Report of Final project - Out-of-Domain PoS tagging

Jiangnan Huang et Jiaxuan Liu

April 30, 2020

Instructor: Caio Corro

Abstract

The goal of this project is to understand the main concepts of statistical learning. We implemented three PoS(Part-of-Speech) taggers, a Perceptron model, a HMM model and a Decision Tree model, which are trained and evaluated on the French treebanks of the corpora of the *Universal Dependencies* project. We evaluated and characterized the impact of changes in domain, then we developed features that are robust to the Out-of-Domain PoS tagging problem. The evaluation of our PoS taggers are considered on the different combination of train and test sets.

Keywords: Part-of-Speech; Out-of-Domain PoS tagging; Machine Learning; Statistical Learning; NLP; Perceptron; HMM; Decision Tree

1 Introduction

In corpus linguistics, a Part-of-Speech (PoS) is a category of words which have similar grammatical properties as nouns, verbs, adjectives, adverbs, etc. Part-of-Speech tagging, which can be considered as a multi-class classification problem, is the process of determining the syntactic category (PoS) of a word from the words in its surrounding context.

Part of speech tagging is often used to help disambiguate natural language phrases because it can be done quickly with high accuracy. Tagging can be used for many NLP tasks like determining correct pronunciation during speech synthesis (for example, work as a noun vs work as a verb), for information retrieval, and for word sense disambiguation.

PoS tagging is a relatively easy task: as long as there are enough training data of the same domain, a simple multi-class classifier with very simple features achieve human-comparable performance. However, PoS taggers performance degrades

significantly when they are applied to sentences that depart from training data. In this work, We aim to evaluate and characterize the impact of changes in domain and develop the features and models that makes our PoS taggers robust to changes in domain.

In Section 2, the data analyse processes are discussed. We describe the french corpus used in our work, and give a quantitative description of the different combination of train and test sets. In section 3, we provide our Perceptron, a part-of-speech (POS) tagger which uses the principal notion of Multi-Classification. The features extracted from the corpus consists of distributional features, suffixes, windows, and word shapes. The advantages of these features are discussed.

In section 4 we provide a classical Hidden Markov Model (HMM) tagger, and in section 5 we present a DecisionTree classifier from sklearn to provide some results as baselines, the word encoding algorithm is also discussed.

The next section describe the results. We give an evaluation on the performance of our three taggers and compare there performance between different combination of train and test sets, the importance of each type of feature is also discussed. Section 6 presents our conclusion.

2 Experimental Data

In this section, we describe the French corpus we used in our work, and give a quantitative description of the differences combination of train and test sets.

Our work is based on on the French treebanks of the corpora of the *Universal Dependencies* project. There are 6 different corpus: 1.fr.ftb, 2.fr.gsd, 3.fr.partut, 4.fr.pud, 5.fr.sequoia, 6.fr.spoken. We also consider two extra treebanks fr.foot and fr.natdis in our experiments.

2.1 Data resource

The 6 corpus from *Universal Dependencies* project:

(1) $fr.ftb$ is the Universal Dependency version of the French Treebank, it contains sentences from the newspaper Le Monde, initially manually annotated with morphological information and phrase-structure and then converted to the Universal Dependencies annotation scheme.
Resource: News Le Monde.

(2) $fr.gsd$ was converted in 2015 from the content head version of the universal dependency treebank v2.0. It is updated since 2015 independently from the previous source.
Resource: News; Blogs; Consumer Reviews.

(3) $fr.partut$ is derived from the already-existing parallel treebank Par(allel)TUT. It's a conversion of a multilingual parallel treebank developed at the University

of Turin, and consisting of a variety of text genres, including talks, legal texts and Wikipedia articles, among others.
Resource: Creative Commons open license; DGT-Translation Memory; Europarl; Facebook; JRC-Acquis multilingual; Articles from Project Syndicate; Universal Declaration of Human Rights; Wikipedia; Web Inventory of Translated Talks.

(4) $fr.pud$ is created for the [CoNLL 2017 shared task on Multilingual Parsing from Raw Text to Universal Dependencies](http://universaldependencies.org/conll17/).
Resource: News, Wikipedia.

(5) $fr.sequoia$ is an automatic conversion of the Sequoia Treebank corpus French Sequoia corpus. The conversion was done with the Grew software and the Graph Rewriting System.
Resource: News, Europarl, Wikipedia, Medicines.

(6) $fr.spoken$ is a Universal Dependencies corpus for spoken French.
Resource: Spoken French.

The 2 extra corpus:

(1) $fr.foot$ contains the conversations about football.
Resource: Tweets.

(2) $fr.natdis$ contains the sentences about the disasters.
Resource: Tweets.

But we did not find the accurate source of these two corpus.

2.2 Size of train/test sets

Table 1 shows the size (number of sentences and words) of the difference of the train and test set, also the proportion of the test set to the training set. From which we can see that the number of sentences and words in corpus 'fr.ftb' is the largest, 14759 and 44228 respectively in train_set.

Table 1: The size of the different corpus

	train_set		test_set	
corpus	# sentences	# words	# sentences	# words
fr.ftb	14759	442228	(17.22%) 2541	(16.98%) 75073
fr.gsd	14450	345009	(2.88%) 416	(2.82%) 9742
fr.partut	803	23324	(13.70%) 110	(10.78%) 2515
fr.pud	803	23324	(124.53%) 1000	(103.49%) 24138
fr.sequoia	2231	49173	(20.44%) 456	(19.81%) 9740
fr.spoken	1153	14952	(62.97%) 726	(66.95%) 10010
fr.foot			743	13985
fr.natdis			622	12044

2.3 Noisiness of corpus

Before training our model, the similarity between the different data sets are measured from multiple angles to characterize the differences between UGC corpora and 'canonical' corpora. If the model can have a good performance on test set which is highly different from train set, then we can conclude that the model is robust enough. We consider three measures of the noisiness of a corpus:

2.3.1 OOV words

OOV words are the words appearing in the test set that are not contained on the train set.

The percentage of Out-of-Vocabulary (OOV) words are measured because the words which are not contained on the train set have less chance to be learned correctly. For example, if we naively consider the word itself as the only feature to predict its PoS, then the OOV words are completely outside the scope of learning. The OOV words force us to find more features according to the context to predict the PoS of one word. In general, the more OOV words exists, the less model accuracy we can get. Our tagger's performance on OOV words of the different data set are discussed in section 6.

2.3.2 KL divergence

Kullback–Leibler divergence is a measure of how one probability distribution is different from a second, reference probability distribution. The KL divergence of 3-grams characters distributions estimated on the train and test sets. The divergence is defined as :

$$KL(c_{test}||c_{train}) = \sum_{n \in N} p_{test}(n) * \log(\frac{p_{test}(n)}{p_{train}(n)})$$

where the sum runs over N the set of all the 3-gram of characters in the train and test sets, and $p_d(c_{i-2}, c_{i-1}, c_i) = \frac{\#\{c_{i-2},c_{i-1},c_i\}+1}{\#N+\#\mathcal{V}*(\#d-2)}$ is the probability to observe the 3-gram $c_{i-2}c_{i-1}c_i$ in data set d with Laplace-smoothing; $\#\mathcal{V}$ is the number of distinct 3-grams of characters in the train and in the test sets and $\#d$ is the number of characters in the corpus d, consequently $d-2$ is the total number of 3-grams in the corpus.

In the simple case, a Kullback–Leibler divergence of 0 indicates that the two sets are identical. The higher the value of KL divergence, the greater the difference between the train set and the test set

2.3.3 Perplexity

Perplexity is a measurement of how well a probability model predicts a sample. In the context of Natural Language Processing, perplexity is one way to evaluate language models. The perplexity is defined as:

$$PP(S) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p(w_i|w_1 w_2 w_3 ... w_{i-1})}}$$

Consider the language model as a trigram model, then the perplexity transformed into:

$$PP(S) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p(w_i|w_{i-2} w_{i-1})}}$$

where N is the number of the words in a sentence. $p(w_i|w_{i-2}w_{i-1})$ is the probability of the word i appears in a position where the words $i-2$ and $i-1$ appear in front.

In this work, the language models are estimated on the train set as a trigram model by KenLM, we also use this package to compute the perplexity: We take all the sentences as samples from the test set and compute the average perplexity over all the samples. With the final perplexity we can measure how well the language models fits the test set. In general, the lower the perplexity, the better the language models fits the test set (also the higher the similarity between train set and test set).

2.4 Metric results

In this subsection, we compute the value of the different metrics and give a quantitative description for the different combination of train and test sets, then we discuss the results.

Table 2: The results of 3 metrics

Corpus	#OOV	P(OOV)	KL divergence	Perplexity
fr.ftb	2529	21.06%	2.18 e^{-05}	**676.55**
fr.gsd	576	17.84%	3.07 e^{-04}	534.63
fr.partut	293	27.73%	**7.74** e^{-04}	280.42
fr.pud	**6540**	**72.56%**	1.60 e^{-05}	670.71
fr.sequoia	899	29.12%	1.58 e^{-04}	437.18
fr.spoken	1783	60.51%	7.70 e^{-05}	308.37

From Table 2 we can conclude that:

For the OOV words: The corpus 'fr.pud' has the largest percentage of OOV words through all the test set. This means the corpus 'fr.pud' has the less similarity between train set and test set in word level. However, the corpus 'fr.gsd' has the least percentage of OOV words which means it has the most similarity in words level.

For the KL divergence: The corpus 'fr.partut' has the largest KL, which means that the distribution of 3-grams characters in test set has much deviate from the distribution in train set. However, the corpus 'fr.pud' has the smallest KL so it has a highest similarity of the distribution of 3-grams characters between train set and test set.

For the Perplexity: The corpus 'fr.ftb' has the largest perplexity, which means that the sentences in its test set has the lowest predictability of distribution of words in each sentences base on the language model estimated on its train set. While the corpus 'fr.partut' has the smallest perplexity so the language model estimated on its train set are less "surprised" by the sentences in its test set.

3 Perceptron model

In machine learning, the Perceptron is an algorithm for supervised learning. It is a type of linear classifier that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. In this section, we talk about how to extract the 4 types of features and how to define the model.

3.1 Feature Extraction

Windows Feature: Given a sentence $s_1, s_2, \ldots, s_n$, we use the words which are in the window of $2l + 1$ around the word s_i as the features of the word $[s_i$: $s_{i-l}, \ldots, s_{i+l}]$. The label of a word depends on not only itself but also the words around it. For example in English:

"Can you answer the question?",

"It's a good answer"

In these two sentences, the word 'answer' has two different labels: 'VERB' in the 1st sentence, 'NOUN' in the 2nd sentence. In the 1st sentence, the word before 'answer' is a 'NOUN', which will be followed by a 'VERB' probably. Similarly, 'ADJ' is usually followed by a 'NOUN' in the 2nd sentence. In order to not only extract features effectively, but also not occupy too much memory, we set the window size $l = 2$.

Suffix Feature: We call the last k characters of a word a suffix. We use suffixes of different lengths to represent different features of a word $w_i : (w_i[-k :], \ldots, w_i[-1 :])$ Suffixes are useful, because the basic morphological rules are the

same in different fields. For the Perceptron model, we simply set k = length of each word to extract all the possible suffix.

Shape Feature: Each word is mapped to a bit string encompassing 8 binary indicators that correspond to different orthographic (e.g., does the word start with a capital letter, does it all capitalize, is it a number, is there hyphen or hyphen low or backslash, is it alphanumeric and what is the length greater than 3)

Distributional Feature: For each word, We count the words and the number of occurrences to the left of this word, and sort by the number of occurrences to get the sort number. The sort number and its word correspond will be the distributional feature. So as the right side of the word. (This feature is implemented but not really used in our final model because it cost too much computing resource which our PC can not afford)

3.2 Model

The Out-of-Speech Tagging is a problem of multi-class classification while each word has a corresponding label: $y_{pre} \in Y$, where Y is a finite set of all possible labels in train set. Our goal is to define a model to learn from the train set and make prediction on the test set.

Loss Function: Firstly we define the loss function which is used to compute the loss by the true label and predicted label.During training step our goal is to minimize it. Like much multi-label classification, the following loss function is $0-1$ loss, which will be 0 if the predicted label is the true label.

$$l^{multi}(y_{pre}, y_{real}) = \begin{cases} 0 & if\ y_{pre} = y_{real} \\ 1 & otherwise \end{cases} \tag{1}$$

Decision function: For each feature of a word, there are multiple possible labels. We use the following decision function to determine the most likely label for a word:

$$label_{pre} = \arg\max_{y \in Y} \sum_{f \in F_x} sigmoid(w[f][y]) \tag{2}$$

Y: the finite set of all possible labels in train set.

F_x: the set of all features of the word x

$W[f][y]$: the weight of the label y corresponding to the feature f which is initialized to 0 and W is saved inside the model.

$sigmoid()$: we implement a sigmoid function as an **activation function** to reduce the effect of particularly large values on results, this could help the model to achieve a better performance. (We also implement ReLU which could efface the effect of the negatives values, but that doesn't really help.)

Update rule: When the predicted label is the same as the correct label, we do nothing. But if they are different, we should update our wight which is defined in the previous item. In practice we increase the weight of all correct labels of the features of input x and reduce the weight w of all wrong labels of the features of input x. The updating formula is as following:

$$\forall f \in F_x, \begin{cases} r[f][y_{pre}] \mathrel{+}= log(0.9)^2 \\ \\ r[f][y_{real}] \mathrel{+}= log(1.1)^2 \end{cases} \tag{3}$$

$$\forall f \in F_x, \begin{cases} w[f][y_{pre}] \mathrel{+}= log(0.9)/(\delta + \sqrt{r[f][y_{pre}]}) \\ \\ w[f][y_{real}] \mathrel{+}= log(1.1)/(\delta + \sqrt{r[f][y_{real}]}) \end{cases} \tag{4}$$

where $log(0.9)$ and $log(1.1)$ are incrementales, r is the sum of the squares of the incrementales, it has the same dimension as w. δ is a smoothing term that avoids division by zero (set to $1e^{-8}$). We imitate the Adagrad algorithm and set the optimization steps, this algorithm can help the model converge more quickly and get better performance.

Accuracy Rate: Finally we define the metric to evaluate the model and we choose accuracy in word-level. It is compute by dividing the number of words which are successfully predicted by the total number of words. The formula is as following:

$$accuracy = \frac{\#correct\ labels}{\#words}$$

$\#correct\ labels$: Number of words predicted correctly.
$\#words$: Number of all words in corpus

Training and Testing: When we are training, we use the decision function to get the predicted labels. When the prediction result is wrong, we constantly update the label weights according to the update rules to obtain more accurate weights. By using $defaultdict(int)$ in Python, the label weights are initialized to 0 by default. However, testing is different from training. During testing, the model will not be updated based on the results. Finally, the accuracy rate is used to evaluate the quality of the model.

4 HMM model

HMM (hidden Markov model) is based on augmenting the Markov chain. A Markov chain is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, like the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. The states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather we only examine today's weather but we won't examine yesterday's weather.

As tomorrow's weather condition has a probability to be influenced by today's weather condition, the PoS tag of word also has a probability to be influenced by the PoS tag of the last word. In this project, we use the Pomegranate library to build a hidden Markov model for PoS tagging.

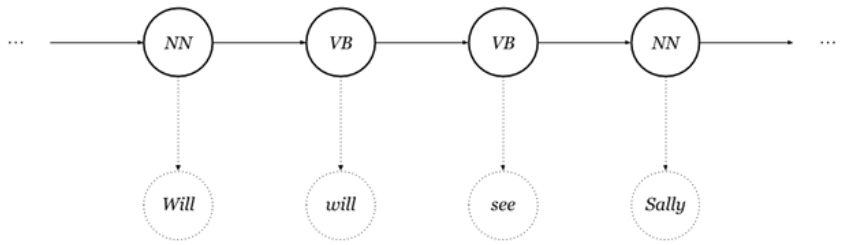

Figure 1: hidden Markov model for PoS tagging.

4.1 Define the objective function

Assume a sentence S consist of words w_i, which has PoS z_i, so we have:

$$\begin{cases} S = (w_1, \ldots, w_i, \ldots, w_n) \\ Z = (z_1, \ldots, z_i, \ldots, z_n) \end{cases}$$

According to the idea of maximum likelihood estimation, here we aim to find a chain of labels Z, which can maximize $P(Z|S)$. Our objective function is:

$$Z^* = \arg\max_Z P(Z|S) \tag{5}$$

By using Bayesian formula, we have $Z^* = \arg\max_Z P(S|Z)P(Z)$.

There are two assumptions in hidden Markov model:

Markov assumption: the hidden state at any moment depends only on the previous hidden state:

$$P(z_i|z_1...z_{i-1}) = P(z_i|z_{i-1})$$

Independent Observations Assumption: the observations depend only on the hidden state at the current time:

$$P(w_i|z_1...z_i...z_n) = P(w_i|z_i)$$

With the assumption the two probabilities in the formula can be reduced to:

$$\begin{cases} P(S|Z) = \prod_{i=1}^{n} P(w_i|z_i) \\ P(Z) = P_{start}(z_1) \prod_{i=1}^{n} P(z_i|z_{i-1}) \end{cases}$$

The objective function transformed into:

$$Z^* = \arg\max_Z \prod_{i=1}^{n} P(w_i|z_i) * P_{start}(z_1) * \prod_{j=2}^{n} P(z_i|z_{i-1}) \tag{6}$$

In the formula (6) There are a lot of probability multiplication operations, and the probability is between 0 & 1, which may eventually lead to floating-point underflow. We can take the logarithm to avoid underflow. Based on this idea, the formula can be further transformed into:

$$Z^* = \arg\max_Z \sum_{i=1}^{n} \log P(w_i|z_i) + \log P_{start}(z_1) + \sum_{j=2}^{n} \log P(z_i|z_{i-1}) \tag{7}$$

We also consider the probability of a word with PoS z_n appeared at the end of a sentence :

$$Z^* = \arg\max_Z \sum_{i=1}^{n} \log P(w_i|z_i) + \log P_{start}(z_1) + \sum_{j=2}^{n} \log P(z_i|z_{i-1}) + \log P_{end}(z_n) \tag{8}$$

Formula (8) is our final objective function.

4.2 Determine the parameters

Our objective function contains four variable parameters, in this subsection we explained the meaning of each parameters, and how to compute their value spaces.

• $P(w_i|z_i)$: this parameter is the **emission probability** of the HMM model, it represents the conditional probability of observing a word w_i with a given PoS z_i. Which can be calculated as the proportion of the word w_i in all the words marked as PoS z_i according to the idea of maximum likelihood estimation:

$$P(w_i|z_i) = \frac{\#\ w_i\ which\ PoS = z_i}{\#\ all\ word\ which\ PoS = z_i}$$

.

Each PoS is an hidden state of the HMM model.

• $P(z_i|z_{i-1})$: this parameter represents the **transition probability** between two states z_{i-1} and z_i, it's computed as the conditional probability of observing a word which PoS is z_i with a given PoS z_{i-1}:

$$P(z_i|z_{i-1}) = \frac{\#\ bigram\ which\ PoS = (z_{i-1}, z_i)}{\#\ all\ word\ which\ PoS = z_i}$$

• $P_{start}(z_1)$: this parameter estimate the **starting probability**. It represents the probability of the state z_1 being the label of the first in a sentence:

$$P_{start}(z_1) = \frac{\#\ starting\ word\ which\ PoS = z_1}{\#\ sentence}$$

• $P_{end}(z_n)$: this parameter estimate the **ending probability**. It represents the probability of a word with PoS z_n appeared at the end of a sentence:

$$P_{end}(z_n) = \frac{\#\ ending\ word\ which\ PoS = z_n}{\#\ all\ word\ which\ PoS = z_n}$$

4.3 Compute the optimal solution

Through the analysis in subsection 4.2, we have determined four parameters and their value space. In the next step, We can roughly enumerate all the parameters to find the combination parameters that maximize the objective function for each sentence in test set, but with a very large time complexity.

We find that it's an optimization problem to solve the objective function. The problem can be decomposed into T steps, each step can be nested recursively inside larger problems, so that dynamic programming methods are applicable for this problem.

In this work, dynamic programming (Viterbi) algorithm in the Pomegranate library is used to optimizes the parameters, the time complexity of this algorithm is $O(N^2 * T)$ where N is the number of types of PoS and T is the sentence length of the test set. This step also named "Decoding" in HMM model.

4.4 Training & Evaluating steps

During the training steps, we first compute all the fours parameters based on the train set of a corpus. Then we use these parameters to "bake" the HMM model.

During the evaluating steps, we use the Viterbi algorithm to decode all the sentences in the test set with the HMM model trained to predict the PoS of each words in the test set.

5 Decision Tree model

Decision Tree (DT) is a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

In this work, we simply use the DT model from the sklearn library Pedregosa et al. (2011) and consider the results of this model as a baseline. The features used for the DT model are exactly the same as the features that we've extracted for our Perceptron model. However, the DT model from sklearn doesn't really accept string type value as input value, so we implement ourselves a "fake" one-hot encoding algorithm.

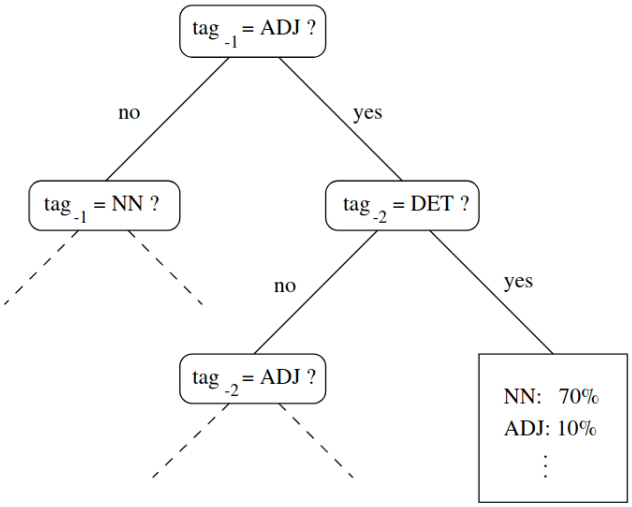

Figure 2: Decision Tree for PoS tagging.Schmid (1994)

5.1 "Fake" one-hot encoding

A real **one-hot encoding** is using a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0) to represent the features of an input value. However, if we use the real one-hot encoding to represent the data in this PoS tagging task, the features extracted will become very sparse (for example, there are thousands of different words which will transform into a vector of thousands of bits...) And this requires a lot of computing resources.(which crashed our PC for several times) To solve this problem, we simply use the numbers from 0 to $N-1$ instead of bits to represent N different types of value.

Windows Feature:

• **train set:** We build a dictionary of words for train set $words\_dict\{`word' : id\}$, the keys of the dictionary are the words in type string and the values are their id number. Each distinct word in train set has a single id number. The ids of the words are considered as the windows features instead of the words themselves, for the places where some of the windows features aren't existed (for example place $l-1$ for the first word of a sentence), we simply mark -1.

• **test set:** We use the same id numbers from the dictionary created with the train set to represent the word in test set. The words which only appear in test set (the OOV words) are also marked -1.

Suffix Feature:

• **train set:** We iterate over all words and build a dictionary of suffix for train set, the keys of the dictionary are the suffix in type string and the values are their id number. The number of suffix is fixed to 10, the words which doesn't have enough suffix will own some -1.

• **test set:** Same as Windows Feature, the same id numbers from the dictionary created with the train set are used to represent the suffix in test set. the OOV suffix are marked -1.

Shape Feature:

As the Shape Feature doesn't have to much types, we could use a real one-hot encoding for it. Each word is mapped to a bit string encompassing 8 binary indicators that correspond to different orthographic (This feature is not changed).

5.2 Some issues

With the features extracted from the train set using our "fake" one-hot encoding algorithm, the DT model can finally achieve an acceptable performance. But this algorithm can also rise some issues.

Words ordered: When we mark all the words with their id numbers, that means these words could be sorted according to id. For example, if we have id of *pomme* = 99 and id of *banana* = 101, then we will have something strange like "*pomme* < *banana*". For the DT model, maybe it will learn a node with "$id < 100 = pomme$; $id > 100 = banana$" which is not logical. The DT model should be able to achieve a better performance if we could use the real one-hot encoding.

OOV words/suffix: All the OOV words and the OOV suffix are all marked -1, this may cause some confusions for the DT model. We should design other strategies for the unknown values in the future work.

6 Results & Evaluation

In this section, we train our 3 models (or taggers) on each of the 6 train sets, then evaluate their accuracy on the 8 test sets. The experimental results for **All accuracy** (accuracy for all words), **OOV accuracy** (accuracy for words not occurring in the train set) and **Ambiguous accuracy** (named 'AMB', accuracy for words that appear with more than one PoS in the train set) are reported in Table 3. The results are analysed with the information of corpus which are described in section 2.2 Metric results of Experimental data, a column's best result is bold.

Table 3: The results

	Perceptron			HMM			DecisionTree		
Corpus	ALL	AMB	OOV	ALL	AMB	OOV	ALL	AMB	OOV
fr.ftb	**95.65**	98.15	80.93	**94.63**	93.30	46.54	**92.26**	99.87	**56.90**
fr.gsd	95.02	99.31	**81.08**	90.82	93.87	42.68	89.71	99.97	56.25
fr.partut	92.60	**99.63**	72.35	93.00	95.65	38.10	84.14	99.92	44.36
fr.pud	82.05	**99.63**	68.35	79.02	80.00	42.03	65.33	99.92	33.24
fr.sequoia	95.04	99.34	78.98	91.40	**96.10**	**53.56**	87.36	**99.99**	45.83
fr.spoken	90.47	99.23	76.05	87.81	93.37	34.78	81.76	99.86	54.51

6.1 The results

All accuracy. We can see from Table 3 that the model Perceptron, HMM and DecisionTree all work best on corpus *ft.ftb*, that's because the corpus has the largest training set (442228 words) so that the model has more chance to learning the relation between the features and labels. And another reason is that the corpus is only from the newspapers which means its grammar is more accurate and easier to be learnt. And also it hasn't much noise in test set from Table 2, only 21% OOV and $2e^{-5}$ KL divergence which is small.

The three models work good (have similar results) on corpus *fr.gsd*, the reason is almost the same as previous, corpus *ft.ftb*. The two corpus have similar size of training set and percentage on OOV. However the results are a little bit worse than which on *ft.ftb*. One reason perhaps is that *ft.gsd* has higher KL divergence($3e-4$ on gsd and $2e-5$ on ftb) which means it has more noise on test set. And another reason is that a part of data of *ft.gsd* comes from *blog* that means the grammar of which is not as rigorous as that from newspaper. The result on corpus *fr.sequoia* is very closed to the result on *ft.gsd* as they have very similar training set and noise(similar percentage OOV and KL divergence.)

However the three models work worst on corpus *fr.pub* because from Table 2 we can see it has much noise on test set (72.56% OOV which is the most, and perplexity 670.71), so the models make predicting on many words/sentences which are never seen. And also this corpus hasn't enough training words(only 23324 training words but 24138 testing words) to learn the relation between the features and the labels.

The corpus *fr.spoken* has big rate of OOV (64% while *fr.pud* has 72%) but the accuracy on it is not bad (90.47% for Perceptron), which is much better than that on fr.pud (82.06%), same result by HMM/DecisionTree. Even *fr.spoken* has more KL divergence ($7e^{-5}$) than *fr.pud*'s KL divergence ($1.6e^{-5}$). One reason perhaps is that from Table 1 *fr.spoken* has less test set percentage (62%) than *fr.pud* (124%) which means less relation not learnt. Another reason is that the perplexity of *fr.spoken* (308.37) is much lower that which of *fr.pud* (670.71).

Accuracy on OOV. Perceptron model works best on corpus *fr.gsd*, obviously because *fr.gsd* has the least rate of OOV. The second is on corpus *fr.ftb* with the same reason as *fr.gsd*. In contrast, *fr.pub* has the most rate of OOV so it has the worst accuracy on OOV. So as the *fr.spoken*, it has the second worst accuracy on OOV as it has the second most rate of OOV.

For HMM, it works badly on OOV on all corpus(40% - 50% accuracy). The reason is that it marks all the words that it never seen as the additional word *UNK*. So HMM make prediction on the OOV words only by the label of their previous word according to the transition probability ($P(z_i|z_{i-1})$). The HMM model can't gather enough features from the context.

For Decision Tree, it works also badly on OOV on all corpus compared with Perceptron even it uses the same features as Perceptron. We believe that the main problem leading to this result is the encoding algorithm, as we mentioned in section 5.2. However, the performance is still better than HMM on most of the corpus (4 in 6).

Accuracy on AMB. Perceptron and Decision Tree both work very well on ambiguous words. That's because the ambiguous words are a part of the train set, with enough input value, these two model can easily become overfitting. Especially for Decision Tree, in theory, this model can achieve 100% accuracy

on train set if the tree is deep enough.

However, HMM works less better on AMB. The reason is still that this model can't gather enough features.

On other data set. Table 4 shows the accuracy got by these three models Perceptron, HMM and DecisionTree trained by the training set *fr.ftb* and evaluated on the 2 other test sets *fr.foot* & *fr.natdis*. From the results we can observe that our Perceptron model is almost the most robust compared to the two others. The results are not good because *fr.foot* contains words about football and *fr.natdis* contains words about disaster which are not often seen in training set of *fr.ftb*.

Table 4: Results on the other 2 test sets

	fr.foot		fr.natdis	
Taggers	ALL	OOV	ALL	OOV
Perceptron	**68.84**	**33.86**	78.39	**25.73**
HMM	63.04	14.05	**79.08**	20.55
DecisionTree	61.60	24.17	74.68	18.16

Overall Perceptron best-perform on ALL and OOV. HMM and DecisionTree work badly on OOV and Decision Tree works best on AMB.

6.2 The importance of Features

In this subsection, we investigate how different modifications of the basic Perceptron model affect performance. We consider the DecisionTree and the HMM model as the baselines, and we observe the effect of:

• omitting one of the two feature types $suffixes$ & $shapes$.

• ignore the $word\ itself$.

• window size $l = 1$.

compared to basic model(using all features) with Perceptron model. The results of all the comparisons are shown in Table 5, a column's best result is bold.

From Table 5 we can conclude:

- For the columns of ALL for the six corpus, the features *shapes* and *words on window size l = 2* did little influence on the result while the feature *suffixes* has more importance and *word itself* has the most importance among the 4 features.
- For the columns of OOV for six corpus, the features *shapes* and *suffixes* are still important for the prediction of the words never seen.

Table 5: The results with different features

		fr.ftb		fr.gsd		fr.partut		fr.pud		fr.sequoia		fr.spoken	
		ALL	OOV	ALL	OOV	ALL	OOV	ALL	OOV	ALL	OOV	ALL	OOV
	DTree	92.26	56.90	89.71	56.25	84.14	44.36	65.33	33.24	87.36	45.83	81.76	54.51
	HMM	94.63	46.54	90.82	42.68	**93.00**	38.10	79.02	42.03	91.40	53.56	87.81	34.78
P	basic	**95.65**	80.93	**95.02**	81.08	92.60	72.35	82.05	68.35	95.04	78.98	90.47	76.05
P	no suffixes	94.66	70.13	94.21	76.22	90.46	66.55	80.38	66.12	93.58	72.86	88.79	73.81
P	no shapes	95.38	74.96	94.83	76.91	92.80	72.01	79.88	60.80	94.86	74.97	88.72	66.80
P	$v\pm\{1,2\}$	92.12	**86.35**	90.79	**86.52**	88.47	**83.15**	77.19	**70.73**	91.17	**83.72**	81.20	76.25
P	$l = 1$	95.46	79.83	94.93	81.60	92.84	75.09	**82.42**	70.29	**95.65**	81.31	**91.20**	**76.61**

- For the columns of OOV for six corpus,if we ignore *word itself*, surprisingly the model can get much better performance on OOV than using it. That's because in this situation, the representations of unknown words are exactly the same type as representations of known words, the feature *word itself* is never been considered during training. For domains where we expect many OOV words, omitting the *word itself* could be helpful.
- In the case when train set is small (for the last 4 corpus on ALL & OOV), using *window size $l = 1$* is better than using *window size $l = 2$* both on ALL and OOV. Because the words in the feature of *window size $l = 2$* can much vary so it is difficult to correctly learn the its relation between the labels when lack of training data. Thus, for domains where we can't gather enough training data, using *window size $l = 1$* should be considered.

7 Conclusion

In this work we've implemented 4 models, Perceptron which is defined by ourselves, HMM, Decision Tree and SVM which can't work for now because of lack of computing resources. We have tried many approach of feature extraction such as window feature, suffix feature, shape feature, distributional feature, n-gram feature, etc. We've fully understood the main concepts of statistical learning. The first three model that we implemented work good on this task while Perceptron works best. Perceptron and Decision Tree work better on AMB than HMM. However HMM and Decision Tree work badly on OOV.

In the future work, we can try to find more features for the words in the PoS taggging, or consider using other approach to extract features such as word embedding. Also, we can use neural network to get hidden representation of the words. The parallelization algorithm also should be considered to limit the time consumed.

References

Jurafsky, D. and Martin, J. H. (2019). Hidden markov models. In *Speech and Language Processing*, pages 548–562. 3th edition.

Martínez Alonso, H., Seddah, D., and Sagot, B. (2016). From noisy questions to Minecraft texts: Annotation challenges in extreme syntax scenario. In *Proceedings of the 2nd Workshop on Noisy User-generated Text (WNUT)*, pages 13–23, Osaka, Japan. The COLING 2016 Organizing Committee.

Nivre, J., Abrams, M., and other (2018). Universal dependencies 2.3. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees, intl. In *Conference on New Methods in Language Processing. Manchester, UK*.

Schnabel, T. and Schütze, H. (2014). FLORS: Fast and simple domain adaptation for part-of-speech tagging. *Transactions of the Association for Computational Linguistics*, 2:15–26.