec_dim),dtype="float32")
val_word2vec1 = np.zeros((len(val_full),Word2Vec_dim),dtype="float32")

for i in range(len(train_full)):
    train_word2vec1[i] = avg_feature_vector(train_full["Text_processed"][i],model1.wv, Word2Vec_dim)

for i in range(len(test_full)):
    test_word2vec1[i] = avg_feature_vector(test_full["Text_processed"][i],model1.wv, Word2Vec_dim)

for i in range(len(val_full)):
    val_word2vec1[i] = avg_feature_vector(val_full["Text_processed"][i],model1.wv, Word2Vec_dim)


train_word2vec2 = np.zeros((len(train_full),Word2Vec_dim),dtype="float32")
test_word2vec2 = np.zeros((len(test_full),Word2Vec_dim),dtype="float32")
val_word2vec2 = np.zeros((len(val_full),Word2Vec_dim),dtype="float32")

for i in range(len(train_full)):
    train_word2vec2[i] = avg_feature_vector(train_full["Text_processed"][i],model4.wv, Word2Vec_dim)

for i in range(len(test_full)):
    test_word2vec2[i] = avg_feature_vector(test_full["Text_processed"][i],model4.wv, Word2Vec_dim)

for i in range(len(val_full)):
    val_word2vec2[i] = avg_feature_vector(val_full["Text_processed"][i],model4.wv, Word2Vec_dim)


train_word2vec1 = pd.DataFrame(train_word2vec1)
train_word2vec1
test_word2vec1 = pd.DataFrame(test_word2vec1)
test_word2vec1
val_word2vec1 = pd.DataFrame(val_word2vec1)
val_word2vec1
train_word2vec2 = pd.DataFrame(train_word2vec2)
train_word2vec2
```

```python
test_word2vec2 = pd.DataFrame(test_word2vec2)
test_word2vec2
val_word2vec2 = pd.DataFrame(val_word2vec2)
val_word2vec2


# # Modeling

# ## LightGBM

lbl = preprocessing.LabelEncoder()
data1 = train_word2vec1
target = lbl.fit_transform(train_full["Class"].astype(str))#将提示的包含错误数据类型这一列进行转换
X_train1, X_test1, y_train1, y_test1 = train_test_split(data1,target,test_size=0.3, random_state = 42)
print("Model1: Train data length:", len(X_train1))
print("Model1: Test data length:", len(X_test1))
btime = datetime.now()
lgb_train1 = lgb.Dataset(X_train1, y_train1)
lgb_eval1 = lgb.Dataset(X_test1, y_test1, reference=lgb_train1)

params = {
    'task':'train',
    'boosting_type':'gbdt',
    'objective': 'multiclass',
    'num_class': 9,
    'metric': 'multi_error',
    'num_leaves': 500,
    'min_data_in_leaf': 100,
    'learning_rate': 0.1,
    'feature_fraction': 0.9,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'lambda_l1': 0.4,
    'lambda_l2': 0.5,
    'min_gain_to_split': 0.2,
    'verbose': -1,
}

gbm1 = lgb.train(params, lgb_train1, num_boost_round=1000, valid_sets=lgb_eval1,
    verbose_eval = 50, early_stopping_rounds=300)
print('all tasks done. total time used:%s s.\n\n'%((datetime.now() - btime).total_seconds()))
gbm1.save_model('model1_CBOW.txt')
gbm1 = lgb.Booster(model_file='model1_CBOW.txt')
y_pred_pa1 = gbm1.predict(X_test1)
y_test_oh1 = label_binarize(y_test1, classes= [1,2,3,4,5,6,7,8,9])
#y_pred_lightGBM1 = [list(x).index(max(x)) for x in y_prob1]
print('auc：', roc_auc_score(y_test_oh1, y_pred_pa1, average='micro'))

y_pred1 = y_pred_pa1.argmax(axis=1)
confusion_matrix(y_test1, y_pred1)

precision_score(y_test1, y_pred1,average='micro')
recall_score(y_test1, y_pred1,average='micro')
f1_score(y_test1, y_pred1,average='micro')

print(classification_report(y_test1, y_pred1))

pred_lightgbm_cbow = gbm1.predict(test_word2vec1)
#pred_lightgbm_cbow = [list(x).index(max(x)) for x in pred_lightgbm_cbow]
#pred_lightgbm_cbow = pd.get_dummies(np.array(pred_lightgbm_cbow) + 1)
```

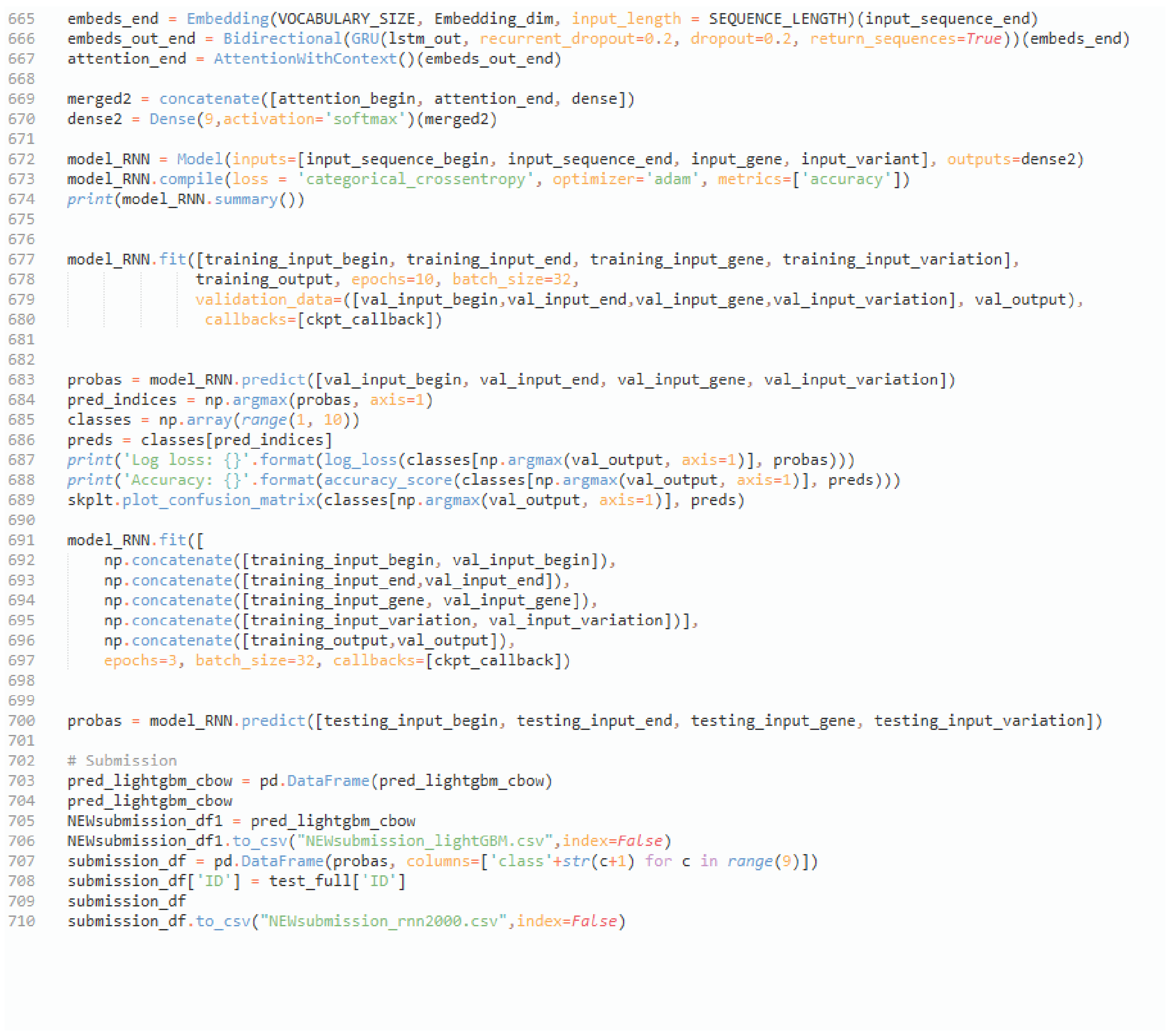
```python
pred_lightgbm_cbow

# pred_lightgbm_skg = gbm2.predict(test_word2vec2)
# # pred_lightgbm_skg = [list(x).index(max(x)) for x in pred_lightgbm_skg]
# # pred_lightgbm_skg = pd.get_dummies(np.array(pred_lightgbm_skg) + 1)
# pred_lightgbm_skg


# ## RNN



NUM_CLASS=9
VOCABULARY_SIZE = 10000
SEQUENCE_LENGTH= 2000
tokenizer = Tokenizer(filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',lower=True,split=" ")
tokenizer.fit_on_texts(train_full["Text_processed"])
vocab = tokenizer.word_index

# Training
training = train_full.sample(frac=1) # shuffle data first
training_input = tokenizer.texts_to_sequences(training['Text_processed'].astype(str))
training_input_r = [list(reversed(x)) for x in training_input]
training_input_begin = pad_sequences(training_input, maxlen=SEQUENCE_LENGTH)
training_input_end = pad_sequences(training_input_r, maxlen=SEQUENCE_LENGTH)
training_output = pd.get_dummies(training['Class']).values

# Testing
testing_input = tokenizer.texts_to_sequences(test_full['Text_processed'].astype(str))
testing_input_r = [list(reversed(x)) for x in testing_input]
testing_input_begin = pad_sequences(testing_input, maxlen=SEQUENCE_LENGTH)
testing_input_end = pad_sequences(testing_input_r, maxlen=SEQUENCE_LENGTH)

# Validation
val_input = tokenizer.texts_to_sequences(val_full['Text_processed'].astype(str))
val_input_r = [list(reversed(x)) for x in val_input]
val_input_begin = pad_sequences(val_input, maxlen=SEQUENCE_LENGTH)
val_input_end = pad_sequences(val_input_r, maxlen=SEQUENCE_LENGTH)
val_output = pd.get_dummies(val_full['Class']).values

print("Training set shape:",training_input_begin.shape, training_input_end.shape, training_output.shape)
print("Testing set shape:",testing_input_begin.shape, testing_input_end.shape)
print("Validation set shape:",val_input_begin.shape, val_input_end.shape, val_output.shape)

# Add gene and variation to predictor
gene_label = LabelEncoder()
ALL_Genes = np.concatenate([train_full['Gene'], val_full['Gene'], test_full['Gene']])
ALL_Variations = np.concatenate([train_full['Variation'], val_full['Variation'], test_full['Variation']])
ALL_Variations = np.asarray([v[0]+v[-1] for v in ALL_Variations])
print ("The number of unique genes: ", len(np.unique(ALL_Genes)))
print ("The number of unique variations:", len(np.unique(ALL_Variations)))


len_train = len(training_input)
len_validation = len(val_input)
len_test = len(testing_input)
print("The length of training input:", len_train)
print("The length of testing input:", len_test)
print("The length of validation input:", len_validation)


gene_encoded = pd.get_dummies(ALL_Genes).values
```



```
variation_encoded = pd.get_dummies(ALL_Variations).values
training_input_gene = gene_encoded[:len_train]
training_input_variation = variation_encoded[:len_train]
testing_input_gene = gene_encoded[-len_test:]
testing_input_variation = variation_encoded[-len_test:]
val_input_gene = gene_encoded[len_train:-len_test]
val_input_variation = variation_encoded[len_train:-len_test]
print (len(training_input_gene))
print (len(testing_input_gene))
print (len(val_input_gene))


def dot_product(x, kernel):
    if K.backend() == 'tensorflow':
        return K.squeeze(K.dot(x, K.expand_dims(kernel)), axis=-1)
    else:
        return K.dot(x, kernel)


# ### Attention with context

class AttentionWithContext(Layer):
    def __init__(self,
                 W_regularizer=None, u_regularizer=None, b_regularizer=None,
                 W_constraint=None, u_constraint=None, b_constraint=None,
                 bias=True, **kwargs):

        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.u_regularizer = regularizers.get(u_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.u_constraint = constraints.get(u_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        super(AttentionWithContext, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight(shape = (input_shape[-1], input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        if self.bias:
            self.b = self.add_weight(shape = (input_shape[-1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)

        self.u = self.add_weight(shape = (input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_u'.format(self.name),
                                 regularizer=self.u_regularizer,
```



```
                                 constraint=self.u_constraint)

        super(AttentionWithContext, self).build(input_shape)

    def compute_mask(self, input, input_mask=None):
        # do not pass the mask to the next layers
        return None

    def call(self, x, mask=None):
        uit = dot_product(x, self.W)

        if self.bias:
            uit += self.b

        uit = K.tanh(uit)
        ait = dot_product(uit, self.u)

        a = K.exp(ait)

        # apply mask after the exp. will be re-normalized next
        if mask is not None:
            # Cast the mask to floatX to avoid float64 upcasting in theano
            a *= K.cast(mask, K.floatx())

        # in some cases especially in the early stages of training the sum may be almost zero
        # and this results in NaN's. A workaround is to add a very small positive number ε to the sum.
        # a /= K.cast(K.sum(a, axis=1, keepdims=True), K.floatx())
        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[-1]

#The number of unique genes:  1522
#The number of unique variations: 347


# ### GRU + Bidirectional

Embedding_dim = 200
lstm_out = 64

# Model saving callback
ckpt_callback = ModelCheckpoint('keras_model', monitor='val_loss', verbose=1, save_best_only=True, mode='auto')

input_sequence_begin = Input(shape=(training_input_begin.shape[1],))
input_sequence_end = Input(shape=(training_input_end.shape[1],))
input_gene = Input(shape=(training_input_gene.shape[1],))
input_variant = Input(shape=(training_input_variation.shape[1],))

merged = concatenate([input_gene, input_variant])
dense = Dense(32, activation='sigmoid')(merged)

embeds_begin = Embedding(VOCABULARY_SIZE, Embedding_dim, input_length = SEQUENCE_LENGTH)(input_sequence_begin)
embeds_out_begin = Bidirectional(GRU(lstm_out, recurrent_dropout=0.2, dropout=0.2, return_sequences=True))(embeds_begin)
attention_begin = AttentionWithContext()(embeds_out_begin)

```



```python
embeds_end = Embedding(VOCABULARY_SIZE, Embedding_dim, input_length = SEQUENCE_LENGTH)(input_sequence_end)
embeds_out_end = Bidirectional(GRU(lstm_out, recurrent_dropout=0.2, dropout=0.2, return_sequences=True))(embeds_end)
attention_end = AttentionWithContext()(embeds_out_end)

merged2 = concatenate([attention_begin, attention_end, dense])
dense2 = Dense(9,activation='softmax')(merged2)

model_RNN = Model(inputs=[input_sequence_begin, input_sequence_end, input_gene, input_variant], outputs=dense2)
model_RNN.compile(loss = 'categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model_RNN.summary())


model_RNN.fit([training_input_begin, training_input_end, training_input_gene, training_input_variation],
              training_output, epochs=10, batch_size=32,
              validation_data=([val_input_begin,val_input_end,val_input_gene,val_input_variation], val_output),
               callbacks=[ckpt_callback])


probas = model_RNN.predict([val_input_begin, val_input_end, val_input_gene, val_input_variation])
pred_indices = np.argmax(probas, axis=1)
classes = np.array(range(1, 10))
preds = classes[pred_indices]
print('Log loss: {}'.format(log_loss(classes[np.argmax(val_output, axis=1)], probas)))
print('Accuracy: {}'.format(accuracy_score(classes[np.argmax(val_output, axis=1)], preds)))
skplt.plot_confusion_matrix(classes[np.argmax(val_output, axis=1)], preds)

model_RNN.fit([
    np.concatenate([training_input_begin, val_input_begin]),
    np.concatenate([training_input_end,val_input_end]),
    np.concatenate([training_input_gene, val_input_gene]),
    np.concatenate([training_input_variation, val_input_variation])],
    np.concatenate([training_output,val_output]),
    epochs=3, batch_size=32, callbacks=[ckpt_callback])


probas = model_RNN.predict([testing_input_begin, testing_input_end, testing_input_gene, testing_input_variation])

# Submission
pred_lightgbm_cbow = pd.DataFrame(pred_lightgbm_cbow)
pred_lightgbm_cbow
NEWsubmission_df1 = pred_lightgbm_cbow
NEWsubmission_df1.to_csv("NEWsubmission_lightGBM.csv",index=False)
submission_df = pd.DataFrame(probas, columns=['class'+str(c+1) for c in range(9)])
submission_df['ID'] = test_full['ID']
submission_df
submission_df.to_csv("NEWsubmission_rnn2000.csv",index=False)
```

Bibliography

[1] J. Ferlay, M. Ervik, F. Lam, M. Colombet, L. Mery, and M. Piñeros, "Global Cancer Observatory: Cancer Today," International Agency for Research on Cancer, Tech. Rep., 2021. [Online]. Available: https://gco.iarc.fr/today

[2] J. J. Salk, E. J. Fox, and L. A. Loeb, "Mutational Heterogeneity in Human Cancers: Origin and Consequences," *Annual Review of Pathology: Mechanisms of Disease*, vol. 5, no. 1, pp. 51–75, 2010. [Online]. Available: http://www.annualreviews.org/doi/10.1146/annurev-pathol-121808-102113

[3] J. Xu, P. Yang, S. Xue, B. Sharma, M. Sanchez-Martin, F. Wang, K. A. Beaty, E. Dehan, and B. Parikh, "Translating cancer genomics into precision medicine with artificial intelligence: applications, challenges and future perspectives," *Human Genetics*, vol. 138, no. 2, pp. 109–124, 2019. [Online]. Available: http://link.springer.com/10.1007/s00439-019-01970-5

[4] R. Leaman, R. Khare, and Z. Lu, "Challenges in clinical natural language processing for automated disorder normalization," *Journal of*

Biomedical Informatics, vol. 57, pp. 28–37, 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1532046415001501

[5] M. Gupta, H. Wu, S. Arora, A. Gupta, G. Chaudhary, and Q. Hua, "Gene Mutation Classification through Text Evidence Facilitating Cancer Tumour Detection," *Journal of Healthcare Engineering*, vol. 2021, pp. 1–16, 2021. [Online]. Available: https://www.hindawi.com/journals/jhe/2021/8689873/

[6] M. D. Yandell and W. H. Majoros, "Genomics and natural language processing," *Nature Reviews Genetics*, vol. 3, no. 8, pp. 601–610, 2002. [Online]. Available: http://www.nature.com/articles/nrg861

[7] S. Velupillai, H. Suominen, M. Liakata, A. Roberts, A. D. Shah, K. Morley, D. Osborn, J. Hayes, R. Stewart, J. Downs, W. Chapman, and R. Dutta, "Using clinical Natural Language Processing for health outcomes research: Overview and actionable suggestions for future advances," *Journal of Biomedical Informatics*, vol. 88, pp. 11–19, 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1532046418302016

[8] H. Farooq, N. Rehmat, S. Kumar, and H. Naveed, "Genetic Mutation Classification using Machine Learning," *Biophysical Journal*, vol. 116, no. 3, p. 292a, 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0006349518328418

[9] M. Singh, A. Murthy, and S. Singh, "Prioritization of Free-Text Clinical Documents: A Novel Use of a Bayesian Classifier," *JMIR Medical Informatics*, vol. 3, no. 2, p. e17, 2015. [Online]. Available: http://medinform.jmir.org/2015/2/e17/

[10] A. Wright, A. B. McCoy, S. Henkin, A. Kale, and D. F. Sittig, "Use of a support vector machine for categorizing free-text notes: assessment of accuracy across two institutions," *Journal of the American Medical Informatics Association*, vol. 20, no. 5, pp. 887–890, 2013. [Online]. Available: https://academic.oup.com/jamia/article-lookup/doi/10.1136/amiajnl-2012-001576

[11] T. K. Ho, "random decision forests," *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1)*, vol. Volume 1, pp. 278–282, 1995. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDAR.1995.598994

[12] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *The Annals of Statistics*, vol. 29, no. 5, Oct. 2001. [Online]. Available: https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy-function-approximation-A-gradient-boosting-machine/10.1214/aos/1013203451.full

[13] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* San Francisco California USA: ACM, Aug. 2016, pp. 785–794. [Online]. Available: https://dl.acm.org/doi/10.1145/2939672.2939785

[14] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," p. 9.

[15] X. Qin, J. Liu, Y. Wang, Y. Liu, K. Deng, Y. Ma, K. Zou, L. Li, and X. Sun, "Natural language processing

was effective in assisting rapid title and abstract screening when updating systematic reviews," *Journal of Clinical Epidemiology*, vol. 133, pp. 121–129, 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0895435621000147

[16] I. Spasic and G. Nenadic, "Clinical Text Data in Machine Learning: Systematic Review," *JMIR MEDICAL INFORMATICS*, p. 19.

[17] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[18] V. Mnih, N. Heess, and A. Graves, "Recurrent Models of Visual Attention," p. 9.

[19] Y. Bao, Z. Deng, Y. Wang, H. Kim, V. D. Armengol, F. Acevedo, N. Ouardaoui, C. Wang, G. Parmigiani, R. Barzilay, D. Braun, and K. S. Hughes, "Using Machine Learning and Natural Language Processing to Review and Classify the Medical Literature on Cancer Susceptibility Genes," *JCO Clinical Cancer Informatics*, no. 3, pp. 1–9, 2019. [Online]. Available: https://ascopubs.org/doi/10.1200/CCI.19.00042

[20] Y. Wang, S. Sohn, S. Liu, F. Shen, L. Wang, E. J. Atkinson, S. Amin, and H. Liu, "A clinical text classification paradigm using weak supervision and deep representation," *BMC Medical Informatics and Decision Making*, vol. 19, no. 1, p. 1, 2019. [Online]. Available: https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-018-0723-6

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv:1810.04805 [cs]*, 2019. [Online]. Available: http://arxiv.org/abs/1810.04805

[22] Y. Su, H. Xiang, H. Xie, Y. Yu, S. Dong, Z. Yang, and N. Zhao, “Application of BERT to Enable Gene Classification Based on Clinical Evidence,” *BioMed Research International*, vol. 2020, pp. 1–13, 2020. [Online]. Available: https://www.hindawi.com/journals/bmri/2020/5491963/

[23] S. Wu, K. Roberts, S. Datta, J. Du, Z. Ji, Y. Si, S. Soni, Q. Wang, Q. Wei, Y. Xiang, B. Zhao, and H. Xu, “Deep learning in clinical natural language processing: a methodical review,” *Journal of the American Medical Informatics Association*, vol. 27, no. 3, pp. 457–470, 2020. [Online]. Available: https://academic.oup.com/jamia/article/27/3/457/5651084