МГУ им.Ломоносова
Факультет ВМК кафедра ММП

Метрические алгоритмы классификации

Батшева Анастасия 317 группа

Москва
2020

Содержание

1 Теоретическая постановка задачи **2**

2 Метод ближайших соседей и его обобщения **2**
- 2.1 Алгоритм 1 ближайшего соседа . . . 2
- 2.2 Алгоритм k ближайших соседей . . . 2
- 2.3 Алгоритм k взвешенных ближайших соседей . . . 3
- 2.4 Выбор метрики . . . 3

3 Формулировка задания **3**

4 Список экспериментов **4**

5 Реализация **5**
- 5.1 MNIST . . . 5
- 5.2 Модули и классы . . . 6
- 5.3 Модуль nearest neighbors и класс KNNClassifier . . . 6
  - 5.3.1 Конструктор . . . 6
  - 5.3.2 Метод fit . . . 7
  - 5.3.3 Метод find-kneighbors . . . 7
  - 5.3.4 Метод predict . . . 7
- 5.4 Модуль cross-validation . . . 8
- 5.5 Модуль distances . . . 9

6 Обработка результатов **10**
- 6.1 Определение 5 ближайших соседей . . . 10
- 6.2 Оценки качества для различных моделей . . . 12
- 6.3 Подсчет точности лучшего алгоритма . . . 13
- 6.4 Объекты, на которых алгоритм ошибся . . . 14
- 6.5 Подбор параметров для повышения качества классификации . . . 15
  - 6.5.1 Вращение цифр . . . 15
  - 6.5.2 Сдвиги цифр . . . 15
  - 6.5.3 Применение фильтра . . . 15

7 Итоги **16**

Метрические алгоритмы классификации используются для решения широкого класса прикладных задач, в которых естественным образом возникает понятие сходства объектов. В основе этих алгоритмов лежит *гипотеза компактности* – предположение о том, что схожие объекты, как правило, лежат в одном классе. Самым непосредственным выражением гипотезы компактности является алгоритм k ближайших соседей kNN, относящий распознаваемый объект к тому классу, которому принадлежит большинство из k ближайших к нему объектов обучающей выборки. Для формализации понятия «сходства» вводится функция расстояния или метрика $\rho(x_1, x_2)$ в пространстве объектов X.

1 Теоретическая постановка задачи

Имеется пространство объектов X и конечное множество имён классов Y. На множестве X задана функция расстояния - метрика $\rho : XX\text{ß}[0, \infty)$. Существует целевая зависимость $y^* : X \to Y$, значения которой известны только на объектах обучающей выборки $X^l = (x_i, y_i)_{i=1}^l, y_i = y^*(x_i)$. Требуется построить алгоритм классификации $a : X \to Y$, аппроксимирующий целевую зависимость $y^*(x)$ на всём множестве X. Обозначим разность всего множества X и обучающей выборки за тестовую выборку.

2 Метод ближайших соседей и его обобщения

Для произвольного объекта из тестовой выборки $x \in test_X$ расположим элементы обучающей выборки $x_1, ..., x_n$ в порядке возрастания расстояний до x: $\rho(x, x_1)\rho(x, x_2)...\rho(x, x_l)$ где через $x_{i,x}$ обозначается i-й сосед объекта x. Аналогичное обозначение введём и для ответа на i-м соседе: $y_i, u = y^{(x_i,x)}$.

2.1 Алгоритм 1 ближайшего соседа

Классифицируемый объект xX^l относится к тому классу, которому принадлежит ближайший обучающий объект: $a(x) = y_{1,x}$. Обучение сводится к элементарному запоминанию выборки X^l.

2.2 Алгоритм k ближайших соседей

Классифицируемый объект x относится к тому классу, которому принадлежит большинство его ближайших соседей: $a(x, k) =_{y \in Y, i=1} X[y_{i,x} = y]$ На практике оптимальное значение параметра k можно определить методом кросс-валидации: перекрестная проверка, когда обсучающая выборка делится произвольным образом на nfolds групп (одинаковой размерности), после чего алгоритм для каждого k тестируется на каждой из групп, которая по очереди становится тестовой выборкой, а оставшиеся именуются валидационной или обучающей. Результаты алгоритма сравниваются с имеюшимися ответами и выводится точность алгоритма с конкретным k. Выбрав наибольшую точность определим оптимальное k.

2.3 Алгоритм k взвешенных ближайших соседей

В алгоритм к ближайших соседей дополнительно вводится строго убывающая последовательность вещественных весов w_i, задающих вклад i-го соседа в классификацию: $a(x,k) =_{y \in Y, i=1} X[y_{i,x} = y] \times w_i$. Практика показывает, что логичнее в качестве весовой функции $w : w_i = w(x_i)$ взять убывающую геометрическую прогрессию: $w_i = q_i$, где знаменатель прогрессии $q \in (0,1)$ является параметром алгоритма.

2.4 Выбор метрики

Популярные разновидности метрик:

1. Евклидова: $\sqrt{\sum_{i=1}^{k} (x_i^1 - x_i^2)^2}$
2. Минковского: $\sum_{i=1}^{k} (x_i^1 - x_i^2)^{p^{\frac{1}{p}}}$
3. Взвешенная Минковского: $\sum_{i=1}^{k} (w_i(x_i^1 - x_i^2))^{p^{\frac{1}{p}}}$
4. Косинусная: $\cos(x^1, x^2)/\|x^1\|\|x^2\|$
5. Манхэттенская: $\sum_{i=1}^{k} |x_i^1 - x_i^2|$
6. Чебышева: $\max_{i \in [1,n]} |x_i^1 - x_i^2|$
7. Хэмминга: $number(x_i^1 \neq x_i^2)/n$
8. Канберра: $\sum \frac{|x_i^1 - x_i^2|}{|x_i^1| + |x_i^2|}$
9. Брэя Кертиса: $\frac{\sum |x_i^1 - x_i^2|}{\sum |x_i^1| + \sum |x_i^2|}$

3 Формулировка задания

1. Написать на языке Python собственные реализации метода ближайших соседей и кросс-валидации.
2. Провести описанные эксперименты с датасетом изображений цифр MNIST.

4 Список экспериментов

1. Исследовать, какой алгоритм поиска ближайших соседей будет быстрее работать в различных ситуациях.

2. Для каждого объекта тестовой выборки найти 5 его ближайших соседей в обучающей для евклидовой метрики.

3. Выбрать подмножество признаков, по которому будет считаться расстояние, размера 10, 20, 100.

4. Оценить по кросс-валидации с 3 фолдами точность (долю правильно предсказанных ответов) и время работы k ближайших соседей в зависимости от следующих факторов:

   (a) a) k от 1 до 10.

   (b) b) Используется евклидова или косинусная метрика.

5. Сравнить взвешенный метод k ближайших соседей, где голос объекта равен $1 = (\text{distance} + \varepsilon)$, где $\varepsilon = 10^5$ с методом без весов при тех же фолдах и параметрах.

6. Применить лучший алгоритм к исходной обучающей и тестовой выборке.

7. Подсчитать точность, сравнить с точностью по кросс-валидации и с точностью, указанной в интернете.

8. Построить и проанализировать матрицу ошибок (confusion matrix).

9. Визуализировать несколько объектов из тестовой выборки, на которых были допущены ошибки. Проанализировать и указать их общие черты.

10. Размножить обучающую выборку с помощью поворотов, смещений и применений гауссовского фильтра.

11. Подобрать по кросс-валидации с 3 фолдами параметры преобразований.

12. Рассмотреть следующие параметры для преобразований и их комбинации:

    (a) a) Величина поворота: 5, 10, 15 (в каждую из двух сторон)

    (b) b) Величина смещения: 1, 2, 3 пикселя (по каждой из двух размерностей)

    (c) c) Дисперсия фильтра Гаусса: 0.5, 1, 1.5

13. Проанализировать, как изменилась матрица ошибок, какие ошибки алгоритма помогает исправить каждое преобразование.

14. Реализовать описанный выше алгоритм, основанный на преобразовании объектов тестовой выборки. Проверить то же самое множество параметров, что и в предыдущем пункте. Проанализировать как изменилась матрица ошибок, какие ошибки алгоритма помогает исправить каждое преобразование. Качественно сравнить два подхода (5 и 6 пункты) между собой.

5 Реализация

Загрузка необходимых библиотек и модулей:

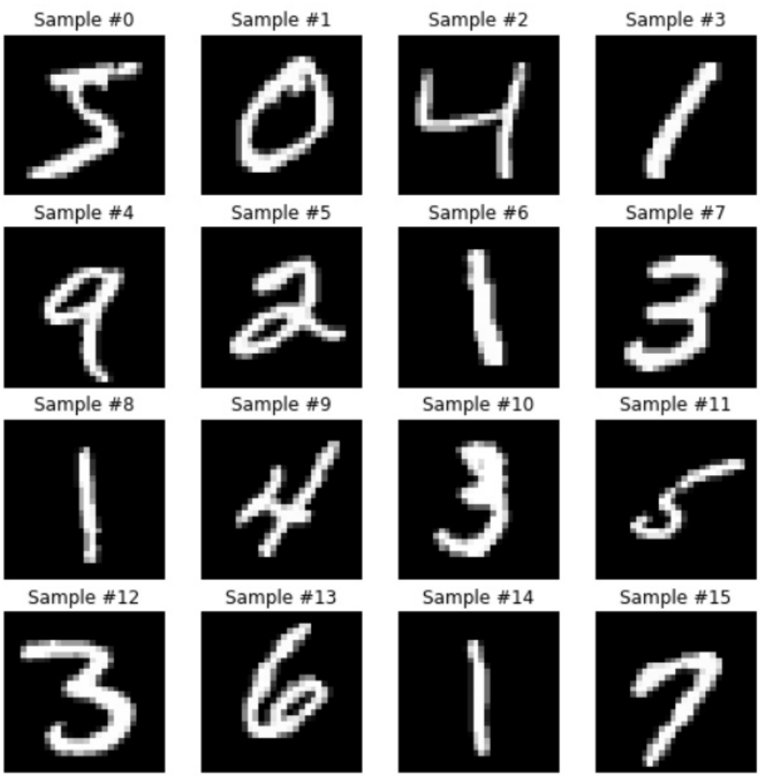
```
1) numpy
2) Собственные модули: KNNClassifier, distances, cross_validation
3) fetch_openml, confusion_matrix, imshow, rotate, AffineTransform, warp,
gaussian, matplotlib.pyplot, seaborn, time, pprint, ceil, product
```

5.1 MNIST

Загрузим датасет MNIST из 70000 растровых изображений из 28*28 пикселей с рукописными цифрами, где каждое изображение "вытянуто"в одномерный признаковый вектор длины 28*28 = 784.

```
X, y = fetch_openml('mnist_784', return_X_y=True)
```

Выведем на наглядности несколько изображений с помощью функции imshow.



Как видно, некоторые цифры легко идентифицировать, в то время как объекты 10, 11 не так очевидны.

После этого разделим датасет на 60.000 обучающих и 10.000 тестовых объектов вместе с их заранее заданными классами.

```
X_train, X_test, y_train, y_test =
X[:-10000],X[-10000:], y[:-10000],y[-10000:]
```

Обучающая выборка X, тестовая выборка X, ответы на обучающую выборку y, ответы на тестовую выборку y соответственно.

5.2 Модули и классы

Приступим к детальному описанию алгоритмов, написанных для решения задачи поиска КНН.

5.3 Модуль nearest neighbors и класс KNNClassifier

5.3.1 Конструктор

Собственная реализация поиска кнн. Заводим конструктор класса с определением собственных полей через параметры:

1. k - число ближайших соседей

2. metric - метрика: евклидова или косинусная. При получении данного параметра сразу определяем функцию metricfunc для поиска расстояния

3. strategy - стратегия поиска (4 типа: собственная, брутфорс или два дерева). В случае, если стратегия несобственная, подключается встроенная в sklearn функция поиска соседей NearestNeighbors

4. weights - опция взвешенного метода

5. testblocksize - размер выборки

6. Xtrain, ytrain, labels - хранимая для собственного метода обучающая выборка.

```python
class KNNClassifier:
    def __init__(self, k: int, strategy: str = 'my_own', metric: str = 'eucledean',
                 weights: bool = False, test_block_size: int = 0, n_jobs: int = 1):
        self.k = k

        self.metric = metric
        if metric == "euclidean":
            self._metric_func = euclidean_distance
        elif metric == "cosine":
            self._metric_func = cosine_distance
        else:
            raise TypeError("Metric <{}> is not supported!".format(metric))

        # only for parallel computation
        num_cores = multiprocessing.cpu_count()
        self.n_jobs = n_jobs if 0 < n_jobs < num_cores else num_cores

        self.strategy = strategy
        if strategy in ['brute', 'kd_tree', 'ball_tree']:
            self.NearestNeighbours = NearestNeighbors(
                n_neighbors=k,
                algorithm=strategy,
                metric=metric,
                n_jobs=self.n_jobs
            )

        elif strategy != 'my_own':
            raise TypeError("Strategy <{}> is not supported!".format(strategy))

        self.weights = weights
        self.test_block_size = test_block_size

        self.X_train = None
        self.y_train = None
        self.labels = None
```

5.3.2 Метод fit

В случае собственного алгоритма fit запоминает в полях класса KNNClassifier обучающую выборку, иначе (в случае брута или деревьев) вызывает встроенную функцию NearestNeighbours.fit(X), обучающую модель на выборке X.

5.3.3 Метод find-kneighbors

Для тестовой выборки возвращается массив рангов ближайших к соседей, а в случае истинности передаваемого параметра return-distance еще и массив непосредственных расстояний до ближайших соседей.

5.3.4 Метод predict

Для тестовой выборки возвращается одномерный массив классов, которые предсказал алгоритм.

5.4 Модуль cross-validation

Перекрестная проверка, выявляющая оптимальный параметр k. На вход подаётся обучающая выборка, которая с помощью функции kfold разделяется на тестовую и валидационную-обучающую выборки. Поскольку ответы для тестовой известны, то результатом сравнения предсказаний с действительными классами объектов из тестовой выборки будет точность алгоритма классификации для конкретных значений параметра k. Функция knn-cross-val-score по известным за счет kfold фолдам разбиения возвращает точность алгоритма.

```python
def kfold(n, n_folds):
    index = np.arange(n)
    folds = np.array_split(index, n_folds)

    result = []
    for fold in folds:
        mask = np.ones(n).astype(bool)
        mask[fold] = False
        result.append((index[mask], index[~mask]))

    return result


def knn_cross_val_score(X: np.ndarray, y: np.ndarray, k_list, score: str = 'accuracy', cv=None, **kwargs):
    if score == 'accuracy':
        score_func = lambda y_pred, y_true: np.mean(np.abs(y_pred == y_true))
    elif score == 'mae':
        score_func = lambda y_pred, y_true: np.mean(np.abs(y_pred != y_true))
    else:
        raise TypeError("Score <{}> is not supported!\nUse <accuracy>.".format(score))

    # KFold splits
    if not cv:
        cv = kfold(X.shape[0], n_folds=3)

    # fix k for validation
    kwargs['k'] = k_list[-1]

    # process score_func for each fold from cv and each k from k_list
    scoring = defaultdict(list)
    for train_ind, val_ind in cv:
        model = KNNClassifier(**kwargs)
        model.fit(X[train_ind], y[train_ind])
        distances, indices = model.find_kneighbors(X[val_ind], True)
        for k in k_list:
            predictions = model.estimate(indices[:, :k], distances[:, :k])
            scoring[k].append(score_func(predictions, y[val_ind]))

        # try to free the memory
        del model

    return scoring
```

5.5 Модуль distances

Реализованы две функции метрики - евклидова и косинусная.

```
def euclidean_distance(X, Y):
    X_norm = np.sum(X * X, axis=1)[:, None]
    Y_norm = np.sum(Y * Y, axis=1)[None, :]
    return np.sqrt(X_norm + Y_norm - 2 * np.dot(X, Y.T))


def cosine_distance(X, Y):
    X_norm = np.sum(X * X, axis=1)[:, None]
    Y_norm = np.sum(Y * Y, axis=1)[None, :]

    X_norm[X_norm == 0] = 1e-5
    Y_norm[Y_norm == 0] = 1e-5
    return 1.0 - np.dot(X, Y.T) / np.sqrt((X_norm  * Y_norm))
```

Так же посредством библиотеки joblib было осуществлено "распараллеливание"процессов вычислений, что привело к снижению затрат времени для собственной реализации поиска my-own практически в 3 раза, сравняв по скорости со встроенными алгоритмами на деревьях.

```
// в конструкторе KNNClassifier
num_cores = multiprocessing.cpu_count()
...self.n_jobs = n_jobs if 0 < n_jobs < num_cores else num_cores
// в find-kneighbor
with parallel_backend('threading', n_jobs=self.n_jobs):
chunks_neighbors = Parallel()(
delayed(chunk_find_kneighbors)(left, right)
for (left, right) in chunks_borders
)
```

Все описанные функции содержатся в прикрепленных к отчету подключаемых модулях. Описан общий принцип работы каждой функции. Некоторые включают в себя дополнительные функции, не описанные выше. Во имя сохранения спокойствия проверяющего данный отчетом и подвижности суставов в пальцах автора данного отчета, опустим промежуточные процедуры и тонкости работы каждого блока.

Поскольку объем датасета для обработки довольно большой (вся работа заняла примерно 8 гб ОП), попробуем сжать его с помощью сокращения признакового вектора до длины 10, 20, 100 соответственно. Признаки, вошедшие в новый признаковый вектор, выбираются случайно, но одинаково для всех объектов выборки. Происходит это с помощью трех (для 10, 20, 100-мерных векторов) генераций массивов случайных индексов от 0 до 784, после чего к вектору каждого объекта применяется итерирование получившимися массивами.

```
feature_indices =
[np.random.randint(0, X.shape[1], size) for size in (10, 20, 100)]
```

6 Обработка результатов

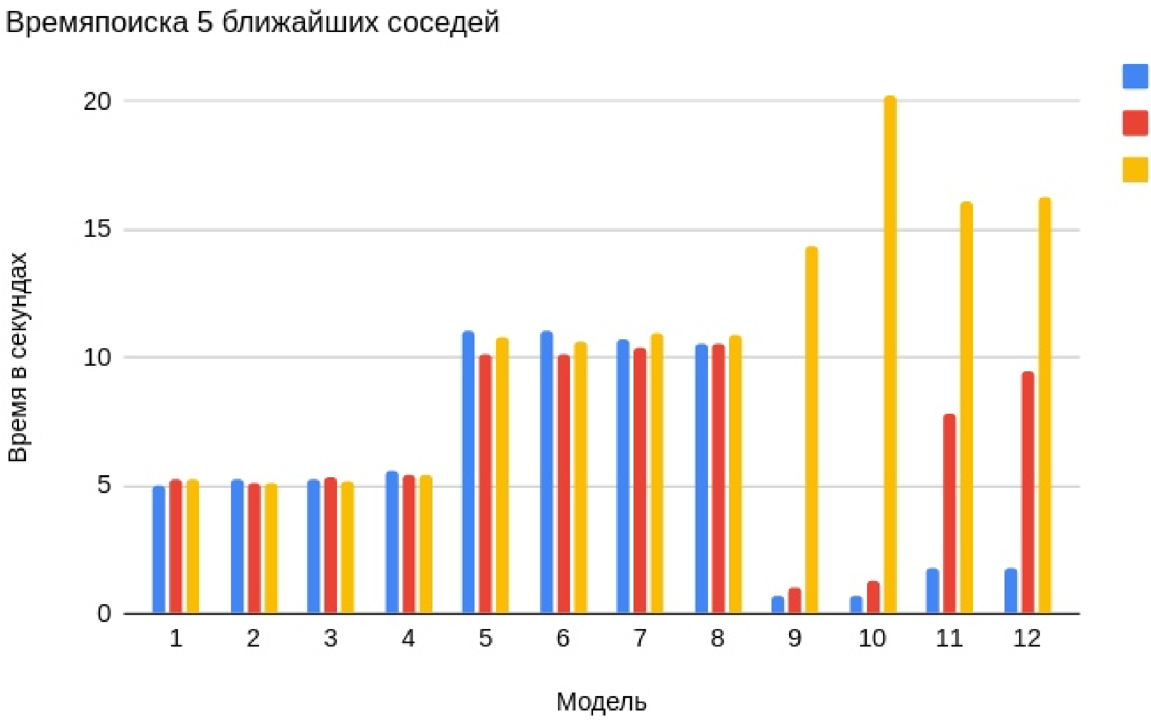
В первую очередь убедимся в количестве моделей по параметрам, их должно быть 4 для собственной реализации (2 метрики и 2 опции взвешенности), 4 на брутфорс (2 метрики и 2 опции взвешенности), 2 и 2 на деревья (доступна только евкидова метрика и 2 опции взвешенности). Итого 12.

```
my_own(5) [euclidean, weighted]
my_own(5) [euclidean]
my_own(5) [cosine, weighted]
my_own(5) [cosine]
brute(5) [euclidean, weighted]
brute(5) [euclidean]
brute(5) [cosine, weighted]
brute(5) [cosine]
kd_tree(5) [euclidean, weighted]
kd_tree(5) [euclidean]
ball_tree(5) [euclidean, weighted]
ball_tree(5) [euclidean]
```

Как видно, все модели учтены.

6.1 Определение 5 ближайших соседей

С помощью каждого метода найдём всем объектам их тестовой выборки 5 ближайших соседей. Время, необходимое каждому алгоритму, отображено на графике:



Модели обозначены в порядке возрастания в соответствии с:

```
1.
my_own(5) [euclidean, weighted] on 10 features: done in 5.0122s
my_own(5) [euclidean, weighted] on 20 features: done in 5.2342s
my_own(5) [euclidean, weighted] on 100 features: done in 5.2478s
2.
my_own(5) [euclidean] on 10 features: done in 5.2716s
my_own(5) [euclidean] on 20 features: done in 5.1296s
my_own(5) [euclidean] on 100 features: done in 5.0625s
3.
my_own(5) [cosine, weighted] on 10 features: done in 5.2397s
my_own(5) [cosine, weighted] on 20 features: done in 5.3188s
my_own(5) [cosine, weighted] on 100 features: done in 5.1582s
4.
my_own(5) [cosine] on 10 features: done in 5.6083s
my_own(5) [cosine] on 20 features: done in 5.4416s
my_own(5) [cosine] on 100 features: done in 5.4228s
5.
brute(5) [euclidean, weighted] on 10 features: done in 11.0825s
brute(5) [euclidean, weighted] on 20 features: done in 10.1217s
brute(5) [euclidean, weighted] on 100 features: done in 10.7710s
6.
brute(5) [euclidean] on 10 features: done in 11.0322s
brute(5) [euclidean] on 20 features: done in 10.1123s
brute(5) [euclidean] on 100 features: done in 10.6144s
7.
brute(5) [cosine, weighted] on 10 features: done in 10.7083s
brute(5) [cosine, weighted] on 20 features: done in 10.3887s
brute(5) [cosine, weighted] on 100 features: done in 10.9503s
8.
brute(5) [cosine] on 10 features: done in 10.5718s
brute(5) [cosine] on 20 features: done in 10.5359s
brute(5) [cosine] on 100 features: done in 10.9063s
9.
kd_tree(5) [euclidean, weighted] on 10 features: done in 0.7030s
kd_tree(5) [euclidean, weighted] on 20 features: done in 1.0057s
kd_tree(5) [euclidean, weighted] on 100 features: done in 14.3313s
10.
kd_tree(5) [euclidean] on 10 features: done in 0.7030s
kd_tree(5) [euclidean] on 20 features: done in 1.3042s
kd_tree(5) [euclidean] on 100 features: done in 20.2431s
\newpage
11.
ball_tree(5) [euclidean, weighted] on 10 features: done in 1.8050s
```

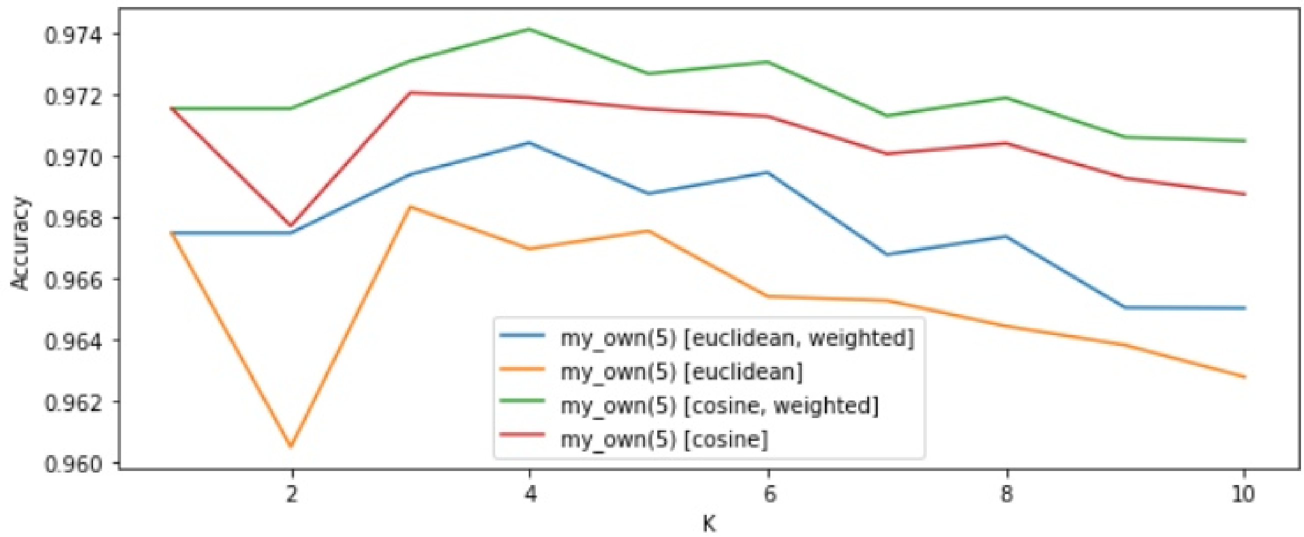
```
ball_tree(5) [euclidean, weighted] on 20 features: done in 7.8182s
ball_tree(5) [euclidean, weighted] on 100 features: done in 16.1321s
12.
ball_tree(5) [euclidean] on 10 features: done in 1.8045s
ball_tree(5) [euclidean] on 20 features: done in 9.5167s
ball_tree(5) [euclidean] on 100 features: done in 16.2338s
```

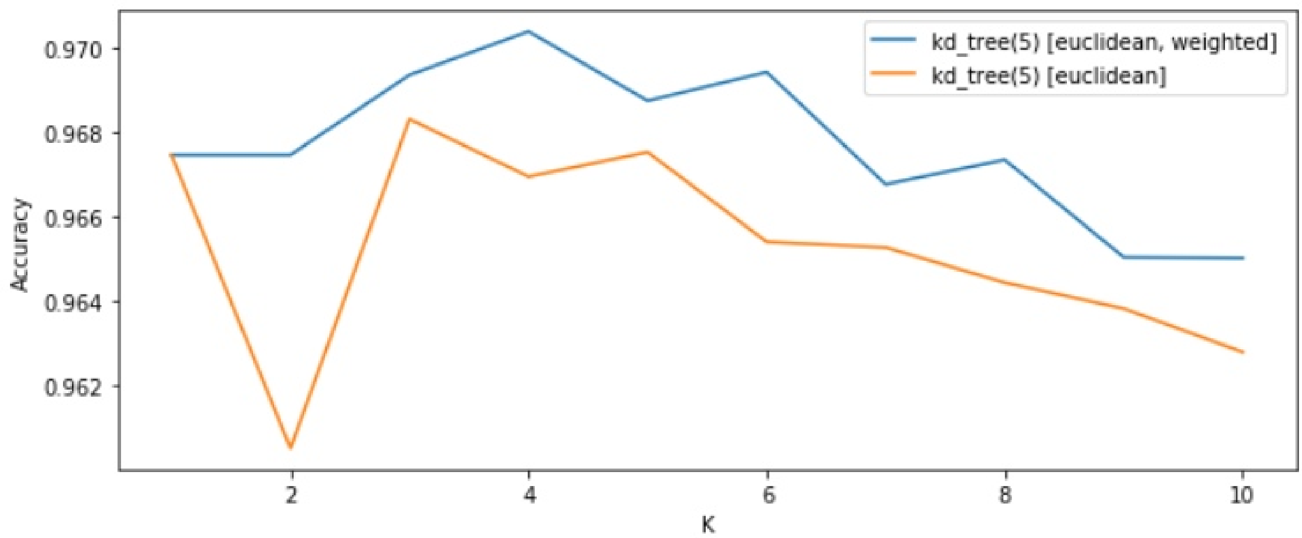
Выделим лучшие алгоритмы по времени для каждой длины признакового вектора.

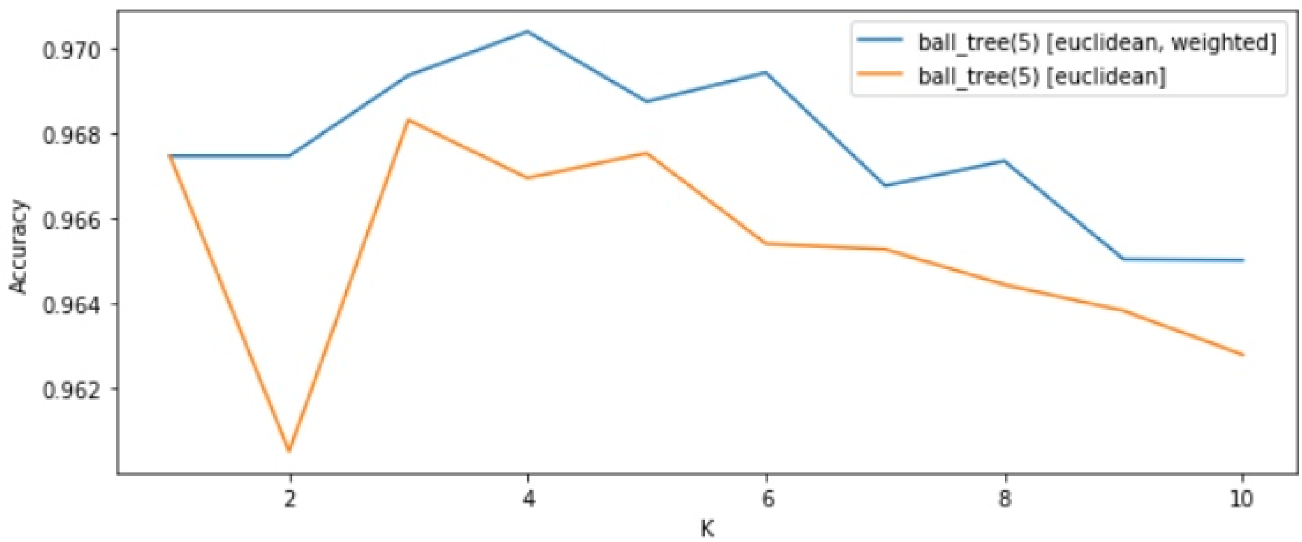
```
Best model for 10 features: kd_tree(5) [euclidean]: done in 0.7030s
Best model for 20 features: kd_tree(5) [euclidean, weighted]: done in 1.00
Best model for 100 features: my_own(5) [euclidean]: done in 5.0625s
```

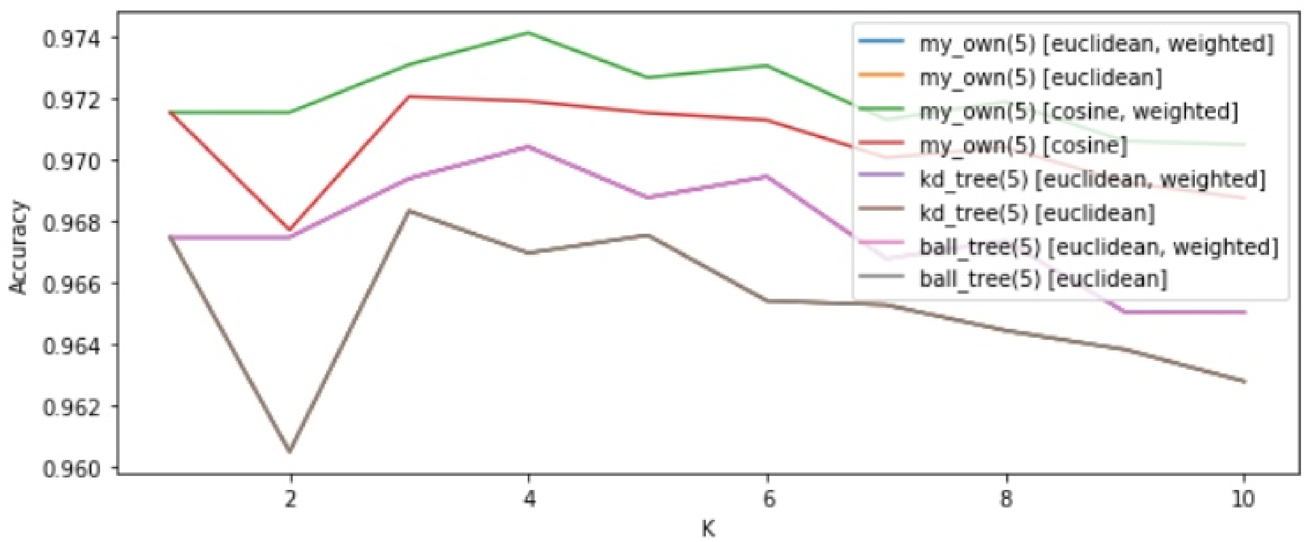
Согласно полученным данным, для 10-мерных объектов самым быстрым является алгоритм невзвешенное kd-дерево, для 20 - взвешенное kd-дерево и для 100 - невзвешенная собственная реализация с евклидовой метрикой.

6.2 Оценки качества для различных моделей









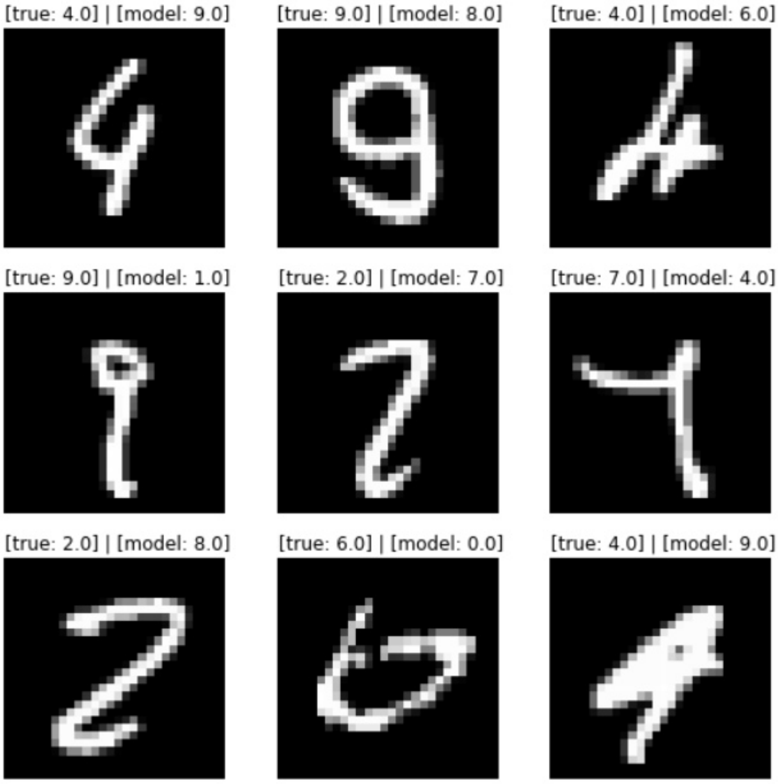
Вывод: Оценка качества одинакова для собственного метода и брута, а также для kd-дерева и ball-дерева на евклидовых метриках. Однако косинусная метрика (не применимая к kd-дерву и ball-дереву) показывает себя лучше для переборных методов. Взвешенные предсказания классов работают лучше для всех методов. Лучший по качеству алгоритм - полностью переборный, однако он существенно медленнее для поиска ближайших соседей.

6.3 Подсчет точности лучшего алгоритма

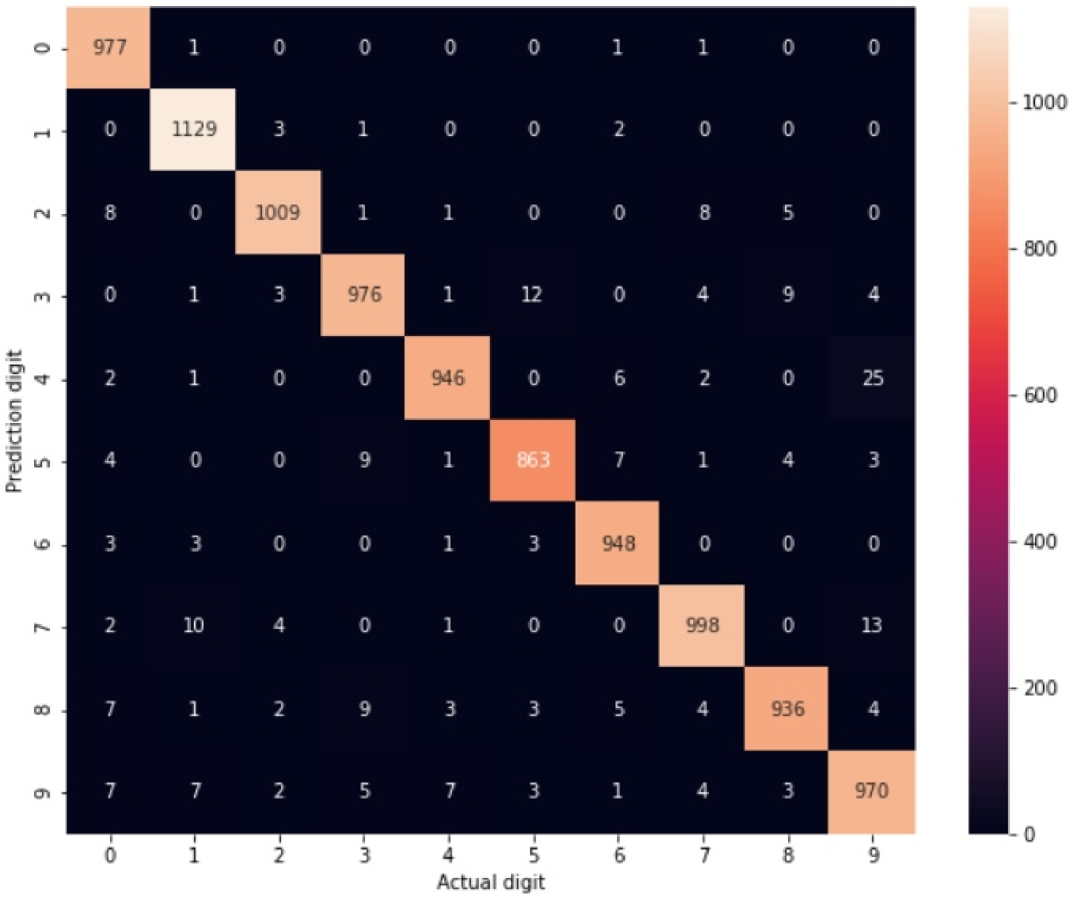
Согласно результатам, точность лучшего алгоритма составила 0.9752. По кросс-валидации этот параметр равен 0.9732. Поскольку лучшим алгоритмов стала собственная реализация, сравнить ее точность со значениями в интернете оказывается проблематично. Однако с оценкой по кросс-валидации результат совпадает до сотых долей.

6.4 Объекты, на которых алгоритм ошибся

Однако она не 1, взглянем на объекты, классифицируя которые алгоритм ошибся:

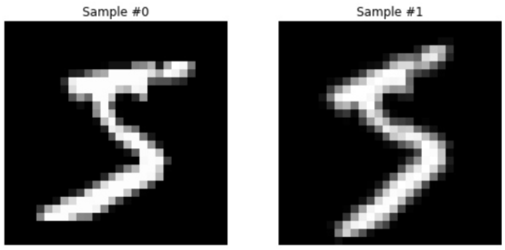


Confusion matrix:



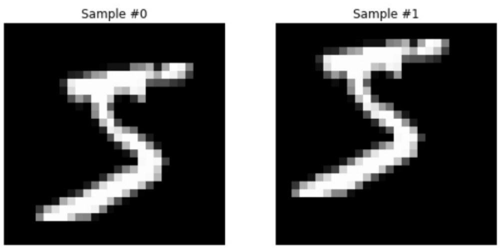
6.5 Подбор параметров для повышения качества классификации

6.5.1 Вращение цифр



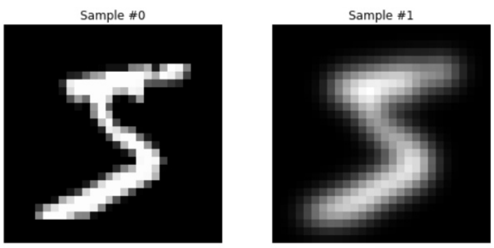
Для начала осуществляется поворот одного объекта из датасета и оценка результата Вывод: согласно результатам кросс-валидации, наилучшим значением для угла поворота является 10 и -10 градусов.

6.5.2 Сдвиги цифр



Для начала осуществляется сдвиг одного объекта из датасета и оценка результата Вывод: согласно результатам кросс-валидации, наилучшим значением для сдвига являются (0, 3) и (3, 0).

6.5.3 Применение фильтра



Для начала фильтр применяется к одному объекту из датасета и оценивается результат Вывод: согласно результатам кросс-валидации, наилучшим значением для дисперсии фильтра Гаусса является 0.5.

Зачем нужно размножение обучающей выборки? Каждое из преобразований убирает некоторую часть ошибок. При использовании всех преобразований для полной выборки это можно было бы проанализировать исходя из confusion matrix, однако размер полученной выборки был бы слишком большим для вычисления на лэптопе.

1. Поворот на некоторый угол позволяет исправить ситуацию, когда тестовое изображение может быть слегка наклонено относительно обучающей выборки (например, если угол наклона почерка человека отличается в разных документах).

2. Сдвиг внутри изображения позволяет учитывать различное положение цифры при начертании. Кто-то напишет цифру не ровно посередине квадрата, а чуть левее или правее, выше или ниже.

3. Фильтр Гаусса позволяет уменьшить влияние степени нажима при начертании. Яркость цифры также становится менее важной. Но самое главное - гауссов фильтр размывает резкие края плохо детализированного изображения. Граница становится мягкой, и одни и те же цифры становятся более похожи друг на друга.

Что будет, если применить те же аугментации к тестовой выборке?

Влияние видов преобразований уже было описано. Если применить их, чтобы размножить тестовую выборку, а не обучающую, мы получим более честные результаты метрики accuracy при тестировании, поскольку это позволит имитировать различные виды начертания цифр различными людьми, которые могли не встречаться в обучающей выборке.

7 Итоги

1. Для каждого подмножества признаков размера 10, 20, 100 для всей тестовой выборки найдены 5 ближайших соседей из обучающей для евклидовой метрики, зафиксировано и отображено время, затраченное на поиск, выделены алгоритмы, выполнившие запрос быстрее всего.

2. Получена оценка точности по кросс-валидации с 3 фолдами (доля правильно предсказанных ответов) и время работы k ближайших соседей в зависимости от следующих факторов:

   (a) a) k от 1 до 10.

   (b) b) Используется евклидова или косинусная метрика.

3. Реализован взвешенный метод k ближайших соседей, где голос объекта равен $1 = (\text{distance} + \varepsilon)$, где $\varepsilon = 10^5$ и сравнен с методом без весов при тех же фолдах и параметрах.

4. После применения лучшего алгоритма к тестовой выборке подсчитана точность, и проведено сравнение с точностью по кросс-валидации.

5. Построена и проанализирована матрица ошибок (confusion matrix).

6. Визуализированы несколько объектов из тестовой выборки, на которых были допущены ошибки.

7. Размножена обучающая выборка с помощью поворотов, смещений и применений гауссовского фильтра.

8. По кросс-валидации с 3 фолдами подобраны оптимальные параметры преобразований.

9. В качестве дополнительного задания написана параллельная реализация поиска.