CSE 156 Assignment 2

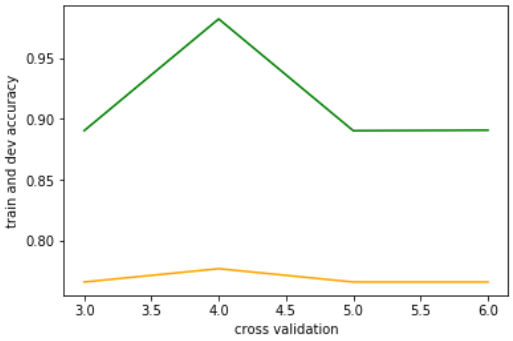
Xinmeng Li A9[redacted], kaggle:xinmeng0506@gmail.com, Teamname: Meng

- Supervised

Hyperparameter search

For every step that I have worked through below, I used GridSearch to find the optimal hyperparameters combination. The max_iter is set to 10000, as it might take more than the default max iterations for the solver to converge. Parameters that I have tuned for logistic regression classifier are {'solver' : ['lbfgs', 'liblinear', 'sag', 'saga'],'tol':[0.00001,0.0001], 'C':[0.001, 0.01, 0.1,1]}. Among them, solvers are optimization algorithms, tol decides the tolerance for stopping criterion, and C represents the regularization value, a small C value means strong regularization.

Parameters that I have tuned for GridSearch is cross validation.
Cv = [3,4,5,6]
trainacc = [0.8904408555216062, 0.9821038847664775, 0.8904408555216062, 0.890877346137058]
devacc = [0.7663755458515283, 0.777292576419214, 0.7663755458515283, 0.7663755458515283]
We observe that the classifier is overfitting on the training data when cv = 4, which is abnormal. However, the performance of classifier on dev dataset is stable. Therefore, we are supposed to choose higher folds to avoid overfitting. As with a higher cv number, we split the training dataset into more subsets, randomly choose train and validation subsets, and thus the probability of overfitting will decrease.



Preprocessing

Considering that the meaning of punctuation and stopwords, they appear frequently but they actually don't provide important information for the prediction of positive or negative. Thus, punctuation and stopwords in each sentence are removed. Also, to avoid the case that same words with uppercase and lowercase are counted separately, all of the words are transferred into lowercase.
However, the accuracy is not improved after we proprocess the text, with cv = 6, I got accuracy on train 0.8878219118288957 and accuracy on dev 0.7554585152838428, both accuracy on train and dev actually decrease. I have printed the decisive feature for classifier with parameter random_state=0, max_iter=10000,C = 0.1, solver= 'saga', tol= 0.0001 without preprocessing text. We observe that there are several stopwords such as *no, why, wasn, should,* but no punctuation in the list. So now it is reasonable for me to get a lower accuracy after processing. Some stopwords, unlike my previous assumption, might have a strong relationship with the prediction. So I give up removing stopwords and choose to only remove punctuation and make all words lowercase, and then get accuracy on train is 0.8893496289829769 and accuracy on dev is 0.7641921397379913.

Top k=50

['find', 'tacos', 'spicy', 'know', 'hidden', 'hands', 'yes', 'recommend', 'right', 'authentic', 'quick', 'attentive', 'enjoyed', 'vegas', 'menu', 'house', 'fast', 'city', 'always', 'quickly', 'yum', 'thai', 'can', 'surprised', 'montreal', 'fun', 'professional', 'style', 'pleased', 'definitely', 'tasty', 'easy', 'happy', 'wine', 'loved', 'town', 'little', 'fantastic', 'wonderful', 'spot', 'perfect', 'friendly', 'favorite', 'love', 'awesome', 'best', 'great', 'excellent', 'delicious', 'amazing']

Bottom k=50

['worst', 'horrible', 'terrible', 'rude', 'not', 'disappointing', 'disappointed', 'slow', 'average', 'asked', 'star', 'bad', 'awful', 'overpriced', 'excited', 'ok', 'went', 'dirty', 'bland', 'poor', 'seems', 'nothing', 'tasted', 'rather', 'should', 'meh', 'waitress', 'expensive', 'told', 'room', 'changed', 'instead', 'group', 'customer', 'order', 'money', 'however', 'high', 'run', 'could', 'wanted', 'no', 'would', 'ordered', 'rooms', 'called', 'unfortunately', 'wasn', 'why', 'disappointment']

TF-IDF weighting

We want to assign higher weights to words that both appeared frequently and have important information for label, so we calculate tf score for the term occurrence in a sentence, and the idf score for the rarity/importance of term through the whole corpus.

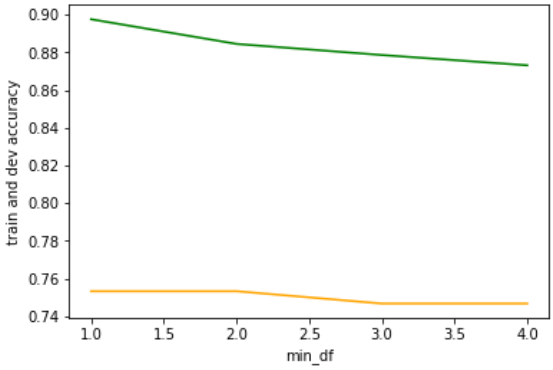
After do GrisSearch on cv6, we get accuracy on train is 0.8884766477520734 and accuracy on dev is 0.7729257641921398 with min_df=2, max_df=0.3, which is a large improvement that agrees on our assumption.

Tf-idf does a better job to find informative and rare features than just count the occurance of terms.

Here I tune the parameters min_df and max_df. Terms occurs strictly less than min_df and higher than the portion of max_df in the document will be ignored. (i.e. integer for absolute counts and float for portion)

If we keep the max_df be the default value 1.0 and tune min_df from 1 to 4, we will get train acc [0.8976429506765604, 0.8845482322130074, 0.8786556089044085, 0.8731994762112615]

dev acc [0.7532751091703057, 0.7532751091703057, 0.7467248908296943, 0.7467248908296943]

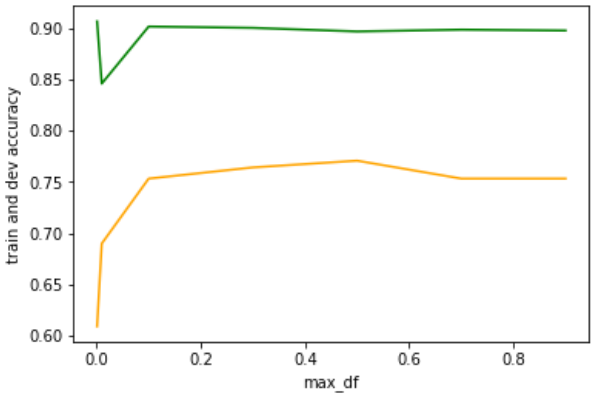


We observe that the trend goes down for both train and dev accuracy.

If we keep min_df to be the default value and tune max_df on 0.001,0.01,0.1,0.3,0.5, 0.7,0.9 we will get

train acc [0.9065910082933217, 0.8459188127455259, 0.9015713662156264, 0.9002618943692711, 0.8967699694456569, 0.898515931907464,0.8976429506765604]

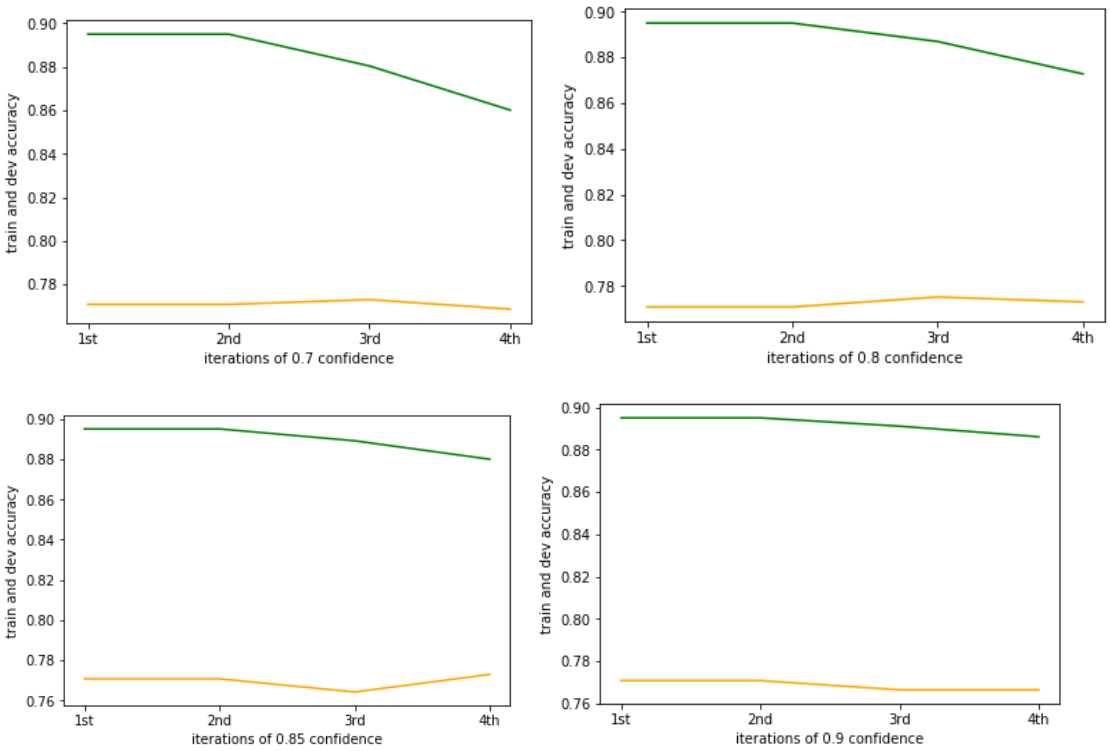
dev acc [0.6091703056768559, 0.6899563318777293, 0.7532751091703057, 0.7641921397379913, 0.7707423580786026,0.7532751091703057,0.7532751091703057]



We observe that the dev accuracy increases at first and then decreases and training acc is fairly stable for max_df>=0.1 as max_df increases.
Combining the above results, I use min_df= 1and max_df=0.5 for the future, which has Accuracy on train is: 0.8967699694456569 and Accuracy on dev is: 0.7707423580786026.
I have submitted this on Kaggle with team name Meng and email address xinmeng0506@gmail.com

- Exploiting the Unlabeled Data

I use self-training for semi-supervised classifier, which is basically train the classifier on the current training dataset, make predictions, and if the predictions are convincing, add their corresponding features to the feature matrix of training text and these predictions to labels of training dataset. Repeat this procedure until the stop criterion is met.
- Did changing the labeling strategy help/ not help?
Two labeling strategies: The first is to add the label with high confidence, the second is to check if the prediction result of logistic regression is equal to another classifier, such as SVM.
Logically, it is unreasonable to blindly add every prediction of unlabeled data into our training corpus, as the accuracy of our existing classifier is not that high (around 0.78) and thus sometimes make mistakes. For example, if in a prediction, positive is assigned 0.55 and negative is assigned 0.45, the classifier is actually not very confident about prediction result, add this prediction to the training set might have a negative impact for the further training. Thus, I choose to set a threshold for the confidence level and tune it. Due to the limitation of computation power and time, I use 30% portion of unlabeled data. As the shape of dev dataset is (458, 9425), 458/100 is around 4, I choose to stop after 4 iterations.
If I add predictions with probability higher than 0.7, I will get Training accuracy array [0.8950240069838499, 0.8950240069838499, 0.8804015713662157, 0.8601047577477084]
Dev accuracy array [0.7707423580786026, 0.7707423580786026, 0.7729257641921398, 0.7685589519650655].
Similarly, for confidence level = 0.8,0.85,0.9, I respectively got 0.8 confidence:
Training accuracy array [0.8950240069838499, 0.8950240069838499, 0.8869489305979922, 0.8727629855958097] Dev accuracy array [0.7707423580786026, 0.7707423580786026, 0.7751091703056768, 0.7729257641921398], 0.85 confidence : Training accuracy array [0.8950240069838499, 0.8950240069838499, 0.889131383675251, 0.8799650807507639]
Dev accuracy array [0.7707423580786026, 0.7707423580786026, 0.7641921397379913, 0.7729257641921398], 0.9 confidence: Training accuracy array [0.8950240069838499, 0.8950240069838499, 0.8910955914447839, 0.8860759493670886]
Dev accuracy array [0.7707423580786026, 0.7707423580786026, 0.7663755458515283, 0.7663755458515283]
Graphs are more intuitive. We observe that, as the confidence level increases, the fluctuation of training and dev accuracy decreases. Although the dev accuracy for confidence 0.7,0.8,0.85 are good, their train accuracy decreases rapidly. So I use 0.9 confidence in the future.



The second strategy is to use another classifier to predict the label, if its result is same as the result of logistic regression, then add this pair of unlabeled data and predicted label to the training corpus. I chose SVM, as it is popular and always performs well on binary classification. I chose LinearSVC as it is fast.

After run LinearSVC as a supervised classifier, I got accuracy on train is: 0.9855958096900916, accuracy on dev is: 0.7838427947598253, which is, just as what I expected, higher than the accuracy of logistic regression that I have got in the first part. With 30% portion, I got train acc: 0.8967699694456569, 0.8780008729812309, 0.8738542121344391, 0.8731994762112615, dev acc: 0.7707423580786026, 0.7663755458515283, 0.7685589519650655, 0.7729257641921398. This result is actually slightly better than the first strategy, but it still does not outperform the highest supervised classifier score.

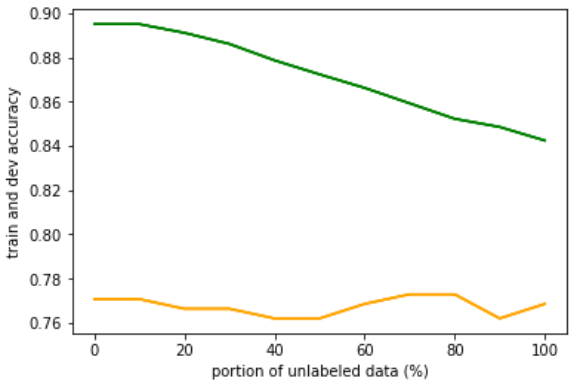
- Did the portion of unlabelled data you use help/ not help?

There is no obvious linear relationship between the portion of unlabelled data and the dev accuracy. As the portion of unlabelled data increases, the training accuracy decreases and dev accuracy fluctuates.

With the portion of unlabelled data increases, for 0.9 confidence, with the confidence level strategy, from 0% to 100%, I got Training accuracy array [0.8950240069838499, 0.8950240069838499, 0.8910955914447839, 0.8860759493670886, 0.8786556089044085, 0.872326494980358, 0.8662156263640332, 0.8592317765168049, 0.8522479266695766, 0.8485377564382366, 0.8424268878219118] Dev accuracy array [0.7707423580786026, 0.7707423580786026, 0.7663755458515283, 0.7663755458515283, 0.7620087336244541, 0.7620087336244541, 0.7685589519650655, 0.7729257641921398, 0.7729257641921398, 0.7620087336244541, 0.7685589519650655].

Note: here I still use the original training set for calculating the training accuracy, since I am not sure if the prediction label that I have added to the training corpus is correct.

The dev accuracy fluctuates might because I use GridSearch when I train the model in every iteration in order to find the optimal logistic regression classifier. Consequently, the hyperparameters settings change in every iteration.



Also, with the portion increases, we observe the decisive features/weightings of words change. For example, in the top 12 words for positive label, ‘good’ does not appear after I use more than 50% portion of unlabeled data. This is because there are words found to be more associated with positive label after we train the model based on a large portion of unlabeled data. Same for the decisive negative words.

Top k=12 for loop 0

['good', 'very', 'favorite', 'always', 'friendly', 'excellent', 'awesome', 'delicious', 'amazing', 'love', 'best', 'great']

Top k=12 for loop 1

['good', 'very', 'favorite', 'always', 'friendly', 'excellent', 'awesome', 'delicious', 'amazing', 'love', 'best', 'great']

Top k=12 for loop 2

['good', 'very', 'favorite', 'always', 'friendly', 'excellent', 'awesome', 'delicious', 'amazing', 'love', 'best', 'great']

Top k=12 for loop 3

['good', 'very', 'favorite', 'always', 'friendly', 'excellent', 'awesome', 'delicious', 'amazing', 'love', 'best', 'great']

Top k=12 for loop 4

['good', 'favorite', 'very', 'always', 'friendly', 'excellent', 'awesome', 'delicious', 'amazing', 'best', 'love', 'great']

Top k=12 for loop 5

['staff', 'favorite', 'very', 'always', 'excellent', 'friendly', 'awesome', 'delicious', 'amazing', 'best', 'love', 'great']

Top k=12 for loop 6

['favorite', 'very', 'staff', 'always', 'excellent', 'friendly', 'awesome', 'delicious', 'amazing', 'best', 'love', 'great']

Top k=12 for loop 7

['favorite', 'very', 'staff', 'always', 'excellent', 'awesome', 'friendly', 'delicious', 'amazing', 'best', 'love', 'great']

Top k=12 for loop 8

['atmosphere', 'very', 'staff', 'excellent', 'always', 'awesome', 'delicious', 'amazing', 'friendly', 'best', 'love', 'great']

Top k=12 for loop 9

['very', 'atmosphere', 'excellent', 'staff', 'always', 'awesome', 'delicious', 'amazing', 'friendly', 'best', 'love', 'great']

Top k=12 for loop 10

['are', 'very', 'excellent', 'staff', 'awesome', 'delicious', 'always', 'amazing', 'best', 'friendly', 'love', 'great']

Bottom k=12 for loop 0

['not', 'worst', 'horrible', 'bad', 'terrible', 'but', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 1

['not', 'worst', 'horrible', 'bad', 'terrible', 'but', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 2

['not', 'worst', 'horrible', 'bad', 'terrible', 'but', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 3

['not', 'worst', 'horrible', 'bad', 'terrible', 'but', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 4

['not', 'worst', 'horrible', 'bad', 'terrible', 'but', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 5

['not', 'worst', 'horrible', 'bad', 'but', 'terrible', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 6

['not', 'worst', 'horrible', 'bad', 'but', 'terrible', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 7

['not', 'worst', 'horrible', 'bad', 'but', 'terrible', 'rude', 'went', 'do', 'would', 'be', 'no']

Bottom k=12 for loop 8

['not', 'horrible', 'worst', 'but', 'bad', 'terrible', 'went', 'rude', 'do', 'be', 'would', 'me']

Bottom k=12 for loop 9

['not', 'horrible', 'worst', 'but', 'bad', 'went', 'terrible', 'rude', 'do', 'be', 'me', 'no']

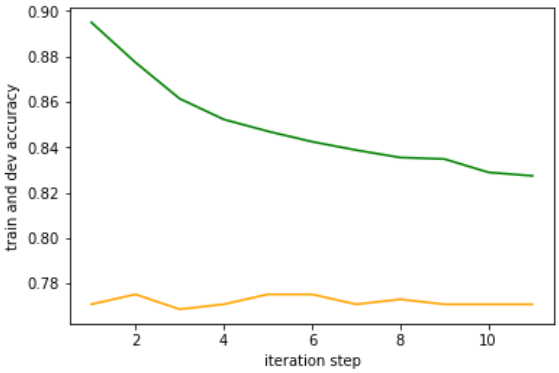
Bottom k=12 for loop 10

['not', 'horrible', 'worst', 'but', 'bad', 'went', 'do', 'terrible', 'rude', 'me', 'no', 'be']

- Did changing stopping criterion help/not help?

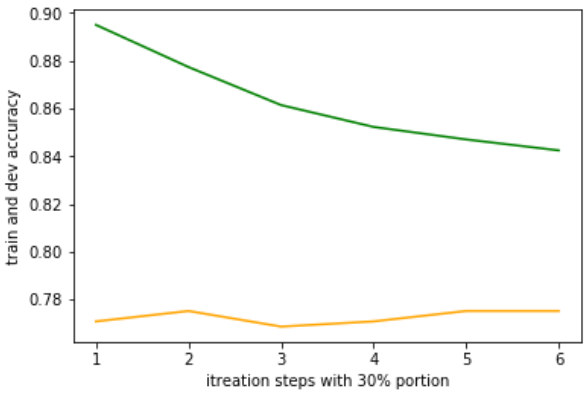
Two stopping criteria: The first is to set iteration steps based on the size of dev dataset, the second is to stop when there is almost no obvious improvement or reduction on dev dataset. I choose to stop the iterations when the increase of dev accuracy is less than 0.002 or the decrease of it is less than 0.01, which means the result almost converge.

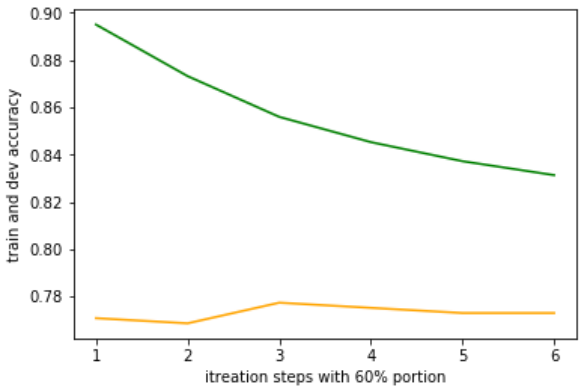
In the previous part, I use 4 as the iteration steps. However, in this part, I want to check whether 4 is enough for convergence. I also want to check whether my second algorithm for convergence works well. So I set the iteration steps to be large: 11. With confidence 0.8 and portion 0.3, I got train acc [0.8950240069838499, 0.8773461370580532, 0.8614142295940638, 0.8522479266695766, 0.8470100392841554, 0.8424268878219118, 0.8387167175905718, 0.8354430379746836, 0.8347883020515059, 0.828895678742907, 0.8273679615888259], dev acc [0.7707423580786026, 0.7751091703056768, 0.7685589519650655, 0.7707423580786026, 0.7751091703056768, 0.7751091703056768, 0.7707423580786026, 0.7729257641921398, 0.7707423580786026, 0.7707423580786026, 0.7707423580786026].



We observe that although on the 5th and 6th iteration, the dev accuracy are the same: 0.7751091703056768. It changes from the 7th iteration, and actually converges after 9th loop. This observation indicates that my second strategy actually has the risk of ‘fake convergence’ (i.e. geometrically, stops at a local window of flat line instead of the global convergence point.)

I also test the second strategy on 60% portion of data. Here is the comparison. It is interesting to observe that as the portion of unlabeled data increases, the speed of train acc reduction increases, but the dev acc converge faster/more stable.





- Conclusion

  Although I have tried multiple strategies, my result is still not ideal. I guess that's because I did not go through all the possible strategies and did not try to design better features in part 3.2.

Kaggle user name: glassm, Kaggle display name: Xinmeng Li, and email:xinmeng0506@gmail.com