Национальный исследовательский университет
“Московский энергетический институт”

Отчет по курсовой работе

по дисциплине: «Программная инженерия»

Тема курсовой работы

«Программная реализации алгоритма извлечения прецедентов с использованием различных метрик»

Группа: А - 05 - 18

Студент: Старикова В.В

Научный руководитель: Варшавский П.В.

Москва 2020

Оглавление

Введение 2
Постановка задачи 2
Теоретическая часть 3
Алгоритм 5
Описание базы данных 5
Пример работы программы 7
Графическое представление работы классификатора KNN 8
Анализ зависимости выбора параметров алгоритма и точности классификатора 10
Вывод 16
Приложение 16
Литература 21

Введение

Метод решения проблем на основе прецедентов(**CBR - Case Based Reasoning**) – это подход, при котором решение принимается по аналогии с подобными случаями с известным исходом, иначе прецедентами.

CBR подход очень популярен, так как он интуитивно понятен, и может быть использован в разных сферах. Например, в медицине – врач ставит диагноз, опираясь на предыдущие случаи заболеваний[1].

Для использования этого метода необходима база прецедентов, на основе которой будет происходить автоматизированный поиск решения[2].

Процесс CBR состоят из четырех этапов[2] :

- *Извлечение* – наиболее похожего прецедента
- *Повторное использование* – использования прецедента для решение текущей проблемы
- *Исправление* – сопоставление полученного решения с текущей проблемой, тестирование в реальном мире и , при необходимости исправление.
- *Сохранение* – после успешного тестирование, добавление случая в базу данных прецедентов для решения новых проблем

Постановка задачи

Целью данной курсовой работы будет является :

- описание этапа извлечения прецедентов
- реализация метода ближайшего соседа с использованием различных метрик
- анализ параметров метода ближайшего соседа

- нахождение наиболее удачных параметров

Теоретическая часть

Целью этапа извлечения прецедента является поиск наиболее адекватного, подобного прецедента для текущей ситуации. Это происходит путем сравнивания текущего случая с данными из базы прецедентов. Сравнение может выполняться с помощью различных алгоритмов, самым популярным является метод ближайших соседей[2].

Для формализации степени сходства объектов в методе ближайших соседей вводится понятие метрики, функции расстояния между двумя точками. В соответствии с выбранной метрикой принимается решения для текущего случая на основе ближайших к нему прецедентов. Этот метод опирается на предположение о том, что схожим объектам, как правило, соответствуют схожие ответы. Это предположение называют *гипотезой компактности*[2-3].

К сожалению, метод ближайшего соседа является ненадежным, так как он не устойчив к погрешностям (объектам, находящимся в окружении объектов чужого класса) и не имеет параметров, настраиваемых по текущей БП, с целью повышения качества классификации[3].

В следствие этого вводится *весовая функция* w_i , которая оценивает степень важности i-го соседа.

При различных вариациях весовой функции получаются новые модифицированные методы[3-4]:

- *Метод k ближайших соседей KNN(k-Nearest Neighbors) – w(i, u)=[i<k]*

  Для принятия решения используется k ближайших соседей метода. Происходит "голосование", текущему случаю присваивается решение наибольшего количества голосов k соседей. Заранее неизвестно каким именно должно быть число k, чтобы эффективность метода была максимальной. Это устанавливается либо опытным путем, либо методом скользящего контроля[3-4].

- *Метод k взвешенных ближайших соседей WKNN*

  w(i) зависит от ранга i-го соседа

  $w_i = q^i, \quad q < 1$ - нелинейная

  $w_i = \frac{K+1-i}{K}$ - линейная

  w(i) зависит от расстояния

  $w_i = \frac{1}{\rho(u,x_i)}$

  Решение выбирают исходя из объекта с большим суммарным весом среди соседей k. Этот метод является решением проблемы KNN, когда одинаковое число соседей имеет разные решения. Стоит отметить , что эффективнее использовать нелинейную функцию весов k[3-6].

- *Метод парзеновского окна* $w(i,u) = K(\frac{\rho(u,x_{i,u})}{h})$ *с фиксированной ширины окна h*

  Вводится функция ядра, не возрастающая на [0, ∞], и принимается как весовая функция, которая зависит от расстояния ρ(u, xi, u) , а не от ранга соседа. Роль ширины окна h очень схожа с выбором k ближайших соседей. Если текущий случай попадает в «окно», он наследует решения объекта этого «окна»[3-4].

- *Метод парзеновского окна* $w(i,u) = K(\frac{\rho(u,x_{i,u})}{\rho(u,x_{k+1,u})})$

Фиксация ширины окна h плохо работает на данных, где объекты неравномерно распределены по пространству, так как метод просто не будет учитывать данные, которые не вошли в окно[3-4].

- *Метод задания порогового значение степени сходства* $1 - \frac{d_{CT}}{d_{max}}$

  Вводится специальная величина H пороговое значение степени сходства прецедентов и текущей проблемы. В результате сравнения выборка решения идет только среди тех соседей, чьи значения степени сходства больше или равны пороговому H [2].

Эффективность метрических методов очень сильно зависит от выбора метрики. Обычно метрика, дающая хороший результат изначально неизвестна, и приходится подбирать ее опытным путем.

Если речь идет о числовых данных, то обычно выбирают следующие метрики:

- Евклидова метрика

$d_{pq} = \sqrt{\Sigma(p_i - q_i)^2}$

- Квадрат Евклидова расстояния

$d_{pq} = \Sigma(p_i - q_i)^2$

- Расстояние Чебышева

$d_{pq} = max(|p_i - q_i|)$

- Косинусное расстояние

$d_{pq} = \frac{\Sigma(p_i q_i)}{\sqrt{\Sigma(p_i)^2}\sqrt{\Sigma(q_i)^2}}$

- Метрика Минковского

$d_{pq} = (\Sigma|p_i - q_i|^p)^{\frac{1}{p}}$, p>1

- Расстояние городских кварталов

$d_{pq} = |p_i - q_i|$

- Нормализованное расстояние Евклида

$d_{pq} = \sqrt{\Sigma\frac{(p_i - q_i)^2}{\sigma_i^2}}$

σ_i^2 - среднеквадратичное отклонение

- Расстояние Канберы

$d_{pq} = \sqrt{\Sigma\frac{|p_i - q_i|}{|p_i + q_i|}}$

Стоит помнить , что перед применением метрики, если диапазон значений параметров слишком велик, их следует нормализовать. Иначе значения с большими значениями будут сильнее влиять на результат метрики(например, объект A ∈ [0.1, 0.5], а объект B ∈ [1000, 5000][5].

Формула *нормализации* имеет следующий вид:

$x' = (x - minX)/(maxX - minX)$

Также стоить учитывать, что некоторые параметры объекта могут быть важнее остальных, поэтому существуют *взвешенные метрики*, которые учитывают веса параметров. Например, взвешенная метрика Евклида будет иметь вид [3]:

$$d_{pq} = \sqrt{\Sigma w_i (p_i - q_i)^2}$$

Алгоритм

Входные данные:

T - текущая ситуация, с заданными параметрами

CL - не пустое множество прецедентов из базы прецедентов

wi …wn - веса параметров

M - множество доступных метрик

q - множество весовых функции i-го соседа

Выходные данные:

SC - множество прецедентов, которые с заданными им весами

Описание алгоритма

В цикле по БП (база прецедентов) рассчитываем расстояние по выбранной метрике для каждого прецедента из БП Cj и T , с учетом коэффициентов важности параметров. Записать данное расстояние в массив данных учитывая весовую функцию для i-го соседа. После того, как все прецеденты из БП были рассмотрены, вернуть массив прецедентов с учетом их веса [2].

Описание базы данных

Для проверки работоспособности алгоритма KNN будем использовать базу данных «Ирисы Фишера».

Рисунок 3 iris setosa

Рисунок 2 iris virginica

Рисунок 1 iris versicolor

«Ирисы Фишера» состоят из данных о 150 экземплярах ириса, по 50 экземпляров из трёх видов — Ирис щетинистый (Iris setosa), Ирис виргинский (Iris virginica) и Ирис разноцветный (Iris versicolor)[7].

Для каждого экземпляра измерялись четыре характеристики (в сантиметрах):

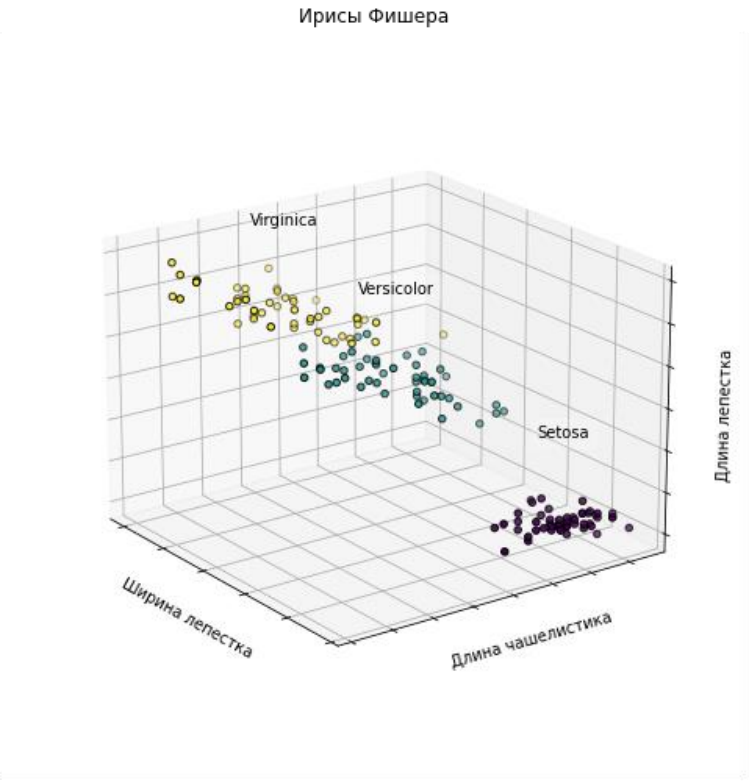
- Длина наружной доли околоцветника (англ. sepal length);

- Ширина наружной доли околоцветника (англ. sepal width);
- Длина внутренней доли околоцветника (англ. petal length);
- Ширина внутренней доли околоцветника (англ. petal width).

Пример записей в БЗ

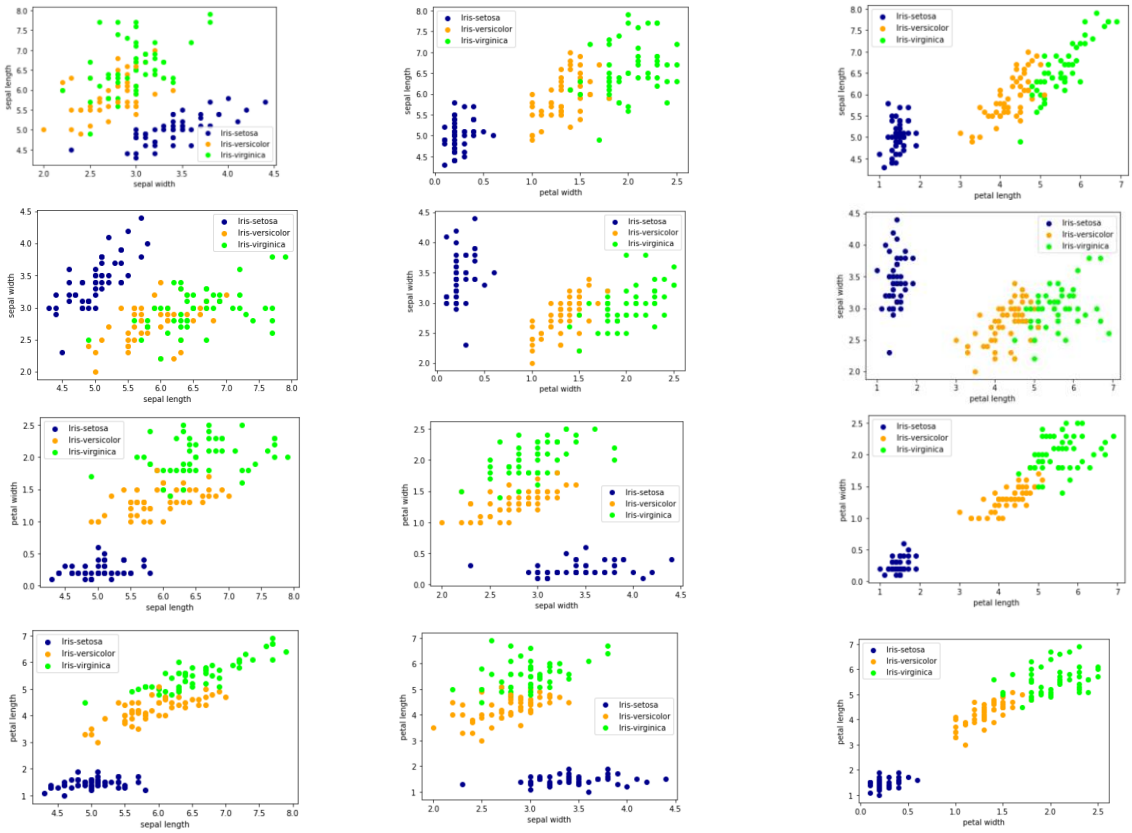
Длина чашелистика	Ширина чашелистика	Длина лепестка	Ширина лепестка	Вид ириса
5.1	3.5	1.4	0.2	*setosa*
4.9	3.0	1.4	0.2	*setosa*
4.7	3.2	1.3	0.2	*setosa*
…	…	…	…	…
7.0	3.2	4.7	1.4	*versicolor*
6.4	3.2	4.5	1.5	*versicolor*
6.9	3.1	4.9	1.5	*versicolor*
…	…	…	…	…
6.3	3.3	6.0	2.5	*virginica*
5.8	2.7	5.1	1.9	*virginica*
7.1	3.0	5.9	2.1	*virginica*
…	…	…	…	…

Для лучшего представления, опишем данные графически



Видно, что класс Iris-setosa линейно отделим от двух остальных[7-8].

Для большей наглядности построим графики зависимостей признаков, раскрашивая точки в зависимости от сортов Ириса.



Из графиков, видно что размеры лепестка (petal width) – это ярко выраженный классифицирующий признак. Petal Width находится в достаточной сильной зависимости с Petal length, так как точки на графиках расположены вдоль прямой линии. Также, они хорошо сгруппированы, поэтому по ним можно будет строить классификацию. Если рассматривать другие вариации признаков, то, например, с помощью пары sepal length и sepal width не удастся построить качественный классификатор, т.к точки классов Versicolor и Virginica перемешаны между собой [8]. Эти знания понадобятся нам во время графического представления работы классификатора.

Пример работы программы

Для демонстрации работы алгоритма базу ирисов случайным образом разделим на прецеденты и тестовую выборку в соотношении 75% на 25%.

Создадим список ответов для тестовой выборки, с помощью которого будем проверять точность работы классификатора.

Для примера будем использовать Евклидово расстояние $d_{pq} = \sqrt{\Sigma(p_i - q_i)^2}$,без весовых функций, нормализации данных и учета весов параметров. Количество соседей возьмем равным 5.

Тогда получим следующий результат:

№	sepal length	sepal width	petal length	petal width	Прогонозируемый вид ириса	Действительный вид ириса
1	4.6	3.1	1.5	0.2	Iris-setosa	Iris-setosa
2	5.0	3.6	1.4	0.2	Iris-setosa	Iris-setosa
3	5.4	3.7	1.5	0.2	Iris-setosa	Iris-setosa
4	5.8	4.0	1.2	0.2	Iris-setosa	Iris-setosa
5	5.1	3.3	1.7	0.5	Iris-setosa	Iris-setosa
6	5.5	4.2	1.4	0.2	Iris-setosa	Iris-setosa
7	5.0	3.5	1.3	0.3	Iris-setosa	Iris-setosa
8	5.0	3.5	1.6	0.6	Iris-setosa	Iris-setosa
9	5.1	3.8	1.6	0.2	Iris-setosa	Iris-setosa
10	5.7	2.8	4.5	1.3	Iris-versicolor	Iris-versicolor
11	5.9	3.2	4.8	1.8	Iris-versicolor	Iris-versicolor
12	6.6	3.0	4.4	1.4	Iris-versicolor	Iris-versicolor
13	6.8	2.8	4.8	1.4	Iris-versicolor	Iris-versicolor
14	5.5	2.4	3.7	1.0	Iris-versicolor	Iris-versicolor
15	6.0	2.7	5.1	1.6	Iris-virginica	Iris-versicolor
16	6.7	3.1	4.7	1.5	Iris-versicolor	Iris-versicolor
17	5.6	3.0	4.1	1.3	Iris-versicolor	Iris-versicolor
18	5.7	2.8	4.1	1.3	Iris-versicolor	Iris-versicolor
19	6.4	2.7	5.3	1.9	Iris-virginica	Iris-virginica
20	6.3	2.7	4.9	1.8	Iris-virginica	Iris-virginica
21	6.1	3.0	4.9	1.8	Iris-virginica	Iris-virginica
22	6.4	2.8	5.6	2.1	Iris-virginica	Iris-virginica
23	6.3	2.8	5.1	1.5	Iris-versicolor	Iris-virginica
24	6.1	2.6	5.6	1.4	Iris-virginica	Iris-virginica
25	7.7	3.0	6.1	2.3	Iris-virginica	Iris-virginica
26	6.4	3.1	5.5	1.8	Iris-virginica	Iris-virginica
27	6.0	3.0	4.8	1.8	Iris-versicolor	Iris-virginica
28	6.7	3.1	5.6	2.4	Iris-virginica	Iris-virginica
29	6.7	3.3	5.7	2.5	Iris-virginica	Iris-virginica
30	6.2	3.4	5.4	2.3	Iris-virginica	Iris-virginica

Число правильно классифицированных случаев: 27

Число неправильно классифицированных случаев: 3

Точность: 90%

Неправильно классифицированные случаи:

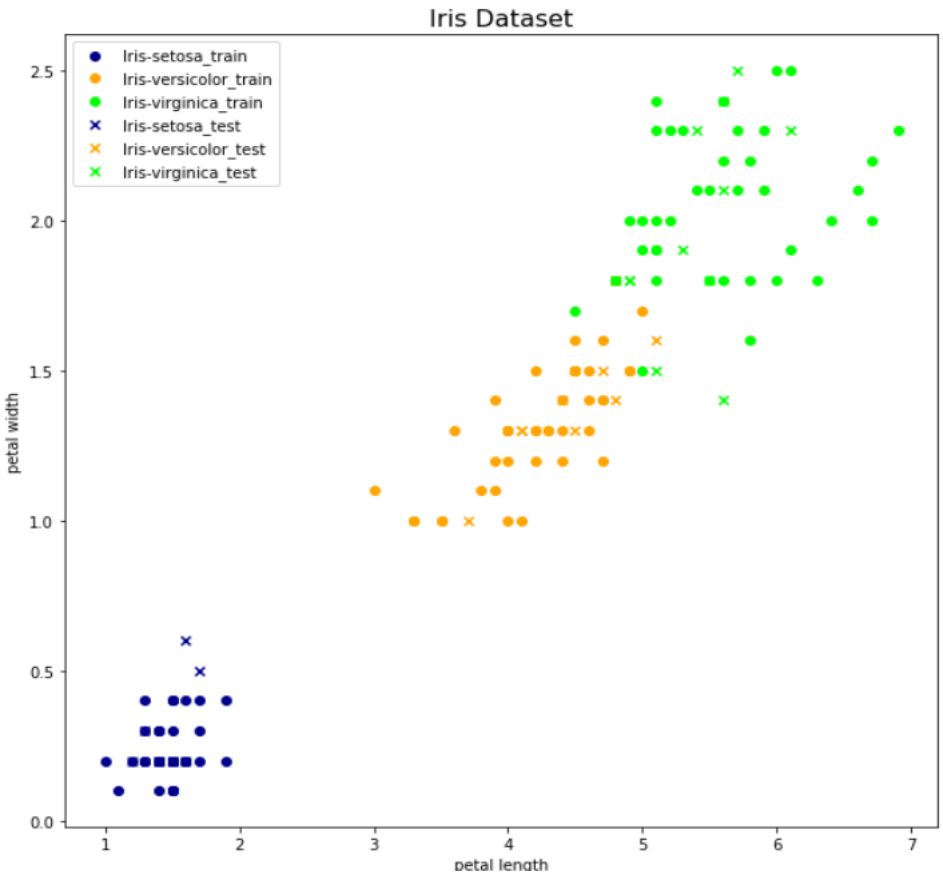
№	sepal length	sepal width	petal length	petal width	Прогонозируемый вид ириса	Действительный вид ириса
15	6.0	2.7	5.1	1.6	Iris-virginica	Iris-versicolor
23	6.3	2.8	5.1	1.5	Iris-versicolor	Iris-virginica
27	6.0	3.0	4.8	1.8	Iris-versicolor	Iris-virginica

Видно, что классификатор путает классы Versicolor и Virginica, параметры которых, как мы уже выяснили, могут быть очень схожи.

Графическое представление работы классификатора KNN

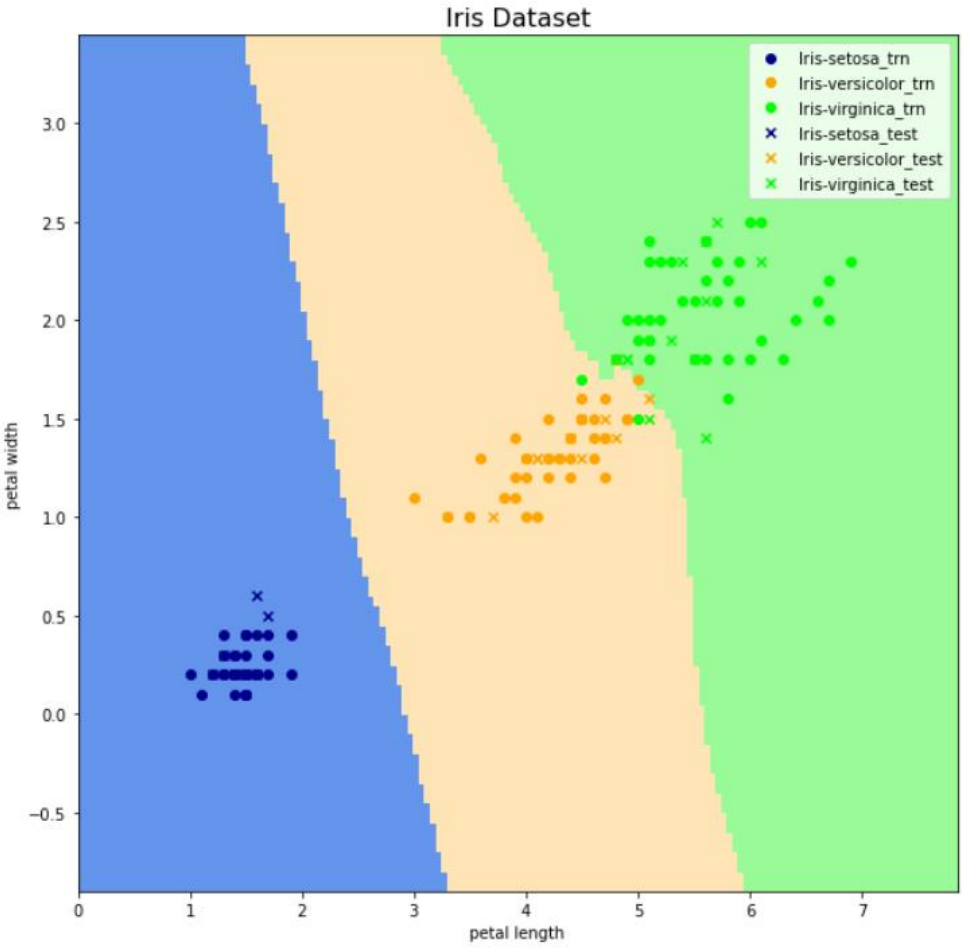
Графически задачу классификации данных можно рассматривать как поиск кривой, наиболее удачной разделяющей классы.

Для иллюстрации работы KNN будем использовать только два признака petal width (ось Oy)и petal length (ось Ox).



Отобразим данные. Точками обозначены прецеденты, крестиками тестовые данные.

Вот как справляется справляет алгоритм KNN с параметрами из предыдущего раздела.



Классификатор, как и ожидалось, отлично справляется с классом Setosa и строит практически прямую границу. А вот между классами Versicolor и Virginica, границей будет являться плавная кривая, которая, достаточно хорошо разделяет данные, но мы все равно видим ошибки, в виде зеленых точек на оранжевом фоне.

Анализ зависимости выбора параметров алгоритма и точности классификатора

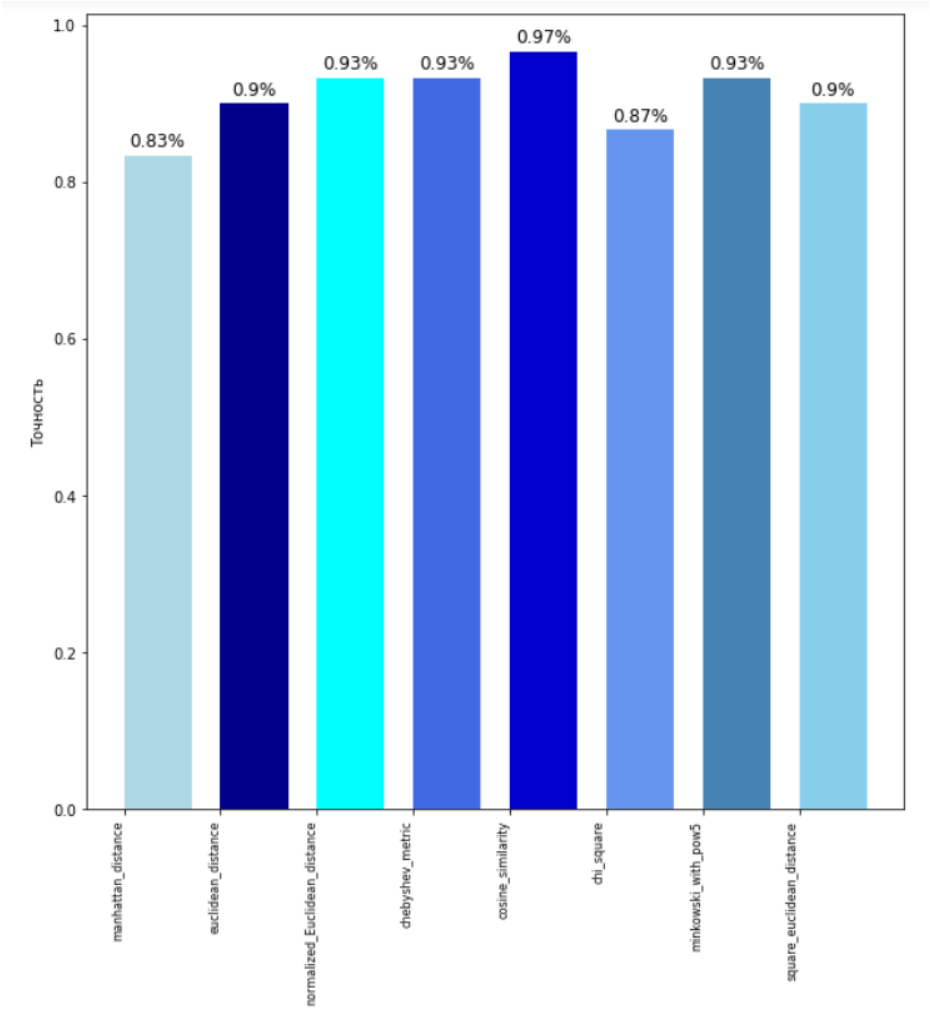
Напомним, что на эффективность алгоритма классификации KNN могут влиять следующие параметры:

- Количество соседей
- Метрика расстояния
- Веса соседей
- Веса параметров
- Нормализация данных

Целью данного раздела опытным путем найти наиболее удачные параметры алгоритма KNN для классификации «Ирисов Фишера».

Для начала выясним: какая метрика дает наибольшую точность классификация для числа соседей равным 5.

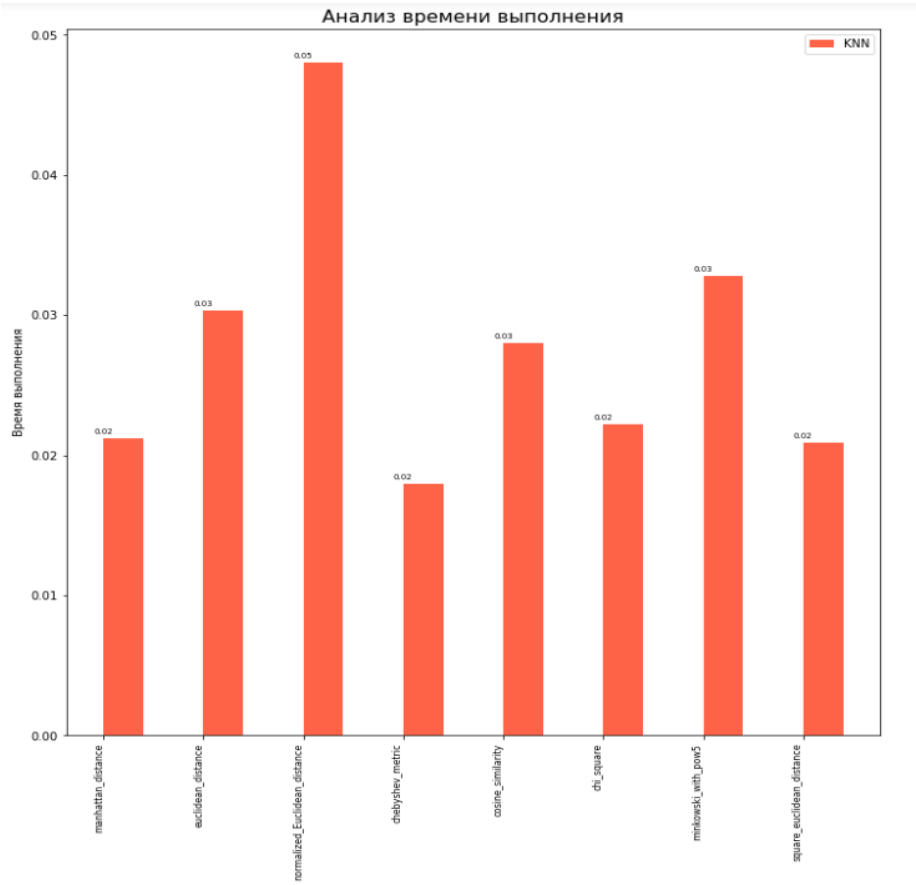
Для визуализации построим график гистограмм.



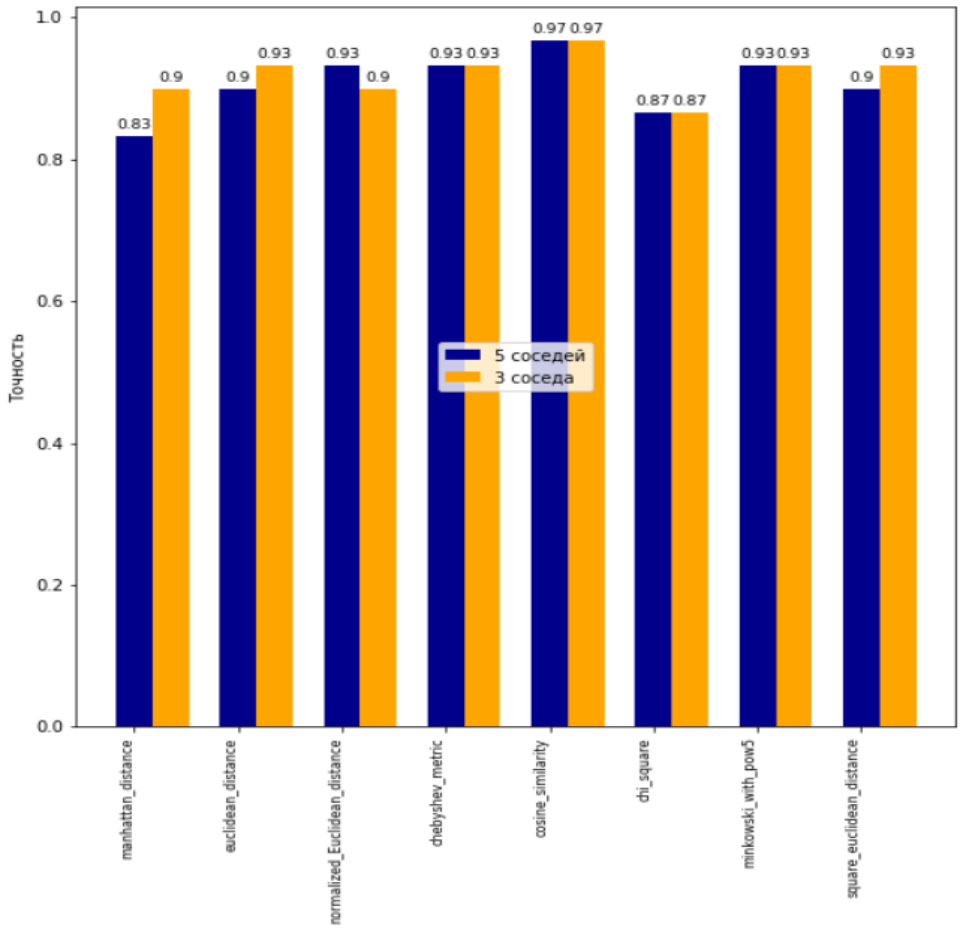
Максимальную точность 97% дает косинусное расстояние.

Если сравнивать время выполнения, то косинусная метрика работает не намного дольше остальных.

Самые эффективная по времени метрика - метрика Чебышева .

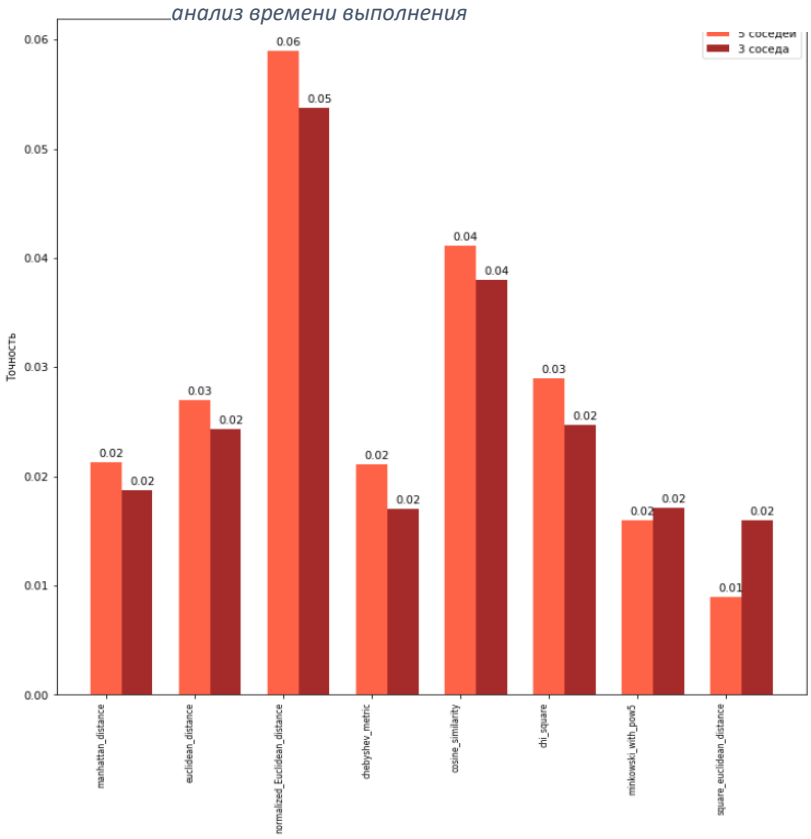


Теперь изменим число соседей на 3

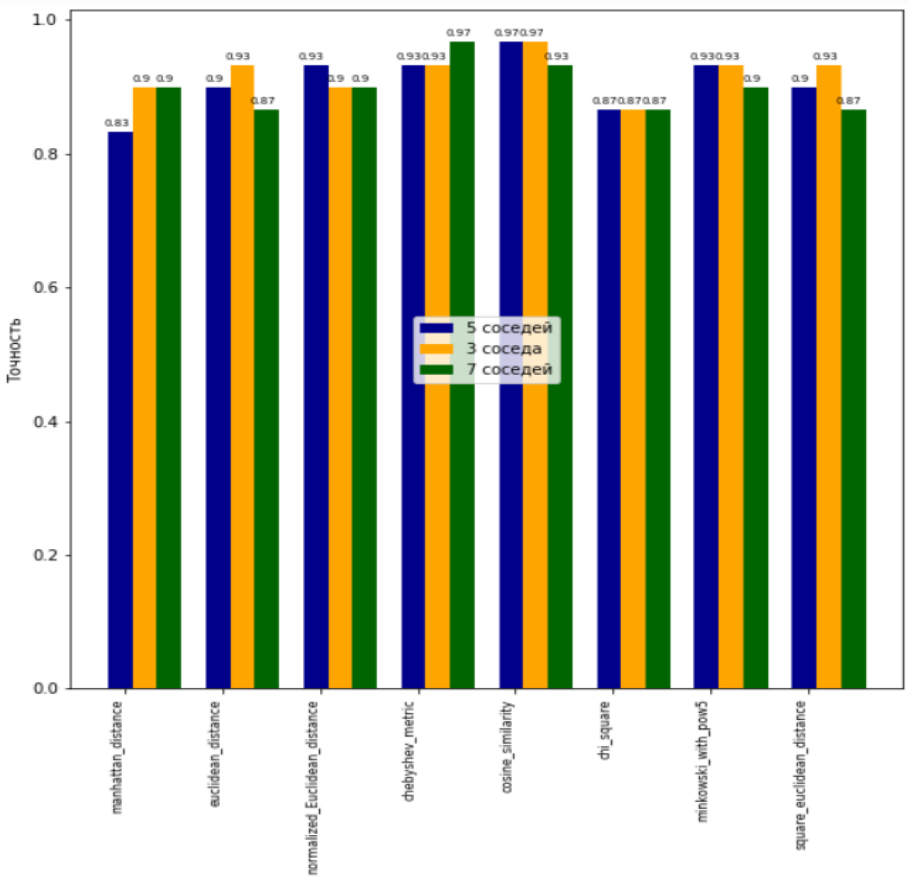


Три метрики улучшили свои показатели: манхэттенская, евклидова, квадрат евклидового расстояния. А вот максимальная точность осталась той же: 97% с косинусным расстоянием.

Что касается времени, то оно уменьшилось.



Попробуем увеличить количество соседей до 7.



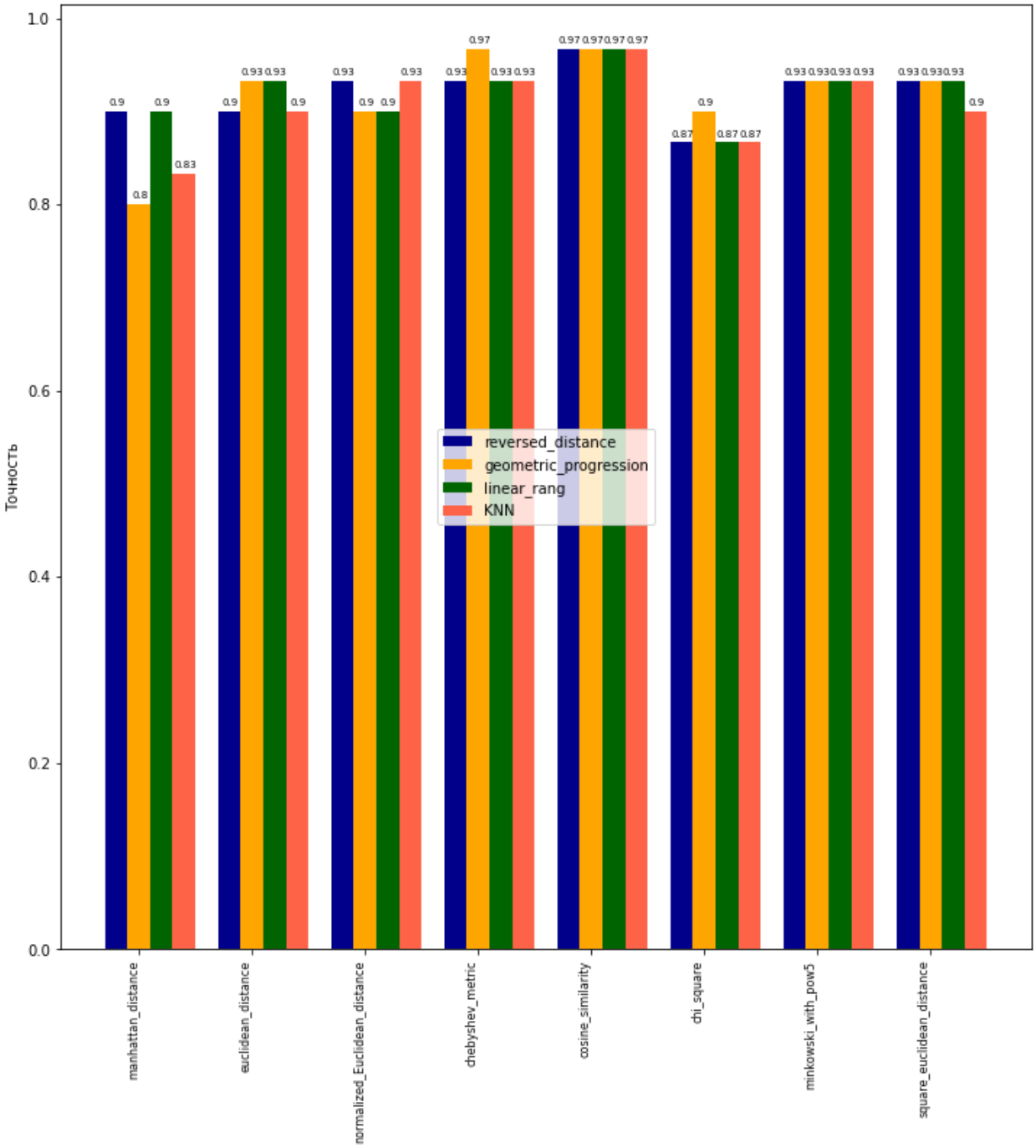
Теперь точности 97% достигает еще и метрика Чебышева. Ситуация по остальным метрикам следующая: половина осталась с той же точностью, что и при трех и пяти соседях (Манхэттенская, нормализованная Евклидова метрика, метрика Канберы), а половина ухудшила свою точность (евклидова, косинусная, метрика Минковского степени 5 и квадрат евклидова расстояния).

Можем сделать вывод, что оптимальным количеством соседей является число 3. Хотя при семи соседях, точность достигается на двух метриках сразу, но на других метриках показатели уменьшаются или остаются прежними, а времени и памяти для семи соседей понадобится больше.

Теперь выясним, как влияют на эффективность веса соседей.

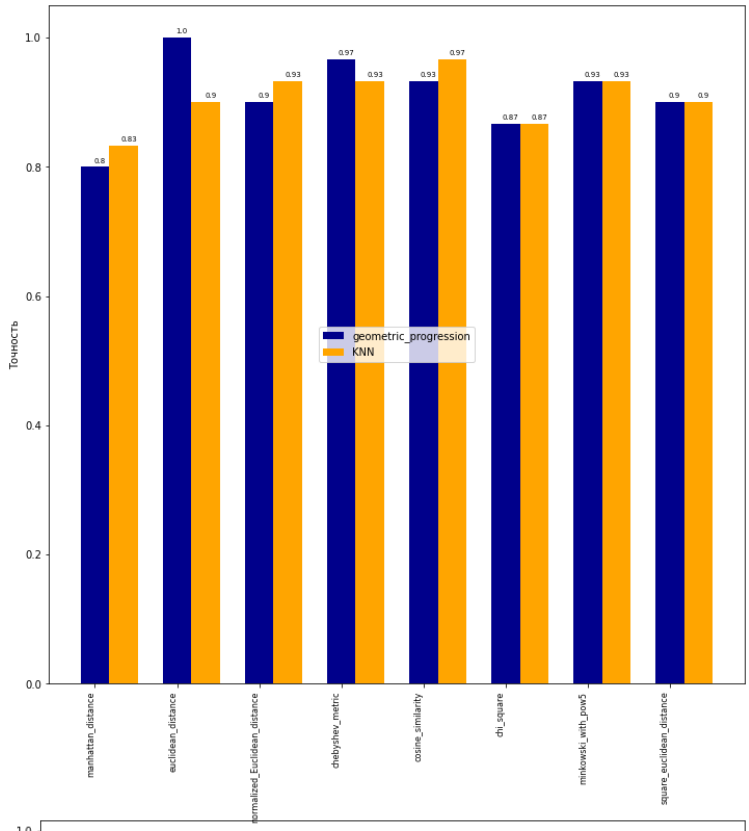
Параметры алгоритма

Количество соседей	5



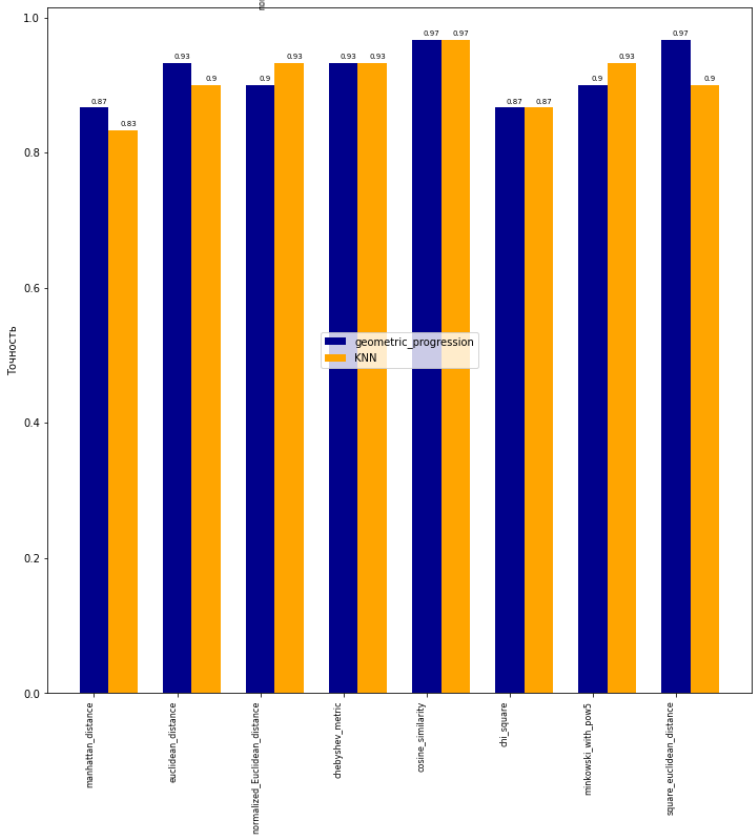
Из графика видно, что введение весов соседей положительно сказывается на эффективности алгоритма. Также, мы убедились, что нелинейные функции весов дают результат лучше, чем линейные. Хотелось бы обратить внимание на функцию $w_i = q^i, \qquad q < 1$ и рассмотреть ее отдельно.

По разному задавая значение q, можно добиться высокой точности для разных метрик.



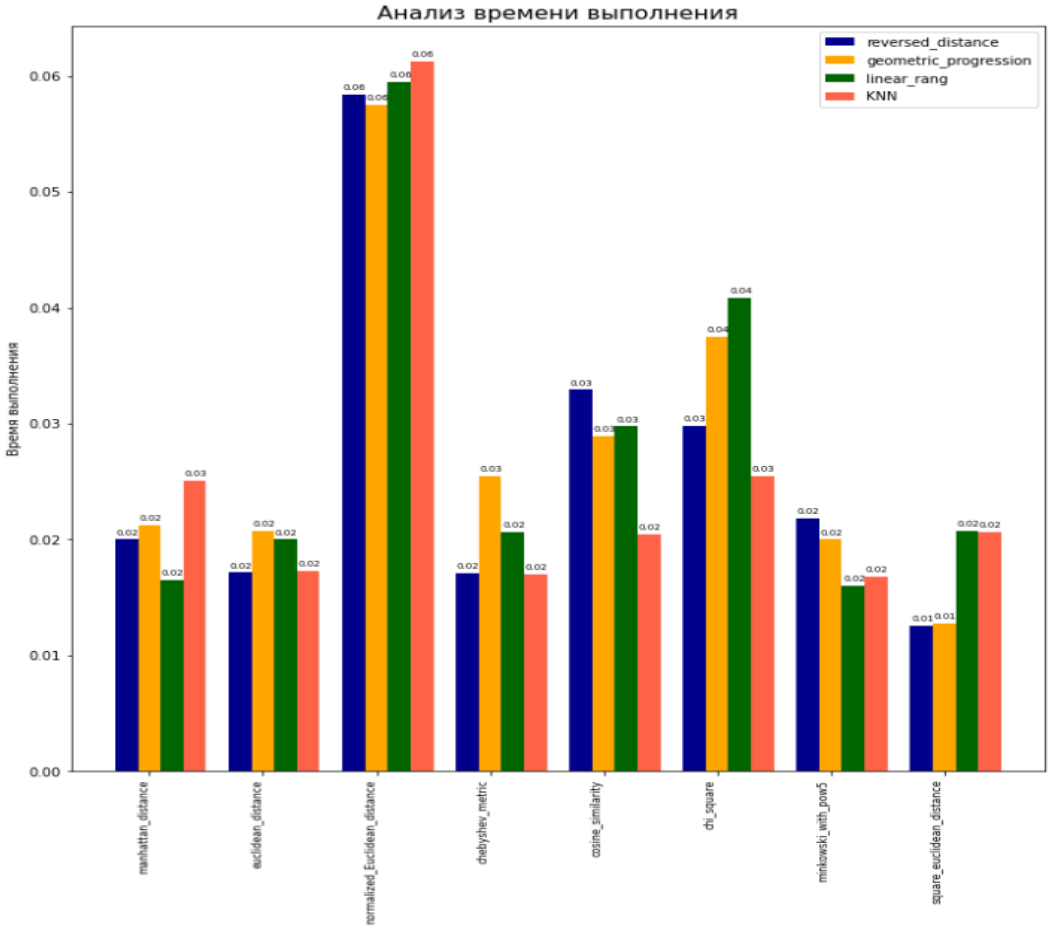
Евклидова метрика достигает точности 100%.

Метрика Чебышева достигает точности 97%.



Квадрат евклидова расстояния 97%

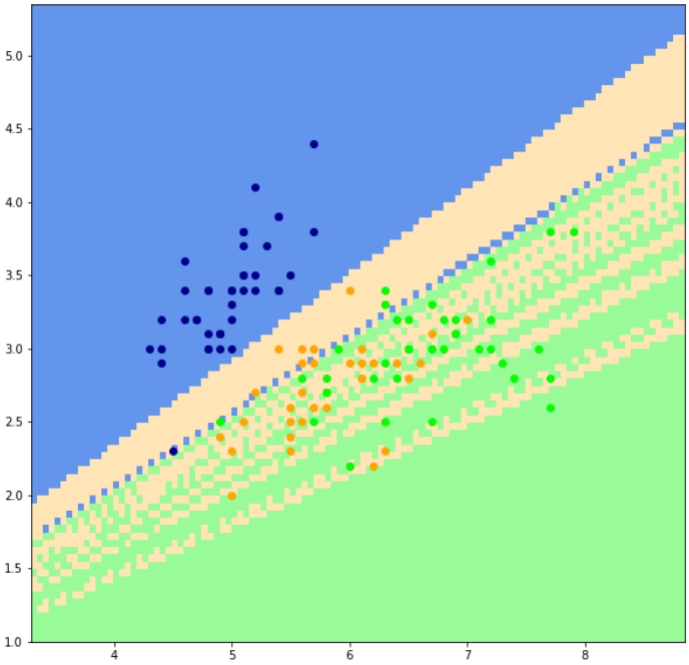
Сравним время работы классификатора с весами функции и без в разных метриках.



Результат для разных метриках различаются. Но все же большинство метрик склоняется к тому, что взвешенный алгоритм ближайших соседей требует больше времени на выполнение.

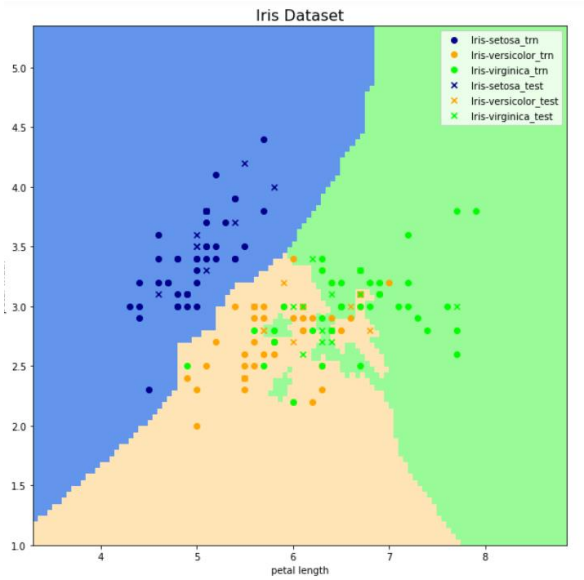
Теперь рассмотрим **графическое представление работы KNN** при разных метриках.

Косинусная метрика

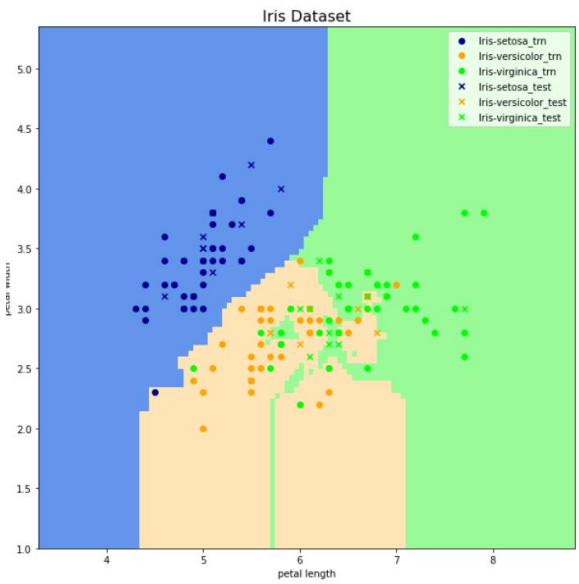


Данные классифицируются хорошо, но границы классов сложные и прерывистые. Скорее всего, новые данные будут классифицированы неправильно.

Метрика Минковского степени 5



Манхэттенская метрика



Вывод

В итоге, можно сказать, что метод ближайших соседей имеет свои плюсы и минусы. Среди плюсов: простота реализации, легкая интерпретация (можно объяснить, почему случай отнесен именно к этому классу), возможность адаптации под задачу(выбор метрики, весов и т.д) [9].

Но также в ходе анализа параметров алгоритма, мы убедились , как сложно найти удачную метрику, веса соседей, веса параметров к тому же это отнимает много времени.

Если говорить про «Ирисы Фишера», то удалось достигнуть точности **100%** с использованием **метрики Евклида** экспоненциально взвешенных ближайших соседей. Хорошо показала себя **косинусная метрика**, которая дает нам точность **97%**, без использования дополнительных параметров алгоритма.

Приложение

Код программы

Реализация алгоритма KNN

```
metrics_array = {'manhattan_distance': manhattan_distance,
                 'euclidean_distance': euclidean_distance,
                 'normalized_Euclidean_distance':normalized_Euclidean_distance,
                 'chebyshev_metric': chebyshev_metric,
                 'cosine_similarity':cosine_similarity,
                 'chi_square':chi_square,
                 'minkowski_with_pow5':minkowski_with_pow5,
                 'square_euclidean_distance':square_euclidean_distance}


def kNN(cases, precendents, num_neighbors, metric_name, weight_name, weights_of_parameters,
normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametrs_list):
    prediction_list = []
    for current_case in cases: #для каждого случая
        if normalize_data_fl: # нормализация данных
            current_case = normalize_vector(current_case, upper_bounds_for_parametrs_list,
lower_bounds_for_parametrs_list)
```

```
        prediction_list.append(k_nearest_neighbors(current_case, precendents, num_neighbors, metric_name,
weight_name,
weights_of_parameters,normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametrs_list))# по
иск и добавление решения в список
    return prediction_list
#-----------------------------------------------------------------------------------------------------------------------------------------------------------

def k_nearest_neighbors(current_case, precendents, num_neighbors, metric_name, weight_name,
weights_of_parameters,normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametrs_list):
    distances = []

    for precendent in precendents: # вычисляем расстояние до каждого прецедента в БП
        answer = precendent[-1]
        # нормализация данных
        if normalize_data_fl:
            precedent_param = normalize_vector(precendent, upper_bounds_for_parametrs_list,
lower_bounds_for_parametrs_list)
        else:
            precedent_param = precendent
        dist = metrics_array[metric_name](precedent_param, current_case, weights_of_parameters)
        distances.append((answer, dist)) #класс и расстояние
    distances.sort(key=lambda tup: tup[1]) #сортируем по возрастанию

    neighbors = []
    if weight_name != '': # взвешенный knn
        weights = make_weights(distances[:num_neighbors],weight_name)
        for i in range(num_neighbors):
                neighbors.append((distances[i][0],weights[i])) #distances[i][0][-1] здесь название класса, получается
список из кортежей, где на первом месте класс, на втором вес
        prediction = voting(neighbors) # классифицируем
    else:
        for i in range(num_neighbors):
                neighbors.append((distances[i][0],1)) #везде одинаковый вес
        # голосование
        output_values = [neib[0] for neib in neighbors]
        prediction = max(set(output_values), key=output_values.count) #текущему случаю присваивается решение
наибольшего количества голосов k соседей
    return prediction
#-----------------------------------------------------------------------------------------------------------------------------------------------------------

def make_weights(dist,weight_name):
    result = np.zeros(len(dist))
    sum = 0.0
    if weight_name == 'reversed_distance':
        for i in range(len(dist)):
            result[i] =reversed_distance(dist[i])
            sum += result[i]
    if weight_name == 'geometric_progression':
        for i in range(len(dist)):
            result[i] = geometric_progression(i)
            sum += result[i]
    if weight_name == 'linear_rang':
        for i in range(len(dist)):
            result[i] = linear_rang(i,len(dist))
            sum += result[i]
    return result / sum
#-----------------------------------------------------------------------------------------------------------------------------------------------------------

def voting(neighbors): #Решение выбирают исходя из объекта с большим суммарным весом среди
соседей k
    votes = {} # словарь, где ключ - класс, значение - сумма значений весов
    for neib in neighbors:
        if (votes.get(neib[0]) == None):
            votes[neib[0]] = neib[1]
        else:
```

```
            votes[neib[0]] += neib[1]
    lst = list(votes.items())
    lst.sort(key = lambda i: i[1],reverse=True) # сортировка по возрастанию суммы весов
    return lst[0][0]
#-------------------------------------------------------------------------------------------------------------------------------------------------------
def normalize_vector(vector, upper_bounds_for_parametrs_list, lower_bounds_for_parametrs_list):
    normalized_vector = []
    for param,max_param,min_param in zip(vector, upper_bounds_for_parametrs_list,
lower_bounds_for_parametrs_list):
            normalized_vector.append((param - min_param ) / (max_param- min_param ))
    return normalized_vector
```

Метрики

```
import numpy as np
from math import sqrt , pow
#-------------------------------------------------------------------------------------------------------------------------------------------------------
def euclidean_distance(current_case, precedent,weights_of_parameters):
    distance = 0.0

    for i_param_current_case,i_param_precedent,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        distance += pow(weight_of_param*(i_param_current_case - i_param_precedent),2)
    return sqrt(distance)

#-------------------------------------------------------------------------------------------------------------------------------------------------------
def square_euclidean_distance(current_case, precedent,weights_of_parameters):
    distance = 0.0

    for i_param_current_case,i_param_precedent,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        distance += pow(weight_of_param*(i_param_current_case - i_param_precedent),2)
    return distance

#-------------------------------------------------------------------------------------------------------------------------------------------------------
def manhattan_distance(precedent, current_case,weights_of_parameters):
    result_distance = 0.0

    for i_param_precedent, i_param_current_case,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        result_distance += weight_of_param * abs(i_param_precedent - i_param_current_case)

    return result_distance

#-------------------------------------------------------------------------------------------------------------------------------------------------------
def chebyshev_metric(precedent, current_case,weights_of_parameters):
    result_distance, current_distance = 0.0, 0.0

    for i_param_precedent, i_param_current_case,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        current_distance = abs(weight_of_param*(i_param_precedent - i_param_current_case))

        if current_distance > result_distance:
            result_distance = current_distance

    return result_distance


def normalized_Euclidean_distance(precedent, current_case, weights_of_parameters):
    norm_data_1,norm_data_2,n_dist = 0,0,0

    for i_param_precedent, i_param_current_case, weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        norm_data_1 += pow(i_param_precedent * weight_of_param, 2)
```

```
        norm_data_2 += pow(i_param_current_case * weight_of_param, 2)

    for i_param_precedent, i_param_current_case, weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        n_dist += abs(weight_of_param*(i_param_precedent/norm_data_1 - i_param_current_case /norm_data_2))

    return np.sqrt(n_dist)
#-----------------------------------------------------------------------------------------------------------------------------------------------------
def cosine_similarity(precedent, current_case, weights_of_parameters):
    mult, norm_a, norm_b = 0,0,0
    for i_param_precedent, i_param_current_case, weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        mult += i_param_precedent * i_param_current_case * weight_of_param
        norm_a += pow(i_param_precedent * weight_of_param, 2)
        norm_b += pow(i_param_current_case * weight_of_param, 2)

    return (1-mult/(sqrt(norm_a)*sqrt(norm_b)))
#-----------------------------------------------------------------------------------------------------------------------------------------------------
def chi_square (precedent, current_case,weights_of_parameters):
    result_distance = 0.0

    for i_param_precedent, i_param_current_case,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        result_distance += abs(weight_of_param * (i_param_precedent - i_param_current_case)) /
abs(weight_of_param * (i_param_precedent + i_param_current_case))

    return np.sqrt(result_distance)
#-----------------------------------------------------------------------------------------------------------------------------------------------------
def minkowski_with_pow5(precedent, current_case, weights_of_parameters):
    result_distance = 0

    for i_param_precedent, i_param_current_case,weight_of_param in zip(precedent,
current_case,weights_of_parameters):
        result_distance += pow(weight_of_param * abs(i_param_precedent - i_param_current_case),5)

    return pow(result_distance, 1/5)
```

Вывод графиков

```
accuracy_5, accuracy_3 = [], []
time_5, time_3 = [], []

for metric in metrics_array:
    start_time = time.time()
    predictions = kNN(cases, precedents, 5, metric, '',
weights_of_parameters,normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametrs_list)
    time_3.append(time.time() - start_time)
    accuracy_3.append(accuracy_check(answer_of_test_cases, predictions)[0])

    start_time = time.time()
    predictions = kNN(cases, precedents, 3, metric, '',
weights_of_parameters,normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametrs_list)
    time_5.append(time.time() - start_time)
    accuracy_5.append(accuracy_check(answer_of_test_cases, predictions)[0])



color = ['lightblue','darkblue','aqua','royalblue','mediumblue','cornflowerblue','steelblue','skyblue']
color_ = ['mediumvioletred','deeppink','hotpink','crimson','pink','purple','darkviolet','tomato']
x = np.arange(len(accuracy_5))
width =0.35
```

```
fig, ax = plt.subplots(figsize=(12,12))

rects2 = ax.bar(x - width/2, time_3, width,color = 'tomato' , align='edge',label = '5 соседей')
rects4 = ax.bar(x + width/2, time_5, width,color = 'brown' , align='edge',label = '3 соседа')
# Add some text for labels, title and custom x-axis tick labels, etc.
'''rects2 = ax.bar(x - width/2, accuracy_5, width,color = 'darkblue' , align='edge',label = 'geometric_progression')
rects4 = ax.bar(x + width/2, accuracy_3, width,color = 'orange' , align='edge',label = 'KNN')
# Add some text for labels, title and custom x-axis tick labels, etc.'''

ax.set_ylabel('Точность')
ax.set_xticks(x)
ax.set_xticklabels(metrics_array.keys(),rotation=90,horizontalalignment='right', fontsize=8)
ax.legend()
def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{}'.format(round(height,2)),
                    xy=(rect.get_x() + rect.get_width() / 2 + 0.06 , height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom',fontsize=10)


autolabel(rects2)
autolabel(rects4)
plt.show()
```

Вывод графического представления KNN

```
num_neighbors = 5

pad = 1
x_min, x_max = lower_bounds_for_parametrs_list[0] - pad,upper_bounds_for_parametrs_list[0] + pad
y_min, y_max = lower_bounds_for_parametrs_list[1] - pad,upper_bounds_for_parametrs_list[1] + pad
h = 0.05
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
precedents = make_list_from_tuple(get_data_from_base(cur,"SELECT {0} FROM {1} WHERE
sampleCode = 'train'".format('one, two, col4',table_name)))
predictions = kNN(zip(xx.ravel(),yy.ravel()), precedents, 3, 'minkowski_with_pow5', weight_function,
weights_of_parameters,normalize_data_fl,lower_bounds_for_parametrs_list,upper_bounds_for_parametr
s_list)
prediction_s = [clss[pred] for pred in predictions]
prediction_s = np.array(prediction_s).reshape(xx.shape)
cmap_light = ListedColormap(['moccasin','cornflowerblue', 'palegreen'])

f = plt.figure(figsize=(10, 10))

plt.pcolormesh(xx, yy, prediction_s, cmap=cmap_light)


precedents_labels =np.array( get_answer(precedents))
cases_labels = np.array( get_answer(cases))
precedents = make_list_from_tuple(get_data_from_base(cur,"SELECT {0} FROM {1} WHERE
sampleCode = 'train'".format('one, two',table_name)))
cases = make_list_from_tuple(get_data_from_base(cur,"SELECT {0} FROM {1} WHERE sampleCode =
'test'".format('one, two',table_name)))

#выводим график точек где крестик это тестовые данные, а о обучающие
```

```
colours =['orange', 'darkblue', 'lime']
legend = ['Setosa', 'Versicolour', 'Virginica']
classes = list(set(precedents_labels))



for i in classes:
    idx = np.where(precedents_labels == i)
    points = np.array([precedents[j] for j in idx[0]])

    plt.scatter(points[:,0],
                points[:,1],
                c=colours[clss[i]],
                label=i + '_trn')
for i in classes:
    idx = np.where(cases_labels == i)
    points = np.array([cases[j] for j in idx[0]])

    plt.scatter(points[:,0],
                points[:,1],
                c=colours[clss[i]],
                label=i + '_test',
                marker='x')
plt.legend()
plt.title('Iris Dataset', fontsize=16)
plt.xlabel('petal length')
plt.ylabel('petal width')
plt.show()
```

Функции оценки точности

```
def get_answer(cases): # создает список ответов на тестовую выборку
            answer_list = []
            for case in cases:
                        answer_list.append(case[-1])
            return answer_list
answer_of_test_cases = get_answer(cases)

def accuracy_check(answer_list, predictions_list): # проверка точности
    count = 0
    idx = []
    i = -1
    for answer,prediction in zip(answer_list,  predictions_list):
        i +=1
        if answer == prediction:
            count += 1
        else:
            idx.append(i)

    return count/len(answer_list), idx,count
```

Литература

1. Рассуждения на основе прецедентов https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%81%D1%83%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D1%8F_%D0%BD%D0%B0_%D0%BE%D1%81%D0%BD%D0%BE%D0%B2

%D0%B5_%D0%BF%D1%80%D0%B5%D1%86%D0%B5%D0%B4%D0%B5%D0%BD%D1%82%D0%BE%D0%B2

2. П. Р. Варшавский, А. П. Еремеев Моделирование рассуждений на основе прецедентов в интеллектуальных системах поддержки принятия решений. Искусственный интеллект и принятие решений, 1/2009

3. Метрические алгоритмы классификации К. В. Воронцов
http://machinelearning.ru/wiki/images/8/8f/Voron-ML-Metric1.pdf

4. Метод ближайших соседей
http://www.machinelearning.ru/wiki/index.php?title=KNN

5. Метод k-ближайших соседей
https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_k-%D0%B1%D0%BB%D0%B8%D0%B6%D0%B0%D0%B9%D1%88%D0%B8%D1%85_%D1%81%D0%BE%D1%81%D0%B5%D0%B4%D0%B5%D0%B9

6. Реализация метода взвешенного соседа
https://visualstudiomagazine.com/articles/2019/04/01/weighted-k-nn-classification.aspx

7. Ирисы Фишера
https://ru.wikipedia.org/wiki/%D0%98%D1%80%D0%B8%D1%81%D1%8B_%D0%A4%D0%B8%D1%88%D0%B5%D1%80%D0%B0

8. Машинное обучение: от Ирисов до Телекома
https://habr.com/ru/company/billing/blog/334738/

9. Открытый курс машинного обучения. Тема 3. Классификация, деревья решений и метод ближайших соседей
https://habr.com/ru/company/ods/blog/322534/#vvedenie