Introduction

Ce TP continue le TP précédent. Nous allons reprendre d'ailleurs les mêmes données et commencer la mise en oeuvre d'un modèle de Markov pour la prédiction des étiquettes:

- une observation est une phrase, représentée comme une séquence de variables aléatoires, une par mot de la phrase
- à cette observation est associée une séquence de variables aléatoires représentant les étiquettes, une par mot de la phrase également

On suppose que la séquence d'observation (une phrase) est générée par un modèle de Markov caché. Les variables cachées sont donc les étiquettes à inférer. Nous allons commencer par écrire une classe python pour représenter le HMM. Cette classe évoluera au fil des TPs.

Pour cela le code de départ suivant est donné:

In [1]:

```
import nltk
from numpy import array, ones, zeros
import sys

# Some words in test could be unseen during training, or out of the vocabulary (OOV) even during
 the training.
# To manage OOVs, all words out the vocabulary are mapped on a special token.
UNK = "<unk>"
UNKid = 0

class HMM:
        def __init__(self, state_list, observation_list,
                 transition_proba = None,
                 observation_proba = None,
                 initial_state_proba = None):
            """Builds a new Hidden Markov Model
            state_list is the list of state symbols [q_0...q_(N-1)]
            observation_list is the list of observation symbols [v_0...v_(M-1)]
            transition_proba is the transition probability matrix
                [a_ij] a_ij = Pr(Y_(t+1)=q_i|Y_t=q_j)
            observation_proba is the observation probablility matrix
                [b_ki] b_ki = Pr(X_t=v_k|Y_t=q_i)
            initial_state_proba is the initial state distribution
                [pi_i] pi_i = Pr(Y_0=q_i)"""
            print "HMM creating with: "
            self.N = len(state_list) # The number of states
            self.M = len(observation_list) # The number of words in the vocabulary
            print str(self.N)+" states"
            print str(self.M)+" observations"
            self.omega_Y = state_list # Keep the vocabulary of tags
            self.omega_X = observation_list # Keep the vocabulary of tags
            # Init. of the 3 distributions : observation, transition and initial states
            if transition_proba is None:
                self.transition_proba = zeros( (self.N, self.N), float)
            else:
                self.transition_proba=transition_proba
            if observation_proba is None:
                self.observation_proba = zeros( (self.M, self.N), float)
            else:
                self.observation_proba=observation_proba
            if initial_state_proba is None:
                self.initial_state_proba = zeros( (self.N,), float )
            else:
                self.initial_state_proba=initial_state_proba
            # Since everything will be stored in numpy arrays, it is more convenient and compact
 to
            # handle words and tags as indices (integer) for a direct access. However, we also n
eed
            # to keep the mapping between strings (word or tag) and indices.
            self.make_indexes()

        def make_indexes(self):
            """Creates the reverse table that maps states/observations names
            to their index in the probabilities arrays"""
            self.Y_index = {}
            for i in range(self.N):
                self.Y_index[self.omega_Y[i]] = i
            self.X_index = {}
            for i in range(self.M):
                self.X_index[self.omega_X[i]] = i
```

Interface avec les données et apprentissage supervisé

Ainsi pour initialiser un HMM, nous allons devoir lire les données (chose faîte lors du TP précédent):

- écrire une fonction permettant d'initialiser le HMM à partir des données d'apprentissage
- écrire une fonction *apprentissage_supervisé* qui permet d'estimer les paramètres

Dans un premier temps, nous limiterons, comme lors du TP précédent, le vocabulaire aux mots apparaissant 10 fois ou plus. Les autres mots sont tous remplacés par la même forme *unk*

Pour cela, le plan de travail peut être envisagé ainsi:

- Lire les données puis générer un corpus de **train** (80%) puis de **test** (10%)
- écrire une fonction qui créer à partir des données d'apprentissage (**train**), tous les comptes nécessaires pour l'estimation supervisée des paramètres du HMM
- écrire 3 fonctions qui estimes les paramètres à partir des comptes, une fonction par distribution: observation, transition, état initial.
- écrire une fonction qui reprend le tout et qui estime tous les paramètres du HMM

Exercice : Algorithme de Viterbi

La question qui se pose est comment calculer la meilleure séquence d'étiquettes pour une phrase donnée connaissant les paramètres du HMM. Par meilleure, on entend la séquence d'étiquettes (ou d'états) la plus probable connaissant la séquence d'obervation.

Proposer et implémenter un algorithme répondant à cette question. Pour vous aider à démarrer, cet algorithme s'appelle Viterbi et regardez cette vidéo https://www.youtube.com/watch?v=RwwfUICZLsA (https://www.youtube.com/watch?v=RwwfUICZLsA), pour comprendre comment il opère.

TODO pour le 18/10/2016

- Finir la partie interface (qui comprend l'apprentissage supervisé)
- Regarder la vidéo et implémenter l'algorithme de Viterbi

TODO pour le 08/11/2016

- Calculer le taux d'erreur de prédiction sur les données de test
  - Implémenter une fonction effectuant l'inférence sur un jeu de donnée et qui calcul le taux d'erreur
- Il peut être avantageux de lisser les distributions d'observation. Expliquer pourquoi et implémenter le
- Choisir la bonne valeur de lissage sur les données de développement (en regardant le taux d'erreur) et constater l'effet sur les données de test

In [ ]: