

CS-UY 4563 Machine Learning

Final Project

Text Classification: Sentence pair in ASR

11:00AM Section

Dec 1, 2022

Professor Linda Sellie

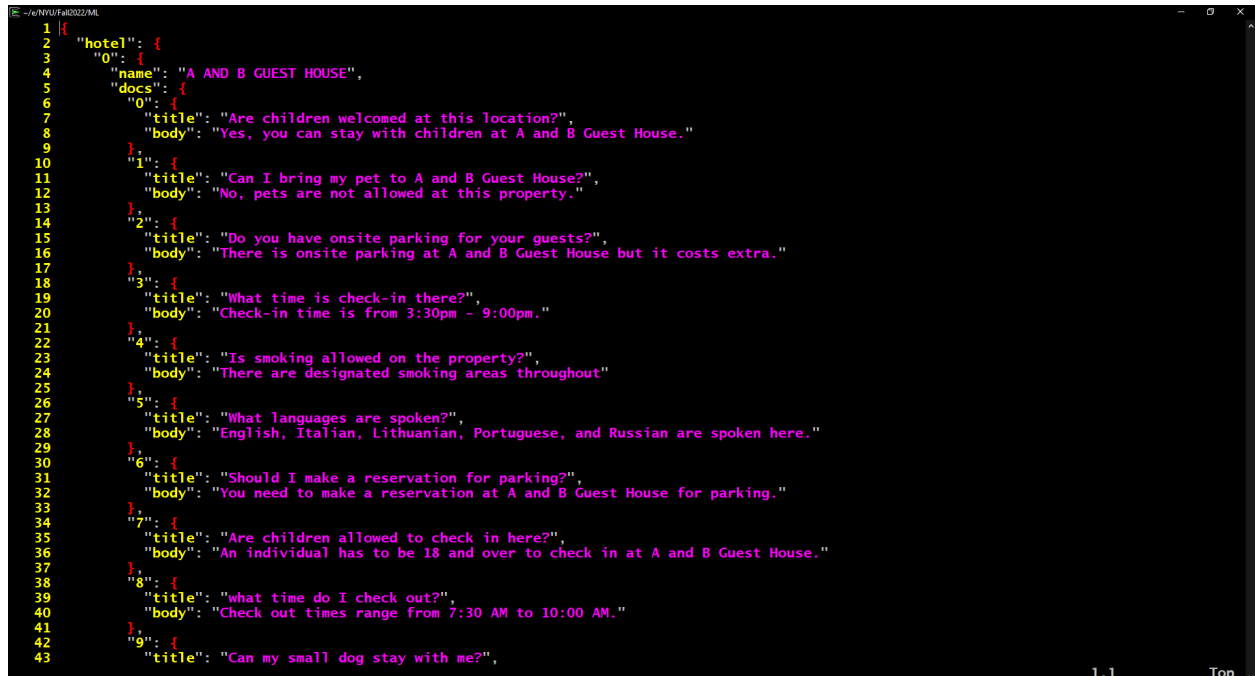
Seeger Zou

Introduction

In this report, I propose approaches to creating a binary classifier that can categorize if a sentence pair is a match. Our dataset consists of a knowledge base in the form of a json file and includes names of hotels, attractions, and restaurants. To do this, I will use various algorithms and methods to train and evaluate each model's results.

Dataset

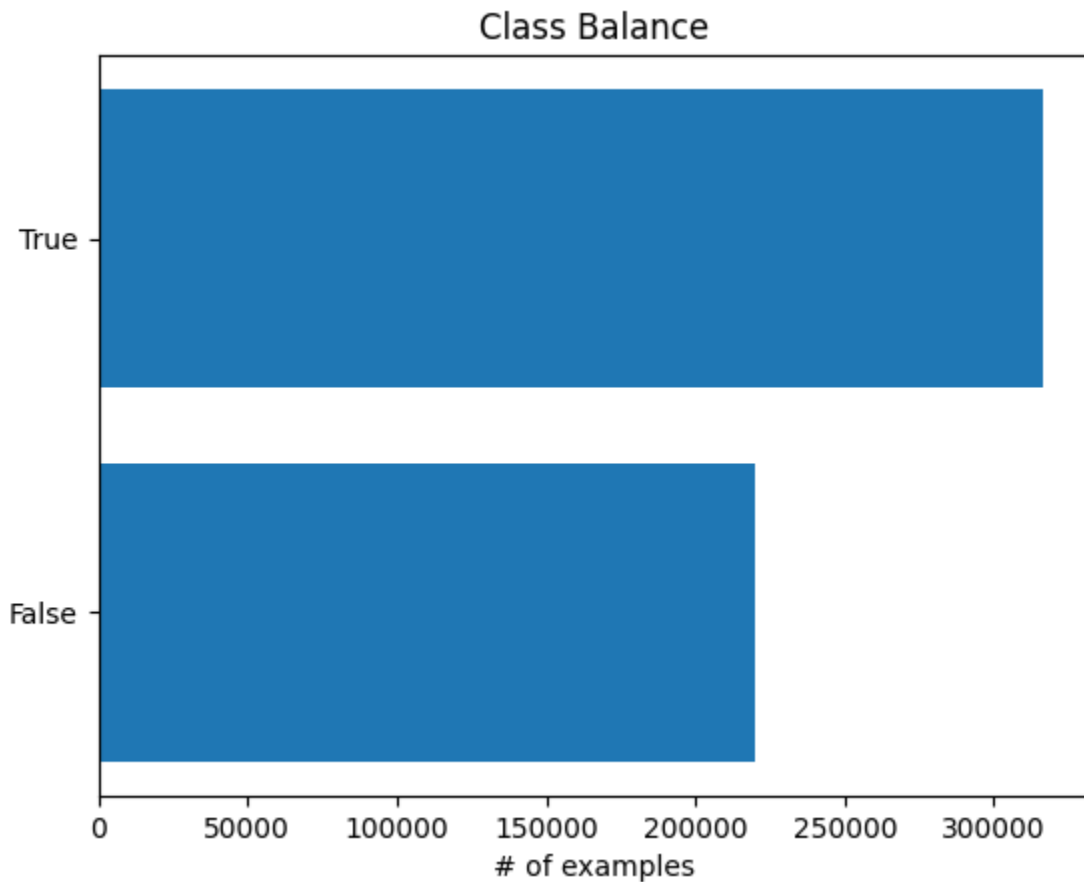
First, let's take a look at the dataset.



```
1 [{
2   "hotel": {
3     "id": "0",
4     "name": "A AND B GUEST HOUSE",
5     "docs": [
6       {
7         "id": "0",
8         "title": "Are children welcomed at this location?",
9         "body": "Yes, you can stay with children at A and B Guest House."
10      },
11      {
12        "id": "1",
13        "title": "Can I bring my pet to A and B Guest House?",
14        "body": "No, pets are not allowed at this property."
15      },
16      {
17        "id": "2",
18        "title": "Do you have onsite parking for your guests?",
19        "body": "There is onsite parking at A and B Guest House but it costs extra."
20      },
21      {
22        "id": "3",
23        "title": "What time is check-in there?",
24        "body": "Check-in time is from 3:30pm - 9:00pm."
25      },
26      {
27        "id": "4",
28        "title": "Is smoking allowed on the property?",
29        "body": "There are designated smoking areas throughout"
30      },
31      {
32        "id": "5",
33        "title": "What languages are spoken?",
34        "body": "English, Italian, Lithuanian, Portuguese, and Russian are spoken here."
35      },
36      {
37        "id": "6",
38        "title": "Should I make a reservation for parking?",
39        "body": "You need to make a reservation at A and B Guest House for parking."
40      },
41      {
42        "id": "7",
43        "title": "Are children allowed to check in here?",
44        "body": "An individual has to be 18 and over to check in at A and B Guest House."
45      },
46      {
47        "id": "8",
48        "title": "What time do I check out?",
49        "body": "Check out times range from 7:30 AM to 10:00 AM."
50      },
51      {
52        "id": "9",
53        "title": "Can my small dog stay with me?",
54        "body": ""
55      }
56    ]
57   }
58 }]
```

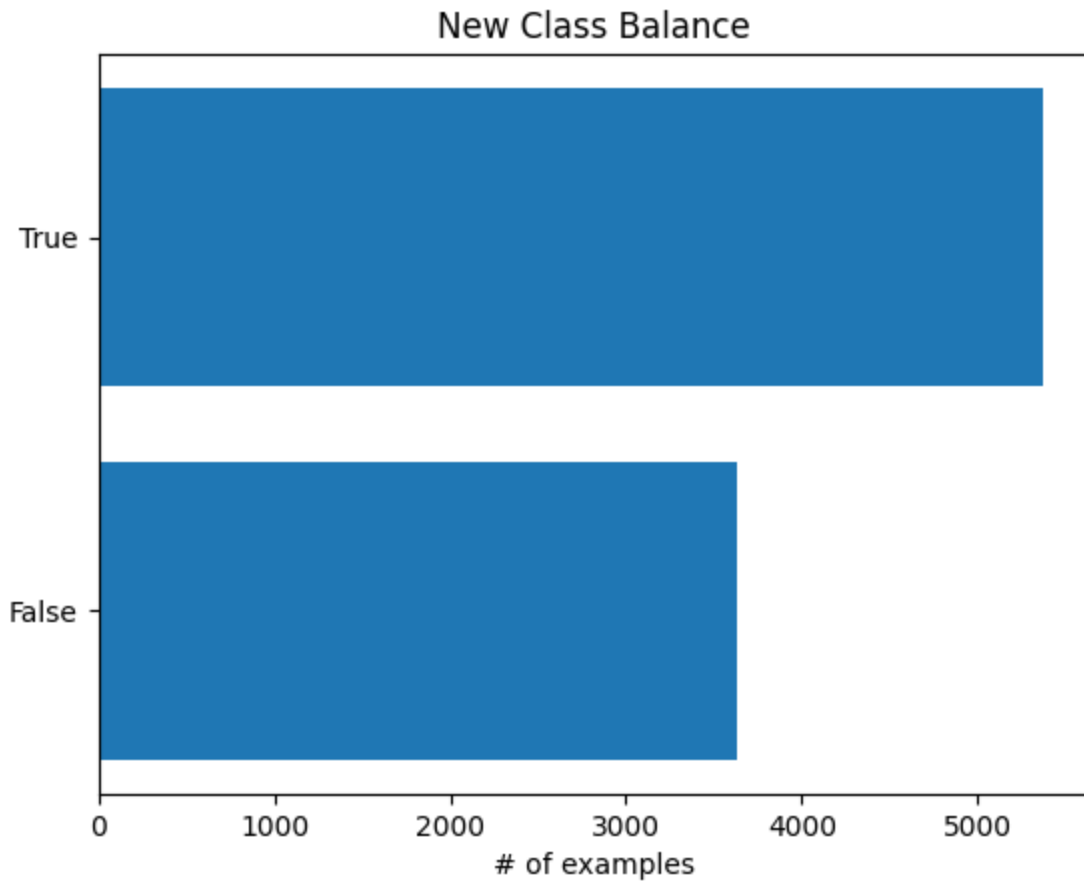
The dataset is not that complicated, it is a json formatted file, and similar to that of a python dictionary, where the first layer specifies the category (ex. hotel), the second layer specifies its unique id, and within that third layer is the information associated with the given id: name, docs, etc. In this task, we only care about the docs, since we need to classify if a given pair of sentences is valid. I have previously worked on this dataset and have preprocessed with Trie-matching Named Entity Recognition methods to delexicalize and clean up the data. Since we did not focus much on preprocessing techniques that I have used, and also for simplicity's sake, I have omitted detailed descriptions of this part of the process. I have

provided the classifier training examples and the classifier validation examples, which are randomly selected with a 9:1 training validation split and the distribution of classes are shown below.



As shown above, the distribution of classes is a 6:4 ratio with true and false values. A logical approach would be to have the same amount of true and false examples, but to encourage the model to predict more “boldly”, I decided to try out this class balance.

Later on, I would change the amount of examples to the same 9:1 training validation split, with the same 6:4 ratio with true and false values, but less examples total at 10,000. This is because I ran into problems with space and time complexity, especially when it comes to the svm models. For detailed explanation, see the **analysis section**.



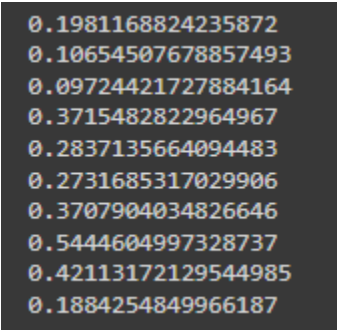
Now, let's look at the final dataset that we will be working with:

```
1 | {
2 |   "title": "What parking is offered?",
3 |   "body": "It offers off street, street, and validated parking.",
4 |   "label": "true"
5 | },
6 | {
7 |   "title": "Can you accommodate large groups?",
8 |   "body": "It does not offer free WiFi.",
9 |   "label": "false"
10 | },
11 | {
12 |   "title": "Is there a gym on site?",
13 |   "body": "It does not have an onsite fitness center.",
14 |   "label": "true"
15 | },
16 | {
17 |   "title": "What are my payment options available?",
18 |   "body": "The Club Quarters Hotel accepts charge or cash.",
19 |   "label": "true"
20 | },
21 | {
22 |   "title": "Do they offer seating outside?",
23 |   "body": "It does not have outdoor seating",
24 |   "label": "true"
25 | },
26 | {
27 |   "title": "Do you have wifi availability?",
28 |   "body": "It does offer its guests free WiFi.",
29 |   "label": "true"
30 | },
31 | {
32 |   "title": "do you have take out?",
33 |   "body": "Yes, it has take-out.",
34 |   "label": "true"
35 | },
36 | {
37 |   "title": "Does it have free WiFi?",
38 |   "body": "No, it is not recommended for groups.",
39 |   "label": "false"
40 | },
41 | {
42 |   "title": "Does it accept credit cards?",
43 |   "body": "It does not have an onsite fitness center.",
44 |   "label": "true"
45 | },
46 | }
```

classifier_examples_train.json [noel] 2414087L, 70434466B 1,1 Top

Some observations can be made: (1) The document consists of a title and body pair, along with a label that is either true or false, representing if the pair is valid or not. (2) Most title sentences end with a question mark, which can be important in determining the end of a question. Let's keep the punctuations in there for later. We have to keep these in mind when we train our model.

That being said, with the task being a text classification problem, let's formulate these queries into data that we can pass into the model. A intuitive way to is to combine the title and body into a single sentence, separated by a space character, and that will be our features that we will later tokenize. The label then will be encoded into a 1 for true and 0 for false and will be our target/predicted variable. In this project, I will use three types of tokenization methods: binary, count, and Tfidf. Each of these methods are then experimented with scaling and how that would effect the accuracy of the models. Below is an example of the nonzero contents within the first element of the array, which is the sentence "What parking is offered? It offers off street, street, and validated parking."



```
0.1981168824235872  
0.10654507678857493  
0.09724421727884164  
0.3715482822964967  
0.2837135664094483  
0.2731685317029906  
0.3707904034826646  
0.5444604997328737  
0.42113172129544985  
0.1884254849966187
```

Here is what a part of the dictionary looks like:

```
1 # vocab of the tokenizer
2 tokenizer.vocabulary_
```

```
'galleria': 675,
'joie': 829,
'vivre': 1578,
'thing': 1464,
'connect': 414,
'hampton': 722,
'downtwon': 535,
'convention': 426,
'drop': 545,
'enough': 575,
'instabul': 803,
'multilingual': 999,
'stated': 1390,
'refundable': 1219,
'ride': 1254,
'flyer': 641,
'pregnant': 1165,
'unaccompanied': 1526,
'40': 43,
'inches': 786,
'taller': 1439,
'noon': 1028,
'moderate': 983,
'ammenities': 159,
'opening': 1071,
```

An important point here is that most of our array will be 0. We categorize our tokenized array a sparse matrix, therefore it is important that we convert it to a dense array for space and time efficiency. Below is a sample of a sparse matrix in compressed sparse row format:

```

(0, 1604)    0.1884254849966187
(0, 1550)    0.42113172129544985
(0, 1402)    0.5444604997328737
(0, 1106)    0.3707904034826646
(0, 1051)    0.2731685317029906
(0, 1047)    0.2837135664094483
(0, 1043)    0.3715482822964967
(0, 815)     0.09724421727884164
(0, 812)     0.10654507678857493
(0, 165)     0.1981168824235872
(1, 1642)    0.17123946985780167
(1, 1617)    0.21902232310438233
(1, 1044)    0.21701672668300245
(1, 1035)    0.1869041190445836
(1, 858)     0.5486673404809553
(1, 815)     0.10677167565523445
(1, 706)     0.36946960039117654
(1, 657)     0.2110448645153775
(1, 520)     0.16121302975125035
(1, 315)     0.23227911491803716
(1, 95)      0.5169621520922558
(2, 1459)    0.16334847919408646
(2, 1336)    0.3160262922127847
(2, 1067)    0.31181700763895026
(2, 1060)    0.3081102636988631
:           :

```

A standard scaler is also sometimes used to experiment with the model results (sklearn's StandardScaler). Now, our data is tokenized and ready to feed into the models.

Logistic regression

Since we do not yet know the relation between each tokenization methods and their scaled counterparts, I conducted scaled and unscaled training on all models this section. I have decided to train using the “liblinear” model from sklearn, and tuning the hyperparameter C to experiment with regularization. The metrics of the corresponding models are shown below:

Unscaled binary:

C=0.001

	precision	recall	f1-score	support
0	0.74	0.05	0.10	443
1	0.57	0.99	0.72	557
accuracy			0.57	1000
macro avg	0.65	0.52	0.41	1000
weighted avg	0.64	0.57	0.44	1000

C=5

	precision	recall	f1-score	support
0	0.92	0.77	0.84	443
1	0.84	0.95	0.89	557
accuracy			0.87	1000
macro avg	0.88	0.86	0.86	1000
weighted avg	0.88	0.87	0.87	1000

C=100000

	precision	recall	f1-score	support
0	0.89	0.75	0.82	443
1	0.82	0.93	0.87	557
accuracy			0.85	1000
macro avg	0.86	0.84	0.84	1000
weighted avg	0.86	0.85	0.85	1000

Scaled binary:

C=0.001

	precision	recall	f1-score	support
0	0.82	0.60	0.69	443
1	0.74	0.89	0.81	557
accuracy			0.76	1000
macro avg	0.78	0.75	0.75	1000
weighted avg	0.77	0.76	0.76	1000

C=5

	precision	recall	f1-score	support
0	0.89	0.75	0.81	443
1	0.82	0.93	0.87	557
accuracy			0.85	1000
macro avg	0.86	0.84	0.84	1000
weighted avg	0.85	0.85	0.85	1000

C=100000

	precision	recall	f1-score	support
0	0.89	0.74	0.81	443
1	0.82	0.93	0.87	557
accuracy			0.85	1000
macro avg	0.86	0.84	0.84	1000
weighted avg	0.85	0.85	0.84	1000

Unscaled count:

C=0.001

	precision	recall	f1-score	support
0	0.70	0.09	0.16	443
1	0.57	0.97	0.72	557
accuracy			0.58	1000
macro avg	0.63	0.53	0.44	1000
weighted avg	0.63	0.58	0.47	1000

C=5

	precision	recall	f1-score	support
0	0.71	0.51	0.59	443
1	0.68	0.83	0.75	557
accuracy			0.69	1000
macro avg	0.69	0.67	0.67	1000
weighted avg	0.69	0.69	0.68	1000

C=100000

	precision	recall	f1-score	support
0	0.70	0.51	0.59	443
1	0.68	0.83	0.75	557
accuracy			0.69	1000
macro avg	0.69	0.67	0.67	1000
weighted avg	0.69	0.69	0.68	1000

Scaled count:

C=0.001

	precision	recall	f1-score	support
0	0.72	0.46	0.56	443
1	0.67	0.86	0.75	557
accuracy			0.68	1000
macro avg	0.69	0.66	0.66	1000
weighted avg	0.69	0.68	0.67	1000

C=5

	precision	recall	f1-score	support
0	0.70	0.51	0.59	443
1	0.68	0.82	0.74	557
accuracy			0.69	1000
macro avg	0.69	0.67	0.67	1000
weighted avg	0.69	0.69	0.68	1000

C=100000

	precision	recall	f1-score	support
0	0.70	0.51	0.59	443
1	0.68	0.83	0.75	557
accuracy			0.69	1000
macro avg	0.69	0.67	0.67	1000
weighted avg	0.69	0.69	0.68	1000

Unscaled Tfidf

C=0.001

	precision	recall	f1-score	support
0	0.00	0.00	0.00	443
1	0.56	1.00	0.72	557
accuracy			0.56	1000
macro avg	0.28	0.50	0.36	1000
weighted avg	0.31	0.56	0.40	1000

C=5

	precision	recall	f1-score	support
0	0.82	0.63	0.71	443
1	0.75	0.89	0.81	557
accuracy			0.77	1000
macro avg	0.78	0.76	0.76	1000
weighted avg	0.78	0.77	0.77	1000

C=100000

	precision	recall	f1-score	support
0	0.85	0.70	0.77	443
1	0.79	0.90	0.84	557
accuracy			0.81	1000
macro avg	0.82	0.80	0.81	1000
weighted avg	0.82	0.81	0.81	1000

Scaled Tfidf

C=0.001

	precision	recall	f1-score	support
0	0.71	0.52	0.60	443
1	0.68	0.83	0.75	557
accuracy			0.69	1000
macro avg	0.70	0.67	0.68	1000
weighted avg	0.70	0.69	0.68	1000

C=5

	precision	recall	f1-score	support
0	0.85	0.69	0.76	443
1	0.79	0.90	0.84	557
accuracy			0.81	1000
macro avg	0.82	0.80	0.80	1000
weighted avg	0.82	0.81	0.81	1000

C=100000

	precision	recall	f1-score	support
0	0.85	0.70	0.77	443
1	0.79	0.90	0.84	557
accuracy			0.81	1000
macro avg	0.82	0.80	0.81	1000
weighted avg	0.82	0.81	0.81	1000

The best performance based on accuracy is unscaled binary with C=5 @ 0.87 accuracy. It would seem that compared to binary and tfidf, count whether scaled or unscaled, is outperformed. Some other observations can be made. For binary tokenization, scaling the data tend to worsen the accuracy, and the opposite is true for tfidf tokenization; scaling the data actually helps with the accuracy. It also appears that regularization is just right at C=5, and the model is worse with too much regularization, compared to the model with too little regularization. This might be the case since I have cleaned the dataset. At first, I thought tfidf would easily outperform the other two tokenization methods. It is possible that the model trained with binary tokenized data is better due to its ability to match keywords, i.e. if the a given word is present once, if it is present again at a later index (say, in the answer), the query is most likely a question answer pair.

Note that before our data sampling where we took out only 10k examples due to the large dataset, training this logistic regression model is possible. The best result I got is 0.90, also using the unscaled binary tokenization method.

Now that we have chosen our best logistic regression model, let's play around with it.

```
[13] 1 log_model.predict(tokenizer.transform(["Does it offer happy hours? It does not allow children below 6."]))
array([0])

[159] 1 log_model.predict(tokenizer.transform(["Does it offer a gym? No, it does not have happy hours."]))
array([1])

[160] 1 log_model.predict(tokenizer.transform(["Does it offer a gym on site? No, it does not have happy hours."]))
array([0])

[142] 1 log_model.predict(tokenizer.transform(["Is there a gym? No, it does not have happy hours."]))
array([1])
```

These examples are very interesting as we can observe some characteristics of the model. The first query was predicted correctly. However, when tweaked a little bit, the model seems to think that there are some correlations between “gym” and “happy hours”. With further testing though, and adding “on site” to the title, the model is able to predict correct again. A possible hypothesis here is that the model will lean more towards predicting true when the sentences are not long enough, as is the case for the fourth query.

Another similar algorithm is the Naive Bayes classifier, but due to time, I was not able to complete any experiments.

SVM

At this point when I was using the original large pool of training set, I ran into the problem with SVM’s memory and runtime. Since SVM is not scaled to large number of samples, and storing the kernel matrix requires memory that scales quadratically with the number of data points, the training time for SVM algorithms scales superlinearly with the number of data points. Generally, SVM algorithms are not feasible for large data sets.

I have tried, however, some tricks to work around this. An approach I used was the Nystrom approximation. Basically, the method approximates the eigenvalues/eigenvectors of a large matrix using a smaller submatrix, effectively thinning down the number of features from 4k+ to 300. Combining this

kernel approximation method with a linear SVM where the feature space mapping is defined implicitly by the kernel function, the SVMs don't explicitly compute feature space representations. Thus, this makes computations efficient for small to medium size datasets, and the feature space is changed from high dimensional to a lower dimensional space. As expected, though, since this is an approximation, the result is not so optimistic:

```
feature_map_nystroem = Nystroem(gamma=.2, random_state=1, n_components=300)
X_train_nystroem = feature_map_nystroem.fit_transform(X_train_tokenized)
X_test_nystroem = feature_map_nystroem.transform(X_test_tokenized)
```

```
▶ 1 support_vec_machine.fit(X_train_nystroem, Y_train)
✕ LinearSVC()

[ ] 1 support_vec_machine.score(X_train_nystroem, Y_train)
    0.774442490633097

[ ] 1 support_vec_machine.score(X_test_nystroem, Y_test)
    0.7742236140625582
```

This is the reason for a smaller dataset, so that these kernelized SVM algorithms can run within reasonable time and would not use up that much space. As discovered in the Logistic Regression models, our best estimates came from datasets with unscaled binary tokenizer and the scaled tfidf tokenizer. I will try these two methods for each kernelized SVM: linear, poly, and rbf. The results are shown below:

Unscaled Binary

Linear

	precision	recall	f1-score	support
0	0.92	0.76	0.83	443
1	0.83	0.95	0.89	557
accuracy			0.86	1000
macro avg	0.88	0.85	0.86	1000
weighted avg	0.87	0.86	0.86	1000

Poly

	precision	recall	f1-score	support
0	0.00	0.00	0.00	443
1	0.56	1.00	0.72	557
accuracy			0.56	1000
macro avg	0.28	0.50	0.36	1000
weighted avg	0.31	0.56	0.40	1000

RBF

	precision	recall	f1-score	support
0	0.85	0.27	0.41	443
1	0.62	0.96	0.76	557
accuracy			0.65	1000
macro avg	0.74	0.61	0.58	1000
weighted avg	0.72	0.65	0.60	1000

Scaled tfidf

Linear

	precision	recall	f1-score	support
0	0.87	0.71	0.78	443
1	0.80	0.92	0.85	557
accuracy			0.82	1000
macro avg	0.83	0.81	0.82	1000
weighted avg	0.83	0.82	0.82	1000

Poly

	precision	recall	f1-score	support
0	0.77	0.26	0.39	443
1	0.62	0.94	0.74	557
accuracy			0.64	1000
macro avg	0.69	0.60	0.57	1000
weighted avg	0.69	0.64	0.59	1000

RBF

	precision	recall	f1-score	support
0	0.88	0.73	0.80	443
1	0.81	0.92	0.86	557
accuracy			0.84	1000
macro avg	0.85	0.83	0.83	1000
weighted avg	0.84	0.84	0.83	1000

It would appear that the best model based on accuracy is the one using unscaled binary tokenizer and a linear kernel. Although, the one with scaled tfidf tokenizer and a rbf kernel is not bad. These results are not better than the logistic regression model, but let's try to see if the queries' response will change.

Warnings are for typeerrors but should not effect prediction.

```
1 support_vec_machine.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer happy hours? It does not allow children below 6."])))
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:593: FutureWarning: np.matrix usage is deprecated in 1.0 and will raise a TypeError in 1.2. Please convert
warnings.warn(
array([0])

1 support_vec_machine.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer a gym? No, it does not have happy hours."])))
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:593: FutureWarning: np.matrix usage is deprecated in 1.0 and will raise a TypeError in 1.2. Please conv
warnings.warn(
array([1])

1 support_vec_machine.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer a gym on site? No, it does not have happy hours."])))
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:593: FutureWarning: np.matrix usage is deprecated in 1.0 and will raise a TypeError in 1.2. Please conv
warnings.warn(
array([0])

1 support_vec_machine.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Is there a gym? No, it does not have happy hours."])))
/usr/local/lib/python3.8/dist-packages/sklearn/utils/validation.py:593: FutureWarning: np.matrix usage is deprecated in 1.0 and will raise a TypeError in 1.2. Please conv
warnings.warn(
array([1])
```

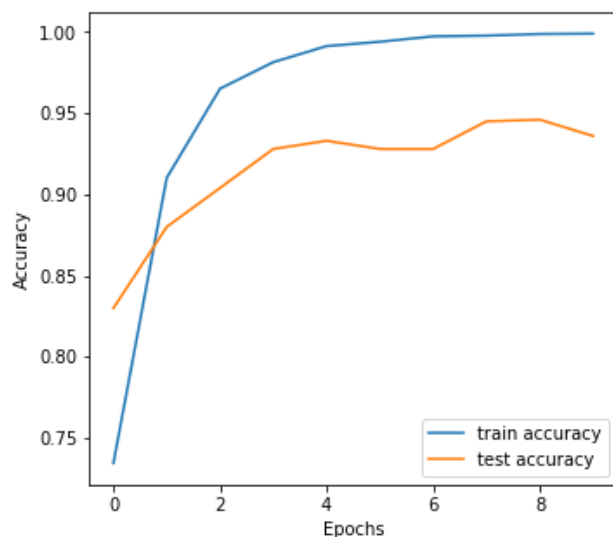
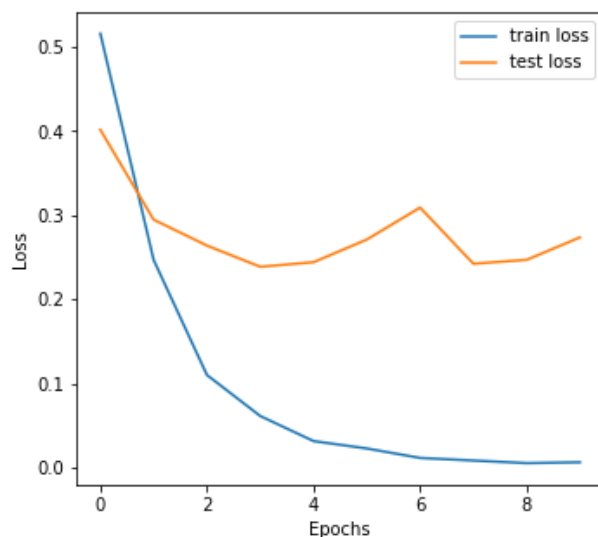

Neural Networks

With the neural network implementation, I decided to use tensorflow and keras to create a 3 layer neural network. I chose to implement the hidden layers using relu activations, and a softmax activation for the final layer. I will also try to utilize l2 and dropout regularization to try to see if I can improve the model performance by running 10 epochs on each model. Using the unscaled binary and scaled tfidf tokenizers for the data, again, the results are as follows. Note that plots are attached to each model.

Unscaled Binary

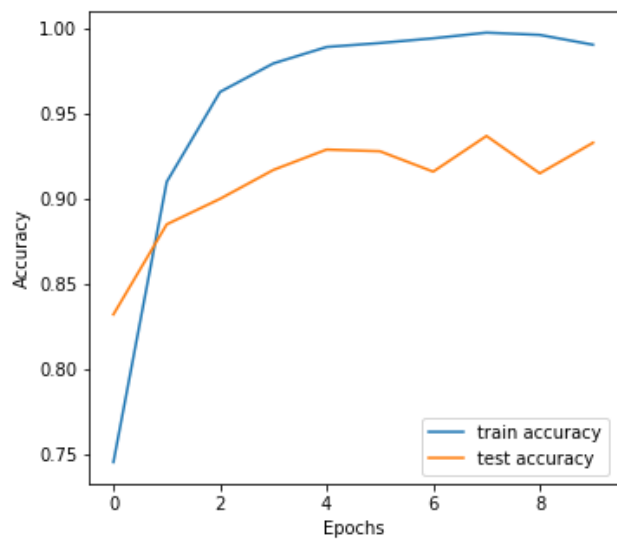
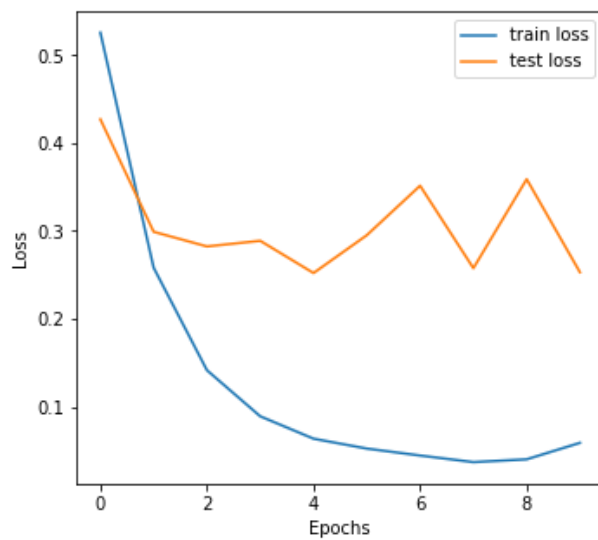
No regularization

```
Epoch 1/10
282/282 - 3s - loss: 0.5161 - accuracy: 0.7346 - val_loss: 0.4017 - val_accuracy: 0.8300 - 3s/epoch - 10ms/step
Epoch 2/10
282/282 - 2s - loss: 0.2468 - accuracy: 0.9102 - val_loss: 0.2947 - val_accuracy: 0.8800 - 2s/epoch - 6ms/step
Epoch 3/10
282/282 - 2s - loss: 0.1100 - accuracy: 0.9651 - val_loss: 0.2640 - val_accuracy: 0.9040 - 2s/epoch - 6ms/step
Epoch 4/10
282/282 - 2s - loss: 0.0612 - accuracy: 0.9814 - val_loss: 0.2387 - val_accuracy: 0.9280 - 2s/epoch - 6ms/step
Epoch 5/10
282/282 - 2s - loss: 0.0314 - accuracy: 0.9913 - val_loss: 0.2442 - val_accuracy: 0.9330 - 2s/epoch - 6ms/step
Epoch 6/10
282/282 - 2s - loss: 0.0226 - accuracy: 0.9940 - val_loss: 0.2711 - val_accuracy: 0.9280 - 2s/epoch - 6ms/step
Epoch 7/10
282/282 - 2s - loss: 0.0113 - accuracy: 0.9973 - val_loss: 0.3092 - val_accuracy: 0.9280 - 2s/epoch - 6ms/step
Epoch 8/10
282/282 - 1s - loss: 0.0082 - accuracy: 0.9978 - val_loss: 0.2423 - val_accuracy: 0.9450 - 1s/epoch - 5ms/step
Epoch 9/10
282/282 - 1s - loss: 0.0051 - accuracy: 0.9988 - val_loss: 0.2470 - val_accuracy: 0.9460 - 1s/epoch - 5ms/step
Epoch 10/10
282/282 - 2s - loss: 0.0061 - accuracy: 0.9990 - val_loss: 0.2737 - val_accuracy: 0.9360 - 2s/epoch - 6ms/step
```



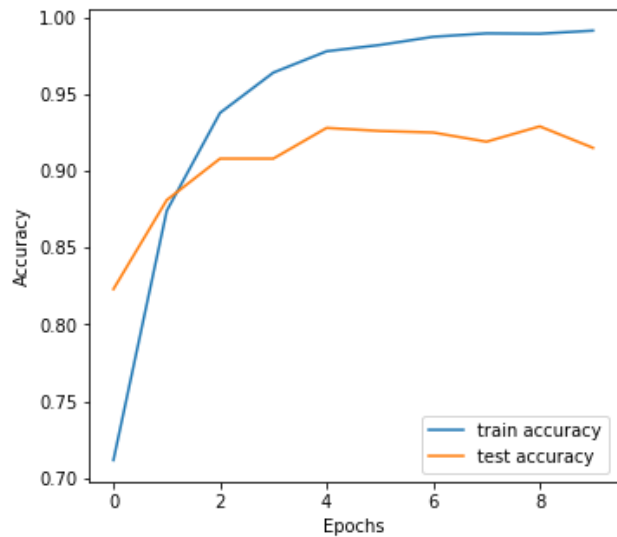
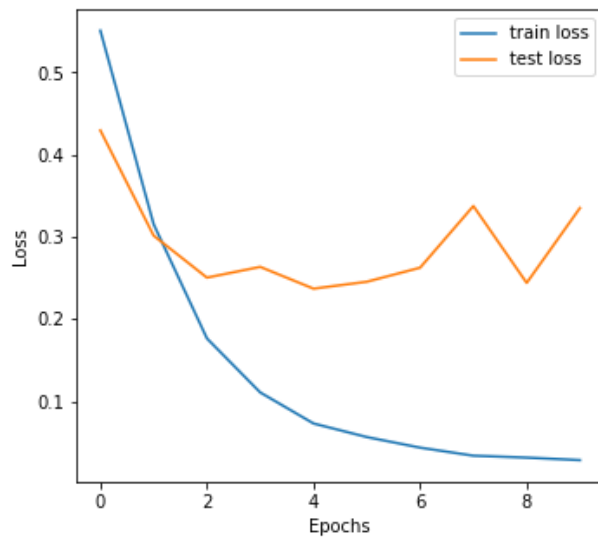
L2

```
Epoch 1/10
282/282 - 3s - loss: 0.5256 - accuracy: 0.7451 - val_loss: 0.4270 - val_accuracy: 0.8320 - 3s/epoch - 10ms/step
Epoch 2/10
282/282 - 2s - loss: 0.2582 - accuracy: 0.9100 - val_loss: 0.2991 - val_accuracy: 0.8850 - 2s/epoch - 7ms/step
Epoch 3/10
282/282 - 2s - loss: 0.1416 - accuracy: 0.9629 - val_loss: 0.2824 - val_accuracy: 0.9000 - 2s/epoch - 7ms/step
Epoch 4/10
282/282 - 1s - loss: 0.0892 - accuracy: 0.9797 - val_loss: 0.2888 - val_accuracy: 0.9170 - 1s/epoch - 5ms/step
Epoch 5/10
282/282 - 1s - loss: 0.0638 - accuracy: 0.9893 - val_loss: 0.2522 - val_accuracy: 0.9290 - 1s/epoch - 5ms/step
Epoch 6/10
282/282 - 2s - loss: 0.0525 - accuracy: 0.9917 - val_loss: 0.2952 - val_accuracy: 0.9280 - 2s/epoch - 6ms/step
Epoch 7/10
282/282 - 2s - loss: 0.0446 - accuracy: 0.9944 - val_loss: 0.3515 - val_accuracy: 0.9160 - 2s/epoch - 7ms/step
Epoch 8/10
282/282 - 2s - loss: 0.0371 - accuracy: 0.9978 - val_loss: 0.2577 - val_accuracy: 0.9370 - 2s/epoch - 7ms/step
Epoch 9/10
282/282 - 2s - loss: 0.0403 - accuracy: 0.9964 - val_loss: 0.3589 - val_accuracy: 0.9150 - 2s/epoch - 6ms/step
Epoch 10/10
282/282 - 2s - loss: 0.0590 - accuracy: 0.9907 - val_loss: 0.2530 - val_accuracy: 0.9330 - 2s/epoch - 7ms/step
```



Dropout

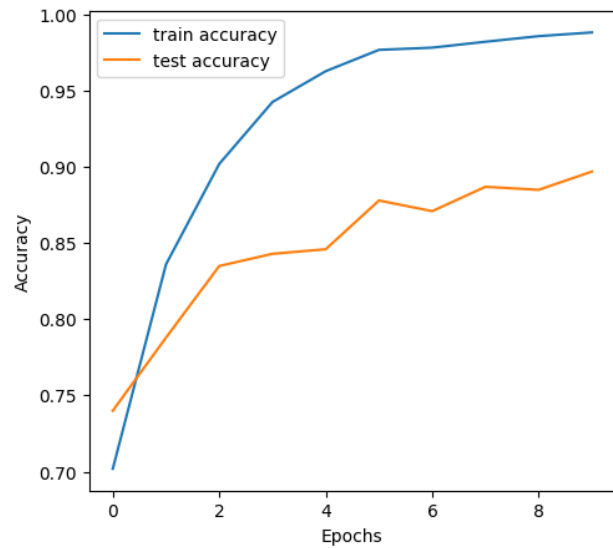
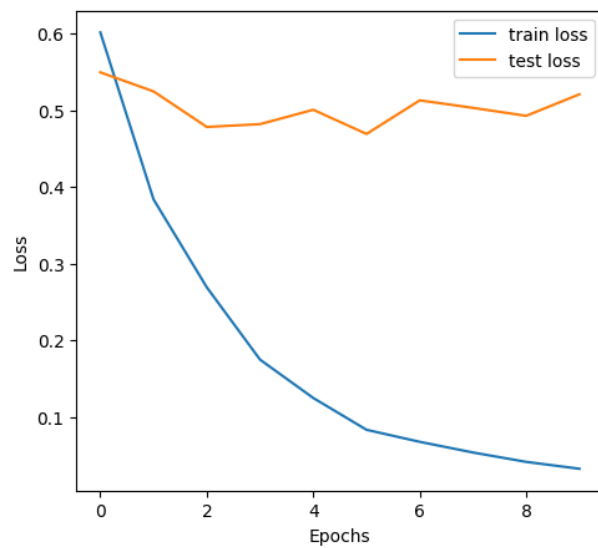
```
Epoch 1/10
282/282 - 2s - loss: 0.5509 - accuracy: 0.7117 - val_loss: 0.4295 - val_accuracy: 0.8230 - 2s/epoch - 9ms/step
Epoch 2/10
282/282 - 2s - loss: 0.3153 - accuracy: 0.8740 - val_loss: 0.3015 - val_accuracy: 0.8810 - 2s/epoch - 5ms/step
Epoch 3/10
282/282 - 2s - loss: 0.1764 - accuracy: 0.9378 - val_loss: 0.2504 - val_accuracy: 0.9080 - 2s/epoch - 5ms/step
Epoch 4/10
282/282 - 2s - loss: 0.1107 - accuracy: 0.9640 - val_loss: 0.2635 - val_accuracy: 0.9080 - 2s/epoch - 6ms/step
Epoch 5/10
282/282 - 2s - loss: 0.0730 - accuracy: 0.9779 - val_loss: 0.2370 - val_accuracy: 0.9280 - 2s/epoch - 6ms/step
Epoch 6/10
282/282 - 2s - loss: 0.0564 - accuracy: 0.9820 - val_loss: 0.2454 - val_accuracy: 0.9260 - 2s/epoch - 6ms/step
Epoch 7/10
282/282 - 1s - loss: 0.0436 - accuracy: 0.9873 - val_loss: 0.2625 - val_accuracy: 0.9250 - 1s/epoch - 5ms/step
Epoch 8/10
282/282 - 2s - loss: 0.0338 - accuracy: 0.9896 - val_loss: 0.3375 - val_accuracy: 0.9190 - 2s/epoch - 6ms/step
Epoch 9/10
282/282 - 2s - loss: 0.0314 - accuracy: 0.9893 - val_loss: 0.2440 - val_accuracy: 0.9290 - 2s/epoch - 5ms/step
Epoch 10/10
282/282 - 1s - loss: 0.0284 - accuracy: 0.9913 - val_loss: 0.3352 - val_accuracy: 0.9150 - 1s/epoch - 5ms/step
```



Scaled tfidf

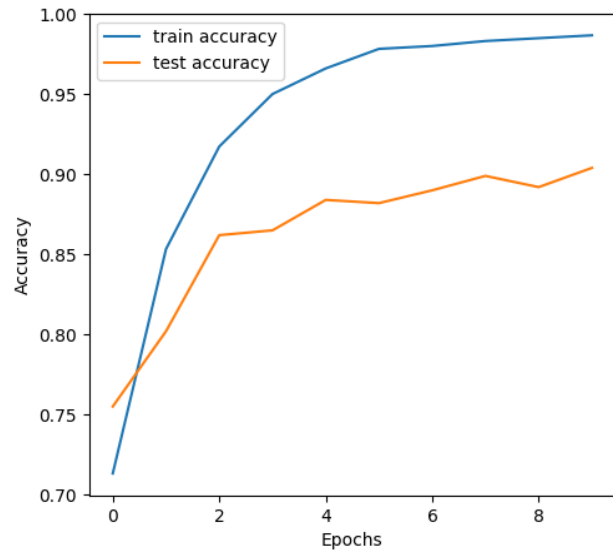
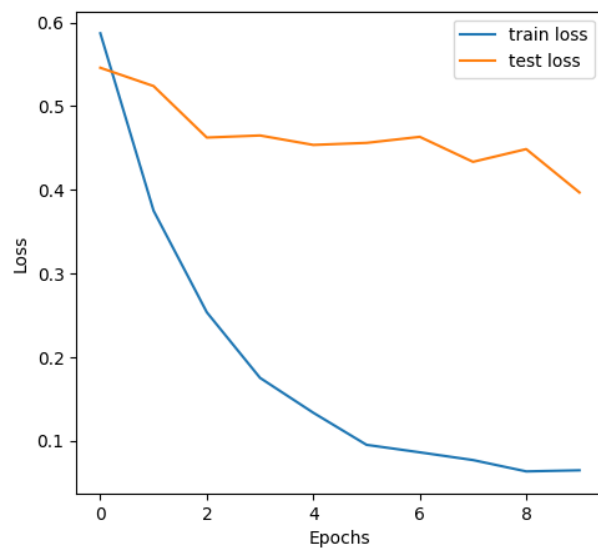
No regularization

```
Epoch 1/10
282/282 - 2s - loss: 0.6013 - accuracy: 0.7019 - val_loss: 0.5493 - val_accuracy: 0.7400 - 2s/epoch - 9ms/step
Epoch 2/10
282/282 - 1s - loss: 0.3836 - accuracy: 0.8361 - val_loss: 0.5245 - val_accuracy: 0.7880 - 1s/epoch - 4ms/step
Epoch 3/10
282/282 - 1s - loss: 0.2690 - accuracy: 0.9020 - val_loss: 0.4781 - val_accuracy: 0.8350 - 1s/epoch - 4ms/step
Epoch 4/10
282/282 - 1s - loss: 0.1748 - accuracy: 0.9427 - val_loss: 0.4817 - val_accuracy: 0.8430 - 1s/epoch - 4ms/step
Epoch 5/10
282/282 - 1s - loss: 0.1248 - accuracy: 0.9629 - val_loss: 0.5005 - val_accuracy: 0.8460 - 1s/epoch - 4ms/step
Epoch 6/10
282/282 - 1s - loss: 0.0834 - accuracy: 0.9769 - val_loss: 0.4690 - val_accuracy: 0.8780 - 1s/epoch - 4ms/step
Epoch 7/10
282/282 - 1s - loss: 0.0677 - accuracy: 0.9783 - val_loss: 0.5128 - val_accuracy: 0.8710 - 1s/epoch - 4ms/step
Epoch 8/10
282/282 - 1s - loss: 0.0538 - accuracy: 0.9822 - val_loss: 0.5030 - val_accuracy: 0.8870 - 1s/epoch - 4ms/step
Epoch 9/10
282/282 - 1s - loss: 0.0415 - accuracy: 0.9859 - val_loss: 0.4926 - val_accuracy: 0.8850 - 994ms/epoch - 4ms/step
Epoch 10/10
282/282 - 1s - loss: 0.0326 - accuracy: 0.9883 - val_loss: 0.5207 - val_accuracy: 0.8970 - 1s/epoch - 4ms/step
```



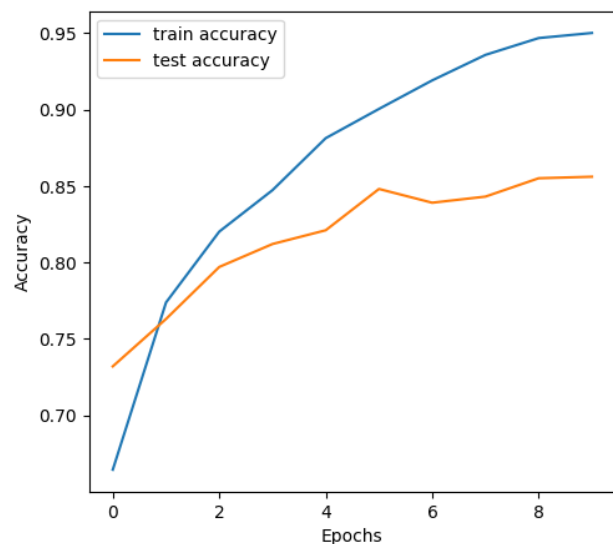
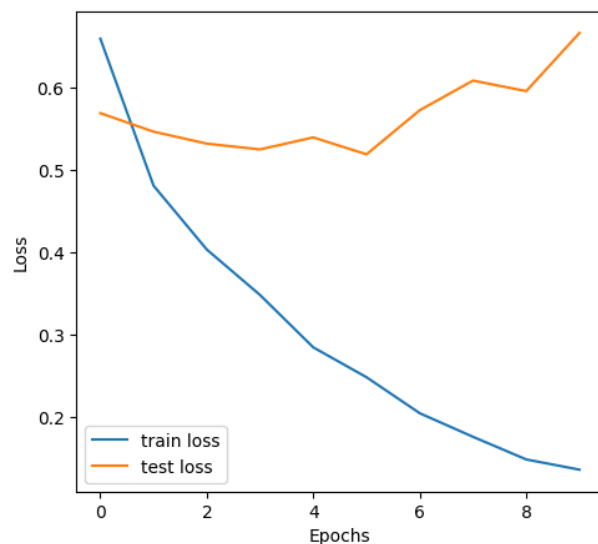
L2

```
Epoch 1/10
282/282 - 2s - loss: 0.5871 - accuracy: 0.7132 - val_loss: 0.5459 - val_accuracy: 0.7550 - 2s/epoch - 8ms/step
Epoch 2/10
282/282 - 1s - loss: 0.3750 - accuracy: 0.8533 - val_loss: 0.5241 - val_accuracy: 0.8020 - 1s/epoch - 5ms/step
Epoch 3/10
282/282 - 1s - loss: 0.2535 - accuracy: 0.9173 - val_loss: 0.4625 - val_accuracy: 0.8620 - 1s/epoch - 5ms/step
Epoch 4/10
282/282 - 1s - loss: 0.1751 - accuracy: 0.9501 - val_loss: 0.4650 - val_accuracy: 0.8650 - 1s/epoch - 5ms/step
Epoch 5/10
282/282 - 1s - loss: 0.1334 - accuracy: 0.9661 - val_loss: 0.4537 - val_accuracy: 0.8840 - 1s/epoch - 5ms/step
Epoch 6/10
282/282 - 1s - loss: 0.0950 - accuracy: 0.9783 - val_loss: 0.4561 - val_accuracy: 0.8820 - 1s/epoch - 5ms/step
Epoch 7/10
282/282 - 1s - loss: 0.0861 - accuracy: 0.9801 - val_loss: 0.4634 - val_accuracy: 0.8900 - 1s/epoch - 5ms/step
Epoch 8/10
282/282 - 1s - loss: 0.0768 - accuracy: 0.9832 - val_loss: 0.4335 - val_accuracy: 0.8990 - 1s/epoch - 5ms/step
Epoch 9/10
282/282 - 1s - loss: 0.0633 - accuracy: 0.9850 - val_loss: 0.4487 - val_accuracy: 0.8920 - 1s/epoch - 5ms/step
Epoch 10/10
282/282 - 1s - loss: 0.0646 - accuracy: 0.9868 - val_loss: 0.3968 - val_accuracy: 0.9040 - 1s/epoch - 4ms/step
```



Dropout

```
Epoch 1/10
282/282 - 2s - loss: 0.6602 - accuracy: 0.6644 - val_loss: 0.5696 - val_accuracy: 0.7320 - 2s/epoch - 7ms/step
Epoch 2/10
282/282 - 1s - loss: 0.4813 - accuracy: 0.7738 - val_loss: 0.5471 - val_accuracy: 0.7630 - 1s/epoch - 4ms/step
Epoch 3/10
282/282 - 1s - loss: 0.4036 - accuracy: 0.8201 - val_loss: 0.5325 - val_accuracy: 0.7970 - 1s/epoch - 4ms/step
Epoch 4/10
282/282 - 1s - loss: 0.3483 - accuracy: 0.8472 - val_loss: 0.5255 - val_accuracy: 0.8120 - 1s/epoch - 4ms/step
Epoch 5/10
282/282 - 1s - loss: 0.2846 - accuracy: 0.8812 - val_loss: 0.5401 - val_accuracy: 0.8210 - 1s/epoch - 4ms/step
Epoch 6/10
282/282 - 1s - loss: 0.2482 - accuracy: 0.9003 - val_loss: 0.5195 - val_accuracy: 0.8480 - 1s/epoch - 4ms/step
Epoch 7/10
282/282 - 1s - loss: 0.2044 - accuracy: 0.9190 - val_loss: 0.5732 - val_accuracy: 0.8390 - 1s/epoch - 4ms/step
Epoch 8/10
282/282 - 1s - loss: 0.1757 - accuracy: 0.9357 - val_loss: 0.6093 - val_accuracy: 0.8430 - 1s/epoch - 4ms/step
Epoch 9/10
282/282 - 1s - loss: 0.1482 - accuracy: 0.9467 - val_loss: 0.5965 - val_accuracy: 0.8550 - 1s/epoch - 4ms/step
Epoch 10/10
282/282 - 1s - loss: 0.1357 - accuracy: 0.9500 - val_loss: 0.6672 - val_accuracy: 0.8560 - 1s/epoch - 4ms/step
```



It would seem that the best model here is the one trained using the unscaled binary tokenizer and with no regularization at 0.946 accuracy. This is better than our logistic regression model. It is interesting since the graph of our test accuracy seems to have a steeper increase for the other models rather than this one. Let's see if the queries' prediction changes now.

```

1 model.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer happy hours? It does not allow children below 6."])))
1/1 [=====] - 0s 108ms/step
array([[0.07494579, 0.9250542 ]], dtype=float32)

1 model.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer a gym? No, it does not have happy hours."])))
1/1 [=====] - 0s 55ms/step
array([[0.02371382, 0.9762862 ]], dtype=float32)

1 model.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Does it offer a gym on site? No, it does not have happy hours."])))
1/1 [=====] - 0s 19ms/step
array([[0.05099352, 0.9490065 ]], dtype=float32)

1 model.predict(scipy.sparse.csr_matrix.todense(tokenizer.transform(["Is there a gym? No, it does not have happy hours."])))
1/1 [=====] - 0s 19ms/step
array([[9.1008429e-04, 9.9908996e-01]], dtype=float32)

```

This is certainly interesting: the predicted label for queries 2 and 3 changed to 0. This means that it recognizes that these queries are not question and answer pairs. However, it did get the 4th query wrong still.

References

Williams and Seeger (2001). Using the Nystroem method to speed up kernel machines.

<https://stats.stackexchange.com/questions/314329/can-support-vector-machine-be-used-in-large-data>

<https://github.com/alexa/alexa-with-dstc10-track2-dataset>

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression.score

<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC.predict>

<https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC>

https://scikit-learn.org/stable/modules/generated/sklearn.kernel_approximation.Nystroem.html#sklearn.kernel_approximation.Nystroem

<https://scikit-learn.org/stable/modules/preprocessing.html>

https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

<https://keras.io/api/layers/regularizers/>

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout