CS563 Natural Language Processing
Assignment: 3

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

May 14, 2020

1 Assignment Description

Designing and implementing a **Hidden Markov Model (HMM)** based model, a **Neural Network (NN)** based model and a **Bidirectional Long Short Term Memory (Bi-LSTM)** based model and applying them for **Part Of Speech (POS) Tagging** on the **Brown Corpus**.

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 *Assignment*3. Please note that the **dataset should be present in the assignment folder**. Use the following command to run the notebook:

```
jupyter notebook <Name of the IPython Notebook>
```

3 Discussion

The primary objective of the assignment is to implement a Hidden Markov Model (HMM) based model, a Neural Network (NN) based model and a Bidirectional Long Short Term Memory (Bi-LSTM) based model and applying them for Part Of Speech (POS) Tagging. 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, classification_report
from collections import defaultdict, Counter
import matplotlib
from matplotlib import pyplot as plt
import seaborn as sns

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

with open('Brown_train.txt', 'r') as f:
    pos_dataset = f.readlines()

sentences = []
words = []
tags = []

for line in pos_dataset:
    line = line.strip().split()
    if len(line) == 0:
        break
    for elem in line:
        elem = elem.rsplit('/', 1)
        words.append(elem[0])
        tags.append(elem[1])
    if len(words) > 0 and len(tags) > 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 3-fold cross validation sets. We obtain the accuracy, precision, recall and f-score on the individual folds. Finally, we obtain the tag-wise performance results as well.

```
# 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)

    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 3-fold cross validation along with accuracy, precision, recall and f-score metrics for overall and tag-wise performance. We further plot the confusion matrix and present tag-wise statistics.

```
# Build the test and training sets of sentences.
kf = KFold(n_splits = 3, 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))

id2tag = {v: k for k, v in tag2id.items()}
print(classification_report([id2tag[i] for i in y_test_idx],
                            [id2tag[i] for i in y_pred_idx]))
```

Overall Performance

```
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: 0.9230769230769231, Precision: 0.9444444444444445,
Recall: 0.9666666666666667, FScore: 0.9481481481481482
Fold 3 Accuracy and Scores:
Viterbi Accuracy: 0.7916666666666666, Precision: 0.7864583333333333,
Recall: 0.7083333333333333, FScore: 0.7116300366300365

Overall Accuracy and Scores:
Viterbi Accuracy: 0.8970098444441581, Precision: 0.81220514710333,
Recall: 0.8324100943805282, FScore: 0.817051032141999
```

Tag-Wise Performance

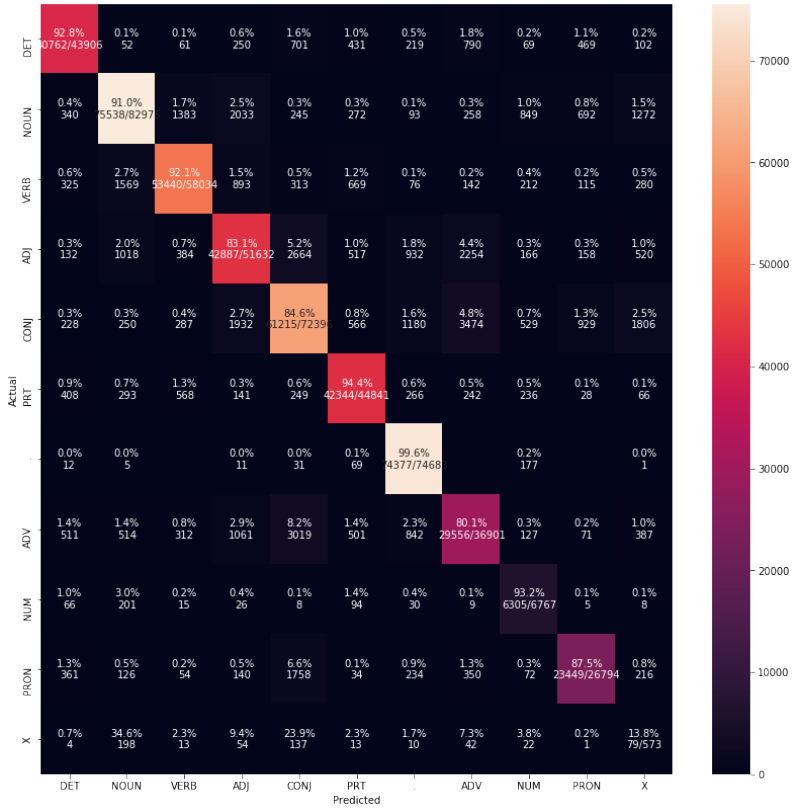
```
              precision    recall  f1-score   support

           .       0.95      0.99      0.97     74854
         ADJ       0.87      0.83      0.85     51942
         ADP       0.88      0.96      0.92     38784
         ADV       0.79      0.79      0.79     37582
        CONJ       0.87      0.83      0.85     73425
         DET       0.94      0.92      0.93     44196
        NOUN       0.95      0.91      0.93     83364
         NUM       0.72      0.93      0.81      6795
        PRON       0.90      0.87      0.88     27098
         PRT       0.92      0.92      0.92     46106
```

```
        VERB       0.94      0.91      0.93     58413
           X       0.02      0.13      0.03       590

    accuracy                           0.90    543149
   macro avg       0.81      0.83      0.82    543149
weighted avg       0.90      0.90      0.90    543149
```

```
def cm_analysis(y_true, y_pred, labels, ymap=None, figsize=(10,10)):
    """
    Generate matrix plot of confusion matrix with pretty annotations.
    The plot image is saved to disk.
    args:
      y_true:    true label of the data, with shape (nsamples,)
      y_pred:    prediction of the data, with shape (nsamples,)
      filename:  filename of figure file to save
      labels:    string array, name the order of class labels in the confusion matrix.
                 use `clf.classes_` if using scikit-learn models.
                 with shape (nclass,).
      ymap:      dict: any -> string, length == nclass.
                 if not None, map the labels & ys to more understandable strings.
                 Caution: original y_true, y_pred and labels must align.
      figsize:   the size of the figure plotted.
    """
    if ymap is not None:
        y_pred = [ymap[yi] for yi in y_pred]
        y_true = [ymap[yi] for yi in y_true]
        labels = [ymap[yi] for yi in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    cm_sum = np.sum(cm, axis=1, keepdims=True)
    cm_perc = cm / cm_sum.astype(float) * 100
    annot = np.empty_like(cm).astype(str)
    nrows, ncols = cm.shape
    for i in range(nrows):
        for j in range(ncols):
            c = cm[i, j]
            p = cm_perc[i, j]
            if i == j:
                s = cm_sum[i]
                annot[i, j] = '%.1f%%\n%d/%d' % (p, c, s)
            elif c == 0:
                annot[i, j] = ''
            else:
                annot[i, j] = '%.1f%%\n%d' % (p, c)
    cm = pd.DataFrame(cm, index=labels, columns=labels)
    cm.index.name = 'Actual'
    cm.columns.name = 'Predicted'
    fig, ax = plt.subplots(figsize=figsize)
    sns.heatmap(cm, annot=annot, fmt='', ax=ax)
```

```
    plt.show()

cm_analysis([id2tag[i] for i in y_test_idx], [id2tag[i] for i in y_pred_idx],
            sorted(tag2id.keys(), key=lambda k: tag2id[k])[1:], ymap=None, figsize=(14, 14))
```

Confusion Matrix




```
print("Tag-wise counts and accuracies\n--\n")
print("Tag\tGold TagCount\tTotalPredicted Correctly\tTotal predicted incorrectly\tAccuracy")
gold_counts = Counter([id2tag[i] for i in y_test_idx])
pred_counts_correct = Counter([id2tag[t] for i, t in enumerate(y_pred_idx)
                               if t == y_test_idx[i]])
pred_counts_incorrect = Counter([id2tag[t] for i, t in enumerate(y_pred_idx)
                                 if t != y_test_idx[i]])
for tag in gold_counts.keys():
  print(f"{tag}\t{gold_counts[tag]}\t\t{pred_counts_correct[tag]}\t\t\t\t"
        + "{pred_counts_incorrect[tag]}\t\t\t\t{pred_counts_correct[tag]/gold_counts[tag]}")
```

```
Tag-wise counts and accuracies
--
```

Tag	Gold Tag Count	Total Correct	Total Incorrect	Accuracy
PRT	46106	42344	3762	0.918405
.	74854	74377	3908	0.993627
CONJ	73425	61215	9171	0.833707
ADJ	51942	42887	6579	0.825670
ADV	37582	29556	8012	0.786440
ADP	38784	37258	4863	0.960653
VERB	58413	53440	3138	0.914864
NOUN	83364	75538	4290	0.906122
DET	44196	40762	2535	0.922300
PRON	27098	23449	2537	0.865340
NUM	6795	6305	2467	0.927888
X	590	79	4677	0.133898

For the notebook codes, please refer *Q1/POS_HMM.ipynb*.

Notebook Code - Neural Network

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
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.metrics import confusion_matrix, classification_report
from collections import defaultdict, Counter
import matplotlib
from matplotlib import pyplot as plt
import seaborn as sns

from keras.models import Sequential
from keras.layers import Dense, LSTM, Conv1D, InputLayer, Bidirectional, TimeDistributed,
Embedding, Activation, Masking, Flatten
from keras.optimizers import Adam, SGD
from keras.preprocessing.sequence import pad_sequences

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

def parse_sentence(sentence):
    '''
```

```
    Function for parsing the words and tags from the
    sentences of the input corpus.
    '''
    word_tag_pairs = sentence.split(" ")
    words = []
    tags = []

    for i, word_tag in enumerate(word_tag_pairs):
        word, tag = word_tag.strip().rsplit('/', 1)
        words.append(word)
        tags.append(tag)

    return words, tags

# Parse the sentences into a list.
parsed_sentences = []

with open('./Brown_train.txt', 'r') as file:
    sentences = file.readlines()

    for sentence in sentences:
        sentence = sentence.strip()
        parsed_sentences.append(parse_sentence(sentence))
```

```
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['PAD'] = 0
    vocabulary2id['UNK'] = 1

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

    tag2id['PAD'] = 0
    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
```

Neural Network Code

We implement a standard Neural Network Model. We split the overall training dataset into 3-fold cross validation sets. We obtain the accuracy, precision, recall and f-score on the individual folds. Finally, we obtain the tag-wise performance results as well.

```
def build_model():
    model = Sequential()
    model.add(InputLayer(input_shape=(5, )))
    model.add(Embedding(len(vocabulary2id), 100))
    model.add(Flatten())
    model.add(Dense(len(tag2id)))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy',
                  optimizer=Adam(0.001),
                  metrics=['accuracy'])
    model.summary()
    return model

def id2onehot(Y, numtags):
    out = []
```

```
    for s in Y:
        out.append(np.zeros(numtags))
        out[-1][s] = 1.0
    return np.array(out)

def make_example(words, vocabulary2id):
    words_new = ['PAD', 'PAD'] + words + ['PAD', 'PAD']
    examples = []
    for i in range(len(words)):
        context_words = words_new[i: i + 5]
        context_word_idx = [vocabulary2id[w] if w in vocabulary2id.keys() else
                            vocabulary2id['UNK'] for w in context_words]
        examples.append(context_word_idx)

    return examples
```

Testing and Analysis

We perform 3-fold cross validation along with accuracy, precision, recall and f-score metrics for overall and tag-wise performance. We further plot the confusion matrix and present tag-wise statistics.

```
# Build the test and training sets of sentences.
kf = KFold(n_splits = 3, shuffle = False)
parsed_sentences = np.asarray(parsed_sentences)
scores = []
scores1 = []
y_pred_idx = []
y_pred_idx1 = []
y_test_idx = []
y_test_idx1 = []

preds_all_folds = []
golds_all_folds = []

for fold_num, (train_index, test_index) in enumerate(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)

    X_train_ids = []
    Y_train_ids = []
    for x_sent, y_sent in zip(X_train, Y_train):
```

```
        X_train_ids.extend(make_example(x_sent, vocabulary2id))
        Y_train_ids.extend([tag2id[word] if word in tag2id.keys() else tag2id['UNK']
                            for word in y_sent])

    X_test_ids = []
    Y_test_ids = []
    for x_sent, y_sent in zip(X_test, Y_test):
        X_test_ids.extend(make_example(x_sent, vocabulary2id))
        Y_test_ids.extend([tag2id[word] if word in tag2id.keys() else tag2id['UNK']
                           for word in y_sent])

    X_train_ids = np.asarray(X_train_ids)
    X_test_ids = np.asarray(X_test_ids)

    Y_train_onehot = id2onehot(Y_train_ids, len(tag2id))
    Y_test_onehot = id2onehot(Y_test_ids, len(tag2id))

    model = build_model()
    model.fit(X_train_ids, Y_train_onehot, batch_size=128, epochs=5, validation_split=0.2)

    predictions = model.predict(X_test_ids)

    predictions_argmax = np.argmax(predictions, axis=-1)

    y_pred_nopad = predictions_argmax[:]
    y_true_nopad = Y_test_ids[:]

    preds_all_folds.append(y_pred_nopad)
    golds_all_folds.append(y_true_nopad)

    y_pred_nopad = np.asarray(y_pred_nopad)
    y_true_nopad = np.asarray(y_true_nopad)
    test_accuracy = (y_pred_nopad == y_true_nopad).mean()
    print('Fold {} test_accuracy: {}'.format(fold_num + 1, test_accuracy))
    prec, rec, fscore, _ = precision_recall_fscore_support(y_true_nopad, y_pred_nopad,
                                                           average = 'weighted')
    print('Fold {} Precision: {} Recall: {} F1-Score: {}'.format(fold_num + 1, prec, rec,
                                                                 fscore))

print("---Averaged Results over all the epochs---")
test_accuracy = (np.asarray(preds_all_folds[0]) == np.asarray(golds_all_folds[0])).mean()
print('Average K-Fold Test Accuracy: {}'.format(test_accuracy))
prec, rec, fscore, _ = precision_recall_fscore_support(preds_all_folds[0],
                                                       golds_all_folds[0],
                                                       average = 'weighted')
print('Average K-Fold Precision: {} Recall: {} F1-Score: {}'.format(prec, rec, fscore))

id2tag = {v: k for k, v in tag2id.items()}
```

```
print(classification_report([id2tag[i] for i in golds_all_folds[0]], [id2tag[i]
                            for i in preds_all_folds[0]]))
```

Overall Performance

```
Fold 1 Accuracy: 0.9175521046178995
Fold 1 Precision: 0.9175708490712462 Recall: 0.9175521046178995 F1-Score: 0.9162935613787996
Fold 2 Accuracy: 0.9538182542571972
Fold 2 Precision: 0.9539599018345375 Recall: 0.9538182542571972 F1-Score: 0.9538492424867576
Fold 3 Accuracy: 0.9489376733476372
Fold 3 Precision: 0.9485933876952576 Recall: 0.9489376733476372 F1-Score: 0.9486331633309958
--------Averaged Results---------
K-Fold Test Accuracy: 0.9175521046178995
K-Fold Precision: 0.9219477576557498 Recall: 0.9175521046178995 F1-Score: 0.9188106478569993
```

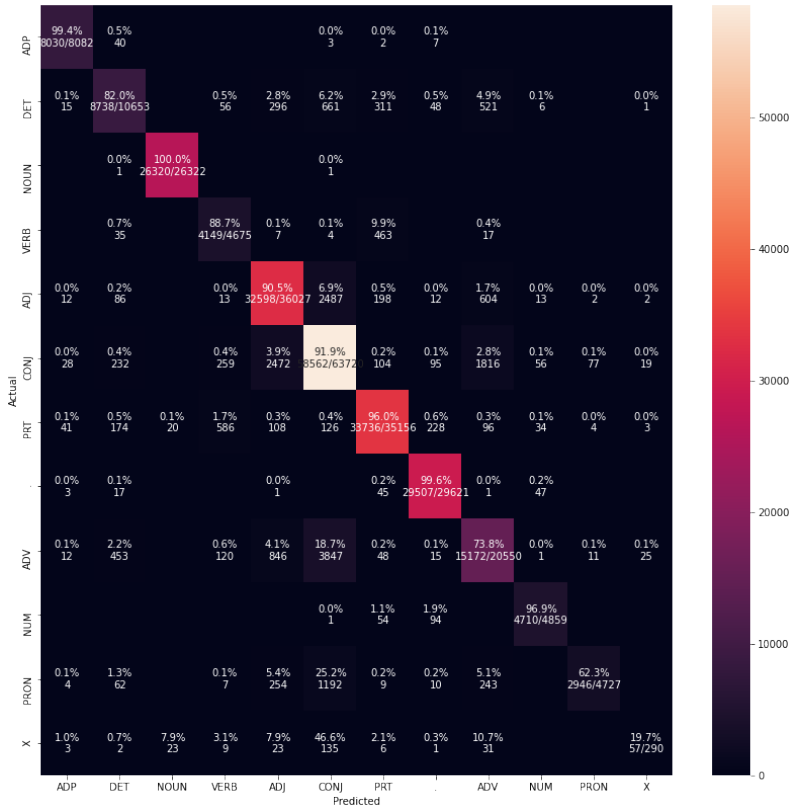
Tag-Wise Performance

```
              precision  recall   f1-score   support

           .       0.98      1.00      0.99     29621
         ADJ       0.89      0.90      0.90     36028
         ADP       0.99      0.99      0.99      8082
         ADV       0.82      0.74      0.78     20552
        CONJ       0.87      0.92      0.90     63728
         DET       0.89      0.82      0.85     10653
        NOUN       1.00      1.00      1.00     26322
         NUM       0.97      0.97      0.97      4859
         PAD       0.00      0.00      0.00         0
        PRON       0.97      0.62      0.76      4728
         PRT       0.96      0.96      0.96     35159
        VERB       0.80      0.89      0.84      4675
           X       0.53      0.19      0.29       293

    accuracy                           0.92    244700
   macro avg       0.82      0.77      0.79    244700
weighted avg       0.92      0.92      0.92    244700
```

```
def cm_analysis(y_true, y_pred, labels, ymap=None, figsize=(10,10)):
    """
    Generate matrix plot of confusion matrix with pretty annotations.
    The plot image is saved to disk.
    args:
      y_true:    true label of the data, with shape (nsamples,)
      y_pred:    prediction of the data, with shape (nsamples,)
      filename:  filename of figure file to save
      labels:    string array, name the order of class labels in the confusion matrix.
                 use `clf.classes_` if using scikit-learn models.
```

```
                with shape (nclass,).
    ymap:       dict: any -> string, length == nclass.
                if not None, map the labels & ys to more understandable strings.
                Caution: original y_true, y_pred and labels must align.
    figsize:    the size of the figure plotted.
    """
    if ymap is not None:
        y_pred = [ymap[yi] for yi in y_pred]
        y_true = [ymap[yi] for yi in y_true]
        labels = [ymap[yi] for yi in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    cm_sum = np.sum(cm, axis=1, keepdims=True)
    cm_perc = cm / cm_sum.astype(float) * 100
    annot = np.empty_like(cm).astype(str)
    nrows, ncols = cm.shape
    for i in range(nrows):
        for j in range(ncols):
            c = cm[i, j]
            p = cm_perc[i, j]
            if i == j:
                s = cm_sum[i]
                annot[i, j] = '%.1f%%\n%d/%d' % (p, c, s)
            elif c == 0:
                annot[i, j] = ''
            else:
                annot[i, j] = '%.1f%%\n%d' % (p, c)
    cm = pd.DataFrame(cm, index=labels, columns=labels)
    cm.index.name = 'Actual'
    cm.columns.name = 'Predicted'
    fig, ax = plt.subplots(figsize=figsize)
    sns.heatmap(cm, annot=annot, fmt='', ax=ax)
    plt.show()

cm_analysis([id2tag[i] for i in golds_all_folds[0]],[id2tag[i] for i in preds_all_folds[0]],
             sorted(tag2id.keys(), key=lambda k: tag2id[k])[1:], ymap=None, figsize=(14,14))
```

Confusion Matrix




```
print("Tag-wise counts and accuracies\n--\n")
print("Tag\tGold Tag Count\tTotal Correct\tTotal Incorrect\tAccuracy")
gold_counts = Counter([id2tag[i] for i in golds_all_folds[0]])
pred_counts_correct = Counter([id2tag[t] for i, t in enumerate(preds_all_folds[0])
                               if t == golds_all_folds[0][i]])
pred_counts_incorrect = Counter([id2tag[t] for i, t in enumerate(preds_all_folds[0])
                                 if t != golds_all_folds[0][i]])
for tag in gold_counts.keys():
  print(f"{tag}\t{gold_counts[tag]}\t\t{pred_counts_correct[tag]}\t\t\t\t"
        + "{pred_counts_incorrect[tag]}\t\t\t\t{pred_counts_correct[tag]/gold_counts[tag]}")
```

```
Tag-wise counts and accuracies
--

Tag    Gold Tag Count    Total Correct    Total Incorrect    Accuracy
PRT    35159             33736            1240               0.959526
.      29621             29507            510                0.996151
CONJ   63728             58562            8457               0.918936
ADJ    36028             32598            4007               0.904796
ADV    20552             15172            3329               0.738224
```

ADP	8082	8030	118	0.993565
VERB	4675	4149	1050	0.887486
NOUN	26322	26320	43	0.999924
DET	10653	8738	1102	0.820238
PRON	4728	2946	94	0.623096
NUM	4859	4710	157	0.969335
X	293	57	50	0.194539

For the notebook codes, please refer *Q1/POS_NN.ipynb*.

Notebook Code - BiLSTM

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
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.metrics import confusion_matrix, classification_report
from collections import defaultdict, Counter
import matplotlib
from matplotlib import pyplot as plt
import seaborn as sns

from keras.models import Sequential
from keras.layers import Dense, LSTM, Conv1D, InputLayer, Bidirectional, TimeDistributed,
Embedding, Activation, Masking, Flatten
from keras.optimizers import Adam, SGD
from keras.preprocessing.sequence import pad_sequences

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

def parse_sentence(sentence):
    '''
    Function for parsing the words and tags from the
    sentences of the input corpus.
    '''
    word_tag_pairs = sentence.split(" ")
    words = []
    tags = []
```

```
    for i, word_tag in enumerate(word_tag_pairs):
        word, tag = word_tag.strip().rsplit('/', 1)
        words.append(word)
        tags.append(tag)

    return words, tags

# Parse the sentences into a list.
parsed_sentences = []

with open('./Brown_train.txt', 'r') as file:
    sentences = file.readlines()

    for sentence in sentences:
        sentence = sentence.strip()
        parsed_sentences.append(parse_sentence(sentence))
```

```
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['PAD'] = 0
    vocabulary2id['UNK'] = 1

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

    tag2id['PAD'] = 0
    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
```

Bi-LSTM Code

We implement a standard Bi-Directional LSTM Model. We split the overall training dataset into 3-fold cross validation sets. We obtain the accuracy, precision, recall and f-score on the individual folds. Finally, we obtain the tag-wise performance results as well.

```
def build_model():
    model = Sequential()
    model.add(Masking(mask_value=float(vocabulary2id['UNK']),input_shape=(padlen,)))
    model.add(Embedding(len(vocabulary2id), 100))
    model.add(Bidirectional(LSTM(int((128+256)/2), return_sequences=True)))
    model.add(TimeDistributed(Dense(len(tag2id))))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy',
                  optimizer=Adam(0.001),
                  metrics=['accuracy'])
    model.summary()
    return model

def id2onehot(Y, numtags):
    out = []
    for s in Y:
        out.append(np.zeros(numtags))
        out[-1][s] = 1.0
    return np.array(out)
```

Testing and Analysis

We perform 3-fold cross validation along with accuracy, precision, recall and f-score metrics for overall and tag-wise performance. We further plot the confusion matrix and present tag-wise statistics.

```
# Build the test and training sets of sentences.
kf = KFold(n_splits = 3, shuffle = False)
parsed_sentences = np.asarray(parsed_sentences)
scores = []
scores1 = []
y_pred_idx = []
y_pred_idx1 = []
y_test_idx = []
y_test_idx1 = []

preds_all_folds = []
golds_all_folds = []

for fold_num, (train_index, test_index) in enumerate(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)

    X_train_ids = []
    Y_train_ids = []
    for x_sent, y_sent in zip(X_train, Y_train):
        X_train_ids.extend(make_example(x_sent, vocabulary2id))
        Y_train_ids.extend([tag2id[word] if word in tag2id.keys() else tag2id['UNK']
                            for word in y_sent])

    X_test_ids = []
    Y_test_ids = []
    for x_sent, y_sent in zip(X_test, Y_test):
        X_test_ids.extend(make_example(x_sent, vocabulary2id))
        Y_test_ids.extend([tag2id[word] if word in tag2id.keys() else tag2id['UNK']
                           for word in y_sent])

    X_train_ids = np.asarray(X_train_ids)
    X_test_ids = np.asarray(X_test_ids)

    Y_train_onehot = id2onehot(Y_train_ids, len(tag2id))
    Y_test_onehot = id2onehot(Y_test_ids, len(tag2id))
```

```
    model = build_model()
    model.fit(X_train_ids, Y_train_onehot, batch_size=128, epochs=5, validation_split=0.2)

    predictions = model.predict(X_test_ids)

    predictions_argmax = np.argmax(predictions, axis=-1)

    y_pred_nopad = predictions_argmax[:]
    y_true_nopad = Y_test_ids[:]

    preds_all_folds.append(y_pred_nopad)
    golds_all_folds.append(y_true_nopad)

    y_pred_nopad = np.asarray(y_pred_nopad)
    y_true_nopad = np.asarray(y_true_nopad)
    test_accuracy = (y_pred_nopad == y_true_nopad).mean()
    print('Fold {} test_accuracy: {}'.format(fold_num + 1, test_accuracy))
    prec, rec, fscore, _ = precision_recall_fscore_support(y_true_nopad, y_pred_nopad,
                                                           average = 'weighted')
    print('Fold {} Precision: {} Recall: {} F1-Score: {}'.format(fold_num + 1, prec, rec,
                                                                 fscore))

print("---Averaged Results over all the epochs---")
test_accuracy = (np.asarray(preds_all_folds[0]) == np.asarray(golds_all_folds[0])).mean()
print('Average K-Fold Test Accuracy: {}'.format(test_accuracy))
prec, rec, fscore, _ = precision_recall_fscore_support(preds_all_folds[0],
                                                       golds_all_folds[0],
                                                       average = 'weighted')
print('Average K-Fold Precision: {} Recall: {} F1-Score: {}'.format(prec, rec, fscore))

id2tag = {v: k for k, v in tag2id.items()}
print(classification_report([id2tag[i] for i in golds_all_folds[0]], [id2tag[i]
                             for i in preds_all_folds[0]]))
```

Overall Performance

```
Fold 1 Accuracy: 0.9010598534402203
Fold 1 Precision: 0.9030468569039867 Recall: 0.9010598534402203 F1-Score: 0.8959284425488834
Fold 2 Accuracy: 0.9260635703539853
Fold 2 Precision: 0.9271398145592178 Recall: 0.9260635703539853 F1-Score: 0.9247742034060448
Fold 3 Accuracy: 0.9274794429144289
Fold 3 Precision: 0.9270271787931963 Recall: 0.9274794429144289 F1-Score: 0.9254782905931985
--------Averaged Results---------
K-Fold Test Accuracy: 0.9010598534402203
K-Fold Precision: 0.9175752724543983 Recall: 0.9010598534402203 F1-Score: 0.9061912643315573
```

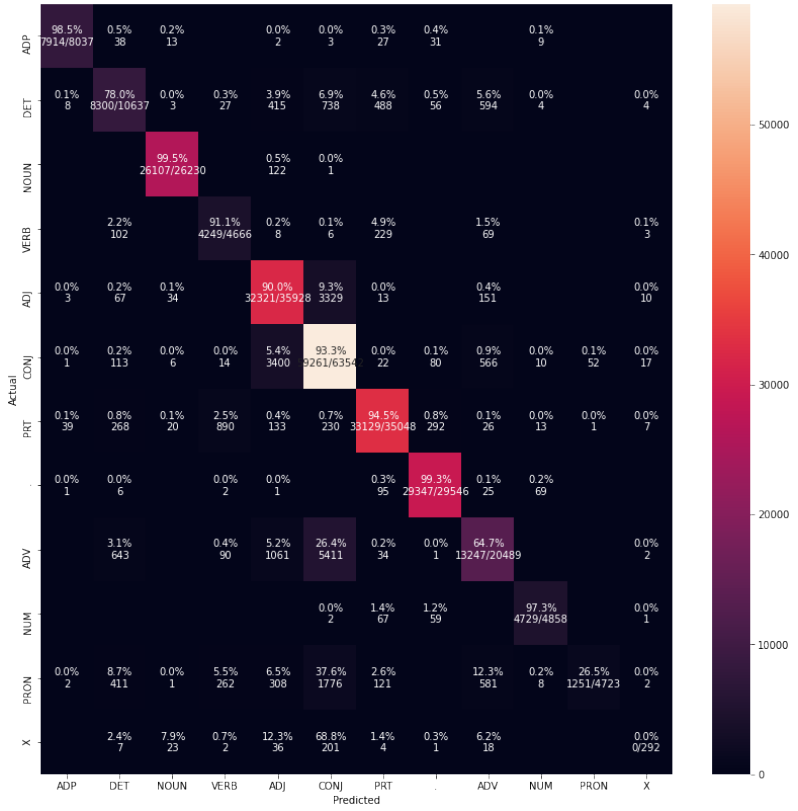
Tag-Wise Performance

	precision	recall	f1-score	support
.	0.98	0.99	0.99	29546
ADJ	0.85	0.90	0.88	35928
ADP	0.99	0.98	0.99	8037
ADV	0.87	0.65	0.74	20489
CONJ	0.84	0.93	0.88	63542
DET	0.83	0.78	0.81	10637
NOUN	1.00	1.00	1.00	26230
NUM	0.98	0.97	0.98	4858
PRON	0.96	0.26	0.42	4723
PRT	0.97	0.95	0.96	35048
VERB	0.77	0.91	0.83	4666
X	0.00	0.00	0.00	292
accuracy			0.90	243996
macro avg	0.84	0.78	0.79	243996
weighted avg	0.90	0.90	0.90	243996

```
def cm_analysis(y_true, y_pred, labels, ymap=None, figsize=(10,10)):
    """
    Generate matrix plot of confusion matrix with pretty annotations.
    The plot image is saved to disk.
    args:
      y_true:    true label of the data, with shape (nsamples,)
      y_pred:    prediction of the data, with shape (nsamples,)
      filename:  filename of figure file to save
      labels:    string array, name the order of class labels in the confusion matrix.
                 use `clf.classes_` if using scikit-learn models.
                 with shape (nclass,).
      ymap:      dict: any -> string, length == nclass.
                 if not None, map the labels & ys to more understandable strings.
                 Caution: original y_true, y_pred and labels must align.
      figsize:   the size of the figure plotted.
    """
    if ymap is not None:
        y_pred = [ymap[yi] for yi in y_pred]
        y_true = [ymap[yi] for yi in y_true]
        labels = [ymap[yi] for yi in labels]
    cm = confusion_matrix(y_true, y_pred, labels=labels)
    cm_sum = np.sum(cm, axis=1, keepdims=True)
    cm_perc = cm / cm_sum.astype(float) * 100
    annot = np.empty_like(cm).astype(str)
    nrows, ncols = cm.shape
    for i in range(nrows):
        for j in range(ncols):
```

```
            c = cm[i, j]
            p = cm_perc[i, j]
            if i == j:
                s = cm_sum[i]
                annot[i, j] = '%.1f%%\n%d/%d' % (p, c, s)
            elif c == 0:
                annot[i, j] = ''
            else:
                annot[i, j] = '%.1f%%\n%d' % (p, c)
    cm = pd.DataFrame(cm, index=labels, columns=labels)
    cm.index.name = 'Actual'
    cm.columns.name = 'Predicted'
    fig, ax = plt.subplots(figsize=figsize)
    sns.heatmap(cm, annot=annot, fmt='', ax=ax)
    plt.show()

cm_analysis([id2tag[i] for i in golds_all_folds[0]],[id2tag[i] for i in preds_all_folds[0]],
             sorted(tag2id.keys(), key=lambda k: tag2id[k])[1:], ymap=None, figsize=(14,14))
```

Confusion Matrix




```
print("Tag-wise counts and accuracies\n--\n")
```

```
print("Tag\tGold Tag Count\tTotal Correct\tTotal Incorrect\tAccuracy")
gold_counts = Counter([id2tag[i] for i in golds_all_folds[0]])
pred_counts_correct = Counter([id2tag[t] for i, t in enumerate(preds_all_folds[0])
                               if t == golds_all_folds[0][i]])
pred_counts_incorrect = Counter([id2tag[t] for i, t in enumerate(preds_all_folds[0])
                                 if t != golds_all_folds[0][i]])
for tag in gold_counts.keys():
  print(f"{tag}\t{gold_counts[tag]}\t\t{pred_counts_correct[tag]}\t\t\t\t"
        + "{pred_counts_incorrect[tag]}\t\t\t\t{pred_counts_correct[tag]/gold_counts[tag]}")
```

```
Tag-wise counts and accuracies
--

Tag   Gold Tag Count   Total Correct   Total Incorrect   Accuracy
PRT   35048            33129           1100              0.945246
.     29546            29347           520               0.993264
CONJ  63542            59261           11697             0.932627
ADJ   35928            32321           5486              0.899604
ADV   20489            13247           2030              0.646542
ADP   8037             7914            54                0.984695
VERB  4666             4249            1287              0.910630
NOUN  26230            26107           100               0.995310
DET   10637            8300            1655              0.780295
PRON  4723             1251            53                0.264874
NUM   4858             4729            113               0.973445
X     292              0               46                0.0
```

For the notebook codes, please refer *Q1/POS_BiLSTM.ipynb*.