АВТОКОРРЕКТОР

ИВАНОВ СЕРАФИМ
СИДОРИНА ДАРЬЯ
ЧАСТУХИН ДАНИИЛ

ГРУППА 0392

РАБОТА СОСТОЯЛА ИЗ ДВУХ ЧАСТЕЙ

1. Корректор, основанный на переборе
2. Корректор на основе машинного обучения

ВВОДНОЕ

Исправление опечаток разделяется на контекстно-независимое и контекстно-зависимое. В первом случае ошибки исправляются для каждого слова в отдельности, во втором – с учетом контекста (*например*, для фразы *«она пошле домой»* в первом случае исправление происходит для каждого слова в отдельности, где мы можем получить **«она пошел домой»**, а во втором случае - **«она пошла домой»**).

Можно выделить несколько основные группы ошибок для контекстно-независимого исправления:

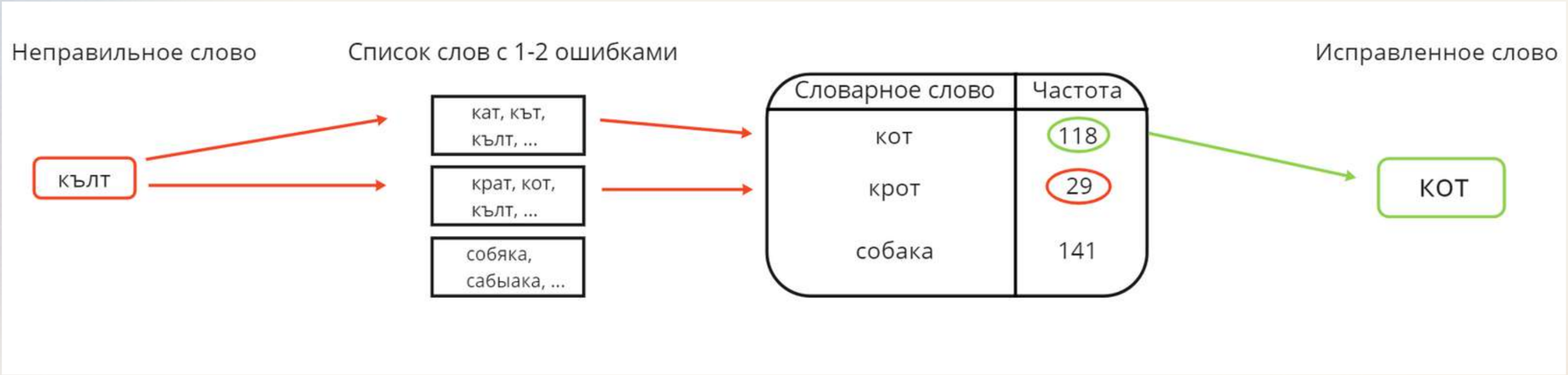
1) ошибки в самих словах (пмрвет → привет), к этой категории относятся всевозможные пропуски, вставки, замены и перестановки букв – 63,7%,
2) слитно-раздельное написание слов – 16,9%,
3) искаженная раскладка (ghbdtn → привет) – 9,7 %,
4) транслитерация (privet → привет) – 1,3%,
5) смешанные ошибки – 8,3%.

АВТОКОРРЕКТОР, ОСНОВАННЫЙ НА ПЕРЕБОРЕ

ОСНОВНАЯ ИДЕЯ

Для каждого словарного слова перебираются все возможные ошибочные варианты с ограниченным числом ошибок или высчитывается расстояние, также учитывается частота.

1)



2)

КОЭФФИЦИЕНТ ЖАККАРА

Коэффициент Жаккара - бинарная мера сходства, вычисляемая следующим образом:

$$k = \frac{c}{a + b - c}$$

где

a, b - количество уникальных символов в строках,

c - количество совпадающих символов

Пример:

коэффициент между строками "собака" и "яблоко" будет равен $\frac{3}{5 + 5 - 3} = 0.43$

Явный минус:

анаграммы имеют коэффициент 1: *k*("абв", "ваб") = 1

РАССТОЯНИЕ ЛЕВЕНШТЕЙНА

Расстояние Левенштейна (*редакционное расстояние*) — метрика, показывающая модуль разности между двумя последовательностями символов. Определяется как минимальное количество односимвольных операций вставки, удаления и замены, необходимых для превращения одной последовательности символов в другую.

Рекуррентная формула для вычисления расстояния

$$d(S_1, S_2) = D\big(len(S_1), len(S_2)\big)$$

$$D(i,j) = \begin{cases} 0, & i = j = 0 \\ i, & i > 0, j = 0 \\ j, & j > 0, i = 0 \\ D(i-1, j-1), & S[i] = S[j] \\ 1 + \min\big(D(i-1,j), D(i,j-1), D(i-1,j-1)\big), & else \end{cases}$$

Расстояние *Дамерау-Левенштейна* является модификацией расстояния *Левенштейна*, включающее в себя операцию транспозиции символов

ПРИМЕР РАСЧЁТА РЕДАКЦИОННОГО РАССТОЯНИЯ

Рассчитаем расстояние между словами "киты" и "кроты"

		к	и	т	ы
	0	1	2	3	4
к	1	0	1	2	3
р	2	1	1	2	3
о	3	2	2	3	4
т	4	3	3	2	3
ы	5	4	4	3	2

- символы совпадают
- замена символа
- вставка символа
- итоговое расстояние

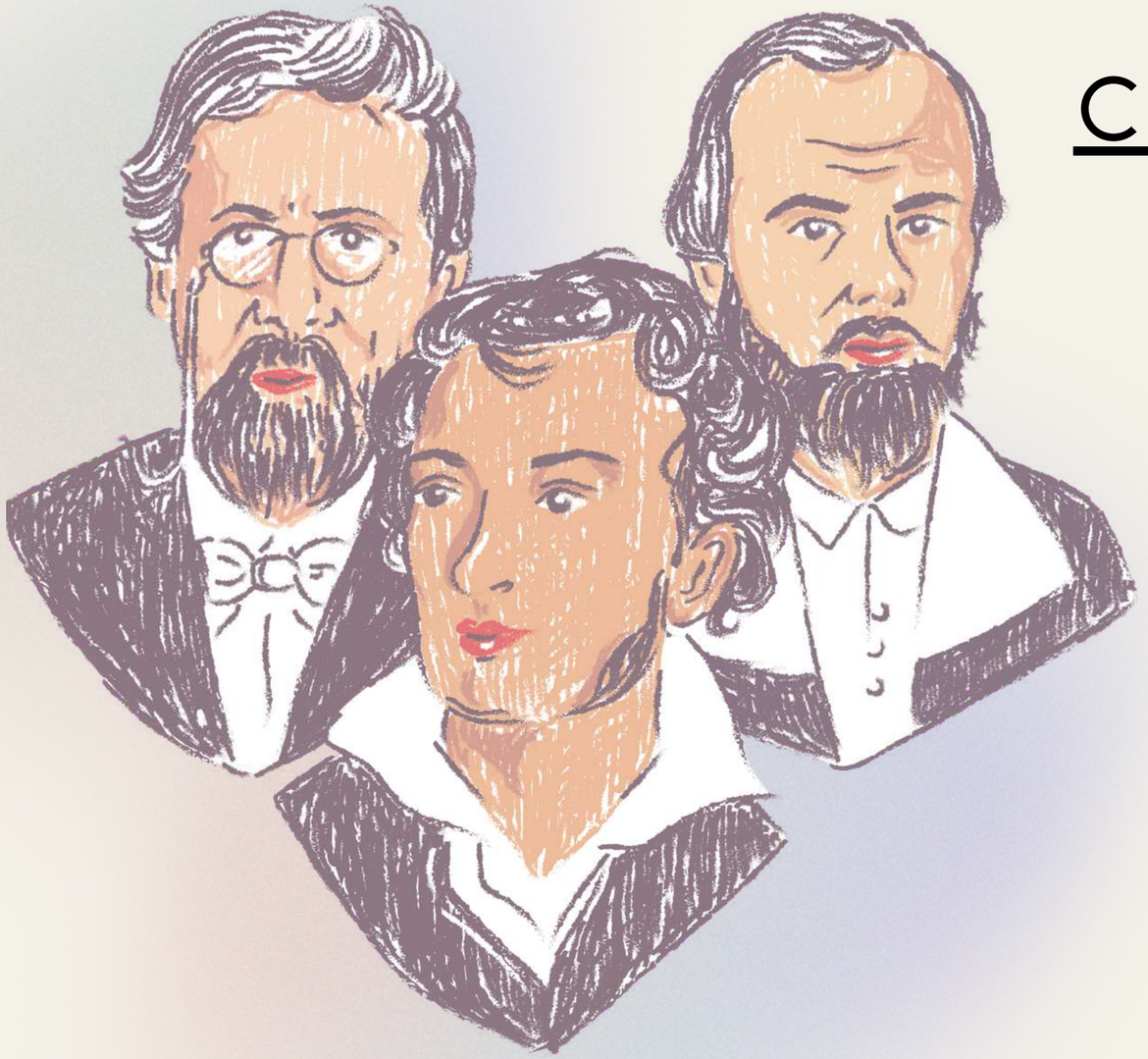
Расстояние Левенштейна между этими двумя строками будет 2

МОДЕЛЬ BAG OF WORDS

В этой модели мы игнорируем порядок слов, но учитываем их частоту.

Представить это можно себе так: вы берете все слова текста и забрасываете их в мешок. Теперь, если вы хотите сгенерировать предложение с помощью этого мешка, вы просто трясете его (слова там перемешиваются) и достаете указанное количество слов по одному (мешок непрозрачный, поэтому слова вы достаете наугад).

Почти наверняка полученное предложение будет грамматически некорректным, но слова в этом предложении будут в +- правильной пропорции (более частые будут встречаться чаще, более редкие - реже).



СЛОВАРЬ СОСТОИТ ИЗ :

- "Капитанская дочка"
- "Преступление и наказание"
- "Каштанка"
- "Котлован"
- "Тихий Дон"
- "Дети Арбата"
- Статьи

ГЕНЕРАЦИЯ НЕВЕРНЫХ СЛОВ

Как было сказано ранее, в слове межно допустить следующие ошибки:
пропуск буквы, дублирование буквы, неверная буква, перестановка букв.

Сгенерировать слова, находящиеся на расстоянии 2, 3 и так далее, можно применив функцию генерации слов на расстоянии 1 дважды, трижды и так далее:

```
def distance2(word):
    return {e2 for e1 in distance1(word) if e1 for e2 in distance1(e1)}
```

Получить все слова, находящиеся на расстоянии 1 можно следующим кодом:

```
# все ошибки на расстоянии 1 (расстояние Левинштейна)
def distance1(word):
    pairs      = splits(word)
    transposes = [a+b[1]+b[0]+b[2:]  for (a,b) in pairs if len(b)>1]                          # перестановки
    replaces   = [a+c+b[1:]          for (a,b) in pairs if b for c in replaces_set[b[0]] if b] # замены:
                                                                                                #  опечатки,
                                                                                                #  пропуски и дублирования букв

    last_replaces = [word[0:-1] + c for c in replaces_set[word[-1]]]                          # замены в конце слова
    return set(replaces + last_replaces + transposes)

def splits(word):
    return [(word[:i], word[i:])
            for i in range (len(word)+1)]
```

где replaces_set(ω) - список всех возможных замен для символа 'c'
В простейшем случае, replaces_set(ω) = { 'а', 'б', … , 'я', 'ωω', '' }

МОДЕЛЬ COUNTER

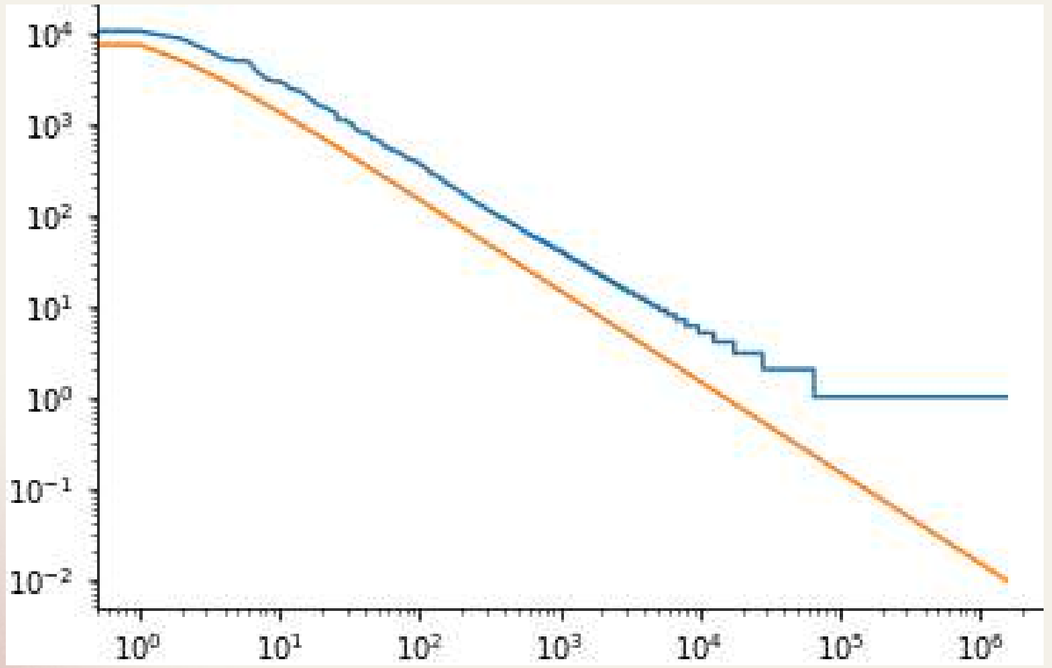
ДРУГОЕ ПРЕДСТАВЛЕНИЕ BAG OF WORDS - **COUNTER**.
ЭТО СЛОВАРЬ, СОСТОЯЩИЙ ИЗ ПАР ВИДА:

{ 'СЛОВО' : КОЛ-ВО ВХОЖДЕНИЙ СЛОВА В ТЕКСТ }

```
for w in tokens('самые редкие слова: Крыжить, Михрютка, Драдедамовый '):
    print(w, COUNTS[w])
```

```
самые 27
редкие 16
слова 196
крыжить 0
михрютка 1
драдедамовый 3
```

ЗАКОН ЦИПФА



/**Из графика видно, что закон Ципфа выполняется для нашей модели **/

В 1935 лингвист Джордж Ципф отметил, что если все слова языка (или просто достаточно длинного текста) упорядочить по убыванию частоты их использования, то частота **n**-го слова в таком списке окажется **~1/n**. Если нарисовать частоты слов, начиная от самого часто встречающегося, на *log-log*-графике, то они должны приблизительно следовать прямой линии, если закон Ципфа верен.

ИСПРАВЛЕНИЕ ОРФОГРАФИЧЕСКИХ ОШИБОК

Применим следующий подход: пусть нам нужно исправить слово w. Найдем всех кандидатов, достаточно близких к w, и выберем более подходящего. Если проверки на близость недостаточно - берем слово с максимальной частотой.

Близость измеряется расстоянием Левенштейна (минимальным количеством удалений, перестановок, вставок, или замен символов, необходимым чтобы одно слово превратить в другое):

Пример: нужно исправить слово 'атец'.

- На *distance0* находится само слово 'атец'
- На *distance1* находятся слова: 'атиец', 'отец', 'ацтец', 'атц', 'атер', 'атеъц' ...
- На *distance2* находятся слова: 'отц', 'тецр', 'тайец', 'отецъ', 'аттцц', 'льтец' ...

ВЕРОЯТНОСТЬ СЛОВА P(W)

Подсчитывать вероятность мы будем с помощью функции ***pdist***, которая на вход принимает мешок слов, а возвращает функцию, выполняющую роль вероятностного распределения на множестве всех возможных слов.

```
def pdist(counter):
    N = sum(list(counter.values()))
    return lambda x: counter[x]/N
```

Пример

```
for w in tokens('То мать пирогов напечет, то бабушка с булочками приедет'):
    print(P(w), w)
```

```
0.0014157824281471007 то
8.80011015667302e-05 мать
2.5882676931391233e-06 пирогов
5.176535386278247e-07 напечет
0.0014157824281471007 то
4.1412283090225975e-06 бабушка
0.0025271845755810403 с
5.176535386278247e-07 булочками
5.694188924906072e-06 приедет
```

РАЗБИЕНИЕ НА СЕГМЕНТЫ

```
@memo
def segment(text):
    if not text:
        return []
    else:
        candidates = ([first] + segment(rest)
                      for (first, rest) in splits(text, 1))
        res = max(candidates, key=Pwords)
        for i in res:
            if i not in WORDS:
                return [text]
        return res
```

```
segment('сдырочкойвправомбоку')
```

```
['с', 'дырочкой', 'в', 'правом', 'боку']
```

```
decl = ('предложениеэтосложнаясинтаксическаяконструкция')
```

```
print(segment(decl))
```

```
['предложение', 'это', 'сложная', 'синтаксическая', 'конструкция']
```

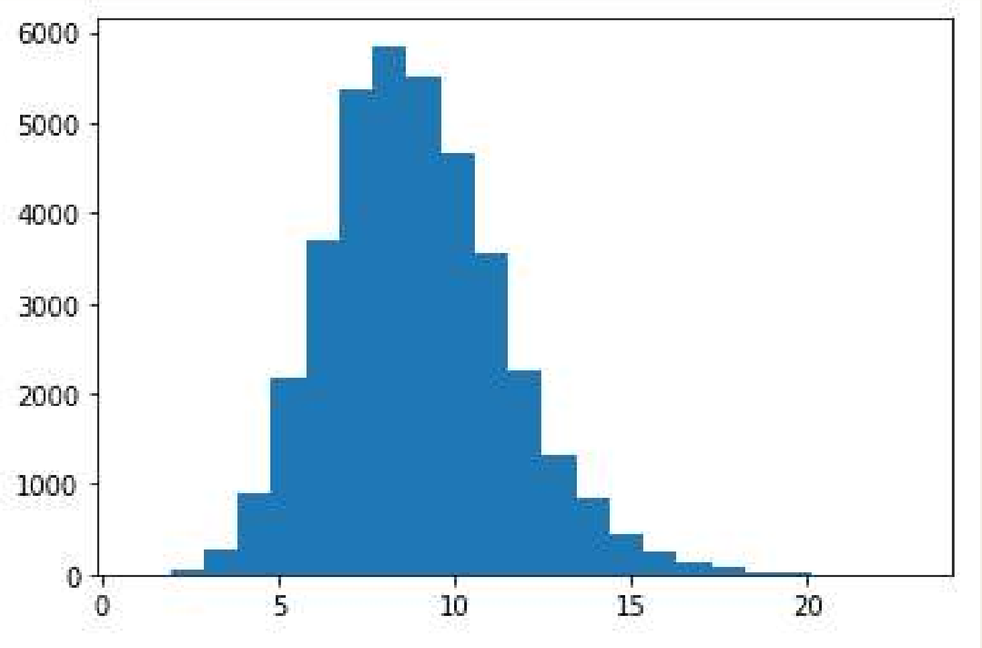
Подход 1: Пронумеруем все возможные разбиения и выберем то, у которого максимальная вероятность.

Подход 2: Делаем одно разбиение - на первое слово и все остальное. Если предположить, что слова независимы, можно максимизировать вероятность первого слова + лучшего разбиения оставшихся букв.

СГЛАЖИВАНИЕ

Что делать со словами, которые не встречаются в словаре? Например, "михрютка" более вероятно, чем "юшркцлвакс". Словарь неидеален: существуют реальные слова, которые мы не включили в него, то есть, их вероятность будет 0. Не будем обнулять их вероятности:

Модель можно представить в виде столбиков вероятностей, где столбик равен вероятности слова, которое в выборке было, и равен 0, если слова в выборке не было.
Мы хотим сгладить распределение вокруг пиков, чтобы модель давала ответ даже для слов, отсутствующих в словаре.
Этот процесс и называется *сглаживанием*.



ИСПРАВЛЕНИЯ "GHBDTN" → "ПРИВЕТ"

Чтобы исправить данный тип ошибок, мы приводим в соответствие буквы английского и русского языков на раскладке клавиатуры

```
eng_rep = ('f,dult`;pbqrkvyjghcnea[wxio]sm\'.z')
```

а далее, если допущена ошибка правописания, то применяются те же правила.

```
if sent[0] not in alphabet:
    sent_new = ''
    for i in range(len(sent)):
        if sent[i] == ' ':
            sent_new += ' '
            continue
        else:
            sent_new += alphabet[eng_rep.find(sent[i])]
    sent = sent_new
```

Пример:

1) введено слово "[fxecgfnm"

2) ему приводится в соответствие "хачуспать"

3) после автокорректора превратится в "хочу спать"

ДОПОЛНЕНИЯ

Чем плох Bag of words? Тем, что для него кошк[у] ≠ кошк[а]

Как это побороть? А что если мы уберем все окончания и часть суффиксов из слов, одним словом сделаем **стемминг Портера**. Тем самым мы будем считать, что формы слова не являются новыми словами.

```
from pymystem3 import Mystem
text = "Красивая мама красиво мыла раму"
m = Mystem()
lemmas = m.lemmatize(text)
print(''.join(lemmas))
```

```
красивый мама красиво мыть рама
```

ДОПОЛНЕНИЯ

Но тогда возникает проблема: существуют слова, для которых стемминг не приведет к одному и тому же слову, например "**шел**" и "**идти**". Тогда на помощь приходят морфологические анализаторы, которые приводят слово к начальной форме (**лемматизация**)

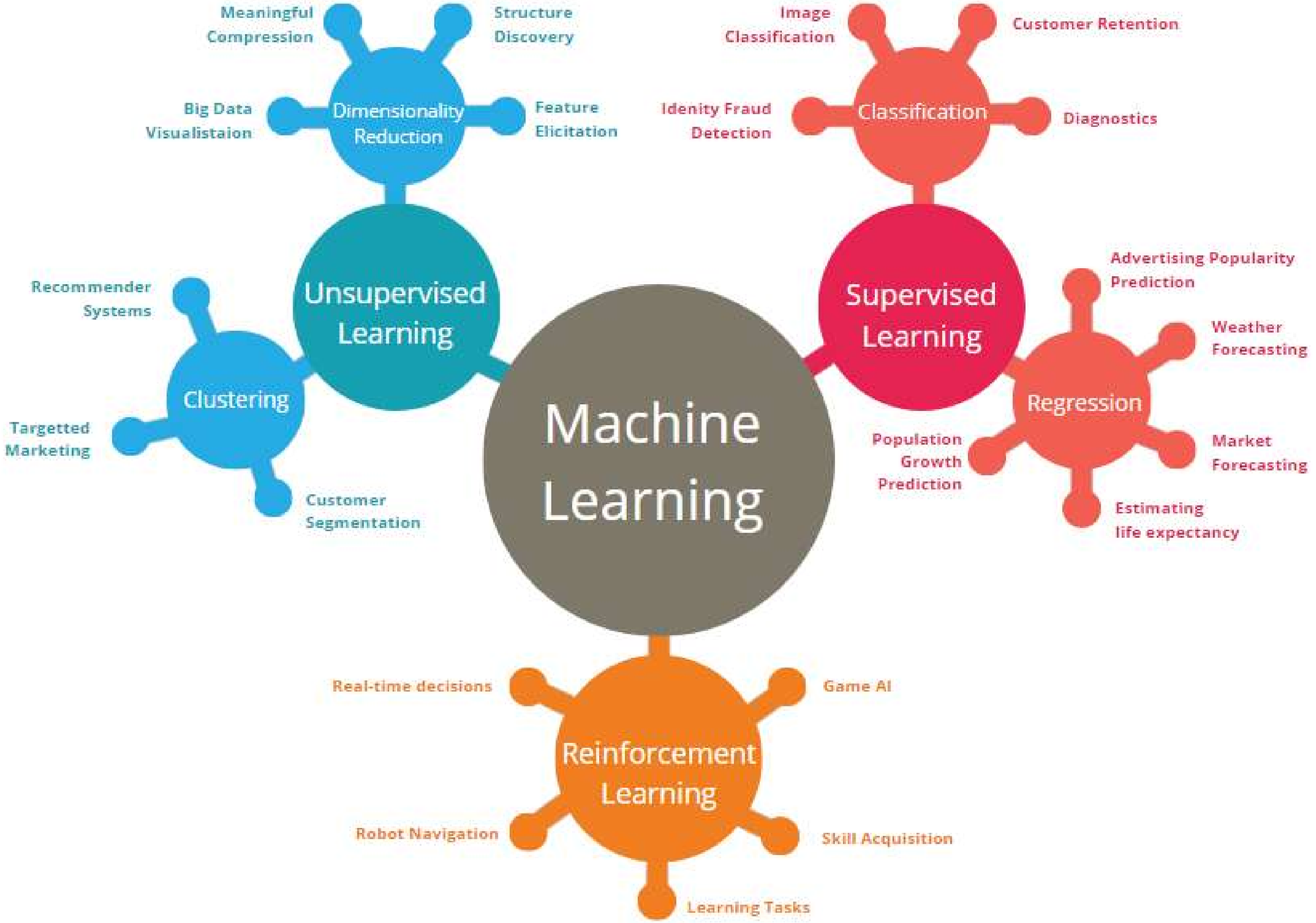
```
import pymystem3
m1=pymystem3.Mystem()

m1.analyze('шел')
```

```
[{'analysis': [{'lex': 'идти', 'wt': 1, 'gr': 'V,несов,нп=прош,ед,изъяв,муж'}],
  'text': 'шел'},
 {'text': '\n'}]
```

Но в таком случае возникает проблема: "*стекло*" (существительное или глагол прошедшего времени?), "*эти типы стали есть в прокатном цехе*" (это про разновидность стали или про голодных людей?). Морфологические анализаторы иногда делают ошибки и без контекста им не разобраться.

МАШИННЫЙ АВТОКОРРЕКТОР

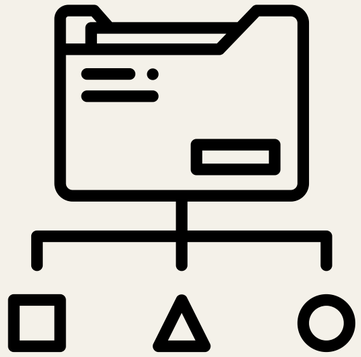

Machine Learning
Unsupervised Learning
Supervised Learning
Reinforcement Learning
Dimensionality Reduction
Clustering
Classification
Regression
Meaningful Compression
Structure Discovery
Big Data Visualistaion
Feature Elicitation
Recommender Systems
Targetted Marketing
Customer Segmentation
Image Classification
Customer Retention
Idenity Fraud Detection
Diagnostics
Advertising Popularity Prediction
Weather Forecasting
Population Growth Prediction
Market Forecasting
Estimating life expectancy
Real-time decisions
Game AI
Robot Navigation
Skill Acquisition
Learning Tasks

КЛАССИФИКАЦИЯ

Машинное обучение — методы искусственного интеллекта, характерной чертой которых является обучение за счёт применения решений множества сходных задач

Классификация - раздел машинного обучения, в котором ставится задача распределения объектов на классы.

Задача классификации - пусть задана обучающая выборка - конечное множество объектов, для которых известно, к каким классам они относятся. Классовая принадлежность остальных объектов не известна. Требуется построить алгоритм, способный классифицировать произвольный объект из исходного множества.



ТОКЕНИЗАЦИЯ

Токенизация - разбиение текста на токены
(токенами могут быть слова, символы, слоги, морфемы).

В случае контекстно - независимого корректора, токенизация производится для слова.
Это нужно для того, чтобы уменьшить количество признаков и избежать переобучения: иначе все неверные слова, сгенерированные для списка верных слов, будут признаками.

Слово проще всего разбить на буквы:

'**ТОКЕНИЗАЦИЯ**' - '**Т**', '**О**', '**К**', '**Е**', '**Н**', '**И**', '**З**', '**А**', '**Ц**', '**И**', '**Я**'
тогда всего признаков будет 33 (по количеству букв), что значительно упростит обучение. Будем далее называть эту токенизацию *CBT*

Но можно точно указать, на какой позиции в слове находится токен (это поможет модели различать анаграммы):

ТОКЕНИЗАЦИЯ' - '**Т0**', '**О1**', '**К2**', '**Е3**', '**Н4**', '**05**', '**З6**', '**А7**', '**Ц8**', '**И9**', '**Я10**'
Здесь количество токенов будет в большей степени зависеть от входных данных и токенов будет значительно больше - на наших входных данных их оказалось 430.
Будем далее называть эту токенизацию *ACBT*

ВЕКТОРИЗАЦИЯ

Комптьютер не может работать с символами и текстом. Поэтому все слова надо представить как набор понятных для компьютера данных. Этот процесс называется векторизацией текста.

Существует много способов произвести векторизацию. Мы используем векторизацию ***TF-IDF***, как наиболее точную и уже доказавшую свою надежность.

TF - IDF

Состоит из двух компонентов: *Term Frequency* (частотность слова в документе) и *Inverse Document Frequency* (инверсия частоты документа).
Они считаются следующим образом:

$$TF_{token_i} = \frac{n_i}{N_i}$$

$$IDF_{token} = \log\frac{P}{p}$$

$$TF - IDF_{token} = TF_{token} \cdot IDF$$

где:

n_i - число вхождений токена в *i*-том документе

N_i - количество токенов в *i*-том документе

p - количество документов, в которых встречается токен

P - общее количеств документов

ПРИМЕР

Пусть словарь **A** = { '**кот**', '**ком**', '**окно**' }

Разобьем каждое слово на токены:

D1 = { '**к**', '**о**', '**т**' }, **D2** = { '**к**', '**о**', '**м**' }, **D3** = { '**о**', '**к**', '**н**', '**о**' }

Token	**TF D1**	**TF D2**	**TF D3**	**IDF**	**D1 TF-IDF**	**D1 TF-IDF**	**D1 TF-IDF**
к	1 / 3	1 / 3	1 / 4	log(3 / 3) = 0	0	0	0
м	0	1 / 3	0	log(3 / 1)	0	0.37	0
о	1 / 3	1 / 3	1 / 2	log(3 / 3) = 0	0	0	0
н	0	1 / 3	1 / 4	log(3 / 2) = 0	0	0.14	0.10
т	1 / 3	0	0	log(3 / 1) = 0	0.37	0	0

Напоминание:

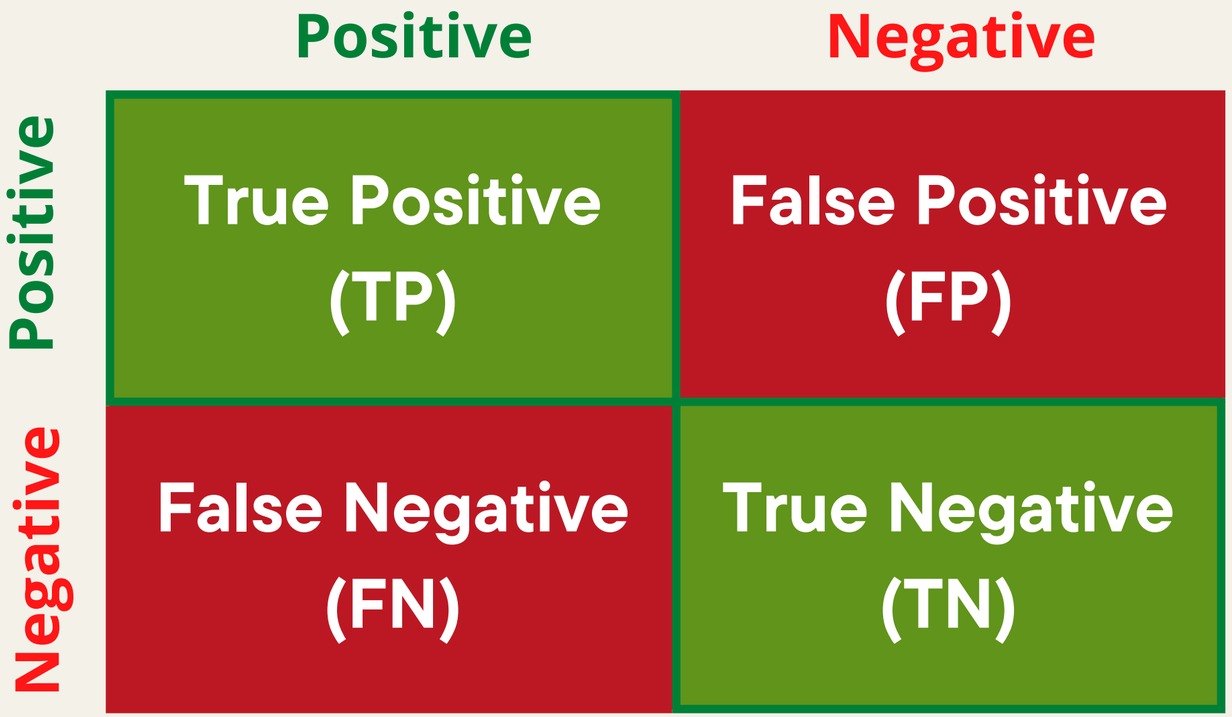
$$TF_{token_i} = \frac{n_i}{N_i}$$

$$IDF_{token} = \log \frac{P}{p}$$

$$TF - IDF_{token} = TF_{token} \cdot IDF$$

МЕТОДЫ ОЦЕНКИ КЛАССИФИКАЦИИ

БИНАРНАЯ КЛАССИФИКАЦИЯ



→ $Precision = \frac{TP}{TP + FP}$

→ $Recall = \frac{TP}{TP + FN}$

→ $Accuracy = \frac{TP}{TP + FP + FN + TN}$

Positive и *Negative* - это предсказания модели (допущена ли ошибка в слове?), а *True* и *False* - это оценка того, правильно ли определила модель наш класс (нужно ли было исправлять слово?) Так:

FP - количество исправленных слов (в словах не было ошибок),

TP - количество исправленных слов (в словах были ошибки),

FN - количество пропущенных слов (в словах были ошибки),

TN - количество пропущенных слов (в словах не было ошибки) .

МНОГОКЛАССОВАЯ КЛАССИФИКАЦИЯ

В случае многоклассовой классификации задача классификации на *K* классов разбивается на *K* задач об отделении класса *i* от остальных $(i = 1, \ldots, K)$, для каждой из задач можно посчитать свою матрицу ошибок. Затем есть два варианта получения итогового значения метрики из K матриц ошибок:

1) Усредняем элементы матрицы ошибок (TP, FP, TN, FN) между бинарными классификаторами, например, $TP = \frac{1}{K}\sum_{i=1}^{K} TP_i$. Затем по одной усредненной матрице считаем *Precision*, *Recall*, *Accuracy*. Одним словом, *микроусреднение*.

2) Считаем Precision, Recall, Accuracy для каждого классификатора отдельно, а потом усредняем, одним словом, *макроусреднение*.

СРАВНЕНИЕ МОДЕЛЕЙ

Посмотрим количественные оценки различных моделей, использующих *TF-IDF* векторизацию на словаре в 10000 слов:

Метод классификации	Тип токенизации	Accuracy	Precision	Recall
KNeighbors	ACBT	0.88	0.96	0.87
KNeighbors	CBT	0.80	0.87	0.80
Decision Tree	ACBT	0.89	0.96	0.88
Decision Tree	CBT	0.61	0.53	0.52
Naive Bayes	ACBT	0.84	0.90	0.80
Naive Bayes	CBT	0.60	0.67	0.53

НАБЛЮДЕНИЯ

1) Заметен прирост точности модели при использовании *ACBT* токенизации;

2) В силу особенностей моделей, основанных на машинном обучении, предсказания модели генерируются быстрее, чем предсказания моделей, основанных на переборе:

```
start = time.time()
pred = model_pipeline.predict(X_test)
end = time.time()
duration = round(end - start, 2)
print("Testing:", (duration), "secs")
print("Average time (ML):", (duration)/len(X_test), "secs")
```

✓ 4.7s

```
Testing: 4.62 secs
Average time (ML): 2.9786465855168148e-05 secs
```

```
import time
start = time.time()
pred = []
for i in X_test:
    pred.append(JDreco([i])[0])
end = time.time()
duration = round(end - start, 4)
print("Average time (JDreco):", (duration)/len(X_test), "secs")
```

✓ 30.4s

```
Average time (JDreco): 0.05863378378378378 secs
```

```
pred = []

start = time.time()
for i in X_test:
    pred.append(levenstein([i])[0])
end = time.time()
duration = round(end - start, 4)
print("Average time (levenstein):", (duration)/len(X_test), "secs")
```

✓ 4m 14.2s

```
Average time (levenstein): 0.4907776061776062 secs
```

3) Модель способна исправлять слова, в которых было допущено больше ошибок, чем в словах, на которых производилось обучение (в нашем случае - расстояние 1):

```
print(model_pipeline.predict(['гийпфруту']))
```

✓ 0.5s

```
['грейпфрут']
```

```
print(model_pipeline.predict(['нчо']))
```

✓ 0.5s

```
['ночь']
```

ВЛОВЛВ

```python
def advanced_levenstein(_word):
    _new_word = levenstein([_word])[0]
    distance = edit_distance(_word, _new_word)
    if(len(_word) > 2):
        split = splits(_word)[1:-1]
        for sp in split:
            sp_corrected = levenstein(sp)
            _new_word_split = ''
            for i in sp_corrected:
                _new_word_split += i + ' '
            _new_word_split = _new_word_split[:-1]
            if edit_distance(_new_word_split, _word) <= distance:
                _new_word = _new_word_split
                distance = edit_distance(_new_word_split, _word)

    return _new_word
```

```python
advanced_levenstein_cont('bcyjdf cnlfzyjxnm')
```

```
'и снова седая ночь'
```

```python
eng_rep = ('f,dult`;pbqrkvyjghcnea[wxio]sm\'.z')
```

```python
def advanced_levenstein_cont(sent):
    if sent[0] not in alphabet:
        sent_new = ''
        for i in range(len(sent)):
            if sent[i] == ' ':
                sent_new += ' '
                continue
            else:
                sent_new += alphabet[eng_rep.find(sent[i])]
        sent = sent_new
    res = ''
    _words = tokens(sent)
    for i in _words:
        res += str(advanced_levenstein(i)) + ' '
    return res[:-1]
```

```python
advanced_levenstein_cont('вканце синтибря стаяла чюдесная пагода')
```

```
'в конце сентября стояла чудесная погода'
```

ОБЗОР ДВУХ КОРРЕКТОРОВ:

- Все модели работоспособны.
- Существенно отличаются по времени работы (ML корректор намного быстрее).
- Несмотря на быстрые предсказания, модели на *ML* обучаются настолько долго, что обработать словарь с более, чем 10000 слов нам не удалось (учитываются формы слов, всего в русском языке порядка 2 миллионов слов).
- Модели, основанные на переборе ошибок, не могут исправить слова, в которых допущено более двух ошибок (иначе, время обработки значительно возрастет)
- Модели, основанные на метриках Жаккара и Левенштейна, имеют точность, сопоставимую с моделями на машинном обучении.

НЕУДАЧНЫЕ ДУБЛИ

```
[368]
                    Вы ввели: молчать
Возможно, Вы имели в виду: сказать
```

— Дшктор, нто сл мнрй?
— У вас опечатка мозга.
— Энто лечтся?
— Надеюсь. Пропишу вам Т9
— Селёдка, доктор, так гнездо лучше.

```python
def editreco(entries = ['мома', 'мфла', 'рму']):
    outcomes = []
    for entry in entries:
        distances = ((edit_distance(entry, word), word) for word in correct_spellings)
        closest = min(distances)
        outcomes.append(closest[1])

    return outcomes



print(editreco('чуство'))
```

✓ 0.6s

```
['чай', 'дух', 'лес', 'вот', 'ваш', 'бог']
```

РАСПРЕДЕЛЕНИЕ РАБОТЫ

Серафим — корректор на основе машинного обучения
Даша — корректор, основанный на переборе
Даниил — корректор на основе машинного обучения и корректор, основанный на переборе