Как написать проверку орфографии («спеллчекер»)

(c) Питер Норвиг

Перевод: Петров Александр

Кто такой Питер Норвиг

Питер Норвиг американец, возглавляет исследовательское отделение компании Google. Почетный член и советник Американской ассоциации Искусственного интеллекта. Написал книгу "Искусственный интеллект: современный подход." в соавторстве со Стюартом Расселом. Основные статьи Норвига посвящены проблемам искусственного интеллекта и компьютерной лингвистики. Он является одним из авторов языка JScheme. Он также является автором известного эссе "Научитесь программировать за десять лет", переведенного на разные языки мира.

И так, приступим...

На прошлой неделе два моих друга (Дин и Билл) независимо друг от друга сказали мне, что они поражены работой Гугловского корректора орфографии. Как здорово и быстро он работает, говорили они. Напишите в строке поиска, например «праферка» и Гугл любезно поправит вас – « Возможно, вы имели в виду: *проверка*». (прим.перев. Стоит, конечно, отметить, что все ломается, если захотеть). Поисковики Microsoft'а и Yahoo работают тоже хорошо. Меня же удивило то, что изначально я думал, что Дин и Билл будучи специалистами в области математики обратили внимание на отличную работу проверки орфографии в Гугле потому, что осознавали те проблемы, которые возникают при статистической обработке текста и при решении таких задач, как проверка орфографии. Однако как оказалось оба моих друга математикой не занимались, и вообщем-то моё предположение оказалось не верным.

Я подумал, что многим будет интересно узнать, как реализуются орфографические корректоры. Не думаю, что стоит полностью описывать детали реализации корректоров, которые используются в Гугле или Yahoo, но планка, которую я поставил для своего корректора достаточно высока – корректор, код которого занимает чуть меньше страницы, будет иметь точность около 80\90 % и будет работать со скоростью 10 слов в секунду. И так вот он – 21 строка Пайтоновского (Python 2.5) кода – и это полностью законченный орфографический проверяльщик:

```
import re, collections
def words(text): return re.findall('[a-z]+', text.lower())
def train(features):
    model = collections.defaultdict(lambda: 1)
        for f in features:
            model[f] += 1
        return model
NWORDS = train(words(file('big.txt').read()))
alphabet = 'abcdefghijklmnopqrstuvwxyz'
def edits1(word):
    n = len(word)
    return set( [word[0:i]+word[i+1:] for i in range(n)] +
# deletion
                [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] +
  # transposition
                [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] +
   # alteration
                [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])
```

```
    # insertion
def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in
NWORDS)
def known(words): return set(w for w in words if w in NWORDS)
def correct(word):
    candidates = known([word]) or known(edits1(word)) or
known_edits2(word) or [word]
    return max(candidates, key=lambda w: NWORDS[w])
```

Использовать этот код нужно следующим образом –

```
>>> correct('speling')
'spelling'
>>> correct('korrecter')
'corrector'
```

Реальная математика – немного теории вероятностей

Как же работают эти 21 строка кода? Для начала немного теории. Пусть дано слово, будем пытаться отыскать слово, в котором с наибольшей вероятностью исправлены допущенные ошибки (если ошибок нет, то таким словом будет данное). Разумеется, мы не сможем гарантировать 100% исправления всех ошибок. (Например, если нам дано слово «пак», то правильным будет слово «паз» или «парк» ?), именно поэтому мы используем вероятностный (или другими словами стохастический) подход. Будем говорить, что мы пытаемся выбрать такое слово c из всех возможных слов-исправлений, что вероятность появления именно слова c при данном слове w будет максимальна:

$$\text{argmax}_c \, P(c|w)$$

Согласно теореме Байеса - выражение, записанное выше, эквивалентно следующему выражению:

$$\text{argmax}_c \, P(w|c) \, P(c) \, / \, P(w)$$

Поскольку $P(w)$ одинакова для всех c мы можем отбросить $P(w)$, что даст нам:

$$\text{argmax}_c \, P(w|c) \, P(c)$$

В этом выражении присутствуют три части. Справа налево:

1. $P(c)$ – вероятность появления слова c (частотность употребления c). Эта вероятность обусловлена самим языком (точнее моделью языка). Иначе говоря, $P(c)$ определяет как часто c встречается в текстах на русском [английском\…] языке. P(«превед») будет достаточно высока, тогда как P(«благоденствовать») будет меньше, а P(«ыгввыцшы») будет около нуля.
2. $P(w|c)$ – вероятность того, что автор опечатался и написал w, хотя имел в виду c. По сути дела эта вероятность обусловлена частотностью тех или иных ошибок в языке (и называется моделью ошибок языка).
3. argmax_c – оператор, перебирающий все возможные c в поиске наиболее (вероятнее всего) подходящего из них (т.е. данный оператор ищет такое допустимое c, которе максимизирует условную вероятность появления w при данном c).

Может возникнуть очевидный вопрос – зачем мы преобразовали простое выражение «$\text{argmax}_c \, P(c|w)$» с помощью какого-то Байеса в более сложное выражение, в котором используются аж две языковые

модели, вместо одной? Дело в том, что P(*c*|*w*) учитывает в себе сразу обе языковых модели, поэтому очевидно, что проще выделить эти модели и работать с ними по отдельности. Предположим у нас есть слово с опечаткой – «езать», это может быть как «ехать», так и «резать». Для какого из исправлений P(*c*|*w*) будет максимально ? Оба исправления имеют примерно одинаковую частотность в русском языке. Хорошо допустим «х» и «з» близко расположены в русской раскладке клавиатуры и это повышает вероятность варианта «ехать», но это не повод, чтоб отбрасывать «резать», ведь «е» и «р» тоже близки. Поэтому лучше не рассматривать P(*c*|*w*) как единую величину, ибо нам приходится учитывать и частность исправления *c* и вероятность исправления *c* для данной опечатки в *w*. Удобнее работать с этими двумя вероятностями по отдельности.

Теперь мы готовы к тому, чтоб посмотреть, как же работает 21-строчный орфографический корректор. Начнем с P(*c*). Мы читаем большой текстовый файл, big.txt, в котором записан миллион слов. Содержание этого файла – это несколько книг из проекта Гутенберг, а также списки самых часто встречающихся слов из Wiktionary и British National Corpus. (Вообще-то в самолете, когда я писал эту статью, у меня в моем ноутбуке были ли лишь рассказы о Шерлоке Холмсе, остальные тексты я добавил позже и перестал добавлять их, когда увидел, что новый текст не улучшает характеристики программы. Я подробно остановлюсь на этом позже, когда мы будем оценивать реализованный корректор)

После того как мы загрузили наш файл, мы извлекаем отдельные слова из файла (используя функцию words, которая считает словом любую последовательность алфавитных символов и при этом является нечувствительной к регистру (words(“The”) -> “the”; words(“Don’t”) -> “don”,”t”.)) Затем мы обучаем нашу стохастическую модель. Это звучит неплохо, но реально, мы просто считаем сколько раз в файле встречается каждое слово. Именно это делает функция train. В коде это выглядит следующим образом:

```
def words(text): return re.findall('[a-z]+', text.lower())
def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model
NWORDS = train(words(file('big.txt').read()))
```

Мы можем узнать сколько раз слово *w* встретилось в файле просто обратившись к таблице NWORDS: NWORDS[w]. Однако при таком подходе мы сталкиваемся с проблемой – что делать со словами, которых не было в файле. Было бы неправильным заявлять, что такие слова имеют вероятность 0, только потому, что они не присутствовали в нашей выборке. Вообще говоря, есть несколько решений данной проблемы. Сейчас мы выберем самое простое – будем считать, что мы «видели» любое слово хотя бы один раз, даже если этого слова не было в нашей файле. В статистике такой подход называется сглаживанием – мы как бы поднимаем провалы на графике распределения вероятностей нашей модели языка. При подсчете числа вхождений слов в файл мы используем collections.defaultdict (это обычный Пайтоновский словарь, т.е. это то, что в других языках иногда называют хэш-таблицей). В силу озвученных выше причин, все значения словаря инициализируются значением 1.

Теперь обратим внимание на проблему перечисления всех возможных исправлений *c* для слова *w*. Часто говорят о расстоянии(edit distance) между двумя словами, имея при этом в виду число символов, которые надо изменить в одном из слов, чтоб слова стали идентичными. Изменением может быть удаление символа, транспозиция символов (когда символы меняются местами), замена одного символа другим или вставка нового символа. Ниже представлен код функции, которая возвращает все слова *c* для данного слова *w* с расстоянием в единицу:

```
def edits1(word):
    n = len(word)
    return set( [word[0:i]+word[i+1:] for i in range(n)] +                     #
deletion
                [word[0:i]+word[i+1]+word[i]+word[i+2:] for i in range(n-1)] +  #
transposition
```

```
                   [word[0:i]+c+word[i+1:] for i in range(n) for c in alphabet] +   #
alteration
                   [word[0:i]+c+word[i:] for i in range(n+1) for c in alphabet])    #
insertion
```

На выходе этой функции - множество слов(set). Для данного английского слова длины n существует n удалений, n-1 перестановок, 26n замен, 26(n+1) вставок, что в итоге дает нам 54n+25 вариантов. Например len(edits1("something")) вернет 494.

Считается, что от 80% до 90% всех опечаток отстоят от оригинала на единицу (в смысле описанного выше расстояния). Как вскоре будет показано ниже, когда я взял множество слов с 270 ошибками, только 76% из них отстояло от правильных вариантов на расстояние 1. Возможно дело в сложности ошибок, которые попали в мою выборку но, в любом случае мне показалось, что стоит учитывать и ошибки с расстоянием 2. К счастью их учет реализуется очень просто – нужно применить функцию edits1 к своему же результату 1 раз.

```
def edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1))
```

(прим.перев. Написать такой код не сложно, но боюсь, что для его исполнения в разумные сроки нам придется арендовать один из суперкомпьютеров NASA) len(edits2("something")) дает 114324 вариантов. Однако edits2 обладает гораздо лучшим покрытием, чем edits1 – на моем тесте из 270 ошибок только 3 ошибки отстоят на расстояние большее чем 2. Таким образом, edits2 покрывает 98.9% всех ошибок, что вполне меня устраивает. Кроме того, поскольку мы не собираемся увеличивать расстояние находимых ошибок, мы можем немного оптимизировать наш код – мы можем сохранять только те варианты, которые ранее были сохранены в NWORDS:

```
def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)
```

Теперича known_edits2("something") возвращает множество всего из трех слов - {«smoothing», «something», «soothing»}, что несколько меньше чем 114324 слова, генерируемых функцией edits2. В результате мы получаем ускорение в 10%.

И так нам осталось рассмотреть реализацию последней части нашей программы – модель ошибок или P($w|c$). И вот тут-то меня подстерегали проблемы. Сидя в самолете, без интернета, я не знал, где б мне найти данные для обучения моей системы. Следуя зову своей интуиции, я решил использовать некоторые эвристики – мы с большей вероятностью можем ошибиться и заменить одну гласную на другую, тогда как замена двух согласных менее вероятна; ошибка в первой букве слова также имеет небольшие шансы на возникновение и т.д. Однако у меня не было фактов, не было результатов экспериментов, чтоб подтвердить мои догадки. Поэтому я поступил проще – я определил тривиальную модель ошибок, которая считает, что все известные слова с расстоянием в единицу от данного слова имеют большую вероятность стать опечаткой, чем слова с расстоянием в 2, но при этом они имеют бесконечно маленькую вероятность по сравнению с известными словами с расстоянием 0. Под известными словами я подразумевал слова из нашего файла (big.txt). Реализовать такую модель можно следующим образом –

```
def known(words): return set(w for w in words if w in NWORDS)
def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or
[word]
    return max(candidates, key=lambda w: NWORDS[w])
```

В начале функция correct строит множество возможных слов-исправлений. В это множество включаются известные слова, отстоящие от данного слова на минимальное расстояние. И как только

такое множество будет построено, из него будет выбран элемент с максимальным значением P(*c*), которое, как мы помним, определяется в словаре NWORDS.

Оценка программы

Самое время оценить насколько хорошо работает наша программа. В полете я потестировал её на нескольких примерах и она с ними вполне справилась. После посадки, я сразу же загрузил словарь ошибок Роджера Миттона (Roger Mitton's Birkbeck spelling error corpus), который храниться в Оксфордском архиве. С помощью этого словаря я построил два тестовых примера. Первый я использовал в процессе тестирования программы при её разработке и настройке.

Второй тест был моим окончательным тестом, в том смысле, что программу на этом тесте я запустил один раз и уже не мог улучшить результаты тестирования, изменяя программу как-либо, для того чтобы подогнать её под этот конкретный тест. Я часто так делаю – использую два тестовых примера. Это помогает мне не привязываться к конкретным входным данным. Ниже показаны два теста и программа тестировщик. Полный текст примеров и программы Вы можете найти здесь - spell.py.

```
tests1 = { 'access': 'acess', 'accessing': 'accesing',
          'accommodation': 'accomodation acommodation acomodation',
          'account': 'acount', ...}
tests2 = {'forbidden': 'forbiden', 'decisions': 'deciscions descisions',
         'supposedly': 'supposidly', 'embellishing': 'embelishing', ...}

def spelltest(tests, bias=None, verbose=False):
    import time
    n, bad, unknown, start = 0, 0, 0, time.clock()
    if bias:
        for target in tests: NWORDS[target] += bias
    for target,wrongs in tests.items():
        for wrong in wrongs.split():
            n += 1
            w = correct(wrong)
            if w!=target:
                bad += 1
                unknown += (target not in NWORDS)
                if verbose:
                    print '%r => %r (%d); expected %r (%d)' % (
                    wrong, w, NWORDS[w], target, NWORDS[target])
    return dict(bad=bad, n=n, bias=bias, pct=int(100. - 100.*bad/n),
                unknown=unknown, secs=int(time.clock()-start) )

print spelltest(tests1)
print spelltest(tests2)  ## only do this after everything is debugged
```

Результаты теста –

```
{'bad': 68, 'bias': None, 'unknown': 15, 'secs': 16, 'pct': 74, 'n': 270}
{'bad': 130, 'bias': None, 'unknown': 43, 'secs': 26, 'pct': 67, 'n': 400}
```

Итого: отладочный тест из 270 слов дал 74% верных «ответов» за 16 секунд, финальный тест показал 67% правильности за 26 секунд при 400 словах. И так, мы получили небольшую, простую и быструю программу, которая, однако, достаточно часто ошибается.

Совершенствование

Давайте прикинем как мы можем сделать нашу жизнь лучше… Или просто попробуем усовершенствовать нашу программу. Мы вновь обратимся к трем параметрам нашей вероятностной модели - P(c); P($w|c$); и argmax_c . Обратим внимание на те примеры, на которых наша программа ошибалась. А затем посмотрим, какие есть ещё факторы, кроме трех, выше перечисленных.

1. P(c) или модель языка. Можно выявить два источника ошибок в языковой модели. Во-первых и главных, это слова, не учтенные при обучении модели – т.е. неизвестные слова. В отладочном примере было 15 неизвестных слов или 5%, в финальном тесте 43 неизвестных слова или 11%. Вот несколько примеров работы функции spelltest при verbose=True :

```
correct('economtric') => 'economic' (121); expected 'econometric' (1)
correct('embaras') => 'embargo' (8); expected 'embarrass' (1)
correct('colate') => 'coat' (173); expected 'collate' (1)
correct('orentated') => 'orentated' (1); expected 'orientated' (1)
correct('unequivocaly') => 'unequivocal' (2); expected 'unequivocally' (1)
correct('generataed') => 'generate' (2); expected 'generated' (1)
correct('guidlines') => 'guideline' (2); expected 'guidelines' (1)
```

   Выше показаны результаты вызовов функции correct (в скобках указана частота встречаемости слова в обучающей выборке, т.е. NWORDS) и слова, которые как ожидалось должна вернуть функция (в скобках указаны их частота встречаемости в обучающей выборке). Эти результаты показывают, что если система не знает такого слова как «econometric», она не способна исправить опечатку в слове «economtric». Мы можем уменьшить количество таких ошибок, расширив обучающую выборку, включив в неё больше разнообразных слов, однако при этом мы повысим вероятность того, что система не сможет выбрать подходящее для данной конкретной опечатки слово из-за большого разнообразия слов. Обратите внимание, что выше в последних четырех строках приведены примеры, в которых указаны слова, которые встречаются в словаре нашей системы, но с измененными окончаниями. Таким образом, нам может потребоваться система, которая может сказать, что окончание “-ed” корректно для глаголов, а “-s” для существительных.

   Вторым источником ошибок в нашей модели является неправильное распределение вероятностей в обучающей выборке – слово, которое часто встречается в выборке, может относительно редко встречаться в повседневной лексике. Я должен отметить, что я не смог найти примеры, в которых именно эта причина являлась главной и основной причиной ошибочной работы системы, обычно большее влияние оказывали другие факторы.

   Давайте посмотрим, насколько сильно объем и репрезентативность обучающей выборки влияет на результаты работы системы. Для этого мы немного поколдуем над нашей тестировочной программой. Будем дублировать в обучающей выборке корректно написанное слово - будем делать 1 дубль этого слова, 10 дублей и т.д. Этим мы имитируем рост объема обучающей выборки и при этом она остается корректной. Эта идея реализована с помощью параметра bias функции spelltest.

   Ниже представлены результаты работы этой функции при изменении данного параметра:

Bias	Dev.	Test
0	74%	67%
1	74%	70%
10	76%	73%
100	82%	77%
1000	89%	80%

   На обоих наших тестовых вариантах мы способны достичь 80-90% правильности работы системы. Следовательно, мы можем предположить, что если бы у нас была достаточно хорошая

вероятностная модель языка, то программа работала бы лучше. С другой стороны, это достаточно самонадеянное предположение, поскольку, как уже было сказано, увеличивая объем обучающей выборки, мы можем добавить в словарь системы слова, которые приведут к новым ошибкам.

Другим путем разрешения проблемы неизвестных слов является разрешение системе выводить слова, которых ещё нет в её словаре. К примеру, на вход системы подается слово «electroencephalographicallz», нам-то с вами очевидно, что нужно поменять последнею «z» на “y”, но система-то этого не знает, кроме того слова «electroencephalographically» по странному (почти мифическому) стечению обстоятельств не оказалось в словаре нашей системы. Мы можем позволить системе вывести это слово, если расширить стохастическую модель языка слогами или суффиксами (нпр., «-ally»), или что проще, расширить модель не на уровне слогов или суффиксов, а последовательностями из 2, 3, 4 букв.

2. P(*w*|*c*) или модель ошибок. Как уже было сказано, используемая в системе модель ошибок тривиальна – чем меньше расстояние между словами, тем меньше ошибка. Это вызывает некоторые проблемы, проиллюстрированные примерами ниже. Во-первых, часто функция correct возвращает предполагаемое исправление, отстоящее от ошибочного слова на 1 символ, тогда как реальное исправление(исправление с высокой частотностью) отстоит на 2 символа.

```
correct('reciet') => 'recite' (5); expected 'receipt' (14)
correct('adres') => 'acres' (37); expected 'address' (77)
correct('rember') => 'member' (51); expected 'remember' (162)
correct('juse') => 'just' (768); expected 'juice' (6)
correct('accesing') => 'acceding' (2); expected 'assessing' (1)
```

В данных примерах, нам хотелось бы, чтобы изменение “d” на “c” классифицировалось системой как «более ошибочное», чем скажем замена буквы “d” на “dd”. Кроме того, встречаются случаи, когда при одинаковом расстоянии правильного слова и неправильного слова от слова с опечаткой, система выбирает неправильное, с точки зрения пользователя, слово.

```
correct('thay') => 'that' (12513); expected 'they' (4939)
correct('cleark') => 'clear' (234); expected 'clerk' (26)
correct('wer') => 'her' (5285); expected 'were' (4290)
correct('bonas') => 'bones' (263); expected 'bonus' (3)
correct('plesent') => 'present' (330); expected 'pleasant' (97)
```

Как и ранее, нам бы хотелось, чтоб система воспринимала замену «a» на «e» в «they» как менее значительное изменение, чем замена “y” на “t”. Нас сколько менее значимым должно быть это изменение? Как минимум в 2.5 раза, чтобы перекрыть вероятностное преимущество слова “that” над “they”.

Очевидно, что наша модель ошибок нуждается в совершенствовании. Руководствуясь здравым смыслом, мы могли бы в ручную присвоить более высокие приоритеты удвоению буквы (т.е. уменьшить значимость такого изменения) или изменению одной гласной на другую (по сравнению с произвольной заменой букв), одним словом, нам требуется лучшая модель, лучшая обучающая выборка по ошибкам. Причем объем такой выборки, содержащей значительное число распространенных описок, также должен быть значительным. Например, если мы хотим рассмотреть вероятность замены одной буквы другой, учитывая при этом две соседние буквы по обеим сторонам от заменяемой, то нам придется рассмотреть 26^6 или около 300 миллионов символов. Причем очевидно, что нам захочется рассмотреть несколько примеров таких замен, и, следовательно, грубо говоря, нам нужно будет рассмотреть около 10 миллиардов символов. Обратите внимание на связь между моделью языка и моделью ошибок. Текущая версия системы имеет очень простую модель ошибок (чем меньше расстояние между *c* и *w* , тем больший

приоритет отдается *c*), что отрицательно влияет на модель языка. Мы не можем бесконтрольно добавлять новые слова в обучающую выборку. Причина этого в том, что если до добавления нового слова программа выбирала в качестве исправления слово, отстоящее на расстояние 2, то после добавления, новое слово может как бы скрыть этот правильный (наиболее употребительный с точки зрения модели языка) вариант, если новое слово имеет расстояние 1. Т.е. модель ошибок влияет (отрицательно) на модель языка. Имея более качественную(разборчивую) модель ошибок, мы могли бы безбоязненно расширять обучающую выборку для модели языка. Вот несколько примеров, иллюстрирующих описанную ситуацию:

```
correct('wonted') => 'wonted' (2); expected 'wanted' (214)
correct('planed') => 'planed' (2); expected 'planned' (16)
correct('forth') => 'forth' (83); expected 'fourth' (79)
correct('et') => 'et' (20); expected 'set' (325)
```

Слова «wonted», «planed», «et» имеют не высокую частотность согласно модели языка (2, 2 и 20 соответственно), и они скрывают более часто употребительные слова из-за выбранной нами модели ошибок.

3. argmax_c или оператор выбора возможных слов-исправлений. Наша программа перебирает все возможные слова-исправления на расстоянии 2 от слова, подозреваемого на наличие опечатки. В отладочном тестовом примере только 3 из 270 слов отстоят на расстояние более 2, но в финальном тесте таких слов 23 (из 400). Вот они:

```
purple perpul
curtains courtens
minutes muinets

successful sucssuful
hierarchy heiarky
profession preffeson
weighted wagted
inefficient ineffiect
availability avaiblity
thermawear thermawhere
nature natior
dissension desention
unnecessarily unessasarily
disappointing dissapoiting
acquaintances aquantences
thoughts thorts
criticism citisum
immediately imidatly
necessary necasery
necessary nessasary
necessary nessisary
unnecessary unessessay
night nite
minutes muiuets
assessing accesing
necessitates nessisitates
```

Возможно, нам стоит расширить модель ошибок, увеличив допустимое расстояние между словом с опечаткой и словом-исправлением до 3. Видимо слова из примеров выше были бы исправлены, если бы мы разрешили вставлять гласную после другой гласной или заменять одну гласную на другую гласную или менять местами близкие согласные такие как «c» и «s».

4. Рассмотрим четвертый (и самый результативный) способ улучшения работы нашей программы: изменить интерфейс (сигнатуру) функции correct так, чтобы она могла учитывать контекст, в который входит то или иное слово, подозреваемое на опечатку. Сейчас correct обрабатывает одно слово за один раз. Абсолютно очевидно, что во многих случаях принять решение, основываясь лишь на одном слове, крайне тяжело. Например:

```
correct('where') => 'where' (123); expected 'were' (452)
correct('latter') => 'latter' (11); expected 'later' (116)
correct('advice') => 'advice' (64); expected 'advise' (20)
```

Мы не можем определенно сказать должен ли вызов correct('where') вернуть нам 'where' или 'were'. Однако если бы вызов функции выглядел бы следующим образом - ('They where going') , то вероятность выбора 'where' могла бы быть выше. (У нас бы появилась дополнительная информация, позволяющая обосновать выбор того или иного варианта) Причём такая стратегия помогла бы нам и в более нетривиальных случаях –

```
correct('hown') => 'how' (1316); expected 'shown' (114)
correct('ther') => 'the' (81031); expected 'their' (3956)
correct('quies') => 'quiet' (119); expected 'queries' (1)
correct('natior') => 'nation' (170); expected 'nature' (171)
correct('thear') => 'their' (3956); expected 'there' (4973)
correct('carrers') => 'carriers' (7); expected 'careers' (2)
```

Должны ли мы выбрать вариант 'there' при исправлении слова 'thear' или мы должны выбрать вариант 'their'. Мы не можем обосновать выбор, основываясь на одном слове, но если бы мы работали с предложением - correct('There's no there thear') , то выбор стал бы очевиден. Для того чтобы построить модель, которая бы учитывала контекст, нам потребуется ещё более большая обучающая выборка `(прим.перев. и похоже нам опять придется вспомнить о суперкомпьютерах)`. К счастью мы можем воспользоваться услугами Google, который предоставляет доступ к базе данных частотности слов, в которой учитывается контекст длиной в пять слов, а сама БД составлялась на основе триллиона слов.

Я уверен, что орфографический корректор пытающийся достичь 90% точности обязательно должен использовать контекст. Думаю, мы с вами займемся этим как-нибудь на досуге…

5. Мы можем несколько повысить точность нашей системы, если будем более тщательно относиться к обучающей выборке. При обучении использовался файл в миллион слов, однако никто не бился об заклад, что среди этих слов нет ошибочно написанных слов, не опечаток и т.д. По хорошему нужно бы найти их и исправить… Тестируя систему на своих тестовых вариантах, я обнаружил по крайней мере три случая, когда тест пытался убедить меня, что моя программа ошиблась, хотя я был уверен, что не прав тест:

```
correct('aranging') => 'arranging' (20); expected 'arrangeing' (1)
correct('sumarys') => 'summary' (17); expected 'summarys' (1)
correct('aurgument') => 'argument' (33); expected 'auguments' (1)
```

Кроме того, следует определиться с каким диалектом английского языка мы собираемся работать. Следующие три ошибки как раз связаны с различиями Американского и Британского английских языков:

```
correct('humor') => 'humor' (17); expected 'humour' (5)
correct('oranisation') => 'organisation' (8); expected 'organization' (43)
correct('oranised') => 'organised' (11); expected 'organized' (70)
```

6. Наконец мы можем улучшить саму реализацию нашей программы, оптимизировав её и повысив скорость выполнения. Скажем, мы можем реализовать её на компилируемом, а не интерпретируемом языке программирования. Можем использовать другой контейнер для строк, специализированный под хранение и сравнение строк. Можем кэшировать результаты наших

проверок и т.д. Только в этом деле всегда нужно помнить о том, что перед тем как оптимизировать что-либо нужно убедиться, что именно этот участок кода является узким местом программы.

Список литературы

- Статья Роджера Миттона, посвященная корректорам орфографии.
- Журафски и Мартин дают обзор поднимаемых в данной статье вопросов в своей статье "Обработка лингвистической информации".
- Маннинг и Щутзе рассматривают стохастические языковые модели.
- На сайте проекта aspell размещено много интересных материалов, в том числе тестовые примеры, которые вроде как будут получше тех, что я использовал.

Спеллчекеры на других языках программирования

После того как я опубликовал эту статью, появилось несколько реализаций корректора орфографии на других языках программирования. Возможно, кому-то будет интересно сравнить реализации на разных языках.

- Erlang: by Federico Feroldi
- F#: by Sebastian G
- Haskell: by Grzegorz
- Perl: by riffraff
- Rebol: by Cyphre
- Ruby: by Brian Adkins
- Scheme: by Shiro
- Scheme: by Jens Axel

Перевод: Петров Александр
Оригинальный текст статьи Вы можете найти здесь.