CS563 Natural Language Processing
Assignment: 1

Mukuntha N S (1601CS27)
Shikhar Jaiswal (1601CS44)

February 4, 2020

1 Assignment Description

Designing and implementing a **Hidden Markov Model (HMM)** based model and a **Recurrent Neural Network (RNN)** based model and applying them for **Named Entity Recognition (NER)** on the **NER Tweet Datasets**.

2 Procedure

Installation

Install the following dependencies either using pip or through conda in a Python 3.5+ environment:
• jupyter
• numpy
• sklearn
• keras

```
python3 -m pip install jupyter numpy sklearn keras
```

or alternatively,

```
conda install -c anaconda jupyter numpy sklearn keras
```

Running The Notebook

To run the program, head to the directory $Assignment1/Q1$ and $Assignment1/Q2$. Please note that the **dataset should be present in the assignment folder**. Use the following command to run the notebook:

```
jupyter notebook
```

3 Features

Throughout most of the model, we consider words to be ordered pairs (or two-element vectors), composed of word and word-feature. The word feature is a simple, deterministic computation performed on each word as it is added to or looked up in the vocabulary. It produces one of the fourteen values stated as follows:

Features

Word Feature	Example Text	Intuition
`twoDigitNum`	90	Two-digit year
`fourDigitNum`	1990	Four digit year
`containsDigitAndAlpha`	A8956-67	Product code
`containsDigitAndDash`	09-96	Date
`containsDigitAndSlash`	11/9/89	Date
`containsDigitAndComma`	23,000.00	Monetary amount
`containsDigitAndPeriod`	1.00	Monetary amount, percentage
`otherNum`	456789	Other number
`allCaps`	BBN	Organization
`capPeriod`	M.	Person name initial
`firstWord`	*first word of sentence*	No useful capitalization information
`initCap`	Sally	Capitalized word
`lowerCase`	can	Uncapitalized word
`other`	,	Punctuation marks, all other words

4 Discussion

The primary objective of the assignment is to implement a Hidden Markov Model (HMM) based model and a Recurrent Neural Network (RNN) based model and applying them for Named Entity Recognition (NER). The following sub-sections contain the explanation for individual code snippets. For a detailed look at the output, please refer the notebook and the individual files provided.

Notebook Code - HMM

Pre-processing and Visualization

Import the Python dependencies, and check the data samples and their values and pre-process the corpus.

```
# Import the required libraries.
import re
import math
import random
import collections
import operator
import numpy as np
from sklearn.model_selection import KFold
```

```
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.metrics import confusion_matrix
from collections import defaultdict, Counter

random.seed(11)
np.random.seed(11)

with open('NER-Dataset-Train.txt', 'r') as f:
    ner_dataset = f.readlines()

sentences = []
words = []
tags = []
for line in ner_dataset:
    line = line.strip()
    if line == '':
        sentences.append((words, tags))
        words = []
        tags = []
    else:
        word, tag = line.split('\t')
        words.append(word)
        tags.append(tag)

if len(words) > 0:
    sentences.append((words, tags))
    words = []
    tags= []

vocab_counts = Counter(sum([a[0] for a in sentences], [])).most_common()
words_to_keep = set([word for word, count in vocab_counts if count > 1])

parsed_sentences = [([w if w in words_to_keep else 'UNK' for w in words], tags) for
                      words, tags in sentences]
```

```
def get_vocab(X_train, Y_train):
    '''
    Function for building the vocabulary from the training set of
    words and tags.
    '''
    vocabulary2id = dict()
    tag2id = dict()
    vocabulary2id['UNK'] = 0

    for sent in X_train:
        for word in sent:
            if word not in vocabulary2id.keys():
                vocabulary2id[word] = len(vocabulary2id)
```

```
    for sent in Y_train:
        for tag in sent:
            if tag not in tag2id.keys():
                tag2id[tag] = len(tag2id)

    return vocabulary2id, tag2id

def get_word_tag_counts(X_train, Y_train, vocabulary2id, tag2id):
    '''
    Function for calculating the counts pertaining to the
    individual word tags.
    '''
    wordcount = defaultdict(int)
    tagcount = defaultdict(int)
    tagpaircount = defaultdict(int)
    tagtriplecount = defaultdict(int)

    for sent in X_train:
        for word in sent:
            wordcount[word] += 1

    for sent in Y_train:
        for tag in sent:
            tagcount[tag] += 1

    for sent in Y_train:
        for i in range(len(sent) - 1):
            tagpaircount[sent[i], sent[i + 1]] += 1

    for sent in Y_train:
        for i in range(len(sent) - 2):
            tagtriplecount[sent[i], sent[i + 1], sent[i + 2]] += 1

    return wordcount, tagcount, tagpaircount, tagtriplecount
```

Hidden Markov Model Code

We implement a standard second-order Hidden Markov Model. We split the overall training dataset into 5 fold cross validation sets. We obtain the accuracy, precision, recall and f-score on the individual folds. Finally, we obtain the prediction results on the test dataset.

```
# Token to map all out-of-vocabulary words (OOVs).
UNK = "UNK"
# Index for UNK
UNKid = 0
epsilon = 1e-100
array, ones, zeros, multiply, unravel_index = np.array, np.ones, np.zeros, np.multiply,
```

```
                                          np.unravel_index

class HMM:
    def __init__(self, state_list, observation_list, transition_proba = None,
                 observation_proba = None, initial_state_proba = None,
                 smoothing_obs = 0.01, transition_proba1 = None, prob_abs = 0.00001):
        '''
        Builds a 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,a_ik = Pr(Y_(t+1)=q_i|Y_t=q_j,Y_(t-1)=q_k)
        * 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)
        '''
        # Number of states.
        self.N = len(state_list)
        # Number of possible emissions.
        self.M = len(observation_list)
        self.prob_abs = prob_abs
        self.omega_Y = state_list
        self.omega_X = observation_list

        if transition_proba1 is None:
            self.transition_proba1 = zeros( (self.N, self.N), float)
        else:
            self.transition_proba1 = transition_proba1

        if transition_proba is None:
            self.transition_proba = zeros( (self.N, 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

        # Build indexes, i.e., the mapping between token and int.
        self.make_indexes()
        self.smoothing_obs = smoothing_obs
```

```
def make_indexes(self):
    '''
    Function for creating the reverse table that maps
    states/observations names to their index in the probabilities
    array.
    '''
    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

def get_observationIndices(self, observations):
    '''
    Function for returning observation indices,
    and dealing with OOVs.
    '''
    indices = zeros( len(observations), int )
    k = 0

    for o in observations:
        if o in self.X_index:
            indices[k] = self.X_index[o]
        else:
            indices[k] = UNKid

        k += 1

    return indices

def data2indices(self, sent):
    '''
    Function for extracting the words and tags and returning a
    list of indices for each.
    '''
    wordids = list()
    tagids  = list()

    for couple in sent:
        wrd = couple[0]
        tag = couple[1]

        if wrd in self.X_index:
```

```
                wordids.append(self.X_index[wrd])
            else:
                wordids.append(UNKid)

            tagids.append(self.Y_index[tag])

        return wordids, tagids

    def observation_estimation(self, pair_counts):
        '''
        Function for building the observation distribution where
        observation_proba is the observation probablility matrix.
        '''
        # Fill with counts.
        for pair in pair_counts:
            wrd = pair[0]
            tag = pair[1]
            cpt = pair_counts[pair]
            # For UNK.
            k = 0

            if wrd in self.X_index:
                k = self.X_index[wrd]

            i = self.Y_index[tag]
            self.observation_proba[k, i] = cpt

        # Normalize.
        self.observation_proba = self.observation_proba + self.smoothing_obs
        self.observation_proba = self.observation_proba /
                                 self.observation_proba.sum(axis = 0).reshape(1, self.N)

    def transition_estimation(self, trans_counts):
        '''
        Function for building the transition distribution where
        transition_proba is the transition matrix with:
        [a_ij] a[i, j] = Pr(Y_(t+1) = q_i | Y_t = q_j, Y_(t-1) = q_k)
        '''
        # Fill with counts.
        for triple in trans_counts:
            i = self.Y_index[triple[2]]
            j = self.Y_index[triple[1]]
            k = self.Y_index[triple[0]]
            self.transition_proba[k, j, i] = trans_counts[triple]

        # Normalize.
        self.transition_proba = self.transition_proba /
                                self.transition_proba.sum(axis = 0).reshape(self.N, self.N)
```

```python
def transition_estimation1(self, trans_counts):
    '''
    Function for building the transition distribution where
    transition_proba is the transition matrix with:
    [a_ij] a[i,j] = Pr(Y_(t+1)=q_i|Y_t=q_j)
    '''
    # Fill with counts.
    for pair in trans_counts:
        i = self.Y_index[pair[1]]
        j = self.Y_index[pair[0]]
        self.transition_proba1[j, i] = trans_counts[pair]

    # Normalize.
    self.transition_proba1 = self.transition_proba1 /
                             self.transition_proba1.sum(axis = 0).reshape(1, self.N)

def init_estimation(self, init_counts):
    '''
    Function for building the initial distribution.
    '''
    # Fill with counts.
    for tag in init_counts:
        i = self.Y_index[tag]
        self.initial_state_proba[i] = init_counts[tag]

    # Normalize.
    self.initial_state_proba = self.initial_state_proba / sum(self.initial_state_proba)

def supervised_training(self, pair_counts, trans_counts, init_counts, trans_counts1):
    '''
    Function for training the HMM's parameters.
    '''
    self.observation_estimation(pair_counts)
    self.transition_estimation(trans_counts)
    self.transition_estimation1(trans_counts1)
    self.init_estimation(init_counts)

def viterbi(self, observations):
    if len(observations) < 2:
        return [hmm.Y_index[z] for z in observations]

    nSamples = len(observations)
    # Number of states.
    nStates = self.transition_proba.shape[0]
    # Scale factors (necessary to prevent underflow).
    c = np.zeros(nSamples)
    # Initialise viterbi table.
```

```
    viterbi = np.zeros((nStates, nStates, nSamples))
    # Initialise viterbi table.
    viterbi1 = np.zeros((nStates, nSamples))
    # Initialise the best path table.
    psi = np.zeros((nStates, nStates, nSamples))
    best_path = np.zeros(nSamples)
    idx0 = self.X_index[observations[0]]
    idx1 = self.X_index[observations[1]]
    viterbi1[:, 0] = self.initial_state_proba.T * self.observation_proba[idx0, :].T

    # Loop through the states.
    for s in range (0, nStates):
        for v in range (0, nStates):
            viterbi[s, v, 1] = viterbi1[s, 0] * self.transition_proba1[s, v] *
                               self.observation_proba[idx1, v]

    psi[0] = 0;

    # Loop through time-stamps.
    for t in range(2, nSamples):
        idx = self.X_index[observations[t]]
        # Loop through the states.
        for s in range (0, nStates):
            for v in range (0, nStates):
                self.transition_proba[np.isnan(self.transition_proba)] = self.prob_abs
                trans_p = viterbi[:, s, t-1] * self.transition_proba[:, s, v]

                if (math.isnan(trans_p[0])):
                    trans_p[0] = 0

                psi[s, v, t], viterbi[s, v, t] = max(enumerate(trans_p),
                                                     key = operator.itemgetter(1))
                viterbi[s, v, t] = viterbi[s, v, t] * self.observation_proba[idx, v]

    cabbar = viterbi[:, :, nSamples - 1]
    best_path[nSamples - 1] = unravel_index(cabbar.argmax(), cabbar.shape)[1]
    best_path[nSamples - 2] = unravel_index(cabbar.argmax(), cabbar.shape)[0]

    # Return the best path, number of samples and psi.
    for t in range(nSamples - 3, -1, -1):
        best_path[t] = psi[int(round(best_path[t + 1])),
                       int(round(best_path[t + 2])), t + 2]

    return best_path

def fwd_bkw(self, observations):
    observations = x
    self = hmm
```

```
nStates = self.transition_proba.shape[0]
start_prob = self.initial_state_proba
trans_prob = self.transition_proba1.transpose()
emm_prob = self.observation_proba.transpose()

# Forward part of the algorithm.
fwd = []
f_prev = {}

for i, observation_i in enumerate(observations):
    f_curr = {}
    for st in range(nStates):
        if i == 0:
            # Base case for the forward part.
            prev_f_sum = start_prob[st]
        else:
            prev_f_sum = sum(f_prev[k] * trans_prob[k][st] for k in range(nStates))

        f_curr[st] = emm_prob[st][self.X_index[observation_i]] * prev_f_sum

    fwd.append(f_curr)
    f_prev = f_curr

p_fwd = sum(f_curr[k] for k in range(nStates))

# Backward part of the algorithm.
bkw = []
b_prev = {}

for i, observation_i_plus in enumerate(reversed(observations[1:] + [None,])):
    b_curr = {}
    for st in range(nStates):
        if i == 0:
            # Base case for backward part.
            b_curr[st] = 1.0
        else:
            b_curr[st] = sum(trans_prob[st][l] *
                             emm_prob[l][self.X_index[observation_i_plus]]
                             * b_prev[l] for l in range(nStates))

    bkw.insert(0,b_curr)
    b_prev = b_curr

p_bkw = sum(start_prob[l] * emm_prob[l][self.X_index[observations[0]]] * b_curr[l]
            for l in range(nStates))

# Merging the two parts.
posterior = []
```

```
        for i in range(len(observations)):
            posterior.append({st: fwd[i][st] * bkw[i][st] / p_fwd for st in range(nStates)})

        assert abs(p_fwd - p_bkw) < 1e-6
        return fwd, bkw, posterior
```

```
def make_counts(X, Y):
    '''
    Function for building the different count tables to train a HMM.
    Each count table is a dictionary.
    '''
    c_words = dict()
    c_tags = dict()
    c_pairs= dict()
    c_transitions = dict()
    c_inits = dict()
    c_transitions1 = dict()

    for sent in zip(X, Y):
        sent = list(zip(*sent))
        for i in range(len(sent)):
            couple = sent[i]
            wrd = couple[0]
            tag = couple[1]

            # Word counts.
            if wrd in c_words:
                c_words[wrd] = c_words[wrd] + 1
            else:
                c_words[wrd] = 1

            # Tag counts.
            if tag in c_tags:
                c_tags[tag] = c_tags[tag] + 1
            else:
                c_tags[tag] = 1

            # Observation counts.
            if couple in c_pairs:
                c_pairs[couple] = c_pairs[couple] + 1
            else:
                c_pairs[couple] = 1

            if i >= 1:
                trans1 = (sent[i - 1][1], tag)
                if trans1 in c_transitions1:
                    c_transitions1[trans1] = c_transitions1[trans1] + 1
```

```
                else:
                    c_transitions1[trans1] = 1

            if i > 1:
                trans = (sent[i - 2][1], sent[i - 1][1], tag)
                if trans in c_transitions:
                    c_transitions[trans] = c_transitions[trans] + 1
                else:
                    c_transitions[trans] = 1
            else:
                if tag in c_inits:
                    c_inits[tag] = c_inits[tag] + 1
                else:
                    c_inits[tag] = 1

    return c_words,c_tags,c_pairs, c_transitions, c_inits, c_transitions1
```

Testing and Analysis

We perform 5-fold cross validation along with accuracy, precision, recall and f-score metrics for both the regular and the 10-types dataset.

```
# Build the test and training sets of sentences.
kf = KFold(n_splits = 5, shuffle = False)
parsed_sentences = np.asarray(parsed_sentences)
scores = []
scores1 = []
y_pred_idx = []
y_pred_idx1 = []
y_test_idx = []
y_test_idx1 = []

for train_index, test_index in kf.split(parsed_sentences):
    train_data = parsed_sentences[train_index]
    test_data = parsed_sentences[test_index]
    X_train = [a[0] for a in train_data]
    Y_train = [a[1] for a in train_data]
    X_test = [a[0] for a in test_data]
    Y_test = [a[1] for a in test_data]

    # Build the vocabulary and word counts.
    vocabulary2id, tag2id = get_vocab(X_train, Y_train)
    wordcount, tagcount, tagpaircount, tagtriplecount = get_word_tag_counts(X_train,
                                    Y_train, vocabulary2id, tag2id)
    cwords, ctags, cpairs, ctrans, cinits, ctrans1 = make_counts(X_train, Y_train)

    state_list = list(ctags.keys())
    observation_list = [a[0] for a in sorted(vocabulary2id.items(), key = lambda x: x[1])]
```

```
    hmm = HMM(state_list = state_list, observation_list = observation_list,
              transition_proba = None, observation_proba = None, initial_state_proba = None,
              smoothing_obs = 0.4, prob_abs = 0)
    hmm.supervised_training(cpairs, ctrans, cinits, ctrans1)

    for x, y_true in zip(X_test, Y_test):
        for i in range(len(x)):
            if x[i] not in vocabulary2id.keys():
                x[i] = 'UNK'

        pred_idx = hmm.viterbi(x)
        y_pred = np.asarray([state_list[int(round(i))] for i in pred_idx])
        y_true = np.asarray(y_true)
        y_pred_idx += np.asarray([tag2id[lab] for lab in y_pred], dtype = np.int32).tolist()
        y_test_idx += np.asarray([tag2id[lab] for lab in y_true], dtype = np.int32).tolist()
        scores += (y_pred == y_true).tolist()

    x, y_true = X_train[0], Y_train[0]

    for x, y_true in zip(X_test, Y_test):
        for i in range(len(x)):
            if x[i] not in vocabulary2id.keys():
                x[i] = 'UNK'

        pred_probs = hmm.fwd_bkw(x)
        pred_idx = [max(probs.items(), key=lambda x: x[1])[0] for probs in pred_probs[2]]
        y_pred = np.asarray([state_list[int(round(i))] for i in pred_idx])
        y_true = np.asarray(y_true)
        y_pred_idx1 +=np.asarray([tag2id[lab] for lab in y_pred], dtype = np.int32).tolist()
        y_test_idx1 +=np.asarray([tag2id[lab] for lab in y_true], dtype = np.int32).tolist()
        scores1 += (y_pred == y_true).tolist()

prec, rec, fscore, _ = precision_recall_fscore_support(y_test_idx, y_pred_idx,
                                                       average = 'macro')

print('Overall Accuracy and Scores:')
print('Only Forward-Backward Accuracy: {}, Precision: {}, Recall: {}, FScore: {}'.format(
    np.asarray(scores1).mean(), prec1, rec1, fscore1))
print('Viterbi Accuracy: {}, Precision: {}, Recall: {}, FScore: {}'.format(
    np.asarray(scores).mean(), prec, rec, fscore))
```

Regular NER Dataset Score

```
Fold 1 Accuracy and Scores:
Viterbi Accuracy: 1.0, Precision: 1.0, Recall: 1.0, FScore: 1.0

Fold 2 Accuracy and Scores:
```

```
Viterbi Accuracy: 0.9615384615384616, Precision: 0.4807692307692308, Recall: 0.5,
FScore: 0.49019607843137253

Fold 3 Accuracy and Scores:
Viterbi Accuracy: 0.8, Precision: 0.26666666666666666, Recall: 0.3333333333333333,
FScore: 0.29629629629629634

Fold 4 Accuracy and Scores:
Viterbi Accuracy: 0.9565217391304348, Precision: 0.5, Recall: 0.4782608695652174,
FScore: 0.4888888888888889

Fold 5 Accuracy and Scores:
Viterbi Accuracy: 0.8888888888888888, Precision: 0.3076923076923077,
Recall: 0.3333333333333333, FScore: 0.32

Overall Accuracy and Scores:
Viterbi Accuracy: 0.8954233409610984, Precision: 0.6493777509943442,
Recall: 0.6547955689413518, FScore: 0.6512071629205967
```

NER 10-Types Dataset Score

```
Fold 1 Accuracy and Scores:
Viterbi Accuracy: 1.0, Precision: 1.0, Recall: 1.0, FScore: 1.0

Fold 2 Accuracy and Scores:
Viterbi Accuracy: 0.9615384615384616, Precision: 0.4807692307692308, Recall: 0.5,
FScore: 0.49019607843137253

Fold 3 Accuracy and Scores:
Viterbi Accuracy: 0.8, Precision: 0.26666666666666666, Recall: 0.3333333333333333,
FScore: 0.29629629629629634

Fold 4 Accuracy and Scores:
Viterbi Accuracy: 1.0, Precision: 1.0, Recall: 1.0, FScore: 1.0

Fold 5 Accuracy and Scores:
Viterbi Accuracy: 0.8888888888888888, Precision: 0.2222222222222222, Recall: 0.25,
FScore: 0.23529411764705882

Overall Accuracy and Scores:
Viterbi Accuracy: 0.9012013729977116, Precision: 0.17299751704223215,
Recall: 0.1306308711792958, FScore: 0.1416866524575724
```

```
with open('NER-Dataset--TestSet.txt', 'r') as f:
    test_dataset = f.readlines()

test_sentences = []
```

```
words = []
for line in test_dataset:
    line = line.strip()
    if line == '':
        test_sentences.append((words,))
        words = []
    else:
        word = line
        words.append(word)

if len(words) > 0:
    test_sentences.append((words,))
    words = []
```

```
parsed_test_sentences = [[w if w in words_to_keep else 'UNK' for w in words[0]]
                          for words in test_sentences]
parsed_test_sentences = np.asarray(parsed_test_sentences)

train_data = parsed_sentences
test_data = np.asarray(parsed_test_sentences)
X_train = [a[0] for a in train_data]
Y_train = [a[1] for a in train_data]
X_test = [a for a in test_data]

# Build the vocabulary and word counts.
vocabulary2id, tag2id = get_vocab(X_train, Y_train)
wordcount, tagcount, tagpaircount, tagtriplecount = get_word_tag_counts(X_train, Y_train,
                                                    vocabulary2id, tag2id)
cwords, ctags, cpairs, ctrans, cinits, ctrans1 = make_counts(X_train, Y_train)

state_list = list(ctags.keys())
observation_list = [a[0] for a in sorted(vocabulary2id.items(), key = lambda x: x[1])]
hmm = HMM(state_list = state_list, observation_list = observation_list,
          transition_proba = None, observation_proba = None, initial_state_proba = None,
          smoothing_obs = 0.4, prob_abs = 0)
hmm.supervised_training(cpairs, ctrans, cinits, ctrans1)

predictions = []
for x in X_test:
    for i in range(len(x)):
        if x[i] not in vocabulary2id.keys():
            x[i] = 'UNK'

    pred_idx = hmm.viterbi(x)
    y_pred = np.asarray([state_list[int(round(i))] for i in pred_idx])
    predictions.append(y_pred)
```

```
test_predictions = [list(s) for s in predictions]
```

```
with open('NER-TestSet-HMM-Predictions.txt', 'w', encoding = 'utf-8') as f:
    for words, predictions in zip(test_sentences, test_predictions):
        assert(len(words[0]) == len(predictions))
        for word, prediction in zip(words[0], predictions):
            f.writelines(word + '\t' + prediction + '\n')
        f.writelines('\n')
```

Additionally please refer *Q1/NER-TestSet-HMM-Predictions.txt* and *Q1/NER-TestSet-10Types-HMM-Predictions.txt* for the predicted outputs for the HMM model trained on the regular and 10-type datasets respectively.

For the notebook codes, please refer *Q1/Q1 - NER Prediction (HMM).ipynb* and *Q1/Q1 - NER Prediction - 10 Types (HMM).ipynb* respectively.

Notebook Code - RNN

Pre-processing and Visualization

Import the Python dependencies, and check the data samples and their values and pre-process the corpus.

```
# Import the required libraries.
import re
import math
import random
import collections
import operator
import numpy as np

from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support, f1_score, accuracy_score
from sklearn.metrics import confusion_matrix
from collections import defaultdict, Counter

from keras.utils import to_categorical
from keras.layers import *
from keras.models import Model
from keras import Model, Sequential
from sklearn.model_selection import StratifiedKFold
from keras.callbacks import *
from sklearn.metrics import classification_report
from sklearn.utils.class_weight import compute_class_weight

random.seed(11)
np.random.seed(11)

with open('NER-Dataset-10Types-Train.txt', 'r') as f:
    ner_dataset = f.readlines()

sentences = []
words = []
```

```
tags = []
for line in ner_dataset:
    line = line.strip()
    if line == '':
        sentences.append((words, tags))
        words = []
        tags = []
    else:
        word, tag = line.split('\t')
        words.append(word)
        tags.append(tag)

if len(words) > 0:
    sentences.append((words, tags))
    words = []
    tags= []

vocab_counts = Counter(sum([a[0] for a in sentences], [])).most_common()
words_to_keep = set([word for word, count in vocab_counts if count > 1])

with open('NER-Dataset--TestSet.txt', 'r') as f:
    test_dataset = f.readlines()

test_sentences = []
words = []
for line in test_dataset:
    line = line.strip()
    if line == '':
        test_sentences.append((words,))
        words = []
    else:
        word = line
        words.append(word)

if len(words) > 0:
    test_sentences.append((words,))
    words = []
```

```
word_features = ['twoDigitNum',
                 'fourDigitNum',
                 'containsDigitAndAlpha',
                 'containsDigitAndDash',
                 'containsDigitAndSlash',
                 'containsDigitAndComma',
                 'containsDigitAndPeriod',
                 'otherNum',
                 'allCaps',
                 'capPeriod',
```

```python
                'firstWord',
                'initCap',
                'lowerCase',
                'other']
```

```python
def get_word_features(sentence):
    features = []
    ## Optimize and use an Enum!
    firstword = True
    for word in sentence:
        if word.isnumeric() and len(word) == 2:
            features.append('twoDigitNum')
        elif word.isnumeric() and len(word) == 4:
            features.append('fourDigitNum')
        elif word.isalnum() and not word.isalpha() and not word.isnumeric():
            features.append('containsDigitAndAlpha')
        elif word.replace('-', '').isnumeric():
            features.append('containsDigitAndAlpha')
        elif word.replace('/', '').isnumeric():
            features.append('containsDigitAndSlash')
        elif word.replace('.', '').replace(',', '').isnumeric() and ',' in word:
            features.append('containsDigitAndComma')
        elif word.replace('.', '').isnumeric():
            features.append('containsDigitAndPeriod')
        elif word.isnumeric():
            features.append('otherNum')
        elif word.isupper():
            features.append('allCaps')
        elif len(word) == 2 and word[0].isupper() and word[1] == '.':
            features.append('capPeriod')
        elif firstword:
            features.append('firstWord')
        elif word[0].isupper():
            features.append('initCap')
        elif word.islower():
            features.append('lowerCase')
        else:
            features.append('other')
        firstword = False
    return features
```

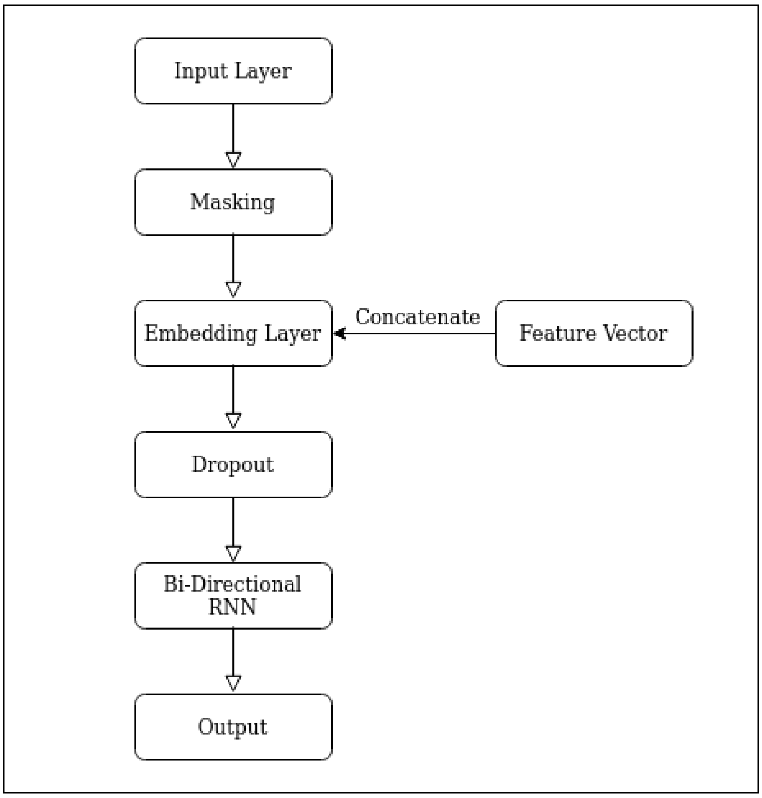
```python
max_len_found = max(len(s[0]) for s in sentences)
max_len = max_len_found + ((50 - (max_len_found % 50)) % 50)
eye_mat = list(np.eye(len(word_features)))
wordfeat2float = {feat: eye_mat[i] for i, feat in enumerate(word_features)}
word2idx = {'UNK': 0, 'PAD': 1}
word2idx.update({word: i + 2 for i, word in enumerate(sorted(words_to_keep))})
```

```
def numberize_sentence(words, max_len = 50):
    features = get_word_features(words)
    word_idx = [word2idx[w] if w in word2idx.keys() else word2idx['UNK'] for w in words]
    feat_np = [wordfeat2float[f] for f in features]
    word_padding = [word2idx['PAD'] for _ in range(max_len - len(word_idx))]
    feat_padding = [np.ones((len(word_features),)) * 2 for _ in range(max_len -
                    len(word_idx))]
    word_idx = np.asarray(word_idx + word_padding)
    feat_np = np.asarray(feat_np + feat_padding)
    return word_idx, feat_np

labels = set.union(*(set(s[1]) for s in sentences))
idx2labels = {i: s for i, s in enumerate(labels)}
n_labels = len(labels)
eye_mat = list(np.eye(len(labels)))
labels2float = {feat: eye_mat[i] for i, feat in enumerate(labels)}

def numberize_labels(gt_labels, max_len=50):
    labels_np = [labels2float[l] for l in gt_labels]
    labels_padding = [labels2float['O'] for _ in range(max_len - len(gt_labels))]
    return np.asarray(labels_np + labels_padding)
```

Recurrent Neural Network Code

We implement a standard Bi-Directional Recurrent Neural Network. We split the overall training dataset into 5 fold cross validation sets. We obtain the accuracy, precision, recall and f-score on the individual folds. Finally, we obtain the prediction results on the test dataset.

Architecture




```
def create_model():
    input_words = Input(shape = (max_len,))
    input_feats = Input(shape = (max_len, len(word_features)))
    masked_words = Masking(mask_value = 1)(input_words)
    masked_feats = Masking(mask_value = 2)(input_feats)
    emb = Embedding(input_dim = (len(word2idx)), output_dim = 50,
                    input_length = max_len)(masked_words)
    drop_emb = Dropout(0.1)(emb)
    concat_out = Concatenate()([drop_emb, masked_feats])
    rnn_out = Bidirectional(SimpleRNN(units = 100, return_sequences = True,
                                  recurrent_dropout = 0.1))(concat_out)
    dense_out = TimeDistributed(Dense(n_labels, activation = "softmax"))(rnn_out)
    model = Model(inputs = [input_words, input_feats], outputs = dense_out)
    model.summary()
    return model
```

```
parsed_sentences = [(numberize_sentence(s[0]), numberize_labels(s[1])) for s in sentences]
parsed_test_sentences = [numberize_sentence(s[0]) for s in test_sentences]
Counter(sum([s[1] for s in sentences], []))
Counter(sum([np.argmax(s[1], axis=-1).tolist() for s in parsed_sentences], []))
```

Testing and Analysis

We perform 5-fold cross validation along with accuracy, precision, recall and f-score metrics for both the regular and the 10-types dataset.

```
# Build the test and training sets of sentences.
kf = KFold(n_splits = 5, shuffle = False)
parsed_sentences = np.asarray(parsed_sentences)
scores = []
y_pred_idx = []
y_test_idx = []

preds = []
fold_count = 0
foldwise_score_outputs = []

for train_index, test_index in kf.split(parsed_sentences):
    fold_count += 1
    y_pred_idx_fold = []
    y_test_idx_fold = []
    scores_fold = []

    train_data = parsed_sentences[train_index]
    test_data = parsed_sentences[test_index]
    X_train = [np.asarray([a[0][0] for a in train_data]),
               np.asarray([a[0][1] for a in train_data])]
    Y_train = np.asarray([a[1] for a in train_data])
    X_test = [np.asarray([a[0][0] for a in test_data]),
              np.asarray([a[0][1] for a in test_data])]
    Y_test = np.asarray([a[1] for a in test_data])
    model = create_model()
    model.compile(optimizer = 'rmsprop',
                  loss = 'categorical_crossentropy',
                  metrics = ['accuracy'])

    model.fit(X_train, Y_train, epochs = 3, validation_split = 0.1, batch_size = 4)

    y_pred_padded = np.argmax(model.predict(X_test), axis = -1)
    y_true_padded = np.argmax(Y_test, axis = -1)

    for i in range(X_test[0].shape[0]):
        for j in range(X_test[0].shape[1]):
```

```
            if X_test[0][i][j] == word2idx['PAD']:
                continue
            else:
                pred = y_pred_padded[i][j]
                true = y_true_padded[i][j]
                y_pred_idx_fold.append(pred)
                y_pred_idx.append(pred)
                y_test_idx_fold.append(true)
                y_test_idx.append(true)
                scores.append(pred == true)
                scores_fold.append(pred == true)

    prec_, rec_, fscore_, _ = precision_recall_fscore_support(y_test_idx_fold,
                                y_pred_idx_fold, average = 'weighted')
    print('[Fold ({}/{})] Accuracy: {}, Precision: {},' +
          ' Recall: {}, FScore: {}'.format(fold_count, kf.n_splits,
          np.asarray(scores_fold).mean(), prec_, rec_, fscore_))
    foldwise_score_outputs.append('[Fold ({}/{})] Accuracy: {}, Precision: {}, Recall: {},'
    + ' FScore: {}'.format(fold_count, kf.n_splits, np.asarray(scores_fold).mean(), prec_,
    rec_, fscore_))

prec, rec, fscore, _ = precision_recall_fscore_support(y_test_idx, y_pred_idx,
                       average = 'weighted')
print('Accuracy: {}, Precision: {}, Recall: {},' +
      ' FScore: {}'.format(np.asarray(scores).mean(), prec, rec, fscore))
```

Regular NER Dataset Score

```
Fold 1 Accuracy and Scores:
Accuracy: 0.9526610644257703, Precision: 0.9485268096309561, Recall: 0.9526610644257703,
FScore: 0.949593291838548

Fold 2 Accuracy and Scores:
Accuracy: 0.9534751773049646, Precision: 0.9505178345375412, Recall: 0.9534751773049646,
FScore: 0.9510871517860034

Fold 3 Accuracy and Scores:
Accuracy: 0.9556722076407116, Precision: 0.9493129500272377, Recall: 0.9556722076407116,
FScore: 0.9487352416691274

Fold 4 Accuracy and Scores:
Accuracy: 0.9653212052302445, Precision: 0.9605602782699517, Recall: 0.9653212052302445,
FScore: 0.9611318240196485

Fold 5 Accuracy and Scores:
Accuracy: 0.9520069808027923, Precision: 0.9422223141228804, Recall: 0.9520069808027923,
FScore: 0.9450074261739186
```

```
Overall Accuracy and Scores:
Accuracy: 0.9558352402745995, Precision: 0.9498458461158625, Recall: 0.9558352402745995,
FScore: 0.9514009972305221
```

NER 10-Types Dataset Score

```
Fold 1 Accuracy and Scores:
Accuracy: 0.9420168067226891, Precision: 0.9142776578972731, Recall: 0.9420168067226891,
FScore: 0.9272401706648893

Fold 2 Accuracy and Scores:
Accuracy: 0.9353191489361702, Precision: 0.9048582847298248, Recall: 0.9353191489361702,
FScore: 0.9173771743040816

Fold 3 Accuracy and Scores:
Accuracy: 0.9431321084864392, Precision: 0.910649828091966, Recall: 0.9431321084864392,
FScore: 0.9217824440455844

Fold 4 Accuracy and Scores:
Accuracy: 0.953951108584423, Precision: 0.9138141168930782, Recall: 0.953951108584423,
FScore: 0.9329121851868873

Fold 5 Accuracy and Scores:
Accuracy: 0.9476439790575916, Precision: 0.916906011125749, Recall: 0.9476439790575916,
FScore: 0.9269635245861122

Overall Accuracy and Scores:
Accuracy: 0.9443935926773456, Precision: 0.9111339327971488, Recall: 0.9443935926773456,
FScore: 0.9257149514228921
```

```
X_test_data = [np.asarray([a[0] for a in parsed_test_sentences]),
               np.asarray([a[1] for a in parsed_test_sentences])]
predictions_full = model.predict(X_test_data)

predictions_list = []
for i, s in enumerate(test_sentences):
    output = []
    for j, w in enumerate(s[0]):
        output.append(np.argmax(predictions_full[i][j]))
    predictions_list.append(output)
```

```
with open('NER-TestSet-10Types-RNN-Predictions.txt', 'w', encoding = 'utf-8') as f:
    for words, predictions in zip(test_sentences, predictions_list):
        assert(len(words[0]) == len(predictions))
        for word, prediction in zip(words[0], predictions):
```

```
            f.writelines(word + '\t' + idx2labels[prediction] + '\n')
        f.writelines('\n')
```

Additionally please refer *Q2/NER-TestSet-RNN-Predictions.txt* and *Q2/NER-TestSet-10Types-RNN-Predictions.txt* for the predicted outputs for the RNN model trained on the regular and 10-type datasets respectively.

For the notebook codes, please refer *Q2/Q2 - NER Prediction (RNN).ipynb* and *Q2/Q2 - NER Prediction - 10 Types (RNN).ipynb* respectively.