SI 630: Homework 1 – Classification

Yu Yan (kuminia)

Kaggle username: kkkyyy1

January 29, 2023

Part 1: Representing Text Data

Task 1.1: Tokenization

I use two ways to tokenize the text, here are the two functions for tokenization I define.

```
def tokenize(string):
    x = string.split()
    token = [*set(x)]
    return token
```

```
def better_tokenize(string):
    string = string.lower()
    first = re.findall("[\w]+",string)
    token = [*set(first)]
    return token
```

In the better_tokenize function, I first make all of the text become the lower case, and then I use regex to get all the words (including numbers) from the text. And I use set to remove the duplicate words. The reason I build this function is that I think the upper case and low case of the same word will not have a huge impact on a model; besides this, I want to delete commas and other punctuation marks, which are not useful in this model.

To use the function, I first make an empty list called txt and throw all the text of each row into the txt list. Then, I make another list called list_token and use the better_tokenize function to tokenize each element in the txt list, and finally add it to the list_token list.

Task 1.2: Building the Term-Document Matrix

Before building the sparse matrix, I first make all of the elements in list_token into a string, and then I implement regex, set, and Counter to count the frequency of each word in the string. Next, I drop the words whose frequency is less than 2. After that, I find that I have 3895 features.

Then, I use row_index, col_index, and data to record the row and col of the appearance of each word for each row in the CSV file. Finally, I implement sparse.coo_matrix to finish building this sparse matrix called mat_coo.

Part 2: Logistic Regression in NumPy

Before building the logistic regression model, I need to add a bias term for the sparse matrix. Therefore, I use np.hstack to add this single column(only contains one) to the sparse matrix.

Next, I build the sigmoid function and log_likelihood function, and I also make the development dataset become a sparse_matrix called dev_mat_coo for future tests.

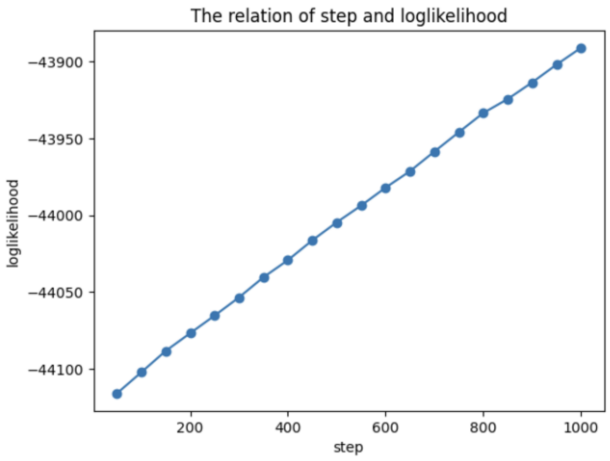
After that, I build a function called get_test_fscore to calculate the f1 score for the development dataset.

I do not build the compute gradient function because I think it can be contained in the logistic regression process very conveniently.

I first implement a function called logistic_regression_test to see how my model is learning, and set the learning rate to 0.00005, the total step to 1000.

```
def logistic_regression_test():
    # give B an initial value
    B = [[1]]*(len(new_token2)+1)
    B = np.array(B)
    d = log_likelihood(B)
    diff = 1
    step = 0
    step_list = []
    loglikelihood_List = []
    t = True
    ran_list = random.sample(range(0, len(list_token)), len(list_token))
    for i in range(1000):
        j = ran_list[i]
        if t:
            x = mat_coo.tocsr()[np.array([j]),:]*B
            B = B - np.transpose(0.00005*((sigmoid(x)-int(train_file["sarcastic"][j]))*mat_coo.tocsr()[np.array([j]),:]
            step = step+1
            if step % 50 == 0 and abs(diff)>0.00001:
                c = log_likelihood(B)
                loglikelihood_List.append(c[0][0])
                diff = c - d
                d = c
                print(f'log-likelihood: {c[0][0]}')
                print(f'difference: {diff[0][0]}')
                step_list.append(step)
            elif abs(diff)<=0.00001:
                return step_list, loglikelihood_List
                t = False
    return step_list, loglikelihood_List
```

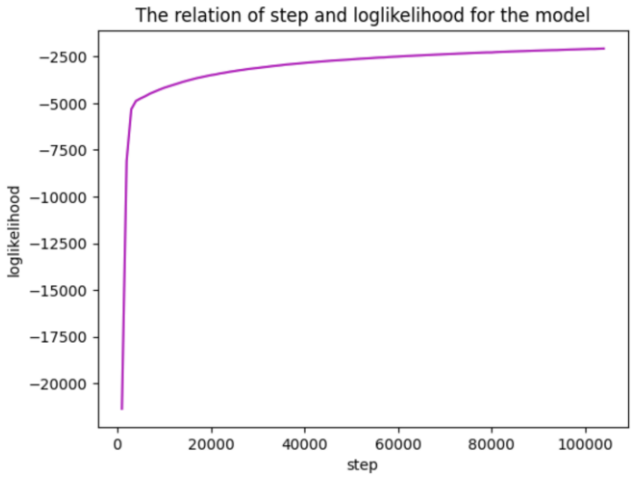
Here is the result, as the step increase, the loglikelihood also increases. It is very reasonable. The model is not converged.

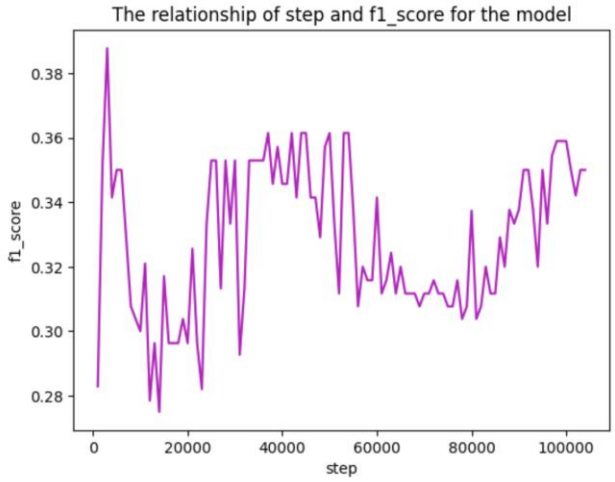


After that, I construct another function called logistic_regression_real, which will train the model until the log-likelihood does not change too much between steps (1e-5), if the epochs are bigger than 30, this function will also stop.

```python
def logistic_regression_real():
    # Initialize B
    B = [[1]]*(len(new_token2)+1)
    B = np.array(B)

    d = log_likelihood(B)
    diff = 1
    epochs = 30

    f1_score_list = []

    loglikelihood_list = []

    step_list = []

    step = 0
    t = True
    for i in range(epochs):
        ran_list = random.sample(range(0, len(list_token)), len(list_token))
        for j in range(len(ran_list)):
            if t:
                k = ran_list[j]
                x = mat_coo.tocsr()[np.array([k]),:]*B
                B = B - np.transpose(0.005*((sigmoid(x)-int(train_file["sarcastic"][k]))*mat_coo.tocsr()[np.array([k])
                step = step + 1
                if step % 1000 == 0 and abs(diff)>0.00001:
                    c = log_likelihood(B)
                    diff = c - d
                    d = c
                    print(f'log-likelihood: {c[0][0]}')
                    print(f'difference: {diff[0][0]}')
                    f1 = get_test_fscore(B)
                    print(f'f1 score: {f1}')
                    loglikelihood_list.append(c[0][0])
                    f1_score_list.append(f1)
                    step_list.append(step)
                elif abs(diff)<=0.00001:
                    return step_list, loglikelihood_list, f1_score_list, B
                    t = False
    return step_list, loglikelihood_list, f1_score_list, B
```

Here is the result of this training. The log-likelihood first increase dramatically and then increase very slowly. For the f1 score, it vibrates between 0.3 and 0.36, and finally gets the result of about 0.36.





I then build a function called predict, use the beta from the training model to predict the result of the test data set, and save the result to a CSV file.

```
def predict(beta,test_set):
    test_mat_coo = get_sparse_matrix(test_set)
    label = []
    for i in range(1400):
        c = sigmoid(test_mat_coo.tocsr()[np.array([i]),:]*beta)
        if c <0.5:
            c = 0
        else:
            c = 1
        label.append(c)
    return label
```

After I submit this file to Kaggle, I got an F1 score of 0.25668.

Part 3: Logistic Regression with PyTorch

3.1 Setting up the Data

First of all, I define a function called to_sparse_tensor with can help convert the matrix into sparse Tensor objects. Then I make another function called get_new_data, which can help me get the data of train, development, and test data and convert it to the format I need.

3.2 Building the Logistic Regression Neural Network

In this part, I first define a LogisticRegression model using the module from PyTorch.

```
class LogisticRegression(nn.Module):

    def __init__(self,n_input_features):
        super(LogisticRegression,self).__init__()
        self.linear=torch.nn.Linear(n_input_features,1)

    def forward(self,x):
        y_predicted=torch.sigmoid(self.linear(x))
        return y_predicted
```

Then I define the criterion and optimizer for the model, the learning rate is given 0.05.

```
criterion = nn.BCELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

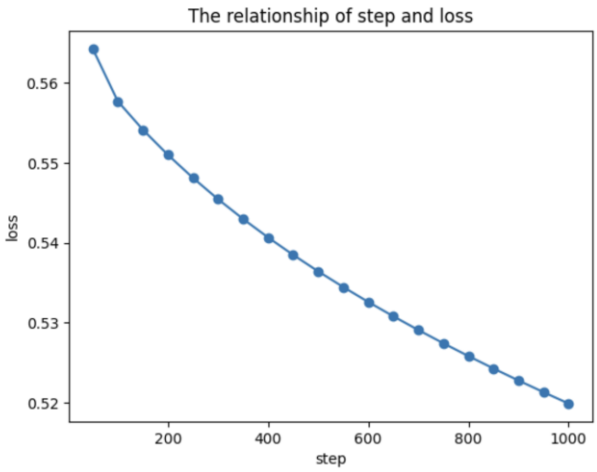
Next, I define the get_f1_score function, which helps me get the f1 score for my test data.

```
def get_f1_score():
    with torch.no_grad():
        true_positive = 0
        false_positive = 0
        false_negative = 0
        y_predicted_1 = model(X_dev)
        y_predicted_cls = y_predicted_1.round()
        for i in range(len(y_predicted_cls)):
            if y_dev[i] == 1 and y_predicted_cls[i] == 1:
                true_positive = true_positive+1
            elif y_dev[i] == 0 and y_predicted_cls[i] == 1:
                false_positive = false_positive + 1
            elif y_dev[i] == 1 and y_predicted_cls[i] == 0:
                false_negative = false_negative+1
        if true_positive+false_positive == 0 or true_positive+false_negative == 0:
            return 0
        else:
            Precision = true_positive/(true_positive+false_positive)
            Recall = true_positive/(true_positive+false_negative)
            if Precision+Recall == 0:
                return 0
            else:
                F1_Score = 2 * (Precision * Recall) / (Precision + Recall)
                return F1_Score
```

3.3 Training and Experiment

3.3.1 Test my model with 1000 steps

First of all, I train my model for a total of 1000 steps and find that the loss is going down as the steps increased.



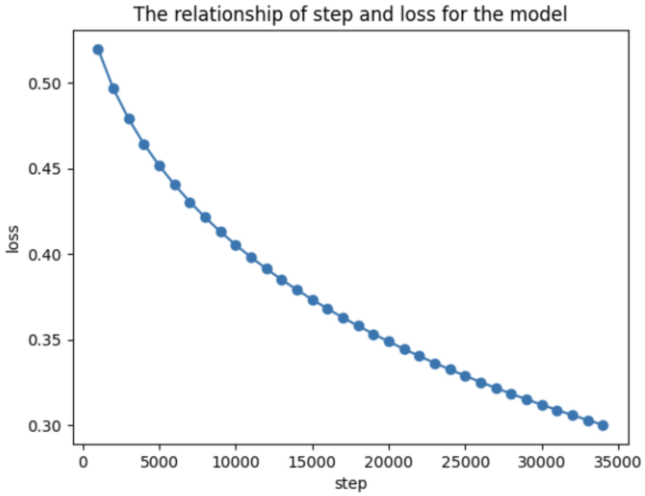
3.3.2 Train my model with 10 epochs

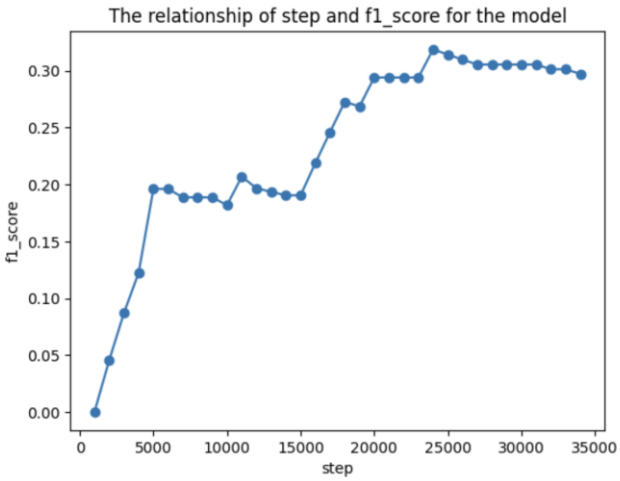
Then, I start to train my model with 10 epochs and a 0.05 learning rate, and I will record the loss and f1 score every 1000 steps for the model. The coding part and results I show below.

```
loss_list = []

f1_list = []

epochs = 10

step = 0

x = []

for epoch in range(epochs):

    for i in range(len(y_train)):

        y_predicted = model(X_train)

        loss = criterion(y_predicted, y_train)

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

        step = step + 1

        if (step+1)%1000 == 0:
            print(f'epoch:{epoch+1}, loss={loss.item():.4f}')
            f1 = get_f1_score()
            print(f'f1:{f1}')
            loss_list.append(loss.item())
            f1_list.append(f1)
            x.append(step+1)
```

Here is the result for the relationship of step with loss and f1 score. We can find as the steps increase from 0 to 10000, the loss goes down dramatically; after that, it decreases relatively slightly way. For the F1 score, we can find as the steps increase, it first goes up and then goes down slightly (it might be because of the overfitting), the final f1 score

for this model is about 0.3.



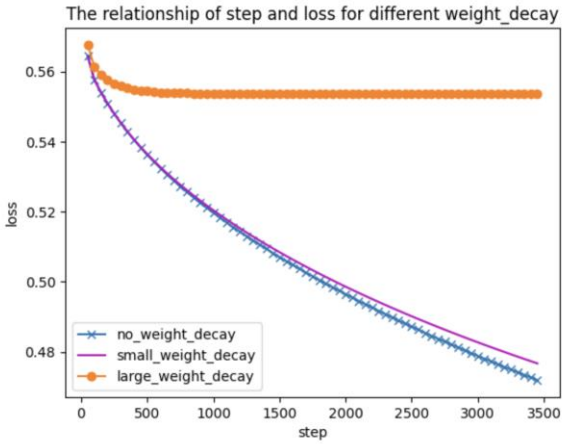


3.3.3 Train my model with L2 penalty

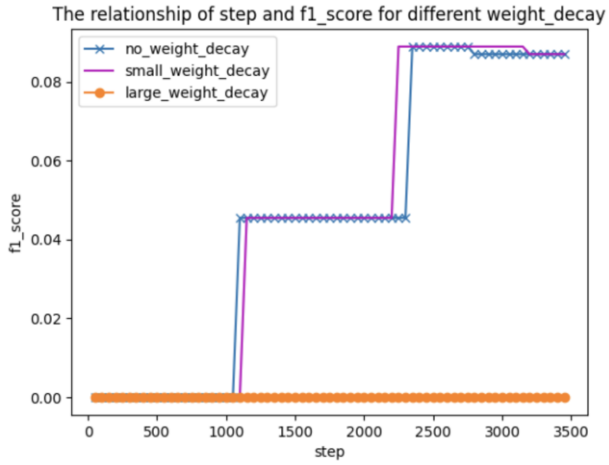
After that, I also use PyTorch to see the effects of adding regularization. I set the epoch as 1, learning as 0.05, and the L2 penalty as 0, 0.001, and 0.1 respectively. I record the loss and f1 score every 50 steps and build two figures for the results.

In the relationship between the step and loss figure, we can see that with the increase of the L2 penalty, the decrease of loss becomes slow. That is because the L2 penalty is designed for combating overfitting by forcing weights to be small; therefore, it will make the change slightly and might avoid the overfitting problem. However, when we

set the L2 penalty to 0.1, it will cause over-regularization which makes the decrease of loss very slow. Therefore, improving L2 will cause a decrease in convergence speed.



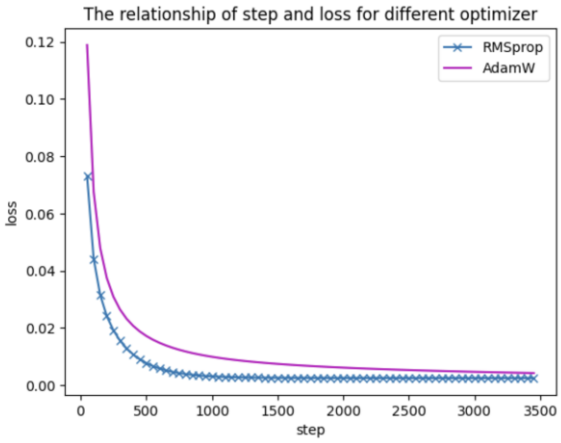
In the relationship between the step and the f1 score figure, we can see that with the increase of the L2 penalty, the increase of the f1 score becomes slow. A small L2 penalty might help improve the performance of this model, but when we set the L2 penalty as 0.1, it will make the model bad.



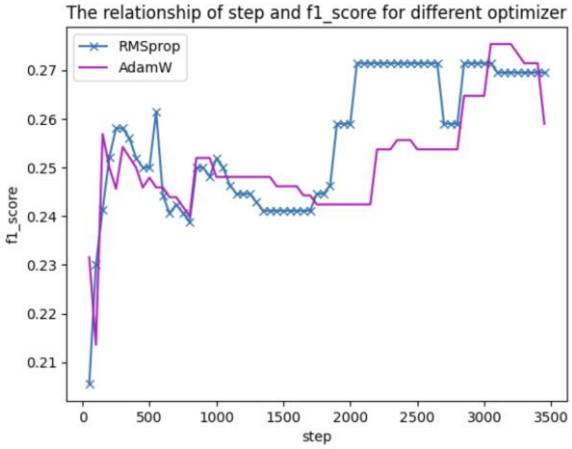
3.3.4 Train my model with RMSprop and AdamW optimizers

Furthermore, I replace my SGD optimizer with RMSprop and AdamW respectively. Here are the results, we can for the two optimizers, both of the loss functions drop dramatically within the first 1000 steps; after that, they keep almost unchanged.

Therefore, RMSprop and AdamW optimizers will cause quick convergence.

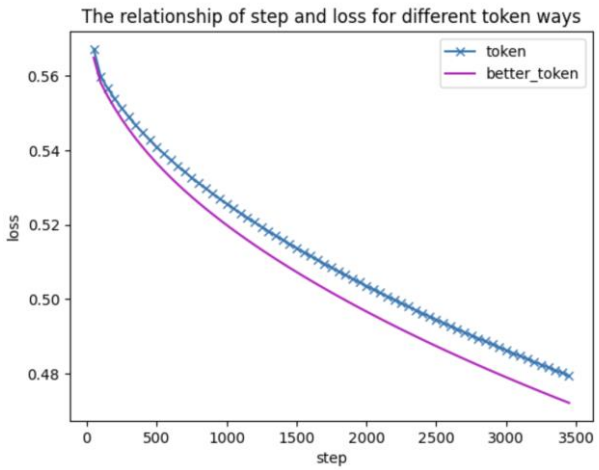


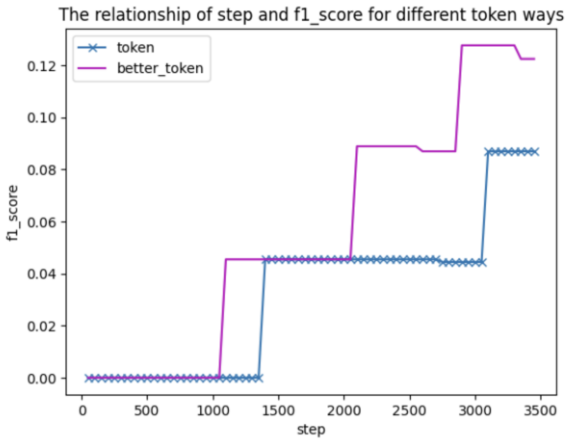
For the f1 score, both of them increase dramatically within the first 250 steps; after that, it vibrates between 0.24 to 0.26, and finally reaches a high f1 score of about 0.27 and continues to vibrate around 0.27. This score is lower than the score I get in 3.2.2 (0.3), the performance of the two optimizers is worse than the SGD optimizer.



3.3.5 Train my model with two different tokenizing ways

We can see that when I use a better tokenizing way, the loss of the model will drop quickly and the increase of the f1 score also go up quickly. Therefore, tokenizing can influence the performance of the model; at least, it can improve the training speed of this model.

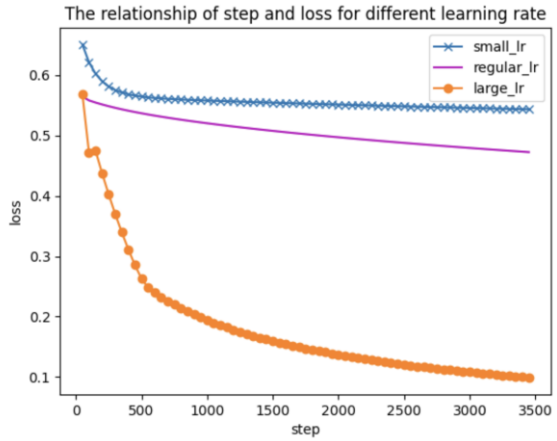


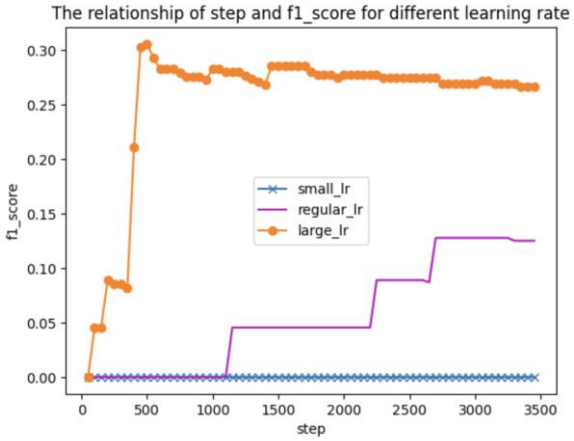


3.3.6 Train my model with three different learning rates

I also pay attention to the influence of the learning rate on the model. I choose three different learning rates and name them small learning rate (0.005), regular learning rate (0.05), and large learning rate (5).

As the figures show, we can see the increase in learning rate will also increase the speed of loss decrease and cause a quick convergence speed. And for the f1 score, we can see the increase in learning rate will promote the speed of the f1 score increase. When the learning rate is 0.005, the f1 score even retains 0, which means we should increase the learning rate to make the model converge quickly; when the learning rate is too big, the training model might converge too quickly and even miss the optimal solution and reach to the suboptimal point, which is also inadvisable.





3.4 Conclusion

Finally, I compare all of the models I build and choose the model I introduced in 3.3.2, which reached the highest f1 score (about 3.2). Next, I build a function called get_predict to predict the result of the test data set and save the result to a CSV file.

```
# Predict for test set
def get_predict():
    y_test_predicted = model(X_test)
    y_test_predicted_cls = y_test_predicted.round()
    y_test_value = y_test_predicted_cls.cpu().detach().numpy()
    return y_test_value

y_test_value = get_predict()

label_list = []
for i in y_test_value:
    label_list.append(int(i[0]))

df = pd.DataFrame(label_list,columns=["prediction"])
df.index.name="index"
df.to_csv('pytorch_predict.csv')
```

After submitting this file to Kaggle, I got an F1 score of 0.29946.

4. Summary

In summary, I use two different methods to predict the f1 score for the test dataset. First, I design the Logistic Regression Algorism by NumPy library and reach the f1 score of 0.25668. Second, I implement the PyTorch library and get the f1 score of 0.29946, which is higher than the score I get using the NumPy library. Therefore, using the PyTorch library will result in a better model for me.