Natural Language Processing

Assignment 2

Pritish Wadhwa 2019440
Rohan Jain 2019095

Task 1: HMM Based POS Tagger implementing Viterbi Algorithm

- Implemented as described in the slides
- Added a start token in all sentences but it didn't help accuracy so removed it
- Assumed the start token to be '.' as the previous sentence must have ended with a '.'

Task 2: MLP Based POS Tagger

- Throughout the tasks, MLP was implemented using Keras. For the same multiple layers with different number of neurons and activation functions were tried upon and the ones with the best outputs are submitted.
- For Word2Vec, pre-trained embeddings, trained on the google news corpus consisting of 3 Billion tokens and vector size of 300 was used.
  - 2 models were trained.
  - In the first one we evaluated the sentences as single entities. Their corresponding embeddings were generated and the MLP model was trained.
  - In the other we considered the sentence as a whole and the taking their context the MLP model was trained.
  - For the second one, we also took into consideration the paddings.
- For Glove embeddings, pre-trained embeddings consisting of 6 Billion tokens and 300 dimension vectors were used.
  - For glove also we did the above two tasks individually.

Experiments:

- 3 fold cross-validation was performed on all the models.
- Precision, Recall, and F1 score was also calculated on all the models and reported.
- Tag-wise precision, recall and F1 score was calculated for each model.
- Statistics of the tags were also calculated and mentioned in each file.
- Word types tagged most incorrectly by HMM include(as ratio):
  - X: miscellaneous tag with very less counts as compared to others

- .: these are mostly found at end token and appear randomly in between texts, hence its classification isn't accurate
  - NUM
  - PRT
  - VERB
  - DET
- The above misclassifications can be explained as there is a lot of ambiguity in the English language and it has a lot of scope for different representations and interpretations. Due to this, a simple model like HMM with the context of only one word was unable to understand classification properly.

- Word types tagged most incorrectly by MLP include:
  - ADV
  - .
  - VERB
  - ADP
  - PRON
  - PRT
  - X
- The above can be explained as the english language contains a lot of anomalies and have a lot of scope for different representations and interpretations. Due to this since in a model like MLP, where the model was not able to understand the context properly it misclassified. eg. It would have been difficult for the model to differentiate between adv from X as it didn't know what the whole context was.

HMM:

Before padding:

```
Average metrics:
Precision:  0.9349713187008618
Recall:  0.9342877645346676
F1 score:  0.9346294139850572
     tag  precision    recall  f1-score
0    ADP   0.927881  0.934325  0.931088
1    DET   0.932602  0.936486  0.933730
2   NOUN   0.952553  0.922200  0.937121
3   VERB   0.948148  0.957994  0.952887
4    ADJ   0.916944  0.874352  0.894771
5   CONJ   0.968926  0.996005  0.981933
6    PRT   0.825178  0.866165  0.844828
7      .   0.918396  0.948445  0.931809
8    ADV   0.941688  0.880705  0.910157
9    NUM   0.875749  0.871675  0.872721
10  PRON   0.984172  0.924222  0.953145
11     X   0.737916  0.291296  0.414717
```

This is after padding both ends:

```
Average metrics:
Precision:  0.8719403286105988
Recall:  0.8944917848753294
F1 score:  0.8830720899376883
        tag  precision    recall  f1-score
0   <START>   0.499982  1.000000  0.666650
1       ADP   0.928043  0.934000  0.931008
2       DET   0.932353  0.937389  0.934007
3      NOUN   0.952233  0.922464  0.937102
4      VERB   0.941095  0.958673  0.949498
5       ADJ   0.917387  0.873383  0.894451
6      CONJ   0.976058  0.995611  0.985518
7       PRT   0.824651  0.866602  0.844749
8         .   0.931883  0.947867  0.938639
9     <END>   0.000000  0.000000       NaN
10      ADV   0.943060  0.880956  0.910927
11      NUM   0.875573  0.872001  0.872771
12     PRON   0.950892  0.924106  0.936745
13        X   0.747552  0.286088  0.410687
```

This is after padding only the start:

```
Average metrics:
Precision:  0.938034318159239
Recall:  0.9375861770087847
F1 score:  0.9378101811679702
     tag  precision    recall  f1-score
0    <s>   0.999927  1.000000  0.999964
1    ADP   0.928043  0.934000  0.931008
2    DET   0.932353  0.937389  0.934007
3   NOUN   0.952233  0.922464  0.937102
4   VERB   0.941095  0.958673  0.949498
5    ADJ   0.917387  0.873383  0.894451
6   CONJ   0.976058  0.995611  0.985518
7    PRT   0.824651  0.866602  0.844749
8      .   0.931883  0.947867  0.938639
9    ADV   0.943060  0.880956  0.910927
10   NUM   0.875573  0.872001  0.872771
11  PRON   0.950892  0.924106  0.936745
12     X   0.747552  0.286088  0.410687
```

Rohan
Train accuracies:
mlp = MLPClassifier(verbose=True, max_iter=50, random_state=42)
0.817775601170213
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(100,50,25), max_iter=50, random_state=42)
0.8221500914113806
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(100,50), max_iter=50, random_state=42)
0.8176577697832454
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(200), max_iter=50, random_state=42)
0.8224980622260191
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(150), max_iter=50, random_state=42)
0.8163192788719118
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(200, 100), max_iter=50, random_state=42)
0.855954811663098
17 min training time for the above model though

```
Iteration 1, loss = 0.99788447
Iteration 2, loss = 0.78981691
Iteration 3, loss = 0.76842451
Iteration 4, loss = 0.75587789
Iteration 5, loss = 0.74649836
Iteration 6, loss = 0.73794131
Iteration 7, loss = 0.73020899
Iteration 8, loss = 0.72221795
Iteration 9, loss = 0.71451940
Iteration 10, loss = 0.70748037
Iteration 11, loss = 0.70037116
Iteration 12, loss = 0.69370962
Iteration 13, loss = 0.68682975
Iteration 14, loss = 0.68054554
Iteration 15, loss = 0.67473479
Iteration 16, loss = 0.66878463
Iteration 17, loss = 0.66295400
Iteration 18, loss = 0.65760459
Iteration 19, loss = 0.65221639
Iteration 20, loss = 0.64750060
Iteration 21, loss = 0.64215625
Iteration 22, loss = 0.63765054
Iteration 23, loss = 0.63340551
Iteration 24, loss = 0.62868830
```

mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(200, 150), max_iter=50, random_state=42)
0.8631351618064288
mlp = MLPClassifier(verbose=True, hidden_layer_sizes=(300, 150), max_iter=50, random_state=42)
0.8927034754735809

```
Iteration 1, loss = 0.95546614
Iteration 2, loss = 0.78234251
Iteration 3, loss = 0.76133896
Iteration 4, loss = 0.74775776
Iteration 5, loss = 0.73593291
Iteration 6, loss = 0.72397008
Iteration 7, loss = 0.71184188
Iteration 8, loss = 0.69810842
Iteration 9, loss = 0.68551215
Iteration 10, loss = 0.67273736
Iteration 11, loss = 0.65978502
Iteration 12, loss = 0.64727762
Iteration 13, loss = 0.63518711
Iteration 14, loss = 0.62378007
Iteration 15, loss = 0.61210722
Iteration 16, loss = 0.60180046
Iteration 17, loss = 0.59148698
Iteration 18, loss = 0.58154647
Iteration 19, loss = 0.57225897
Iteration 20, loss = 0.56327269
Iteration 21, loss = 0.55492853
Iteration 22, loss = 0.54681922
Iteration 23, loss = 0.53987331
Iteration 24, loss = 0.53288433
```

1. sklearn mlp, 150 epochs (default config), embs for each word, y_train as direct label: 85.65%
2. sklearn mlp, 50 epochs (default config), embs for each word, y_train as one hot encoding: 81.86%
3. Keras
   ```
   model = Sequential()
   model.add(Dense(100, activation='relu', input_dim=len(embs[0])))
   model.add(Dropout(0.5))
   model.add(Dense(64, activation='relu'))
   model.add(Dropout(0.4))
   model.add(Dense(y_train.shape[1], activation='softmax'))
   model.compile(loss='categorical_crossentropy',
                 optimizer='adam', metrics=['accuracy'])
   ```
   30 epochs: 84.xx%
4.
   ```
   model = Sequential()
   model.add(Dense(100, activation='relu', input_dim=len(embs[0])))
   model.add(Dense(y_train.shape[1], activation='softmax'))
   model.compile(loss='categorical_crossentropy',
                 optimizer='adam', metrics=['accuracy'])
   ```

model.summary()
acc: 85.19%

5.