Assignment 1

February 17, 2016

1 Assignment 1

This assignment will involve the creation of a spellchecking system and an evaluation of its performance. You may use the code snippets provided in Python for completing this or you may use the programming language or environment of your choice

Please start by downloading the corpus `holbrook-tagged.dat` from Blackboard

You may process the file using the following code. You should inspect the code to ensure that you understand its operation.

Note: the "yield" statement makes a function into an "iterator"

Note: There are snippets of code to help you build your answers. These are provided as a guide and do not need to be followed exactly.

Comments like "# Write your code here."_ indicate where you could write your code._

```
In [1]: from nltk.tokenize.regexp import WordPunctTokenizer

        tokenizer = WordPunctTokenizer()

        class SpellingSentence:
            """Store a sentence in original form and any spelling errors."""
            def __init__(self, text, spans, error_spans, corrections):
                """Constructor parameters are as follows:
                text: the original sentence.
                spans: an iterable that returns a span (start and end indices) for each word in the tex
                error_spans: a list of spans for mis-spelled words.
                corrections: a list of corresponding correctly spelled words.
                """
                self.text = text
                self.spans = list(spans)
                self.error_spans = error_spans
                self.corrections = corrections

            def tokens_with_errors(self):
                """Returns an iterator over the original (possibly mis-spelled) words as strings."""
                for x, y in self.spans:
                    yield self.text[x:y]

            def tokens_corrected(self):
                """Returns an iterator over the corrected words as strings."""
                token_spans = []
                for s1 in self.spans:
                    is_not_in_error_span = True
                    for s2 in self.error_spans:
                        is_not_in_error_span = (is_not_in_error_span and
```

```
                                            not (s1[0] >= s2[0] and s1[1] <= s2[1]))
                    if is_not_in_error_span:
                        token_spans.append(s1)

                token_spans = token_spans + self.error_spans
                token_spans = sorted(token_spans, key = lambda s1: s1[0])

                for x,y in token_spans:
                    if (x, y) in self.error_spans:
                        yield self.corrections[self.error_spans.index((x, y))]
                    else:
                        yield self.text[x:y]

            def errors(self):
                """Returns an iterator over the mis-spelled words as strings."""
                for x, y in self.error_spans:
                    yield self.text[x:y]


        def read_sentence(sentence):
            """Reads a sentence from the holbrook spelling corpus (as text),
            returning a SpellingSentence instance for that senetence."""
            error_spans = []
            corrections = []
            while "<ERR" in sentence:
                i = sentence.index("<ERR")
                j = sentence.index(">")
                correction = sentence[(i+10):j]
                k = sentence.index("</ERR>")
                if i > 0:
                    sentence = sentence[:(i-1)] + sentence[(j+1):(k-1)] + sentence[(k+6):]
                else:
                    sentence = sentence[(j+1):(k-1)] + sentence[(k+6):]
                error_spans.append((i, k - (j - i  + 3)))
                corrections.append(correction)
            return SpellingSentence(sentence, tokenizer.span_tokenize(sentence),
                                    error_spans, corrections)

        # Example usage:
        ss = read_sentence("My <ERR targ=sister> siter </ERR> <ERR targ=goes> go </ERR> to Tonbury.")

        print(list(ss.tokens_with_errors()))
        print(list(ss.tokens_corrected()))
        print(ss.error_spans)
        print(ss.spans)
        print(list(ss.errors()))
        print(ss.corrections)
```

```
['My', 'siter', 'go', 'to', 'Tonbury', '.']
['My', 'sister', 'goes', 'to', 'Tonbury', '.']
[(3, 8), (9, 11)]
[(0, 2), (3, 8), (9, 11), (12, 14), (15, 22), (22, 23)]
```

```
['siter', 'go']
['sister', 'goes']
```

You can load all the data as follows:

Note: you need to have downloaded the file "holbrook-tagged.dat" from blackboard into your notebook directory.

```
In [2]: data = [read_sentence(line) for line in open("holbrook-tagged.dat").readlines()]

        test = data[1:100]
        train = data[101:]
```

1.1 Task 1:

Calculate the frequency, *ignoring case*, of all words and bigrams (sequences of two words) satisfying the following assertions from the corrected *training* sentences:

```
In [ ]: def unigram(word):
            # Write your code here.
            return 0


        def bigram(words):
            # Write your code here.
            return 0

        assert unigram("the") == 1525
        assert bigram("it will") == 9
```

1.2 Task 2:

Using the following as a basis find all words that have the minimal edit distance from a given word (i.e., there does not exist another word with lower edit distance in the corrected training corpus):

Note: Edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other

```
In [3]: from nltk.metrics.distance import edit_distance

        # Edit distance returns the number of changes to transform one word to another
        print(edit_distance("hello", "hi"))

        all_words = set()

        for line in train:
            all_words.update(line.tokens_corrected())
```

```
In [ ]: def get_candidates(token):
            # Write your code here.
            for t in all_words:
                ed = edit_distance(token, t)
                # Write your code here.
            return []

        # get_candidates() should find the following
        assert get_candidates("calld") = ['calls', 'call', 'called']
```

1.3 Task 3:

Write a function that corrects spelling based on n-gram language model probabilities. It should work as follows

- Check if each word is the same as a correctly spelled word from the corpus
- If not, find the corpus words with minimum edit distance to that word
- If multiple corpus words have the same edit distance
  - Choose the most likely word based on the bigram probability, recall:
    - $p(w_n|w_{n-1}) = \frac{c(w_{n-1}w_n)}{c(w_{n-1}\cdot)}$ where $c()$ is the count in the corpus
  - If the bigram probability does not exist (or the spelling error is in the first word), use the unigram probability, recall:
    - $p(w_n) = \frac{c(w_n)}{N}$ where N is the total number of *tokens* (not types) in the corpus

```
In [ ]: def correct(tokens):
            for token in tokens:
                # Write your code here.
                # if is_misspelled(token):
                    # candidates = get_candidates(token)
                    # if len(candidates) > 1
                        # Find candidate with highest bigram value
                        # If first word or not bigram exists
                            # Find candidate with highest unigram value
                        # yield best_candidate
                    # else
                        # yield candidates[0]
                # else
                    yield token

        assert list(correct(["so", "they", "calld", "it", "the", "murder", "car"]))[2] != "calld"
```

1.4 Task 4:

Using the test corpus evaluate the *accuracy* of your method, i.e., how many words from your systems output match the corrected sentence (you should count words that are already spelled correctly and not changed by the system).

```
In [ ]: # Write your code here.
```

Notice that your system currently corrects many words that do not need correction, the following code detects all the words that are in the test but not the training set and if they are errors or not:

```
In [4]: # True spelling mistakes
        true_positives = []
        # Correctly spelled but unrecognized words
        false_positives = []

        for sentence in test:
            for word in sentence.tokens_with_errors():
                if word not in all_words:
                    if word in sentence.errors():
                        true_positives.append(word)
                    else:
                        false_positives.append(word)
```

```
print(len(true_positives))
print(len(false_positives))
```

```
72
88
```

1.5 Task 5:

Using an SVM (see Lab 3) write a classifier that classifies a novel word as a misspelling or a new word based on the following features

- The minimum edit distance to the nearest known English word
- Whether it starts with a capital
- Whether it is all in capitals
- One other feature of your choice

```
In [ ]: def ed(token):
            return 0

        def init_cap(token):
            # Write your code here.
            return 1

        def all_caps(token):
            # Write your code here.
            return 0

        def my_feature(token):
            # Write your code here.
            return 0

        X = [[ed(token), init_cap(token), all_caps(token), my_feature(token)]
             for token in true_positives + false_positives]
        y = [1] * len(true_positives) + [0] * len(false_positives)

        from sklearn import svm

        svc = svm.SVC()
        svc.fit(X, y)
```

1.6 Task 6:

Using the `cross_validation` methods from Scikit Learn (covered in Lab 4) perform 10-fold cross validation and report your precision, recall and F-Measure.

```
In [ ]: # Write your code here.
```