Viterbi.py

```
import argparse
from collections import Counter
from collections import defaultdict
import multiprocessing
import re
from sklearn.metrics import accuracy_score


class POS_Holder():
    def __init__(self, tags, c_val):
        self.tags = tags
        self.c_val = c_val

    def get_last_tag(self):
        return self.tags[-1]

    def get_c_val(self):
        return self.c_val

    def get_params(self, tag, mval, adjust_by=0):
        """
        Returns parameters to copy current state
        Useful for one to many transitions as well as many to one
transitions

        :param tag: selected tag to be added to history
        :param mval: current max value after consideration of all
possible tags
        :return: parameters to be forwarded to next state
        """
        t = self.tags.copy()
        t.append(tag)
        return t, mval+adjust_by


class UtilityFunctions():
    def __init__(self, filename):
        self.filename = filename
        with open(self.filename, "r") as f:
            self.myfile = f.read()

        self.pos = self.process_lines()

        self.bigrams = self.get_bigrams()

        self.pos_count = Counter(self.pos)

        self.tag_emmission = Counter(self.bigrams)
```

```
        self.lpos = len(self.pos)

        self.tags = [i[1] for i in self.pos]
        self.tag_count = Counter(self.tags)
        self.words = [i[0] for i in self.pos]
        self.word_count = Counter(self.words)

        self.tag_emmission_counts = Counter(self.pos)

        self.existing_tag_set = self.get_existing_tag_set()

    def get_tag_list(self, words):
        """
        Returns the list of tags for a given list of words
        :param words: list of words
        :return: list of tags
        """
        # return [self.get_tag(word) for word in words]
        pass

    def default_policy(self, word):
        """
        General policy is that if the first char is uppercase make
it a NP otherwise NN.
        We can change it in multiple ways but since most of the
unknown might be NP or NN
        it would be fine.
        """
        if(word[0].isupper()):
            default = "NP"
        else:
            default = "NN"

        mval = 0
        return {default}

    def get_tag(self, word):
        """
        Returns all possible tags for a given word
        To add complicated default rules, change default policy.py

        :param word: word
        :return: all possible tags for a given word
        """
        if(word in self.existing_tag_set):
            return self.existing_tag_set[word]
        else:
            return self.default_policy(word)

    def get_tag_emmission_prob(self, word, tag):
        """
```

```
        Calculates the tag emmission probability of a word given a
tag

        :param word: word
        :param tag: tag
        :return: tag emmission probability
        """
        return (self.tag_emmission_counts[(word, tag)]+1)/
(self.tag_count[tag]+len(self.existing_tag_set))
        # return (self.tag_emmission_counts[(word, tag)] + 1)/
(self.word_count[word]+len(self.existing_tag_set))

    def get_existing_tag_set(self):
        """
        Returns the set of all tags for each word in pos
        """
        d = dict()
        for i in self.pos:
            if(i[0] in d):
                d[i[0]].add(i[1])
            else:
                d[i[0]] = set({i[1]})
        return d

    def get_tag_transition_prob(self, tag1, tag2):
        """
        Returns transition probability from tag1 to tag2

        :param tag1: tag1
        :param tag2: tag2
        :return: transition probability
        """
        val = (self.tag_emmission[(tag1, tag2)] /
(self.tag_count[tag1]))
        return val

    def get_bigrams(self):
        """
        Returns bigram
        """
        lis = []
        for i in range(1, len(self.pos)):
            lis.append((self.pos[i-1][1], self.pos[i][1]))

        return lis

    def process_lines(self):
        """
        Processes the lines of the file
        :return pos: list of word,pos
        """
        lines = self.myfile.split("\n")
```

```
        lines = [self.add_start_end(line) for line in lines]
        pos = [self.find_tags(i) for line in lines for i in
line.split(" ")]
        return pos

    def add_start_end(self, line):
        """
        Adds a linestartshere/START token at the beginning of the
sentence
        and lineendshere/END token at the end of sentence
        """
        line = "linestartshere/START "+line+" lineendshere/END"
        return line

    def find_tags(self, rawword):
        """
        Finds the word,tags in the raw word

        :param rawword: raw word
        :return: word,tags
        """
        pattern = "(.*)\/([A-Z,.\$:#)('`\.,\|]+)"
        try:
            return re.findall(pattern, rawword)[0]
        except:
            # In case parsing fails
            return "SOMEWORD", "NN"


class Runner():
    def __init__(self, utility_functions):
        self.utility_functions = utility_functions

    def make_one_transition(self, tag1, tag2, word, pval=1):
        """
        Simple, previously calculated max value * transition *
emmission

        :params tag1,tag2,word,pval:
        :returns value:
        """
        return pval *
self.utility_functions.get_tag_emmission_prob(word, tag2) *
self.utility_functions.get_tag_transition_prob(tag1, tag2)
        # return pval *
self.utility_functions.get_tag_emmission_prob(word, tag2)

    def step(self, next_word, plis=[POS_Holder(["START"], 1)]):
        """
        Single step of transition
        """
        next_dict = dict()
```

```
        # Calculate possisble tags for next word
        word_tag_set = self.utility_functions.get_tag(next_word)

        # Calculate score for all tags belonging to a word from
all previous tags
        # Previous tags are there in plis, e.g. START for first
word
        # It goes something like this for tag belongs to
next_word, check against
        # all previous tags and keep max in mval and keep
reference in ref.
        # then add ref to next_dict for this current tag
        for next_tag in word_tag_set:
            mval = 0
            ref = plis[0]
            for i in plis:
                calc = self.make_one_transition(
                    i.get_last_tag(), next_tag, next_word,
pval=i.get_c_val())
                # print(calc)
                if(calc >= mval):
                    mval = calc
                    ref = i
                else:
                    # print(calc)
                    pass

            params = ref.get_params(next_tag, mval)
            # print("Choosen Tag: ", params[0][-2],
            #       " ", params[0][-1], " : ", mval)
            next_dict[next_tag] = POS_Holder(params[0], params[1])

        # Finally return list of values because while developing
        # I made it like this XD since assuming that we are
working with lists is
        # easier than working with dictionaries... :P
        return list(next_dict.values())

    def run(self, my_test_str="Pierre Vinken , 61 years old , will
join the board as a nonexecutive director Nov."):
        my_test_str = my_test_str.split(" ")
        plis = self.step(my_test_str[0])
        for i in range(1, len(my_test_str)):
            plis = self.step(my_test_str[i], plis)

        # Calculate Path with MAX PROBABILITY
        mmax = 0
        for i in plis:
            if(i.get_c_val() >= mmax):
                ref = i
                mmax = i.get_c_val()
```

```
        return ref


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Parts of Speech
Tagger")
    parser.add_argument("train", help="Training file")
    parser.add_argument("test", help="Test file")

    args = parser.parse_args()

    # train_file = "POS.train"
    train_file = args.train
    test_file = args.test

    # TRAINING PHASE
    u = UtilityFunctions(train_file)

    # Initalize Runner
    runner = Runner(u)

    # TESTING PHASE

    with open(test_file, "r") as f:
        test_str = f.read()

    test_str = test_str.split("\n")

    z = []
    for i in range(len(test_str)):
        z.append(test_str[i].split(" "))

    a = []
    for lis in z:
        n = []
        for rawword in lis:
            n.append(u.find_tags(rawword))
        unzipped_list = list(zip(*n))
        a.append(unzipped_list)

    # Just testing...
    # ref = runner.run(
    #     my_test_str="Hey , I am Sarvesh Bhatnagar .")
    # print(ref.tags)

    # Calculate accuracy sentence by sentence
    tot_acc = 0
    for i in range(len(a)):
        ref = runner.run(my_test_str=" ".join(a[i][0]))
        tot_acc += accuracy_score(list(a[i][1]), ref.tags[1:])
    print(tot_acc/len(a))
```

Baseline.py

```
import argparse
from collections import Counter
from collections import defaultdict
import multiprocessing
import re
from sklearn.metrics import accuracy_score


class POS_Holder():
    def __init__(self, tags, c_val):
        self.tags = tags
        self.c_val = c_val

    def get_last_tag(self):
        return self.tags[-1]

    def get_c_val(self):
        return self.c_val

    def get_params(self, tag, mval, adjust_by=0):
        """
        Returns parameters to copy current state
        Useful for one to many transitions as well as many to one
transitions

        :param tag: selected tag to be added to history
        :param mval: current max value after consideration of all
possible tags
        :return: parameters to be forwarded to next state
        """
        t = self.tags.copy()
        t.append(tag)
        return t, mval+adjust_by


class UtilityFunctions():
    def __init__(self, filename):
        self.filename = filename
        with open(self.filename, "r") as f:
            self.myfile = f.read()

        self.pos = self.process_lines()

        self.bigrams = self.get_bigrams()

        self.pos_count = Counter(self.pos)
```

```
        self.tag_emmission = Counter(self.bigrams)
        self.lpos = len(self.pos)

        self.tags = [i[1] for i in self.pos]
        self.tag_count = Counter(self.tags)
        self.words = [i[0] for i in self.pos]
        self.word_count = Counter(self.words)

        self.tag_emmission_counts = Counter(self.pos)

        self.existing_tag_set = self.get_existing_tag_set()

    def get_tag_list(self, words):
        """
        Returns the list of tags for a given list of words
        :param words: list of words
        :return: list of tags
        """
        # return [self.get_tag(word) for word in words]
        pass

    def default_policy(self, word):
        """
        General policy is that if the first char is uppercase make
it a NP otherwise NN.
        We can change it in multiple ways but since most of the
unknown might be NP or NN
        it would be fine.
        """
        if(word[0].isupper()):
            default = "NN"
        else:
            default = "NN"

        mval = 0
        return {default}

    def get_tag(self, word):
        """
        Returns all possible tags for a given word
        To add complicated default rules, change default policy.py

        :param word: word
        :return: all possible tags for a given word
        """
        if(word in self.existing_tag_set):
            return self.existing_tag_set[word]
        else:
            return self.default_policy(word)

    def get_tag_emmission_prob(self, word, tag):
        """
```

```
        Calculates the tag emmission probability of a word given a
tag

        :param word: word
        :param tag: tag
        :return: tag emmission probability
        """
        # NOTE made changes here.
        # return (self.tag_emmission_counts[(word, tag)]+1)/
(self.tag_count[tag]+len(self.existing_tag_set))
        return (self.tag_emmission_counts[(word, tag)] + 1)/
(self.word_count[word]+len(self.existing_tag_set))

    def get_existing_tag_set(self):
        """
        Returns the set of all tags for each word in pos
        """
        d = dict()
        for i in self.pos:
            if(i[0] in d):
                d[i[0]].add(i[1])
            else:
                d[i[0]] = set({i[1]})
        return d

    def get_tag_transition_prob(self, tag1, tag2):
        """
        Returns transition probability from tag1 to tag2

        :param tag1: tag1
        :param tag2: tag2
        :return: transition probability
        """
        val = (self.tag_emmission[(tag1, tag2)] /
(self.tag_count[tag1]))
        return val

    def get_bigrams(self):
        """
        Returns bigram
        """
        lis = []
        for i in range(1, len(self.pos)):
            lis.append((self.pos[i-1][1], self.pos[i][1]))

        return lis

    def process_lines(self):
        """
        Processes the lines of the file
        :return pos: list of word,pos
        """
```

```
        lines = self.myfile.split("\n")
        lines = [self.add_start_end(line) for line in lines]
        pos = [self.find_tags(i) for line in lines for i in
line.split(" ")]
        return pos

    def add_start_end(self, line):
        """
        Adds a linestartshere/START token at the beginning of the
sentence
        and lineendshere/END token at the end of sentence
        """
        line = "linestartshere/START "+line+" lineendshere/END"
        return line

    def find_tags(self, rawword):
        """
        Finds the word,tags in the raw word

        :param rawword: raw word
        :return: word,tags
        """
        pattern = "(.*)\/([A-Z,.\$:#)('`\.,\|]+)"
        try:
            return re.findall(pattern, rawword)[0]
        except:
            # In case parsing fails
            return "SOMEWORD", "NN"


class Runner():
    def __init__(self, utility_functions):
        self.utility_functions = utility_functions

    def make_one_transition(self, tag1, tag2, word, pval=1):
        """
        Simple, previously calculated max value * transition *
emmission

        :params tag1,tag2,word,pval:
        :returns value:
        """

        # NOTE Made changes here
        # return pval *
self.utility_functions.get_tag_emmission_prob(word, tag2) *
self.utility_functions.get_tag_transition_prob(tag1, tag2)
        return pval *
self.utility_functions.get_tag_emmission_prob(word, tag2)

    def step(self, next_word, plis=[POS_Holder(["START"], 1)]):
        """
```

```
        Single step of transition
        """
        next_dict = dict()

        # Calculate possisble tags for next word
        word_tag_set = self.utility_functions.get_tag(next_word)

        # Calculate score for all tags belonging to a word from
all previous tags
        # Previous tags are there in plis, e.g. START for first
word
        # It goes something like this for tag belongs to
next_word, check against
        # all previous tags and keep max in mval and keep
reference in ref.
        # then add ref to next_dict for this current tag
        for next_tag in word_tag_set:
            mval = 0
            ref = plis[0]
            for i in plis:
                calc = self.make_one_transition(
                    i.get_last_tag(), next_tag, next_word,
pval=i.get_c_val())
                # print(calc)
                if(calc >= mval):
                    mval = calc
                    ref = i
                else:
                    # print(calc)
                    pass

            params = ref.get_params(next_tag, mval)
            # print("Choosen Tag: ", params[0][-2],
            #       " ", params[0][-1], " : ", mval)
            next_dict[next_tag] = POS_Holder(params[0], params[1])

        # Finally return list of values because while developing
        # I made it like this XD since assuming that we are
working with lists is
        # easier than working with dictionaries... :P
        return list(next_dict.values())

    def run(self, my_test_str="Pierre Vinken , 61 years old , will
join the board as a nonexecutive director Nov."):
        my_test_str = my_test_str.split(" ")
        plis = self.step(my_test_str[0])
        for i in range(1, len(my_test_str)):
            plis = self.step(my_test_str[i], plis)

        # Calculate Path with MAX PROBABILITY
        mmax = 0
        for i in plis:
```

```
            if(i.get_c_val() >= mmax):
                ref = i
                mmax = i.get_c_val()
        return ref


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Parts of Speech
Tagger")
    parser.add_argument("train", help="Training file")
    parser.add_argument("test", help="Test file")

    args = parser.parse_args()

    # train_file = "POS.train"
    train_file = args.train
    test_file = args.test

    # TRAINING PHASE
    u = UtilityFunctions(train_file)

    # Initalize Runner
    runner = Runner(u)

    # TESTING PHASE

    with open(test_file, "r") as f:
        test_str = f.read()

    test_str = test_str.split("\n")

    z = []
    for i in range(len(test_str)):
        z.append(test_str[i].split(" "))

    a = []
    for lis in z:
        n = []
        for rawword in lis:
            n.append(u.find_tags(rawword))
        unzipped_list = list(zip(*n))
        a.append(unzipped_list)

    # Just testing...
    # ref = runner.run(
    #     my_test_str="Hey , I am Sarvesh Bhatnagar .")
    # print(ref.tags)

    # Calculate accuracy sentence by sentence
    tot_acc = 0
    for i in range(len(a)):
        ref = runner.run(my_test_str=" ".join(a[i][0]))
```

```
        tot_acc += accuracy_score(list(a[i][1]), ref.tags[1:])
    print(tot_acc/len(a))
```