Московский авиационный институт
(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа № 1

по курсу «Машинное обучение»

Студент: Гаврилов М.С.

Группа: 80-306б

Преподаватель:

Оценка:

Москва, 2022

1. Постановка задачи

Формально говоря вам предстоит сделать следующее:

1) реализовать следующие алгоритмы машинного обучения: Linear/ Logistic Regression, SVM, KNN, Naive Bayes в отдельных классах

2) Данные классы должны наследоваться от BaseEstimator и ClassifierMixin, иметь методы fit и predict (подробнее: https://scikit-learn.org/stable/developers/develop.html)

3) Вы должны организовать весь процесс предобработки, обучения и тестирования с помощью Pipeline (подробнее: https://scikit-learn.org/stable/modules/compose.html)

4) Вы должны настроить гиперпараметры моделей с помощью кросс валидации (GridSearchCV,RandomSearchCV, подробнее здесь: https://scikit-learn.org/stable/modules/grid_search.html), вывести и сохранить эти гиперпараметры в файл, вместе с обученными моделями

5) Проделать аналогично с коробочными решениями

6) Для каждой модели получить оценки метрик:Confusion Matrix, Accuracy, Recall, Precision, ROC_AUC curve (подробнее: Hands on machine learning with python and scikit learn chapter 3, mlcourse.ai, https://ml-handbook.ru/chapters/model_evaluation/intro)

7) Проанализировать полученные результаты и сделать выводы о применимости моделей

8) Загрузить полученные гиперпараметры модели и обученные модели в формате pickle на гит вместе с jupyter notebook ваших экспериментов

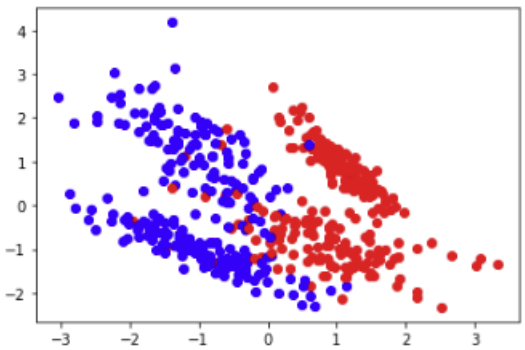
2. Выполнение работы

1. Реализация собственных версий алгоритмов машинного обучения.

В этом разделе демонстрируются классы, реализующие модели Линейной/Логистической регрессии, SVM, KNN, Наивный Байесовский алгоритм. Также показана работоспособность моих реализаций требуемых метрик.

Для тестирования алгоритмов используется искусственный датасет для бинарной классификации, созданный с помощью skl.

Вот его визуализация:



1.1 Линейная / логистическая регрессия

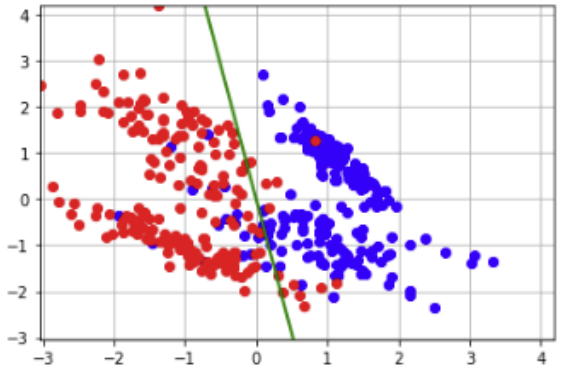
1.1.1. Линейная регрессия

```
class linear_regression(skl.base.ClassifierMixin):

    def __init__(self,n_feat,lr = 0.05,d = 0):
        self.w = np.zeros(n_feat + 1)
        self.d = d
        self.lr = lr

    def wts(self):
        return self.w

    def set_d(self,new_d):
        self.d = new_d

    def predict(self,feats_): # кассификация (положительный/отрицательный результат в зависимости от того,
                              # преодолевает ли предстказанное значение дискриминант)
        feats = feats_[0]

        score = np.sum((np.dot(feats,self.w[0:len(feats)])))

        return int(score > self.d)

    def linres(self,feats_): # предсказанное значение f(feats)
        feats = feats_[0]

        score = np.sum((np.dot(feats,self.w[0:len(feats)])))

        return score

    def accuracy(self,data,labels):
        total = 0
        correct = 0
        for i in range(len(labels)):
            total += 1
            if(self.predict([data[i]]) == labels[i]):
                correct += 1
        return correct/total

    def score(self,data,labels):
        return self.accuracy(data,labels)

    def fit(self,data,labels,rr = 50):
        self.w = np.dot(np.dot(np.linalg.inv(np.dot(np.transpose(np.matrix(data)),
                                                    np.matrix(data))),
                               (np.transpose(np.matrix(data)))),
                        np.transpose(np.matrix(labels)))
```

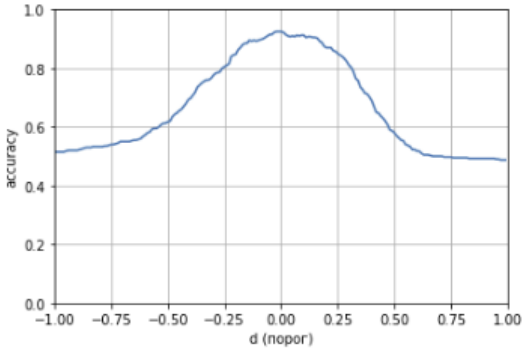
Метрики для данной модели.

Разделяющая прямая:

```
accuracy with train set: 0.925
accuracy with test set: 0.9
```



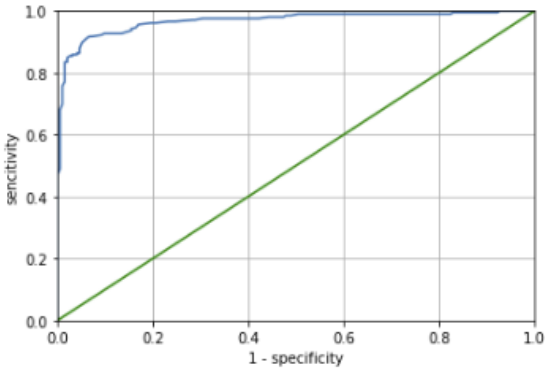
Кривая «порог – точность»:
Она очень помогает в выборе оптимального значения порога



Основные метрики бинарной классификации:

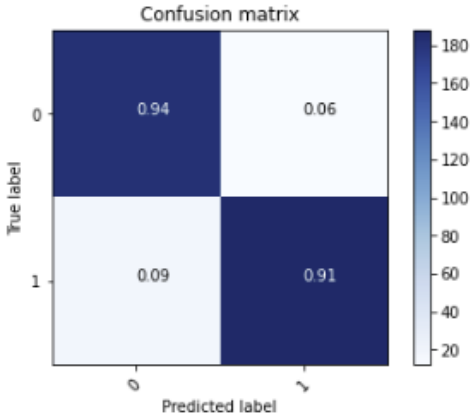
```
accuracy: 0.925 (чем больше ложных, тем меньше)
precision: 0.94 (чем больше ложнопозитивных, тем меньше)
recall: 0.912621359223301 (чем больше ложнонегативных, тем меньше)
specificity: 0.9381443298969072 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.912621359223301 (чем больше позитивных классифицировано неверно, тем меньше)
```

Кривая ROC:



Матрица спутывания:

```
Normalized confusion matrix
```

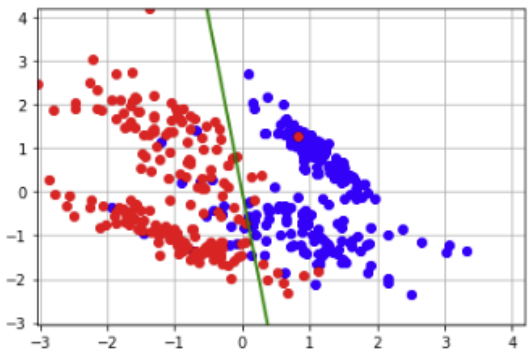


1.1.2. Логистическая регрессия

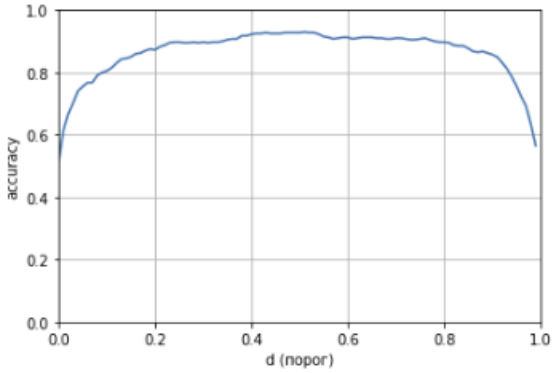
```
class logistic_regression(skl.base.ClassifierMixin):

    def __init__(self,n_feat,lr = 0.1,d = 0.5):
        self.w = np.zeros(n_feat + 1)
        self.d = d
        self.lr = lr

    def wts(self):
        return self.w

    def predict(self,feats_,d = -1):
        feats = feats_[0]

        score = np.sum(np.dot(self.w[0:len(feats)],feats))

        if(d == -1 or (d < 0) or (d > 1)):
            d = self.d
        prob = 1/(1 + np.exp(-score))

        return int(prob > d)

    def logit(self,feats_):
        feats = feats_[0]

        score = np.sum(np.dot(self.w[0:len(feats)],feats))
        return 1/(1 + np.exp(-score))

    def set_d(self,new_d):
        self.d = new_d

    def accuracy(self,data,labels):
        total = 0
        correct = 0
        for i in range(len(labels)):
            total += 1
            if(self.predict([data[i]]) == labels[i]):
                correct += 1
        return correct/total

    def score(self,data,labels):
        return self.accuracy(data,labels)

    def fit(self,data,labels,rr = 50):

        if(rr == 0):
            rr = len(labels) + 1
        print("")

        for i in range(len(labels)):
            lr = self.lr

            sc = self.logit([data[i]])
            self.w += (labels[i] - sc)*data[i]*lr
```

Разделяющая прямая:

```
accuracy with train set: 0.9275
accuracy with test set: 0.9
```



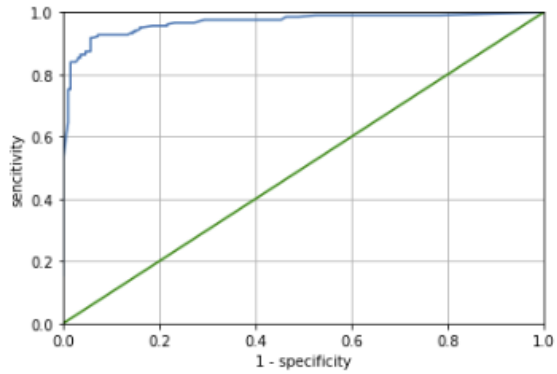
Кривая «порог – точность»:



Основные метрики бинарной классификации:

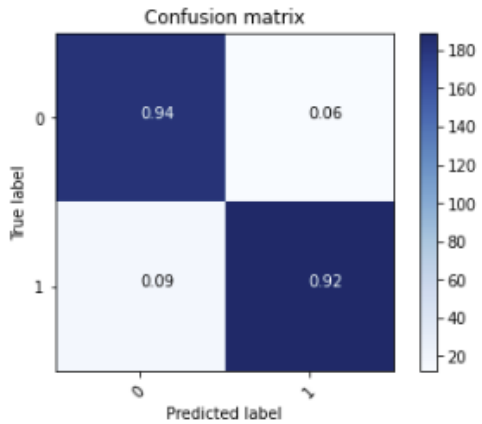
```
accuracy: 0.9275 (чем больше ложных, тем меньше)
precision: 0.9402985074626866 (чем больше ложнопозитивных, тем меньше)
recall: 0.9174757281553398 (чем больше ложнонегативных, тем меньше)
specificity: 0.9381443298969072 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.9174757281553398 (чем больше позитивных классифицировано неверно, тем меньше)
```

Кривая ROC:



Матрица спутывания:

```
Normalized confusion matrix
```

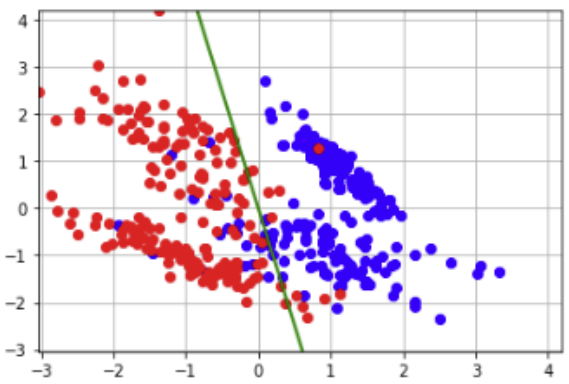


1.2 SVM

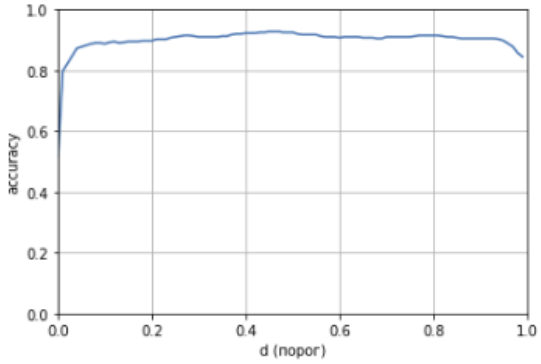
```
class SVM(skl.base.ClassifierMixin):

    def __init__(self,n_feat,lr = 0.05,d = 0.9):
        self.w = np.zeros(n_feat + 1)
        self.d = d
        self.lr = lr

    def wts(self):
        return self.w

    def predict(self,feats_):
        feats = feats_[0]

        score = np.sum(np.dot(self.w[0:len(feats)],feats))

        prob = 1/(1 + np.exp(-score))

        return int(prob > self.d)

    def set_d(self,new_d):
        self.d = new_d

    def accuracy(self,data,labels):
        total = 0
        correct = 0
        for i in range(len(labels)):
            total += 1
            if(self.predict([data[i]]) == labels[i]):
                correct += 1
        return correct/total

    def score(self,data,labels):
        return self.accuracy(data,labels)

    def fit(self,data,labels, alpha = 0.1,silent = True):

        for i in range(len(labels)):
            lr = self.lr

            cl = self.predict([data[i]])


            if(cl < self.d and labels[i] == 1):
                self.w += (data[i])*lr

            if(cl > self.d and labels[i] == 0):
                self.w -= (data[i])*lr

            self.w += alpha * self.w * lr
```

Разделяющая прямая:

```
accuracy with train set: 0.905
accuracy with test set: 0.9
```

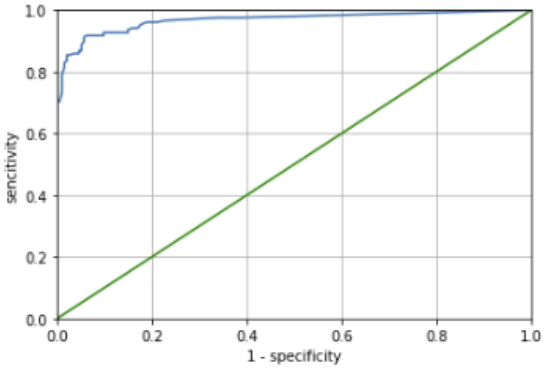


Кривая «порог – точность»:



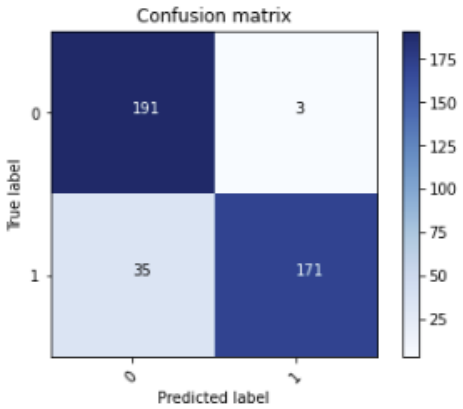
Основные метрики бинарной классификации:

```
accuracy: 0.905 (чем больше ложных, тем меньше)
precision: 0.9827586206896551 (чем больше ложнопозитивных, тем меньше)
recall: 0.8300970873786407 (чем больше ложнонегативных, тем меньше)
specificity: 0.9845360824742269 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.8300970873786407 (чем больше позитивных классифицировано неверно, тем меньше)
```



Матрица спутывания:

```
Confusion matrix, without normalization
```



1.3. KNN

```
class KNN(skl.base.ClassifierMixin): #svm (soft-margin)

    def __init__(self,k = 5):
        if(k % 2 == 0):
            k+=1

        if(k <= 0):
            return -1

        self.k = k

    def set_d(self,new_d):
        self.k = new_d

    def predict(self,feats_,silent = True):
        feats = feats_[0]

        mins = np.zeros([self.k,2]) - 1

        j=0
        for cur in self.examples:
            j+= 1
            dist = np.linalg.norm(feats - cur[0]) #рассояние между точками

            for i in range(len(mins)):
                if(dist < mins[i][0] or mins[i][0] < 0):
                    mins[i][0] = dist
                    mins[i][1] = cur[1]
                    break

        if(not silent):
            print(mins)

        score = np.zeros(len(self.classes))
        for a in mins:
            score[int(a[1])] += 1

        if(not silent):
            print(score)

        return (np.argmax(score))

    def accuracy(self,data,labels):
        total = 0
        correct = 0
        for i in range(len(labels)):
            total += 1
            if(self.predict([data[i]]) == labels[i]):
                correct += 1
        return correct/total


    def score(self,data,labels):
        return self.accuracy(data,labels)

    def fit(self,data,labels):
        self.examples = []
        for i in range(len(data)):
            self.examples.append([data[i],labels[i]])
        self.examples = np.array(self.examples,dtype=object)
        self.classes = np.unique(labels)
```

Точность на обучающей и тестовой выборке:

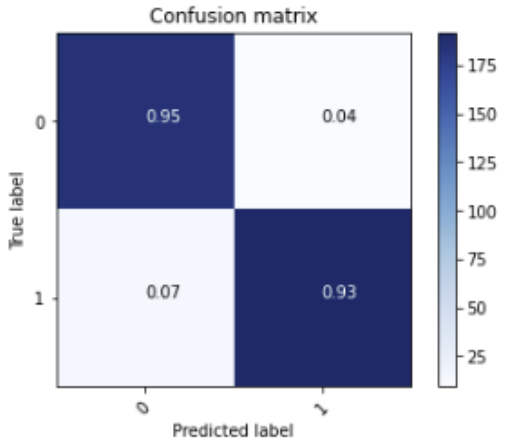
```
accuracy with train set: 0.9425
accuracy with test set: 0.93
```

Основные метрики бинарной классификации:

```
accuracy: 0.9425 (чем больше ложных, тем меньше)
precision: 0.9552238805970149 (чем больше ложнопозитивных, тем меньше)
recall: 0.9320388349514563 (чем больше ложнонегативных, тем меньше)
specificity: 0.9536082474226805 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.9320388349514563 (чем больше позитивных классифицировано неверно, тем меньше)
```

Матрица спутывания:

```
Normalized confusion matrix
```



Так как алгоритм KNN не включает в себя порогового значения или его аналогов, кривые ROC и порог-точность не применимы

1.4. Наивный Байесовский алгоритм

```
class Naive_Bayes(sk1.base.ClassifierMixin):
    #базовый наивный байесовский алгоритм
    def __init__(self):
        0 == 0

    def predict(self,feats_,silent = True):
        feats = feats_[0]
        score = copy.deepcopy(self.cl_freq)

        for i in range(len(score)):
            for cur in feats:
                if(np.array(cur,dtype = np.intc).tolist() in self.uni.tolist()):
                    score[i] *= self.feat_freq[i][self.uni.tolist().index(np.array(cur,dtype = np.intc).tolist())]
        cl = np.argmax(score)

        return cl

    def _score(self,feats_):
        feats = feats_[0]
        score = copy.deepcopy(self.cl_freq)

        for i in range(len(score)):
            for cur in feats:
                if(np.array(cur,dtype = np.intc).tolist() in self.uni.tolist()):
                    score[i] *= self.feat_freq[i][self.uni.tolist().index(np.array(cur,dtype = np.intc).tolist())]
        return score

    def accuracy(self,data,labels):
        total = 0
        correct = 0
        for i in range(len(labels)):
            total += 1
            if(self.predict([data[i]]) == labels[i]):
                correct += 1
        return correct/total

    def score(self,data,labels):
        return self.accuracy(data,labels)

    def status(self):
        print("classes | freq:")
        print(self.cl_freq)
        print("features | freq: ")
        print(self.feat_freq)

    def fit(self,data,labels):
                            #должны учитываться только признаки, которые есть во всех классах

        #data представляет из себя массив массивов возмможно переменной длины,
        #в каждом из которых находятся теги, принадл. данному эементу

        self.cl_freq = np.zeros(np.max(labels) + 1)
        for i in labels:
            self.cl_freq[i] += 1

        #список уникальных признаков
        #признаками считаются элементы массива признаков целиком
        #([[1,2],[3,4]) - признак №1 = [1,2], признак №2 = [3,4]

        self.uni = []
        for cur in data:
            for a in cur:
                b = np.array(a,dtype = np.intc).tolist()
                if not b in self.uni:
                    self.uni.append(b)
        self.uni = np.array(self.uni)

        #частоты признаков в классах
        self.feat_freq = np.zeros([len(self.cl_freq),len(self.uni)])

        for i in range(len(labels)):
            for j in range(len(data[i])):
                #в списке уникальных признаков находим текущий признак, и по его индексу инкрементируем элемент в списке частот
                self.feat_freq[int(labels[i])][self.uni.tolist().index(np.array(data[i][j],dtype = np.intc).tolist())] += 1
```

Точность на обучающей и тестовой выборке:

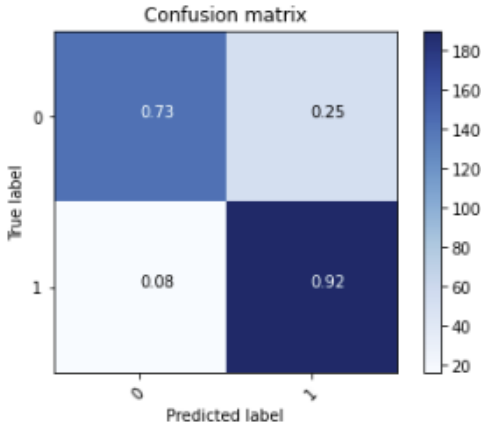
```
accuracy with train set: 0.83
accuracy with test set: 0.77
```

Основные метрики бинарной классификации:

```
accuracy: 0.83 (чем больше ложных, тем меньше)
precision: 0.7851239669421488 (чем больше ложнопозитивных, тем меньше)
recall: 0.9223300970873787 (чем больше ложнонегативных, тем меньше)
specificity: 0.7319587628865979 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.9223300970873787 (чем больше позитивных классифицировано неверно, тем меньше)
```

Матрица спутывания:

```
Normalized confusion matrix
```



Интересно, что, не смотря на специфику классических применений этого алгоритма, он довольно универсален. Например, для приемлемой классификации точек на двумерной плоскости, я округлил их координаты и представил координатные пары, как признаки. Можно предположить, что, если увеличить разрешение (округлять не до целых, а до 0.5, например), то точность еще повысится, но это не основная тема лабораторной, так что предположение осталось непроверенным.

Теперь можно проанализировать алгоритмы и сделать некоторые выводы об их особенностях. Во-первых, благодаря кривой порог-точность, можно заметить основную разницу между линейной регрессией, логистической регрессией и методом опорных векторов с точки зрения пользователя: Линейная регрессия имеет лишь небольшой диапазон значений дискриминанта, в котором точность максимальна. При использовании логистической регрессии этот диапазон существенно расширяется, но все равно ближе к краям точность падает. У SVM же, благодаря максимизации зазора, точность примерно равна почти на любых значениях порога, пусть и не всегда столь же высока, как максимальная точность линейной / логистической регрессии.

Самым точным алгоритмом оказался KNN, самым неточным – Наивный Байесовский (хотя его показатели, наверное, можно улучшить, изменив выбор признаков)

Несмотря на, казалось, бы, очевидное превосходство KNN, у него есть существенный недостаток – все классифицируемые объекты должны принадлежать пространству одной размерности. Наивный Байесовский алгоритм же может принимать на вход объекты разной размерности и, при этом, успешно их классифицировать.

2. Реализация поставленной в нулевой лабораторной работе задачи.

Поставленная задача:

Классификация слов русского языка по частям речи.

Так как при классификации слов я, так или иначе, буду подавать на вход алгоритмам буквы алфавита, я сомневаюсь в пригодности для решения моей задачи алгоритма линейной регрессии и ему подобных, так как решение задачи регрессии подразумевает нахождение в непрерывном векторном пространстве, в то время как пространство букв алфавита дискретно, не векторно, и не может быть спроецировано на векторное пространство (то, что буквы «А» и «Б» находятся рядом в алфавите, нахождение буквы «А» на конце слова не делает его похожим на слово с буквой «Б» на конце)

Я предполагаю, что наибольшего успеха получится добиться при использовании наивного Байесовского классификатора, так как он не работает с признаками, как с координатами. Также, возможно, KNN даст неплохой результат благодаря нахождению точных совпадений, и потому что он лучше всех себя показал при тестировании на искусственном датасете, может это о чем-то говорит.

При классификации будет использоваться датасет из слов, относящихся к одной из 7 частей речи: имя существительное, предлог, имя прилагательное, глагол, наречие, союз, причастие.

2.1. Наивный подход

Для начала попробуем использовать в качестве признаков буквы слова – каждая буква, входящая в слово, есть отдельный признак. Единственный доступный при таком подходе классификатор – наивный Байесовский.

Используем такую функцию-трансформер (wordAsArray = слово в виде массива алфавитных индексов букв)

```
class transform(skl.base.TransformerMixin):

    def fit(self,a,b):
        return self

    def transform(self,a):
        res = copy.deepcopy(a)
        for i in range(len(a)):
            res[i] = wordAsArray(a[i])

        return res
```

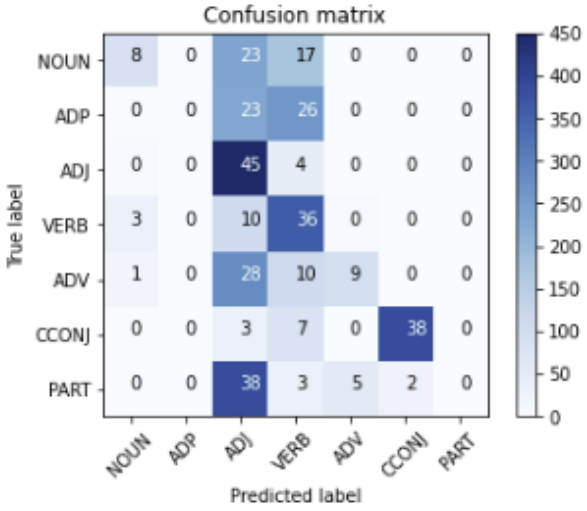
Точность, хоть и лучше случайного угадывания (0.14 для семи классов), но, все же, неприемлемо низкая.

```
accuracy on train set:
0.39371428571428574
accuracy on test set:
0.3914285714285714
```

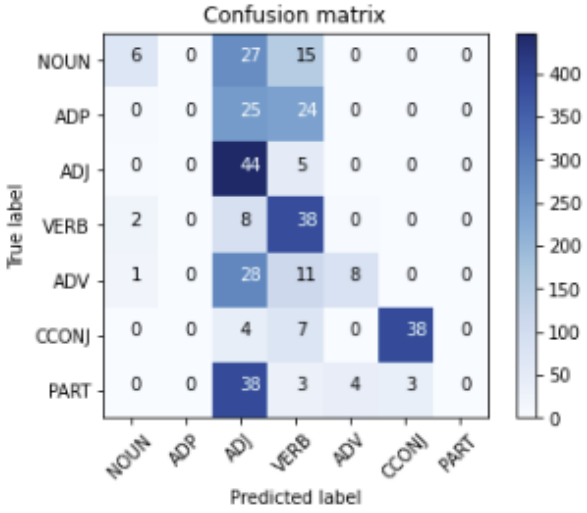
Матрицы спутывания также выглядят неважно:

Все числа на матрицах спутывания, представленных ниже, уменьшены в 10 раз, чтобы корректно уместить их на рисунке.

```
Confusion matrix, without normalization
```



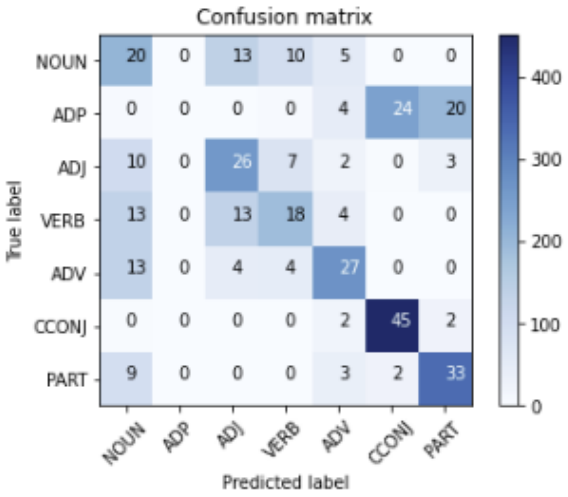

```
Confusion matrix, without normalization
```



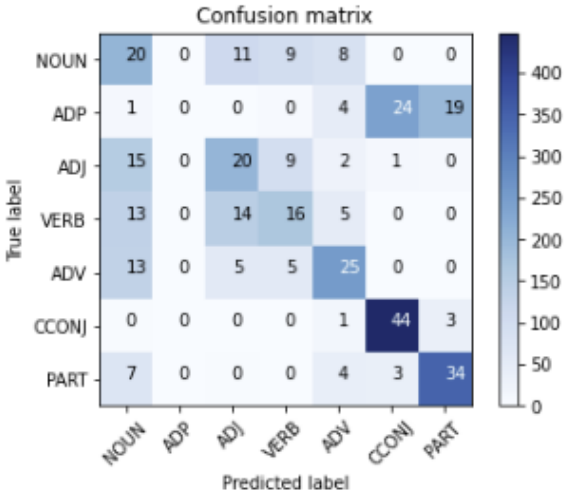
2.2. Выделение важных признаков.

Будем использовать массив из признаков, которые являются ключевыми для определения части речи. Для начала включим в него только длину слова. Используемый классификатор – по-прежнему – наивный Байесовский.

```
accuracy on train set:
0.48942857142857144
accuracy on test set:
0.4634285714285714
train set:
Confusion matrix, without normalization
```




```
test set:
Confusion matrix, without normalization
```

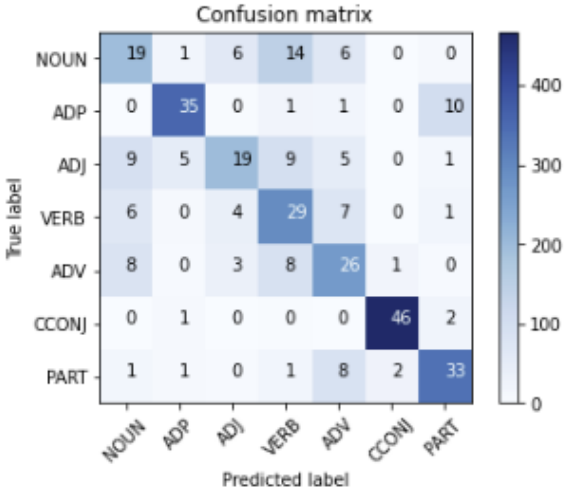


Результат уже лучше. Видно, что, так как предлоги и союзы имеют одинаковую длину, модель совершенно их не различает.

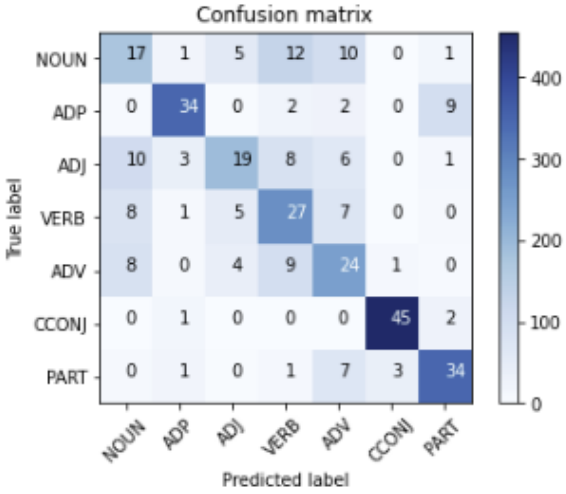
Проверим, что получится, если в массив добавить первую букву слова, последнюю букву слова и различные комбинации трех вышеперечисленных признаков.

Длина + первая буква

```
accuracy on train set:
0.604
accuracy on test set:
0.5857142857142857
train set:
Confusion matrix, without normalization
```

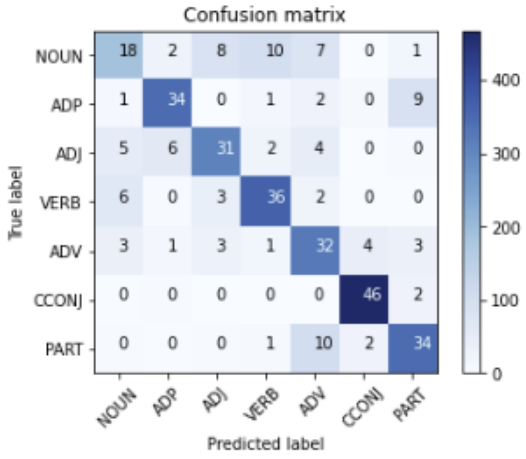



```
test set:
Confusion matrix, without normalization
```

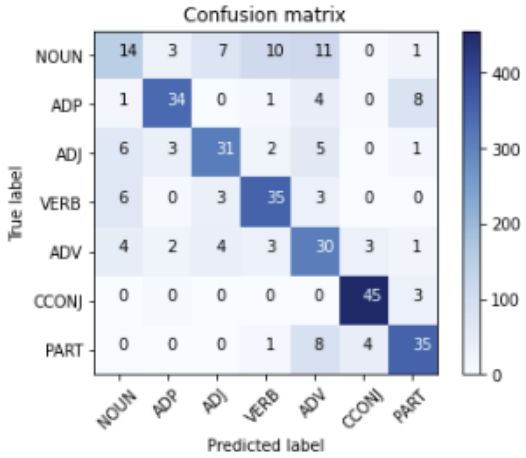


Длина + последняя буква

```
accuracy on train set:
0.6717142857142857
accuracy on test set:
0.6502857142857142
train set:
Confusion matrix, without normalization
```

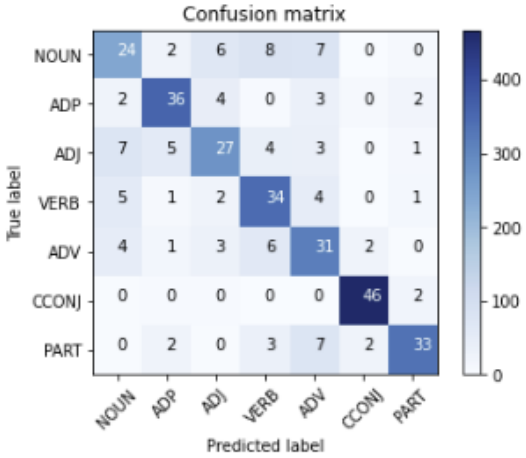



```
test set:
Confusion matrix, without normalization
```

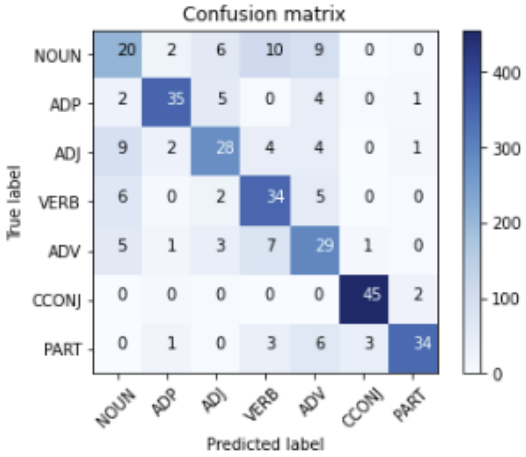


Длина + обе буквы

```
accuracy on train set:
0.6702857142857143
accuracy on test set:
0.6528571428571428
train set:
Confusion matrix, without normalization
```

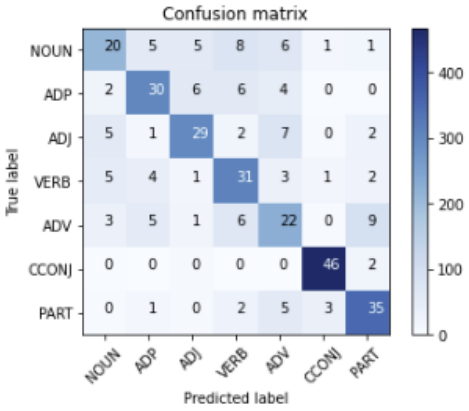



```
test set:
Confusion matrix, without normalization
```

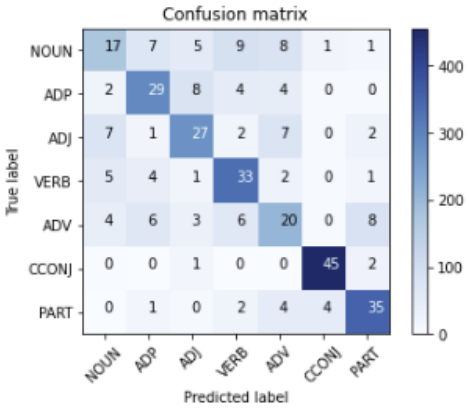


Обе буквы без длины

```
accuracy on train set:
0.6202857142857143
accuracy on test set:
0.598
train set:
Confusion matrix, without normalization
```



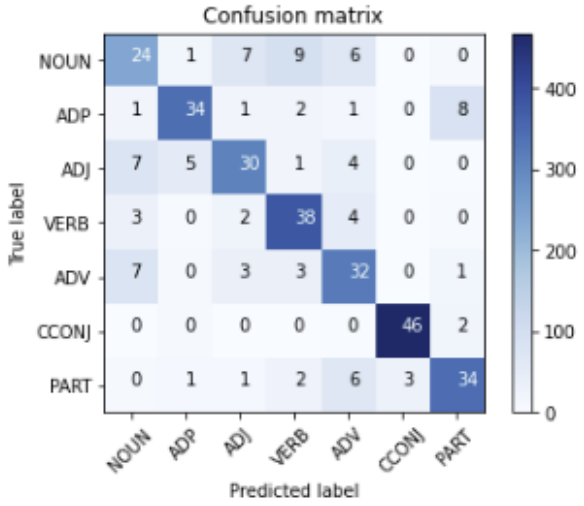

```
test set:
Confusion matrix, without normalization
```



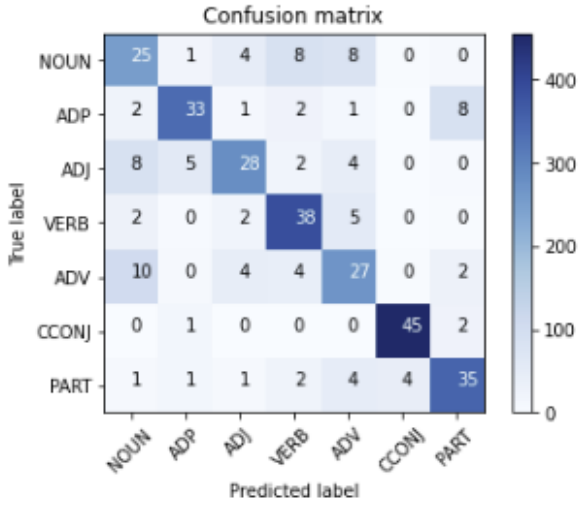
По сути, добавление любой буквы существенно увеличивает точность предсказаний и приводит матрицу спутывания в приличный вид. Ожидаемо, что длина + обе буквы – лучшая комбинация, интересно, что обе буквы без длины тоже показали неплохой результат.

Наилучший результат получается, если скомбинировать длину, первую букву, последнюю букву, предпоследнюю букву и букву перед предпоследней.

```
accuracy on train set:
0.69
accuracy on test set:
0.6688571428571428
Confusion matrix, without normalization
```




```
Confusion matrix, without normalization
```



Даже с оптимальным массивом признаков достичь высоких значений точности, используя наивный Байесовский алгоритм не получается. Следующий алгоритм, который будет испытан – KNN.

Используется трансформер, строящий лучший массив признаков:

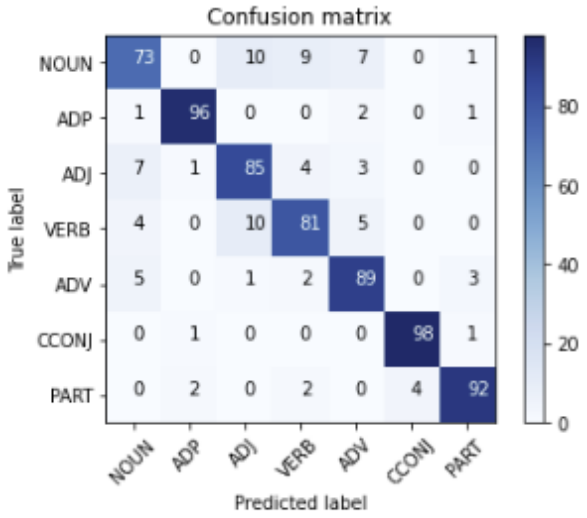
```
def CarArr(_word):
    word = copy.deepcopy(_word)

    if(len(word) < 3):
        res = np.array([len(word),
                        LetterPos(word[0]),
                        LetterPos(('a')),
                        LetterPos((word[len(word) - 2])),
                        LetterPos((word[len(word) - 1]))], dtype = object)
    else:
        res = np.array([len(word),
                        LetterPos(word[0]),
                        LetterPos((word[len(word) - 2])),
                        LetterPos((word[len(word) - 2])),
                        LetterPos((word[len(word) - 1]))], dtype = object)
    return res
```

```
class transform(skl.base.TransformerMixin):

    def fit(self,a,b):
        return self

    def transform(self,a):
        res = copy.deepcopy(a)
        for i in range(len(a)):
            res[i] = CarArr(a[i])

        return np.array(res,dtype = 'int')
```

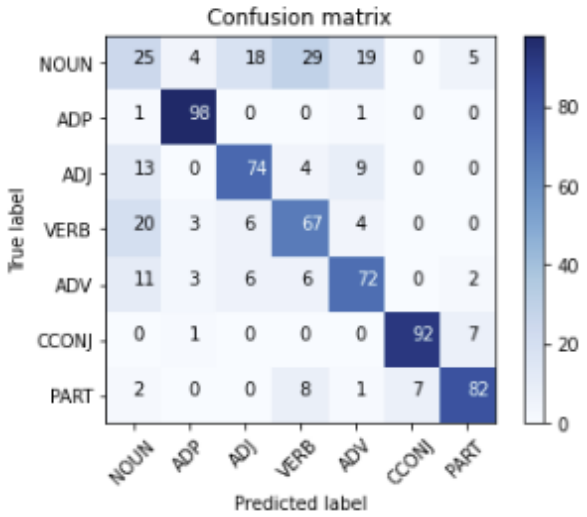
Так как моя реализация KNN никак не оптимизирована и работает крайне медленно, я сгенерировал уменьшенный датасет в 100 экземпляров для тренировки и 100 для тестирования.

Метрики полученной модели: (на матрицах спутывания истинные числа)

```
accuracy on train set:
0.8771428571428571
accuracy on test set:
0.7285714285714285
train set:
Confusion matrix, without normalization
```

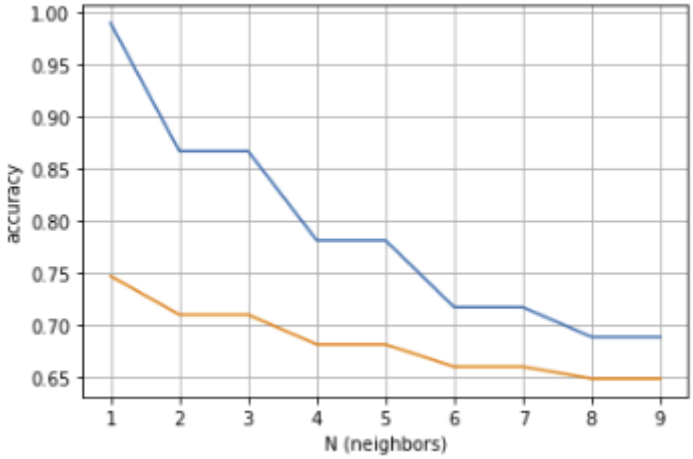



```
test set:
Confusion matrix, without normalization
```



Получена поразительная точность. Идея использования KNN определенно стоит дальнейших исследований.

Определим, какое число соседей дает наилучшую точность.



По графику видно, что чем меньше соседей рассматривается, тем лучше. При k = 1 классификатор превращается в своего рода словарь, который ищет классифицируемое слово в памяти, и, если не находит, выдает наиболее близкое по признакам слово.

Почему KNN работает?

KNN, также, как и регрессии основан на представлении данных, как координат в гиперпространстве параметров, потому кажется странным, что, несмотря на то, что буквы алфавита невозможно представить в таком виде, не добавив новых связей между ними, KNN дает такой высокий результат.

Мое предположение заключается в том, что, благодаря обработке входных слов, алгоритм часто находит среди данных для обучения слово, массив признаков которого после обработки в точности соответствует массиву признаков искомого слова. То, что соседние буквы алфавита дают похожий результат, не должно негативно влиять на точность классификации таких примеров, а ошибки должны чаще возникать, когда отличие находится в коде буквы, а не в длине.

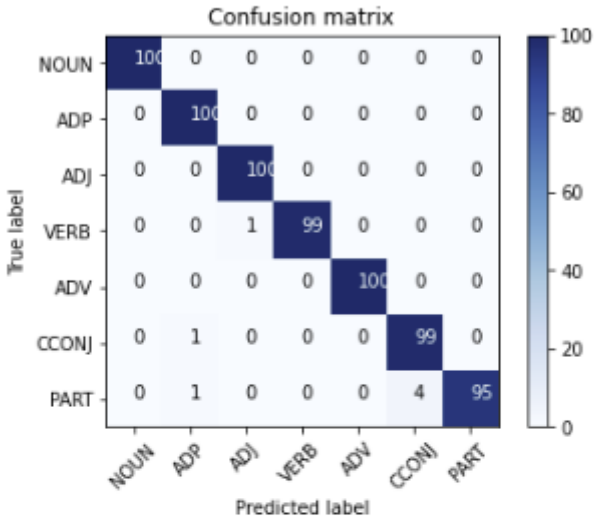
Данную гипотезу можно проверить двумя способами.

1. Увеличить расстояние между соседними кодами букв, чтобы расстояние между объектами с соседними буквами алфавита было существенно больше, чем расстояние между объектами с близкими длинами. Тогда алгоритм будет, где это возможно, выбирать пары слов, содержащие одни и те же буквы на ключевых позициях, а не пары слов одинаковой длины. Точность должна увеличиться. Точно также, если увеличить расстояние между соседними длинами, точность должна упасть.
2. Проверить, какие различия приводят к ошибкам, напрямую посмотрев данные о ближайшем соседе.

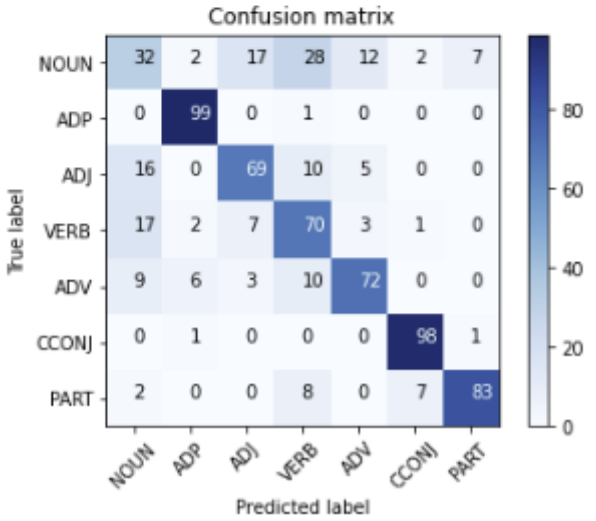
Так как наибольшую точность дает значение k = 1, т.е. алгоритм ищет лишь одного ближайшего соседа, будем экспериментировать, используя его.

Расстояние между соседними значениями длины равно расстоянию между буквами

```
accuracy on train set:
0.99
accuracy on test set:
0.7471428571428571
train set:
Confusion matrix, without normalization
```

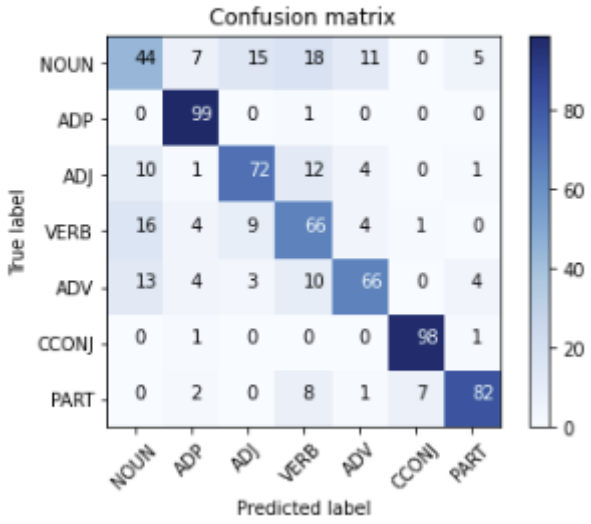



```
test set:
Confusion matrix, without normalization
```



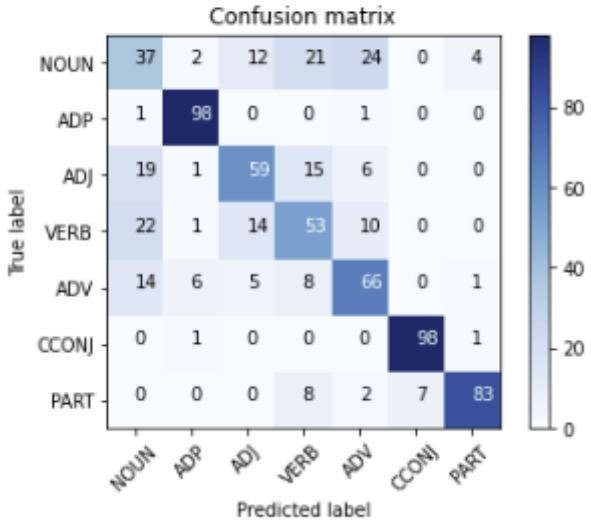
Расстояние между буквами больше, чем между длинами в 10 раз

```
accuracy on test set:
0.7528571428571429
test set:
Confusion matrix, without normalization
```

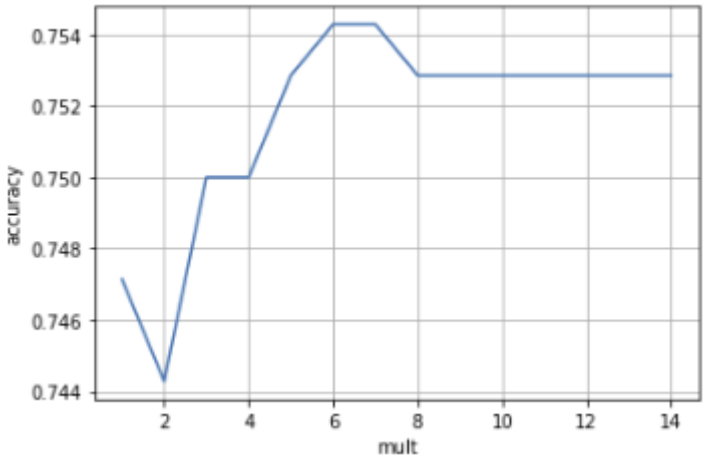


Расстояние между длинами больше, чем между буквами в 10 раз

```
accuracy on test set:
0.7057142857142857
test set:
Confusion matrix, without normalization
```



Как видно, первая гипотеза подтвердилась. По матрице спутывания видно, что при увеличении расстояния между буквами, точность определения некоторых классов выросла, в то время, как других – упала. Возможно, можно подобрать такой множитель, что результирующая общая точность будет еще больше.



Наиболее вымокая точность, которую удалось достичь:

```
mult = 6.0
acc = 0.7542857142857143
```

Она не сильно больше, чем точность при равном расстоянии между длинами и буквами.

Для проверки второго предположения, я, используя KNN с числом соседей k = 1, выводил элемент, который оказался наиболее близким к классифицируемому, и сравнивал их между собой. Сравнения дали такой результат:

```
avg count of missmatched letters per correctly identified word: 1.0597014925373134
avg count of missmatched letters per wrongly   identified word: 2.515151515151515

avg difference in length with correctly identified words: 1.9850746268656716
avg difference in length with wrongly   identified words: 3.393939393939394

how often do lengths match (higher = more often)
avg difference in length with correctly identified words: 0.4925373134328358
avg difference in length with wrongly   identified words: 0.030303030303030304

% of perfect matches in correctly identified words: 35.82089552238806
% of perfect matches in wrongly   identified words: 0.0

% of perfect len matches with letter missmatch in correctly identified words: 13.432835820895523
% of perfect len matches with letter missmatch in wrongly   identified words: 3.0303030303030303

% of perfect letter matches with len missmatch in correctly identified words: 7.462686567164178
% of perfect letter matches with len missmatch in wrongly   identified words: 0.0
```

В целом, ничего удивительного. Чем дальше слово от ближайшего соседа, тем скорее оно будет определено неверно.

Интересно, что среди слов, классифицированных неверно, есть слова, у которых совпала длина, но не совпали буквы, но нет слов, у которых совпали буквы, но не совпала длина, т.е. все такие слова были определены верно. Это подтверждает то, что для верного определения части речи с помощью KNN важнее, чтобы совпали буквы на ключевых позициях, а не длина. Также всегда, когда признаки слова совпадали с ближайшим соседом, его часть речи определялась верно.

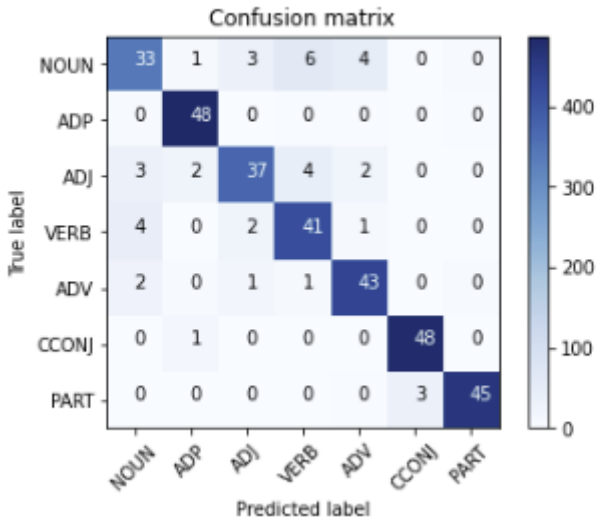
Разумеется, стоит помнить, что это исследование было проведено на выборке в 700 слов, и потому, точные цифры могут несколько отличаться, однако, как вытаскивать ближайших соседей из классификатора из skl я не знаю, а на моем классификаторе большее количество слов классифицировать будет слишком долго.

В отличие от моей реализации, KNN из модуля skl работает очень быстро и не сильно замедляется при увеличении выборок.

Пример классификатора KNN, созданного с помощью модуля skl. И классифицирующего сбалансированный датасет в 3500 слов (числа на матрице спутывания уменьшены в 10 раз)

```
pipe = Pipeline([
    ('transformer',transform(n = 6)),
    ('classifier',KNeighborsClassifier(n_neighbors = 1))])
pipe.fit(larrDC,larrLB)

print("accuracy on train set:")
print(pipe.score(larrDC,larrLB))
print("accuracy on test set:")
print(pipe.score(tarrDC,tarrLB))

confusion_matrix(pipe,tarrDC,tarrLB,
                 class_names = Classes,
                 normalize = False,
                 shrink = 10)
```

```
accuracy on train set:
0.9857142857142858
accuracy on test set:
0.8542857142857143
Confusion matrix, without normalization
```

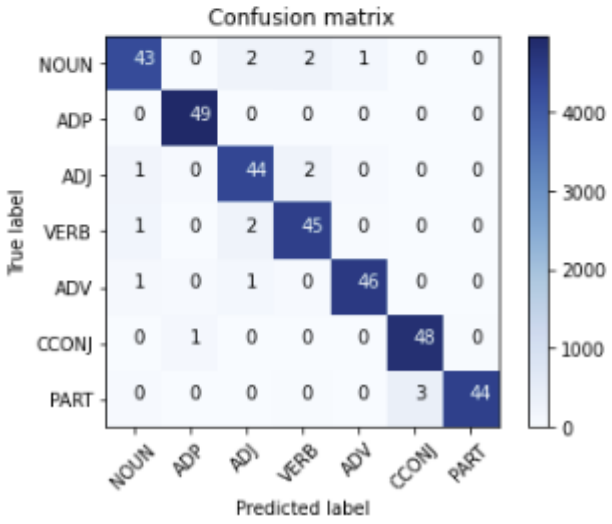


Использование KNN на датасете длинною в 7*5000 (35000) слов (числа на матрице спутывания уменьшены в 100 раз)

```
larrLB,larrDC,tarrLB,tarrDC = create_dataset(selLen = 50000, ClLen = 5000)
```

```
pipe = Pipeline([
    ('transformer',transform(n = 6)),
    ('classifier',KNeighborsClassifier(n_neighbors = 1))])
pipe.fit(larrDC,larrLB)
print("accuracy on train set:")
print(pipe.score(larrDC,larrLB))
print("accuracy on test set:")
print(pipe.score(tarrDC,tarrLB))

confusion_matrix(pipe,tarrDC,tarrLB,
                 class_names = Classes,
                 normalize = False,
                 shrink = 100)
```

```
accuracy on train set:
0.9685428571428571
accuracy on test set:
0.9193714285714286
Confusion matrix, without normalization
```

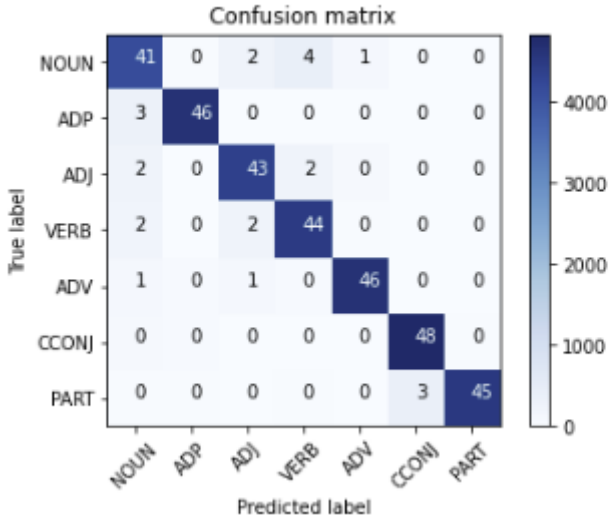


Что неудивительно, с увеличением обучающей выборки точность на тестовом датасете растет.

Также, в этом примере гораздо более выражено то, что умножение кодов букв действительно увеличивает точность. Та же модель, но с мультипликатором трансформера = 1:

```
pipe = Pipeline([
    ('transformer',transform(n = 1)),
    ('classifier',KNeighborsClassifier(n_neighbors = 1))])
pipe.fit(larrDC,larrLB)
print("accuracy on train set:")
print(pipe.score(larrDC,larrLB))
print("accuracy on test set:")
print(pipe.score(tarrDC,tarrLB))

confusion_matrix(pipe,tarrDC,tarrLB,
                 class_names = Classes,
                 normalize = False,
                 shrink = 100)
```

```
accuracy on train set:
0.9601714285714286
accuracy on test set:
0.9066
Confusion matrix, without normalization
```



Раз уж KNN оказалась неожиданно эффективной в классификации слов по частям речи, стоит также проверить, как покажут себя модели-регрессии. Очевидно, что для реализации n-class classification с помощью регрессий, предназначенных для бинарной классификации, нужно решать задачу выделения класса k. Для этого напишем такой модификатор датасета:

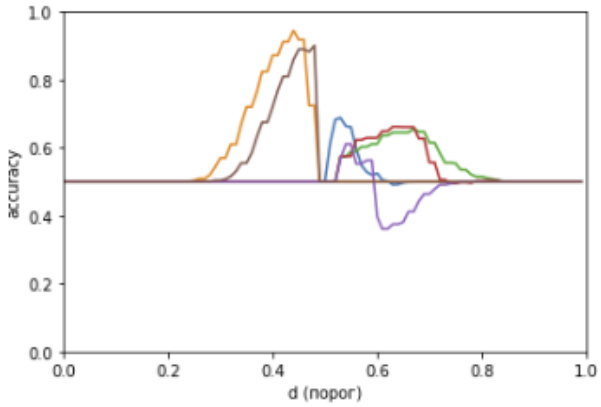
```
def separate_class(cl,larrDC,larrLB):
    count = [0,0]
    lim = 500
    words = []
    sel_class = cl
    larr_L = []
    for i in range(len(larrLB)):
        if(larrLB[i] == sel_class):
            if(count[1] < lim):
                words.append(larrDC[i])
                larr_L.append(1)
                count[1] += 1

        else:
            if(count[0] < lim):
                words.append(larrDC[i])
                larr_L.append(0)
                count[0] += 1
    return words,larr_L
```

Он преобразует датасет с n классами в сбалансированный бинарный датасет, где класс cl обозначен единицей, а остальные классы – нулем. Так как эта функция воздействует не только на значения признаков, но и на лейблы, представить ее в виде трансформера в конвейере не получилось.

Попробуем выделить каждый класс, имея в качестве признаков только длину слова:

```
pipe = Pipeline([('transformer',transform()),('classifier',logistic_regression(1,lr = 0.0001))])
```

```
pipe.fit(separate_class(0,larrDC,larrLB)[0],separate_class(0,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(0,larrDC,larrLB)[0],separate_class(0,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(2,larrDC,larrLB)[0],separate_class(2,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(2,larrDC,larrLB)[0],separate_class(2,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(3,larrDC,larrLB)[0],separate_class(3,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(3,larrDC,larrLB)[0],separate_class(3,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(4,larrDC,larrLB)[0],separate_class(4,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(4,larrDC,larrLB)[0],separate_class(4,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(5,larrDC,larrLB)[0],separate_class(5,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(5,larrDC,larrLB)[0],separate_class(5,larrDC,larrLB)[1],d_span = [0,1])
```

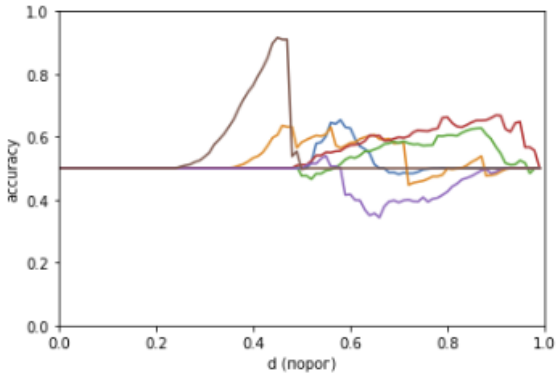


Единственные два класса, которые получилось выделить хорошо (высокие пики на графике) – это предлог и союз. Это было ожидаемо, однако, с введением других признаков, ситуация становится только хуже.

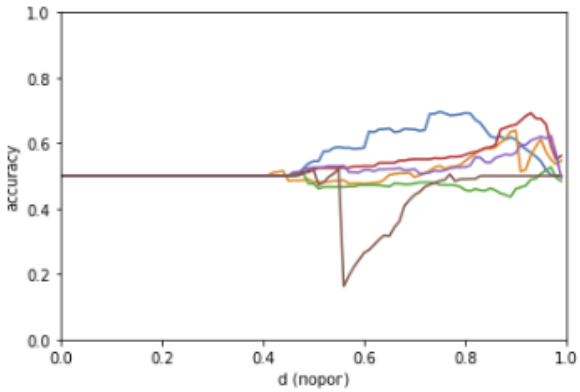
Длина + последняя буква

```
pipe = Pipeline([('transformer',transform()),('classifier',logistic_regression(2,lr = 0.0001))])
```

```
pipe.fit(separate_class(0,larrDC,larrLB)[0],separate_class(0,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(0,larrDC,larrLB)[0],separate_class(0,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(2,larrDC,larrLB)[0],separate_class(2,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(2,larrDC,larrLB)[0],separate_class(2,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(3,larrDC,larrLB)[0],separate_class(3,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(3,larrDC,larrLB)[0],separate_class(3,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(4,larrDC,larrLB)[0],separate_class(4,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(4,larrDC,larrLB)[0],separate_class(4,larrDC,larrLB)[1],d_span = [0,1])
pipe.fit(separate_class(5,larrDC,larrLB)[0],separate_class(5,larrDC,larrLB)[1])
DACforPipe(pipe,separate_class(5,larrDC,larrLB)[0],separate_class(5,larrDC,larrLB)[1],d_span = [0,1])
```



Все признаки



(я мог бы подписать графики, но и так видно, что ничего не работает, так что какая разница)

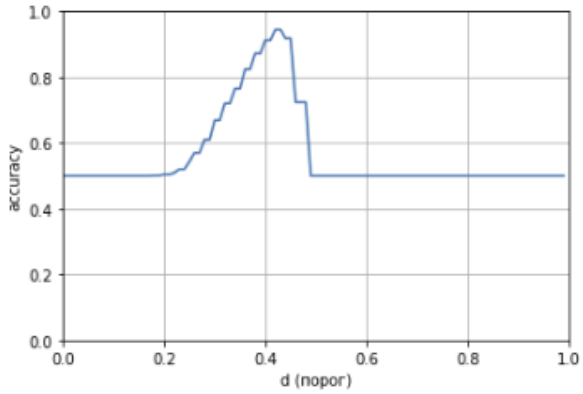
Разумеется, регрессии можно применять для весьма точного выделения предлогов по длине, но ту же задачу можно решить, просто вручную задав порог длины в 3-4 буквы, которые предлоги редко преодолевают. Тот же KNN прекрасно справляется с задачей выделения предлогов внутри задачи многоклассовой классификации, что видно по его матрице спутывания.

Пример реализации задачи выделения предлогов с помощью логистической регрессии

```
pipe = Pipeline([('transformer',transform()),('classifier',logistic_regression(1,lr = 0.0001))])
pipe.fit(separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1])
pipe[1].d = 0.41
print("accuracy on train set:")
print(pipe.score(separate_class(1,larrDC,larrLB)[0],separate_class(1,larrDC,larrLB)[1]))
print("accuracy on test set:")
print(pipe.score(separate_class(1,tarrDC,tarrLB)[0],separate_class(1,tarrDC,tarrLB)[1]))
```

```
accuracy on train set:
0.912
accuracy on test set:
0.856
```

Кривая порог-точность

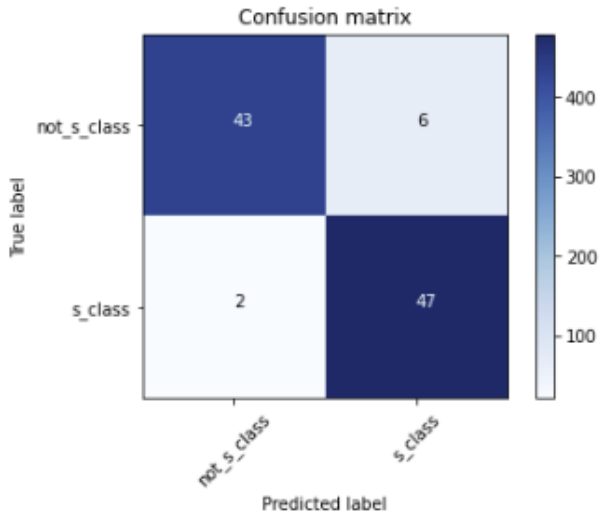


Базовые метрики

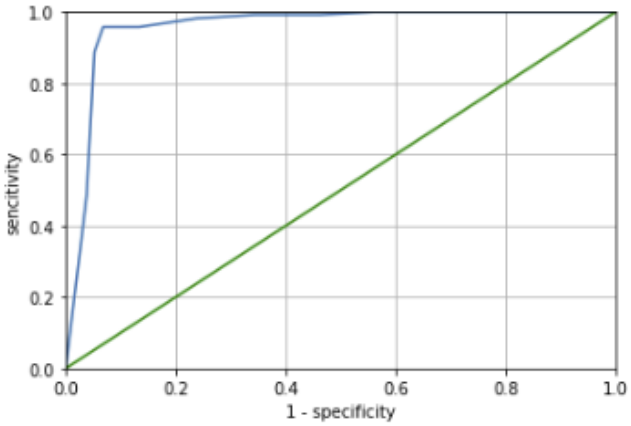
```
accuracy: 0.912 (чем больше ложных, тем меньше)
precision: 0.8772893772893773 (чем больше ложнопозитивных, тем меньше)
recall: 0.958 (чем больше ложнонегативных, тем меньше)
specificity: 0.866 (чем больше негативных классифицировано неверно, тем меньше)
sencitivity: 0.958 (чем больше позитивных классифицировано неверно, тем меньше)
```

Матрица спутывания

```
Confusion matrix, without normalization
```



Кривая ROC



2.3. Сохранение наиболее эффективной модели.

Очевидно, что наиболее эффективной для решения данной задачи моделью оказалась KNN. Она единственная показала приемлемый результат и, на большом датасете перешла границу точности в 90% на тестовой выборке. На втором месте оказался наивный байесовский классификатор с точностью ~60%, остальные модели не удалось эффективно применить для решения поставленной задачи. Сохраним эти две модели.

KNN skl

```
with open('skl_knn.pkl', 'wb') as pickle_out:
    pickle.dump(pipe, pickle_out)

with open('skl_knn_dataset.pkl', 'wb') as pickle_out:
    pickle.dump((larrLB,larrDC,tarrLB,tarrDC), pickle_out)
```

My KNN

```
with open('my_knn.pkl', 'wb') as pickle_out:
    pickle.dump(pipe, pickle_out)

with open('my_knn_dataset.pkl', 'wb') as pickle_out:
    pickle.dump((larrLB,larrDC,tarrLB,tarrDC), pickle_out)
```

Оптимальная настройка naive bayes

```
with open('my_nb.pkl', 'wb') as pickle_out:
    pickle.dump(pipe, pickle_out)

with open('my_nb_dataset.pkl', 'wb') as pickle_out:
    pickle.dump((larrLB,larrDC,tarrLB,tarrDC), pickle_out)
```

3. Выводы

В ходе выполнения этой лабораторной работы я реализовал классификаторы LR, LogR, SVM, KNN и NB, определил наиболее подходящий классификатор для решения задачи классификации слов русского языка по частям речи и подобрал оптимальные гиперпараметры.

Я научился использовать конвейеры в skl и стал лучше понимать принципы работы и основные отличия вышеперечисленных алгоритмов. Также я получил опыт работы с настоящим nlp-датасетом, что может пригодиться мне в будущем.

У меня получилось добиться классификации тестового датасета с точностью в 91.94%, что больше, чем я ожидал, но все же недостаточно хорошо, чтобы использовать такой классификатор в настоящих nlp-системах. Возможно, точность можно улучшить, совместив несколько разнотипных классификаторов (KNN + NB), либо используя бутстрап или бустинг.

Интересно, что KNN показал наибольшую точность как при испытании на искусственном датасете, так и при решении основной задачи. Единственным, но при этом критическим недостатком этого алгоритма оказалась низкая скорость классификации, хотя реализация в skl работает относительно быстро.