50.007 Machine Learning

Summer 2023 1D Project

Group 12

Group members:

1003424	Ooi Jia Sheng
1005983	Brandon Tan Rui En
1006340	Darren Chan Yu Hao
1006240	Michelle Chrisalyn Djunaidi

Tables of Contents

Introduction

In this project, the team explores what was taught in the last 4 weeks of the machine learning module, 50.007. Parts 1 to 3 was a semi guided implementation of a sequence labelling model for informal texts such as those from Twitter or Weibo, in this case, the texts chosen were in the languages of Spanish and Russian. This was done via Hidden Markov Model (HMM) and specifically its Viterbi algorithm.

In part 4 of the project, the team was tasked with improving the sentiment analysis system. The main change that the team did was to improve the pre-processing of data in order to have cleaner, less noisy data for the model to learn from. In addition, a pseudo-character level language model was also implemented as an attempt to improve the score as an alternative to dealing with Out-of-Vocabulary (OOV) words.

Part 1

Reading data and emission parameters

To begin, there are two parameters that need to be implemented and trained before we can use the Viterbi algorithm to generate our sequence labelling. They are the emission parameters and the transition parameters. In part 1, we will implement the emission parameters.

Estimating Emission Parameters using MLE

First, we read the training data into a list, with each element containing the respective word and label. Then, we separate the list into its respective labels and words and count the unique number of tags and words in the training data.

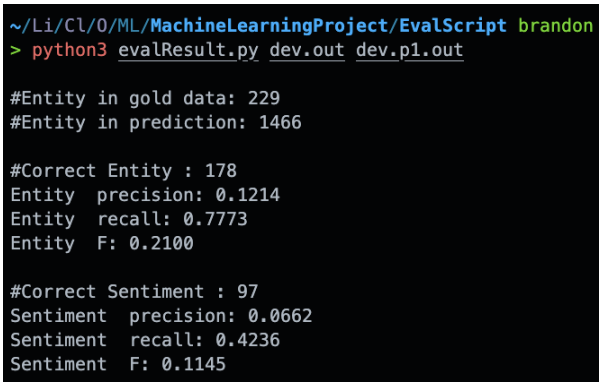
To get the estimated emission parameters, we store all $Count(y)$'s in a dictionary. Then, we count the number of times $Count(y \rightarrow x)$ appears and store it in a dictionary as well. What this means in the layman is basically, for every tag, e.g I-positive, we count the number of times it occurs in the corpus. We then count the number of times the tag "I-positive" is associated with a word, e.g. "hello" is tagged to I-positive 10 times. Setting $k = 1$, we then calculate emission parameters $e(x|y) = \frac{Count(y \rightarrow x)}{Count(y) + k}$ for every single unique word. We also account for unknown words that might appear in the test set, to deal with this, we implemented the #UNK# token, setting any new words to "UNK" and setting the unknown emission parameter as $\frac{k}{Count(y) + k}$.

Implementing a simple sentiment analysis system

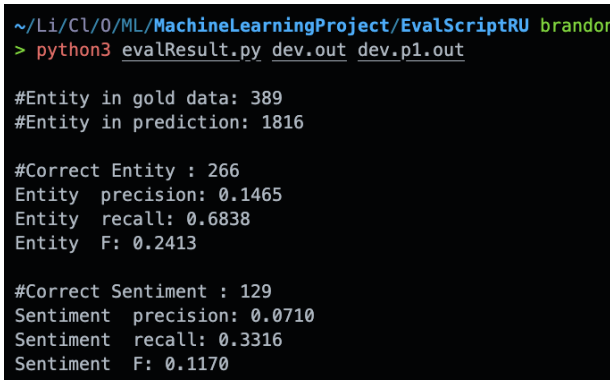
To predict the label of a word, we iterate through the test list of words and assign the label to a word which has the highest emission probability. If there are any unknown words in the test set, we assign the word the label with the highest "UNK" probability.

Results:

Spanish:



Russian:



Part 2

Estimating Transition parameters

Now that the emission parameters have been implemented, we move on to implement the transition parameters. Transition parameters essentially inform the likelihood for a word to transition from one to another, E.g from “Hello” to “world”.

In order to estimate the transition parameters, we use the equation as stated in the project handout:

$$q(y_i|y_{i-1}) = \frac{Count(y_{i-1}, y_i)}{Count(y_{i-1})}$$

Once again, we simply count the number of times the tag goes from one state to another, and divide it by the total number of times that unique tag appeared in the corpus.

Implementing Viterbi Algorithm

Once the transition parameters are learned, we proceed to implement the Viterbi algorithm as the maximum likelihood algorithm to label our sequence. We store $\pi_{j,v}$ for each node in memo[{ } for j in range(n+1)] and the best path in parent_arr [{ } for j in range(n+1)]. To prevent log underflow, we log both the transition and emission parameters. Hence, if probability = 1 , log(1) = 0, if probability = 0 , log(0) = -infinity.

We start the Viterbi algorithm with the initial step from “START” to the first node, where

$$log(\pi(j+1, u)) = max_v(log(\pi(j, v)) + log(b_u(x_{j+1})) + log(a_{u,v})) \quad \text{——— (1)}$$

And store the results in the first index of memo and parent_arr. Then, we continue with the recursive step of the Viterbi algorithm from j =1 to j=n using equation 1. If the word is known but the label does not emit the word, we set the emission probability to -infinity. If the word is unknown, we use the “#UNK” emission probability which corresponds to the label. For every previous node v, we check for the max v probability and store it in the memo, and node v in parent_arr. Then, we complete the termination step which is just $\pi(n+1, STOP) = max_v\{\pi(n, v) \times a_{v, STOP}\}$

To backtrack, we iterate the parent_arr backwards to get the maximum probability path sequence.

For part 2, the main functions that implement most of our project logic is log_underflow_prevention() and Viterbi(), with the bulk of the processing being in write_predictions_part_2().

Results:

Spanish:

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScript brand
> python3 evalResult.py dev.out dev.p2.out

#Entity in gold data: 229
#Entity in prediction: 542

#Correct Entity : 134
Entity  precision: 0.2472
Entity  recall: 0.5852
Entity  F: 0.3476

#Correct Sentiment : 97
Sentiment  precision: 0.1790
Sentiment  recall: 0.4236
Sentiment  F: 0.2516
```

Russian:

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScriptRU
> python3 evalResult.py dev.out dev.p2.out

#Entity in gold data: 389
#Entity in prediction: 488

#Correct Entity : 189
Entity  precision: 0.3873
Entity  recall: 0.4859
Entity  F: 0.4310

#Correct Sentiment : 129
Sentiment  precision: 0.2643
Sentiment  recall: 0.3316
Sentiment  F: 0.2942
```

Part 3

Finding 2nd best and 8th best output from Viterbi algorithm

For part 3, in order to get the k-best output sequences, our group decided to store the k-best probabilities and paths in each node in k_best_paths = [(“current_node”, path_probability, [previous_path]) for _ in range(k)]. We went with the k approach instead of hardcoding the values so we can reuse the code for the 2nd best and the 8th best output. We set variable k=8 as the best-8 number of sequences we want to store for each node in the HMM.

$$log(\pi(j+1, u)) = max_v(log(\pi(j, v)) + log(b_u(x_{j+1})) + log(a_{u,v}))$$

We use the same recursive step for the Viterbi algorithm as above, but change what is stored. We modified the Viterbi algorithm such that for a current node v_j , it calculates and stores all possible next $\pi_{j+1,\ u}$ for all u in path_probability. We also store the path from all previous nodes up to v and append node u into [previous_path] list. Then, we sort the probabilities in descending order and only keep the top-8 probabilities.

Since we store the best-8 probabilities and paths in each node, backtracking is trivial. We simply take the best-8 probabilities and paths from the “STOP” node and sort them in descending order. The first entry in k_best_paths corresponds to the best output sequence for the document. Since we want the 2nd best and 8th best output sequences, we simply take the 2nd and 8th entries of k_best_paths. However, some documents might not have 8 valid sequences. Hence, if such a case occurs, we let the 8th entry be equal to the last entry of k_best_paths.

Results:

Spanish:

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScript brandon
> python3 evalResult.py dev.out dev.p3.2nd.out

#Entity in gold data: 229
#Entity in prediction: 503

#Correct Entity : 123
Entity  precision: 0.2445
Entity  recall: 0.5371
Entity  F: 0.3361

#Correct Sentiment : 72
Sentiment  precision: 0.1431
Sentiment  recall: 0.3144
Sentiment  F: 0.1967
```

2nd Best

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScript brandon
> python3 evalResult.py dev.out dev.p3.8th.out

#Entity in gold data: 229
#Entity in prediction: 623

#Correct Entity : 121
Entity  precision: 0.1942
Entity  recall: 0.5284
Entity  F: 0.2840

#Correct Sentiment : 70
Sentiment  precision: 0.1124
Sentiment  recall: 0.3057
Sentiment  F: 0.1643
```

8th Best

Russian:

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScriptRU brandon
> python3 evalResult.py dev.out dev.p3.2nd.out

#Entity in gold data: 389
#Entity in prediction: 774

#Correct Entity : 193
Entity  precision: 0.2494
Entity  recall: 0.4961
Entity  F: 0.3319

#Correct Sentiment : 122
Sentiment  precision: 0.1576
Sentiment  recall: 0.3136
Sentiment  F: 0.2098
```

2nd Best

```
~/Li/Cl/O/ML/MachineLearningProject/EvalScriptRU brandon
> python3 evalResult.py dev.out dev.p3.8th.out

#Entity in gold data: 389
#Entity in prediction: 912

#Correct Entity : 189
Entity  precision: 0.2072
Entity  recall: 0.4859
Entity  F: 0.2905

#Correct Sentiment : 113
Sentiment  precision: 0.1239
Sentiment  recall: 0.2905
Sentiment  F: 0.1737
```

8th Best

Part 4

Preliminary Inspection

In part 4, the task at hand is to handle a new dataset 'test.in', such that the predictions will be optimal without knowledge on the 'test.out'. The task allows us to use any model at hand, but the restriction is placed on no external machine learning libraries.

There has been thought to utilize neural network models such as Long-Short-Term-Model, during our process of due diligence during the preliminary process of the possible tools. However, in our attempts that has been difficult without using packages due to its inherent complexity that requires quite a fair bit of optimization that is beyond our current depth of knowledge.

Thus, instead of reinventing the wheel, the team has made a conscious decision to work on the existing model, the Hidden Markov Model, and to dissect any possible factors that have a detrimental impact towards its output performance and improve upon it.

Main Process

A current issue that we have detected is that the 'train' dataset does not have the full spectrum of words that are found in the other datasets. As what we have done in Part 1

- One problem with estimating the emission parameters is that some words that appear in the test set do not appear in the training set. One simple idea to handle this issue is as follows. We introduce a special word token #UNK#, and make the following modifications to the computation of emission probabilities:

$$e(x|y) = \begin{cases} \frac{\text{Count}(y \rightarrow x)}{\text{Count}(y)+k} & \text{If the word token } x \text{ appears in the training set} \\ \frac{k}{\text{Count}(y)+k} & \text{If word token } x \text{ is the special token \#UNK\#} \end{cases}$$

The performance has not been very ideal as we inspect how the predictions worked when comparing with the gold standard. This does not consider the linguistic semantics of the word itself for those #UNK# thus translating into suboptimal predictions. As such, the current steps are to enhance the current HMM model such that the performance is enhanced..

Data Pre-processing

During both training and prediction phases, it's essential to convert all words to lowercase. For instance, consider the words "Plato" and "plato." While they may appear different due to variations in their ASCII codes, they are linguistically identical. However, without converting them to lowercase, they would be treated as distinct words in the process. That has been proven as we checked through our unique words detection function that they are indeed treated differently. This practice serves as a necessary and justifiable form of prior knowledge.

Dealing with unknown words

The main issue we realized about trying to conduct sentiment analysis using a word based language model is that there will also be out-of-vocabulary (OOV) words. Based on our research, we note that there have been significant strides in dealing with OOVs. Below, we will discuss 3 of them which we took inspiration from.

MIMICK

While researching, we found a novel approach called MIMICK, which addresses a significant limitation in the effectiveness of word embeddings for downstream natural language processing (NLP) tasks, particularly in the context of Hidden Markov Models (HMMs). (Pinter,2017) Word embeddings are powerful representations that capture semantic relationships between words using distributional information from unlabeled data. However, their utility is hampered by out-of-vocabulary (OOV) words, which lack corresponding embeddings.

MIMICK proposes a solution by generating compositional embeddings for OOV words, specifically tailored for HMM-based tasks, without requiring re-training on the original word embedding corpus. This is achieved by learning a mapping function from word spellings to distributional embeddings. The key innovation is that learning is performed at the type level, i.e., across all instances of a word, rather than at the token level. This approach enables MIMICK to effectively address OOV words, which are a common challenge in many NLP applications.

The enhancement that MIMICK provides to HMM-based approaches is notable. In HMMs, the transition probabilities between hidden states and the emission probabilities from hidden states to observed symbols (words) play a crucial role. By introducing compositional embeddings for OOV words, MIMICK enriches the emission probabilities by providing meaningful representations for words that were previously missing from the embedding space. This, in turn, improves the HMM's ability to accurately model and predict the observed sequence of words.

Intrinsic and extrinsic evaluations of MIMICK demonstrate its efficacy. It outperforms word-based baselines in tagging part-of-speech and morphosyntactic attributes across multiple languages. The paper also highlights that MIMICK's performance is competitive with, and can be complementary to, character-based models, especially in low-resource settings. This indicates that MIMICK's approach of generating OOV embeddings compositionally effectively addresses a critical limitation of word embeddings, leading to improved performance in HMM-based NLP tasks.

SentencePiece

Another source of inspiration was sentencepiece, a subword tokenizer by Google. What it does is that given a sentence, it will be able to deconstruct the words into subwords, thereby creating more “words” for usage in the training, improving and alleviating the issue of OOV. For instance, it can decompose the word “Hello” into many forms such as: [H,e,ll,o], [Hell,o], [H,e,l,l,o], [He,ll,o] to name a few. (Github,n.d)

With those subwords, you can then do text construction to formulate new words that might not have been in your vocabulary such as [Hell] or [elo]. This increases the number of “learned words”.

Character Level Language

Now that we learnt some interesting methods such as word embedding and making words into subwords, we continued researching and found out about character level languages. Character level language is essentially that instead of looking at words, we only look at the sequence of characters directly. (Soulpage ,2023)

Upon consuming a large amount of data, it will then be able to generate subsequent characters in sequence and generate “words”. With a character level language model, you will rarely encounter OOV issue as there are only a fixed number of characters in a given language and unless your dataset is small, every character should be present in your dataset. Essentially, it has increased flexibility in that it has unrestricted vocabulary.

However, character level language downside is the increase in time complexity. For example: the word “Hello” only needs to be trained once but in a character level language, it has to be trained 5 times as there are 5 characters. Additionally, character level language lacks semantics. Really, it is just a sequence of characters with no meaning behind it, neither does it know any structure.

Regardless, as our dataset is not too large but enough such that most characters should be present in the dataset, computation time and complexity is not a huge issue

for us. In addition, by understanding that sentencepiece subwording as a concept can work, we wanted to try to see if character level only can work as well. This sounded interesting to us and so we took the idea of subwords in sentence piece and character level language and tried to generate sentiment based upon it.

Our implementation

In our implementation, due to the difficulty and time constraint of rewriting the entire model as specified in our opening above, we took inspiration from character level language modeling and sentencepiece by Google.

What we ended up implementing is a combination of character and word level language model. For any words that can be found in the current dataset, we retain the usage of the word level to predict the emission parameters. However, if the word does not exist, insead of using an UNK token, our implementation will split that word into its individual characters. For instance, “Hello” will be turned into [“H”,”E”,”L”,”L”,”O”]. Then each character , already trained and labeled, will have an emission parameter. With that, the word “Hello” might be mapped as:

H I-positive
E O
L I-negative
L I-positive
O O

We then use that and randomly select from this 5 sentiment, one sentiment to represent the word “Hello”. In a way, this is similar to subword language model like those employed by sentencepiece but character level instead. The reason we used character level even though as explained above that there are several cons, such as lack of semantic understanding is because firstly, we do not have the large dataset of subwords that sentencepiece has available to it, nor are we allowed to utilize it. In addition, subwords are language specific and nobody on our team is fluent in Spanish or Russian and we do not wish to randomly import random datasets without knowing whether it is correct or not.

Surprisingly, well character level lacks semantic meaning, our results substantially, especially for the alphabet based Spanish language.

```
PS C:\Users\littl\OneDrive\Documents\GitHub\MachineLearningProject\EvalScript> python3 .\evalResult.py dev.ru.out dev.p4.RU.out

#Entity in gold data: 389
#Entity in prediction: 311

#Correct Entity : 188
Entity  precision: 0.6045
Entity  recall: 0.4833
Entity  F: 0.5371

#Correct Sentiment : 133
Sentiment  precision: 0.4277
Sentiment  recall: 0.3419
Sentiment  F: 0.3800
PS C:\Users\littl\OneDrive\Documents\GitHub\MachineLearningProject\EvalScript> python3 .\evalResult.py dev.out dev.p4.ES.out

#Entity in gold data: 229
#Entity in prediction: 224

#Correct Entity : 139
Entity  precision: 0.6205
Entity  recall: 0.6070
Entity  F: 0.6137

#Correct Sentiment : 108
Sentiment  precision: 0.4821
Sentiment  recall: 0.4716
Sentiment  F: 0.4768
PS C:\Users\littl\OneDrive\Documents\GitHub\MachineLearningProject\EvalScript>
```

Observed Results from RU and ES, trained with dev.in respectively

Observing from the derived F score, that has been greatly favorable, considering the F score from the base case without any augmentation. Thus, we decided that there is merit in continuing to study this method.

Side Explorations

"Success is stumbling from failure to failure with no loss of enthusiasm." - Winston Churchill. Before we arrived at our main method, we explored and coded out various different possibilities; all of the code for this part can be found at Part_4_Research.ipynb and p4ResearchAttempts.ipynb with their respective Markdown headers: Decoding with Forward-Backward, Perceptron Implementation, and Prior-Knowledge and Text Preprocessing.

Prior Knowledge

We have attempted to use the concept that we have learnt from Homework 4, the question on prior knowledge.

Question 3. The Viterbi algorithm discussed in class is used for finding the optimal $\mathbf{y}$ sequence based on the following:

$$y_1^*, \ldots, y_n^* = \arg\max_{y_1,\ldots,y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_n)$$

Now, consider the problem of part-of-speech tagging. Sometimes we have some prior knowledge about certain tags for certain observations. For example, assume the observation x_i="*the*", we are almost certain that it is not a verb (*i.e.*, we believe $y_i \neq \mathrm{V}$). In this case, we would like to do the decoding in the following way, where we would like to incorporate the prior knowledge $y_i \neq \mathrm{V}$ (and find optimal values for all other y_k in the sequence, where $k = 1, \ldots, n, k \neq i$):

$$y_1^*, \ldots, y_{i-1}^*, y_{i+1}^*, \ldots, y_n^* = \arg\max_{y_1,\ldots,y_{i-1},y_{i+1},\ldots,y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_{i-1}, y_{i+1} \ldots, y_n | y_i \neq \mathrm{V})$$

Clearly describe how to modify the Viterbi algorithm to perform such a new decoding task. *(10 points)*

Figure: Homework Question 3

On the topic of prior knowledge, we are aware of the fact that numbers (Eg. 0123456789) and punctuations (Eg. .,!()) do not carry sentiment; this knowledge has been validated through inspection of the dataset. In theory, it will be valuable to assume that having these to be 'O' makes sense, and thus the execution was according to how we approached Question 3, in which we accounted for the presence of prior knowledge during the recursion process of moving forwards during HMM.

The yielded results seemed to cause a constant small drop of F score, roughly 0.1 decrease for both languages. This is due to an increase in our precision but a decrease in our recall, which resulted in worse F scores as it is a harmonic mean of both values. Our hypothesis is that the model is already robust in its implementation, as we observed the predictions from dev.in and the gold standard dev.out, as observed that the punctuations and numbers were assigned the 'O' tag, and the modifications that we have made hampered the prediction process.

Forward-backward Algorithm

We tried implementing the forward-backward algorithm as explained in the notes HMM III and IV and as touched in the homework. With the algorithm, we can estimate the most probable state sequence for each observation pair sequence in test.in, such that each hidden state is the most probable considering the sentence after this position (forward) and the sentence before this position (backward).

We built our forward and backward scores recursively, following the equations stated in HMM IV.

1. Forward probability:

The forward probabilities are defined as:

$$\alpha_u(j) = p(x_1, \ldots, x_{j-1}, y_j = u; \theta) \tag{1}$$

- Base case:

$$\alpha_u(1) = a_{\text{START},u} \quad \forall u \in 1, \ldots, N-1 \tag{2}$$

- Recursive case:

$$\alpha_u(j+1) = \sum_v \alpha_v(j) a_{v,u} b_v(x_j) \quad \forall u \in 1, \ldots, N-1, j = 1, \ldots, n-1 \tag{3}$$

Figure: Forward Probability from HMM IV notes, pg. 1

Due to how the forward probability is calculated (recursively sum of all (previous forward probability * transmission probability * emission probability)), we realized that logging the transmission and emission probabilities would have made it impossible to reverse the forward probabilities back to the real value, as exp(sum(log(forward) + log(transmission) + log(emission))) =/= sum(exp(log(forward)) + exp(log(transmission) + log(emission))). A similar conundrum applies for backward probabilities.

So, we circumvent this limitation by multiplying all probabilities by a constant c, so that we can restore the final forward score by sum(forward*transmission*emission) = sum(modified_forward*modified_transmission*modified_emission)/c^3. Similarly, we adjust the backward probabilities. After some testing with c values such as 10000000 and 100000, we chose c to be 100.

Nevertheless, as explained by HMM notes, the Viterbi algorithm yields better solutions than max-marginal decoding using the forward-backward algorithm. In the end, we decided not to go with this approach.

Perceptron Implementation

We referred to the 2002 paper on Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. (Collins,2002)

Our implementation is as follows: we first proceed with the normal HMM training with Viterbi as usual. Then, as suggested by the paper, we check our training output (zi) with the training golden standard (yi). For every sentence, we execute standard perceptron protocol: if the model does not correctly predict the output, we adjust our weights accordingly to make it closer to the real output.

Inputs: Training examples (x_i, y_i)
Initialization: Set $\bar{\alpha} = 0$
Algorithm:
For $t = 1 \ldots T$, $i = 1 \ldots n$
Calculate $z_i = \arg\max_{z \in \mathbf{GEN}(x_i)} \Phi(x_i, z) \cdot \bar{\alpha}$
If$(z_i \neq y_i)$ then $\bar{\alpha} = \bar{\alpha} + \Phi(x_i, y_i) - \Phi(x_i, z_i)$
Output: Parameters $\bar{\alpha}$

Figure: The paper's proposed variant of the perceptron algorithm

In our code, we used a simple indicator function as our ϕ; the end result basically checks whether or not every position's predicted tag matches the original golden data's tag.

References

Pinter, Y. (2017, July 21). Mimicking Word Embeddings using Subword RNNs. arXiv.org. https://arxiv.org/abs/1707.06961

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002), pages 1–8. Association for Computational Linguistics. https://aclanthology.org/W02-1001/

GitHub -Unsupervised text tokenizer for Neural Network-based text generation. GitHub. https://github.com/google/sentencepiece

Character-level language. (2023, June 16). Soulpage IT Solutions. https://soulpageit.com/ai-glossary/character-level-language-explained/