Московский авиационный институт
(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа № 2

по курсу «Машинное обучение»

Студент: Гаврилов М.С.

Группа: 80-306б

Преподаватель:

Оценка:

Москва, 2022

1. Постановка задачи

Вы построили базовые (слабые) модели машинного обучения под вашу задачу. Некоторые задачи показали себя не очень, некоторые показали себя хорошо. Как выяснилось, вашим инвесторам показалось этого мало и они хотят, чтобы вы построили модели посерьезней и поточнее. Вы вспомнили, что когда то вы проходили курс машинного обучения и слышали что есть способ улучшить результаты вашей задачи: ансамбли: беггинг, пастинг, бустинг и стекинг, а также классификация путем жесткого и мягкого голосования и вы решили это опробовать. Требования к написанным классам вы оставляете теми же, что и в предыдущей работе. Будьте аккуратны в оптимизации целевой метрики и учитывайте несбалансированность классов. Требования к отчету сохраняются такими же)

Ваша задача:

1. Используя модели которые вы реализовали в предыдущей лабораторной работе, реализовать два подхода для построения ансамблей: жесткое и мягкое голосование, однако учтите, некоторые модели не предусматривают оценку вероятностей, например SVM и потому вам необходимо будет оценивать вероятности
2. Реализовать дерево решений
3. Реализовать случайный лес
4. Воспользоваться готовой коробочной реализацией градиентного бустинга для решения вашей задачи
   Для всех моделей провести fine-tuning.

2. Выполнение работы

Поставленная в лабораторной работе №0 задача заключается в классификации слов русского языка по частям речи. Для получания размеченных данных используется датасет nerus.

В лабораторной работе №1 была достигнута точность классификации 91.94% с помощью коробочной реализации алгоритма KNN на выборке из 35000 слов и ~75% с использованием собственной реализации KNN на выборке из 700 слов (в первую очередь из-за того, что собственная реализация работала медленно). В этой лабораторной работе попытаемся с использованием ансамблей превзойти эти значения.

1. Беггинг

Прежде чем реализовывать беггинг, нужно сделать бутстрап. Это метод разделения выборки, который создает k подвыборок длиной $l \leq L$, где L — длина основной выборки. Элементы добавляются в подвыборки случайно, один элемент может несколько раз попасть в одну и ту же выборку.

Функция, реализующая бутстрап:

```
def bootstrap(data_X_,data_L_,num_sel,sel_size = None):
    #по умолчанию бьет выборку на равные части (случайно выбирая элементы с повторением)
    #sel size в пределах от 0 до 1 задает размер частей относительно общего размера выборки
    #sel size больше 1 задает абсолютный размер частей
    data_X,data_L = skl.utils.shuffle(data_X_,data_L_)
    if(sel_size == None):
        sel_size = int(len(data_X) / num_sel)
    if(sel_size <= 1):
        sel_size = int(sel_size*len(data_X))

    mod_base = [[data_X[i],data_L[i]] for i in range(len(data_X))]

    all_sel = []
    for k in range(num_sel):
        cur_sel = []
        for i in range(sel_size):
            cur_sel.append(random.choice(mod_base))
        all_sel.append(cur_sel)

    return np.array(all_sel,dtype = object)
```

Основная идея беггинга: выборка разделяется бутстрапом, затем на каждой подвыборке обучается отдельный классификатор. Все классификаторы однотипны. При классификации происходит голосование (жесткое или мягкое) моделей. При жестком голосовании выбирается класс, который выбрало большинство моделей. При мягком производится учет вероятности, с которой, по «мнению» моделей, объект принадлежит к выбранному им классу.

Реализация беггинга:

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

    def __init__(self,model_exmpl,model_count,soft_voting = False, sel_size = None):
        self.models = []
        for i in range(model_count):
            self.models.append(copy.deepcopy(model_exmpl))
            if(soft_voting):
                self.models[i].set_return_probability(True)

        self.spl_cnt = model_count
        self.soft_voting = soft_voting
        self.sel_size = sel_size

    def fit(self,data,labels,verbal = False):
        self.class_cnt = np.unique(labels)
        splits = bootstrap(data,labels,self.spl_cnt,self.sel_size)
        for i in range(self.spl_cnt):
            self.models[i].fit(splits[i,:,0],splits[i,:,1])

    def predict(self,data):
        class_score = np.zeros_like(self.class_cnt)

        if(self.soft_voting):
            for i in self.models:
                pred = i.predict([data])
                class_score[pred[0]] += pred[1]
        else:
            for i in self.models:
                pred = i.predict([data])
                class_score[pred] += 1

        return np.argmax(class_score)

    def accuracy(self,data,labels,verbal = False):
        correct = 0
        overall = 0
        sep_acc = []
        for mod in self.models:
            sep_acc.append(mod.score(data,labels))

        for i in range(len(data)):
            overall += 1
            cl = self.predict(data[i])
            if(cl == labels[i]):
                correct += 1
            if(verbal):
                print(pred)
                print(cl)
            res = [correct/overall]
            for i in sep_acc:
                res.append(i)
        return np.array(res)

    def score(self,data,labels,verbal = False):
        return self.accuracy(data,labels,verbal = verbal)
```

Для использования мягкого голосования необходимо, чтобы базовые классификаторы поддерживали метод **set_return_probability()**, и в режиме возврата вероятностей выдавали результат классификации в виде [n,p], где n — определенный класс, p — «уверенность» классификатора.

Функция, реализующая кросс-валидацию:

```
class RCV():

    def __init__(self,arr_train,arr_test,pipe = None):
        self.train_X = arr_train[0]
        self.train_L = arr_train[1]
        self.test_X = arr_train[0]
        self.test_L = arr_train[1]
        self.pipe = pipe

    def apply_model(self,pipe):
        self.pipe = pipe

    def validate(self,i = 10,pipe = None,ratio = 0.5,verbal = False):
        if(pipe == None):
            pipe = self.pipe
        if(pipe == None):
            return None

        arr = []
        validation_array_X = np.concatenate([self.train_X,self.test_X],axis = 0)
        validation_array_Y = np.concatenate([self.train_L,self.test_L],axis = 0)

        for j in range(i):
            cur_validation_array_X, cur_validation_array_Y =
                skl.utils.shuffle(validation_array_X,validation_array_Y)
            train_X,test_X = np.split(cur_validation_array_X,
                                      [int(len(cur_validation_array_X)*ratio)])
            train_L,test_L = np.split(cur_validation_array_Y,
                                      [int(len(cur_validation_array_Y)*ratio)])

            train_X = train_X.tolist()
            train_L = train_L.tolist()

            test_X = test_X.tolist()
            test_L = test_L.tolist()

            self.pipe.fit(train_X,train_L)
            score = self.pipe.score(test_X,test_L)
            arr.append(score)

            if(verbal):
                print("test {} of {} | done".format(j,i))

        return np.array(arr,dtype=object)
```

Данная функция осуществляет кросс-валидацию случайными разбиениями: выборка перемешивается и разбивается надвое (по умолчанию), на одной половине обучается классификатор, на второй выполняется проверка. Преимущество данного метода кросс-валидации в том, что валидацию можно выполнять сколько угодно раз, вычисляя среднее значение метрик с очень высокой точностью. Эта функция возвращает только точности на тестовых выборках.

Оценка эффективности беггинга:

Для тестирования используются два алгоритма, показавшие себя наилучшим образом в работе, посвященной слабым алгоритмам машинного обучения: knn и наивный байесовский алгоритм. Вывод значений точности ансамблей осуществляется следующим образов: в массиве на первой позиции стоит точность всего ансамбля в целом, дальше под ряд записаны значения точности каждого классификатора по отдельности. Такой способ записи позволяет быстро оценить эффективность ансамбля (насколько точность целиком выше точности составных элементов).

Тесты с использованием наивного байесовского алгоритма на сбалансированной выборке из 3500 слов (7 классов):

Тесты с NB

20x NB, hard voting

```
bg = bagging(Naive_Bayes(),20,sel_size = 300)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.6270357142857141, 0.5397857142857142, 0.5369642857142857,
       0.5367857142857142, 0.5259285714285713, 0.526892857142857,
       0.5277142857142856, 0.5271785714285715, 0.5385, 0.5282142857142856,
       0.5305714285714285, 0.5283214285714285, 0.54025,
       0.5387857142857143, 0.55025, 0.5349285714285714,
       0.5379642857142859, 0.5432857142857143, 0.5468571428571428,
       0.54225, 0.5333214285714286], dtype=object)
```

Точность ансамбля ~0.62, точность составляющих ~0.53

20x NB, soft voting

```
bg = bagging(Naive_Bayes(),20,soft_voting = True,sel_size = 300)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.6989214285714291, 0.5291714285714285, 0.5395285714285712,
       0.5352285714285716, 0.5398357142857141, 0.5349857142857142,
       0.5410714285714286, 0.5321785714285712, 0.5460071428571429,
       0.5421571428571427, 0.5366357142857141, 0.5327785714285715,
       0.5375142857142857, 0.5339857142857142, 0.5365642857142856,
       0.5273357142857145, 0.5326428571428571, 0.5321714285714286,
       0.5365428571428572, 0.5403642857142859, 0.5335499999999999],
      dtype=object)
```

Точность ансамбля ~0.69, точность составляющих ~0.53

20x NB, soft voting, на больших подвыборках

```
bg = bagging(Naive_Bayes(),20,soft_voting = True,sel_size = 0.5)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.7002714285714284, 0.6487571428571427, 0.6452857142857146,
       0.6452999999999999, 0.6420285714285717, 0.6408857142857143,
       0.6426285714285714, 0.6448999999999999, 0.6453714285714288,
       0.6483428571428571, 0.6453714285714287, 0.6442857142857144,
       0.6387428571428572, 0.6428857142857143, 0.6467714285714287,
       0.6430571428571428, 0.6444285714285714, 0.6421000000000001,
       0.641442857142857, 0.6388714285714284, 0.6414714285714284],
      dtype=object)
```

Точность ансамбля ~0.7, точность составляющих ~0.64

20x NB, soft voting, на подвыборках размером с базовую выборку.

```
bg = bagging(Naive_Bayes(),20,soft_voting = True,sel_size = 1)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.7038857142857143, 0.6703428571428574, 0.6723571428571431,
       0.6689857142857142, 0.6720714285714284, 0.6747, 0.6667857142857142,
       0.6682857142857144, 0.6694, 0.6705714285714284, 0.6663142857142859,
       0.6667857142857139, 0.6662571428571431, 0.6718571428571427,
       0.6704000000000001, 0.6698714285714287, 0.6703857142857141,
       0.6699142857142859, 0.6693714285714285, 0.6682285714285715,
       0.6700571428571429], dtype=object)
```

Точность ансамбля ~0.7, точность составляющих ~0.67

Тесты с KNN

k = 1 (такая конфигурация дала наилучший результат при тестировании с одиночным KNN), hard voting.

```
bg = bagging(KNN(k = 1),20,soft_voting = False,sel_size = 0.5)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.8210714285714287, 0.7486428571428572, 0.7502857142857142,
       0.729642857142857, 0.7516428571428572, 0.7460714285714285,
       0.7440714285714287, 0.7445714285714286, 0.7517142857142858,
       0.7569285714285714, 0.7485714285714286, 0.7449285714285716,
       0.7353571428571428, 0.7461428571428572, 0.7602857142857142,
       0.7486428571428572, 0.7500000000000001, 0.7567142857142858,
       0.7340714285714285, 0.7572142857142856, 0.7458571428571428],
      dtype=object)
```

Точность ансамбля ~0.82, точность составляющих ~0.74

k = 1, soft voting.

```
bg = bagging(KNN(k = 1),20,soft_voting = True,sel_size = 0.5)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.8227142857142857, 0.7518571428571429, 0.7464285714285716, 0.75,
       0.7404999999999999, 0.7397857142857143, 0.7442857142857143,
       0.7329285714285716, 0.7577142857142858, 0.7434285714285713,
       0.7477857142857143, 0.7592142857142858, 0.749642857142857,
       0.7374999999999999, 0.7544285714285713, 0.7517857142857143, 0.741,
       0.7549285714285714, 0.7444999999999998, 0.7535000000000001,
       0.7572857142857141], dtype=object)
```

Точность ансамбля ~0.82, точность составляющих ~0.74

Так как «уверенность» классификатора в knn определяется тем, сколько элементов из тех, по которым был определен класс данного объекта имеет тот же класс, что и определенный, очевидно, что при k = 1 разницы между жестким и мягким голосованием не будет. Попробуем k > 1.

k = 3, hard voting

```
bg = bagging(KNN(k = 3),20,soft_voting = False,sel_size = 0.5)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.7420000000000002, 0.6723571428571429, 0.6833571428571429,
       0.6742142857142857, 0.6717142857142857, 0.6534285714285712,
       0.6737857142857144, 0.6737857142857142, 0.6646428571428571,
       0.6671428571428571, 0.6677857142857142, 0.667, 0.66399999999
       0.6678571428571429, 0.6672142857142858, 0.6734285714285715,
       0.6694285714285715, 0.6657142857142857, 0.6692142857142855,
       0.6682142857142858, 0.672], dtype=object)
```

Точность ансамбля ~0.74, точность составляющих ~0.67

k = 3, soft voting

```
bg = bagging(KNN(k = 3),20,soft_voting = True,sel_size = 0.5)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

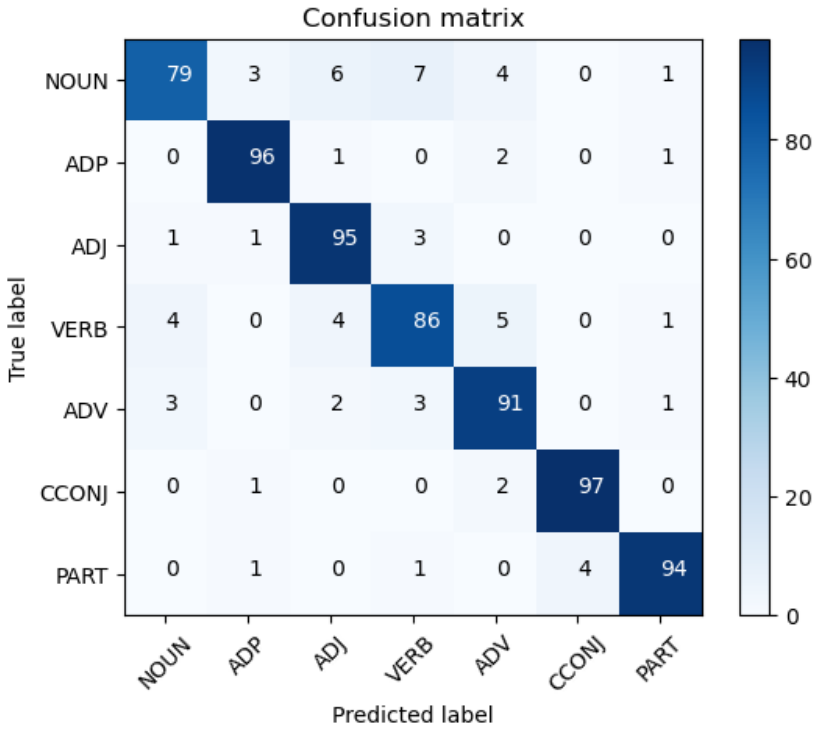
```
array([0.750642857142857, 0.6752142857142858, 0.6698571428571429,
       0.6712142857142858, 0.6702142857142857, 0.6742857142857143, 0.6695,
       0.6719285714285715, 0.68, 0.6757857142857143, 0.6729999999999999,
       0.6763571428571428, 0.6705000000000001, 0.6676428571428572,
       0.6528571428571428, 0.6705714285714286, 0.677, 0.6680000000000003,
       0.6639999999999999, 0.6742857142857144, 0.6663571428571429],
      dtype=object)
```

Точность ансамбля ~0.75, точность составляющих ~0.67

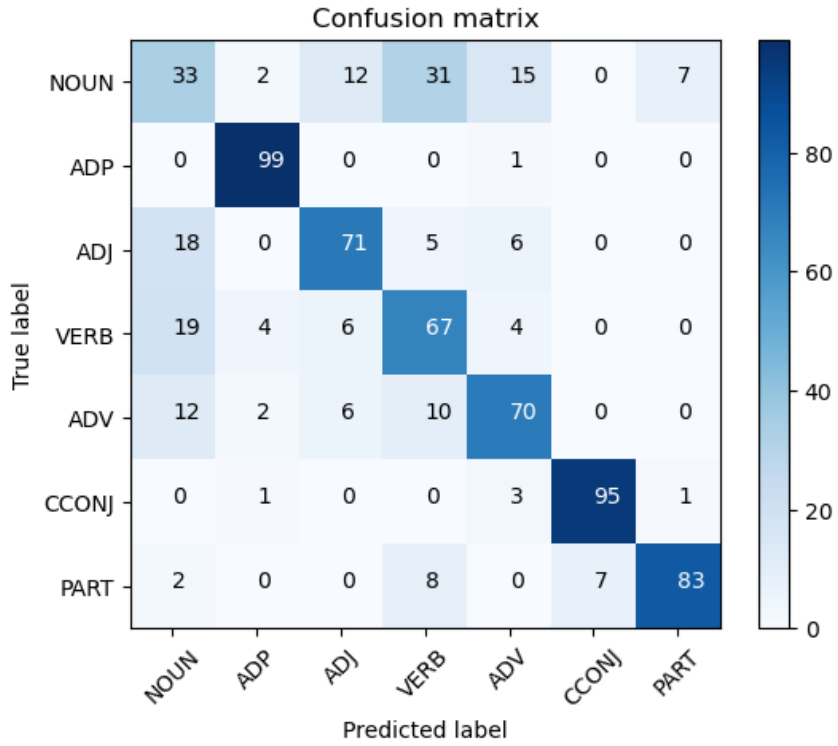
Наблюдается увеличение точности при мягком голосовании, но сама точность в сравнении с k = 1 столь низка, что рассматривать такую конфигурацию не имеет смысла.

Матрица спутывания для беггинга (20xknn[k=1], мягкое голосование отключено, размер подвыборок = половине основной выборки

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




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



2. Пастинг

Этот алгоритм работает, как беггинг, только в подвыборках не допускаются повторения.

Для этого используется новая функция разбиения на подвыборки;

```
def unique_bootstrap(data_X_,data_L_,num_sel,sel_size = None):
    #то же, что и bootstrap, только без повторений в выборках
    data_X,data_L = skl.utils.shuffle(data_X_,data_L_)
    if(sel_size == None):
        sel_size = int(len(data_X) / num_sel)
    if(sel_size <= 1):
        sel_size = int(sel_size*len(data_X))

    mod_base = [[data_X[i],data_L[i]] for i in range(len(data_X))]

    all_sel = []
    for k in range(num_sel):
        cur_sel = []
        ind_taken = []
        for i in range(sel_size):
            cur = int(random.random()*len(mod_base))
            while(cur in ind_taken):
                cur = int(random.random()*len(mod_base))
            cur_sel.append(mod_base[cur])
            ind_taken.append(cur)
        all_sel.append(cur_sel)

    return np.array(all_sel,dtype = object)
```

Сама функция, реализующая пастинг, не отличается от функции, реализующей беггинг ничем, кроме метода разбиения выборки.

Оценка эффективности пастинга:

Сравнение результатов на алгоритме NB с мягким голосованием и без:

```
pt = pasting(Naive_Bayes(),20)
pipe = Pipeline([('transformer',transform()),('classifier',pt)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.6186071428571428, 0.5438214285714285, 0.5354642857142856,
       0.5195714285714286, 0.5346785714285713, 0.5289285714285714,
       0.5117499999999999, 0.5288214285714288, 0.5246071428571429,
       0.5190357142857144, 0.534, 0.5230714285714286, 0.524999999999
       0.5424999999999999, 0.5183571428571428, 0.5438571428571428,
       0.5152857142857141, 0.5293214285714285, 0.542, 0.542571428571
       0.5141428571428571], dtype=object)
```

```
pt = pasting(Naive_Bayes(),20,soft_voting = True)
pipe = Pipeline([('transformer',transform()),('classifier',pt)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.6779285714285714, 0.5249642857142857, 0.5294642857142857,
       0.5269285714285715, 0.5107857142857142, 0.5261785714285716,
       0.5345714285714287, 0.5308928571428572, 0.5523571428571429,
       0.5224285714285714, 0.5164642857142857, 0.5309285714285714,
       0.5244285714285714, 0.5315000000000001, 0.5264285714285715,
       0.5340357142857142, 0.532607142857143, 0.5363571428571428,
       0.5246785714285713, 0.5401428571428571, 0.5117499999999999],
      dtype=object)
```

Hard voting:

Точность ансамбля ~0.61, точность составляющих ~0.53

Soft voting:

Точность ансамбля ~0.61, точность составляющих ~0.53

Результаты повторяют картину, полученную при исследовании бегинга, но с меньшей точностью в обоих случаях.

Самый интересный эффект, при сравнении бегинга с пастингом возникает при биении выборки на подвыборки размером с нее саму:

```
pt = pasting(Naive_Bayes(),20,soft_voting = True,sel_size = 1)
pipe = Pipeline([('transformer',transform()),('classifier',pt)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.6990714285714283, 0.6990714285714283, 0.6990714285714283,
       0.6990714285714283, 0.6990714285714283, 0.6990714285714283,
       0.6990714285714283, 0.6990714285714283, 0.6990714285714283,
       0.6990714285714283, 0.6990714285714283, 0.6990714285714283,
       0.6990714285714283, 0.6990714285714283, 0.6990714285714283,
       0.6990714285714283, 0.6990714285714283, 0.6990714285714283],
      dtype=object)
```

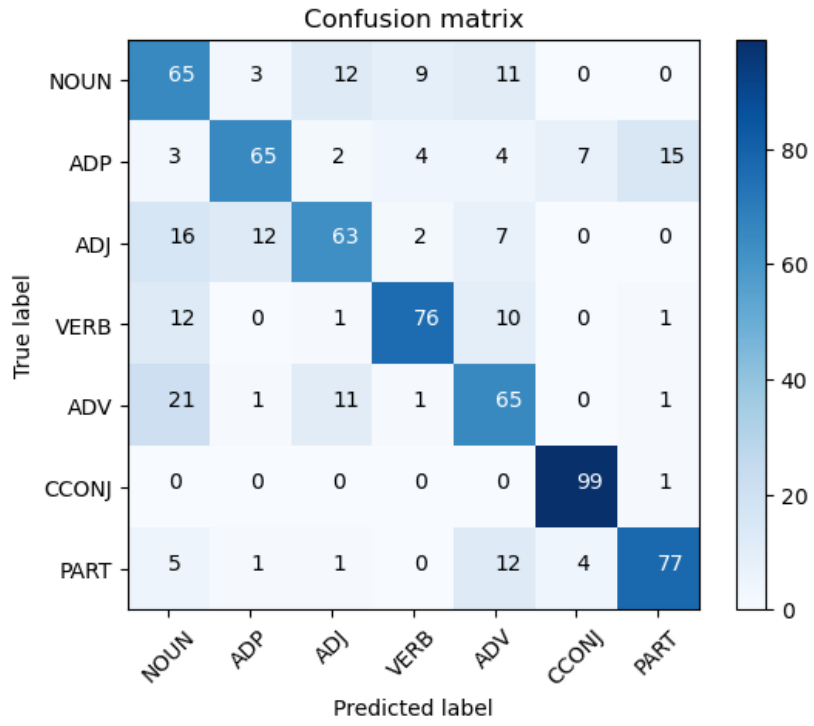
При использовании пастинга все базовые алгоритмы становятся идентичны друг другу и ансамбль вырождается, в свою очередь, бегинг продолжает давать, хоть и небольшой, но выигрыш в точности:

```
bg = bagging(Naive_Bayes(),20,soft_voting = True,sel_size = 1)
pipe = Pipeline([('transformer',transform()),('classifier',bg)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

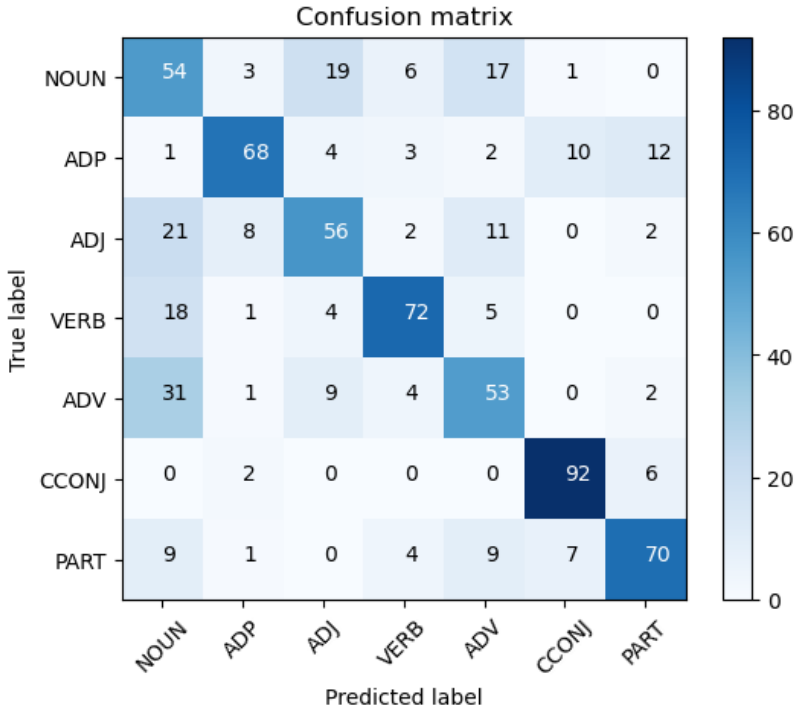
```
array([0.7038857142857143, 0.6703428571428574, 0.6723571428571431,
       0.6689857142857142, 0.6720714285714284, 0.6747, 0.6667857142857142,
       0.6682857142857144, 0.6694, 0.6705714285714284, 0.6663142857142859,
       0.6667857142857139, 0.6662571428571431, 0.6718571428571427,
       0.6704000000000001, 0.6698714285714287, 0.6703857142857141,
       0.6699142857142859, 0.6693714285714285, 0.6682285714285715,
       0.6700571428571429], dtype=object)
```

Матрица спутывания для пастинга. 20XNB, мягкое голосование включено, размер подвыборок = половине основной выборки)

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




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



Использовались наивные байесовские алгоритмы для разнообразия, так как knn уже применялся с бегингом. По сути, оба ансамбля похожи, но беггинг лучше, так как дает большую точность. Оба ансамбля дают существенно более высокую точность при использовании мягкого голосования.

3. Стекинг

Идея этого алгоритма: Выборка разбивается на n+1 частей, n — число базовых классификаторов. Базовые классификаторы могут быть разнородны, каждый из них обучается на своей подвыборке. Затем результат классификации базовых алгоритмов анализируется метаалгоритмом. Метаалгоритм обучается на результатах классификации последней подвыборки базовыми алгоритмами.

Функция, реализующая стекинг:

```
class Stacking(skl.base.ClassifierMixin):
    def __init__(self,base_mods,metamod):
        self.base_models = base_mods
        self.metamodel = metamod

        self.spl_cnt = len(self.base_models) + 1

    def accuracy(self,data,labels,verbal = False):
        data,labels = skl.utils.shuffle(data,labels)
        accuracy_base = []
        for i in range(len(self.base_models)):
            accuracy_base.append(self.base_models[i].score(data,labels))

        base_res_pre = []
        for i in range(len(self.base_models)):
            cur_res = []
            for a in data:
                cur_res.append(self.base_models[i].predict([a]))
            base_res_pre.append(cur_res)

        base_res = np.array([np.array(base_res_pre)[:,i] for i in range(len(base_res_pre[0]))])
        base_res_feats = np.array(base_res)

        accuracy_meta = self.metamodel.score(np.array(base_res_feats),labels)

        errors = 0
        fair = 0
        for i in range(len(base_res_feats)):
            predl = self.metamodel.predict([base_res_feats[i]])
            if(predl != labels[i]):
                errors += 1
                if(verbal):
                    print("base: {} pred: {} act: {}".format(base_res[i],predl,labels[i]))
                if(predl == base_res[i][0] or (predl == base_res[i][1])):
                    #предположил класс, один из коих был в базовой классификации
                    fair += 1
        if(verbal):
            print("errors: {} fair: {} fair part: {}".format(errors,fair,fair/errors))

        res = [accuracy_meta]
        for i in accuracy_base:
            res.append(i)

        return np.array(res)

    def predict(self,data):
        pre_res = []
        for mod in self.base_models:
            pre_res.append(mod.predict([data]))

        base_res = np.array(pre_res)

        return self.metamodel.predict([base_res])

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

```
    def fit(self,data,labels,verbal = False):
        data,labels = skl.utils.shuffle(data,labels)
        ids = [int(len(data)/self.spl_cnt)*i for i in range(1,self.spl_cnt)]
        splitsXX = np.split(np.array(data),ids)
        splitsLB = np.split(np.array(labels),ids)

        for i in range(len(self.base_models)):
            self.base_models[i].fit(splitsXX[i],splitsLB[i])

        ln = len(splitsXX) - 1

        base_res_pre = []
        for i in range(len(self.base_models)):
            cur_res = []
            for a in splitsXX[ln]:
                cur_res.append(self.base_models[i].predict([a]))

            base_res_pre.append(cur_res)

        base_res = np.array([np.array(base_res_pre)[:,i] for i in range(len(base_res_pre[0]))])
        #base_res_feats = np.array([[(t[0],t[1],1)] for t in base_res])
        base_res_feats = np.array(base_res)

        self.metamodel.fit(base_res_feats,splitsLB[ln])

        if(verbal):
            print("pred lables")
            print(base_res.tolist())

            print("true lables")
            print(splitsLB[ln])

        test_res = []
        for i in range(len(base_res_feats)):
            test_res.append(self.metamodel.predict([base_res_feats[i]]))

        if(verbal):
            print("test lables")
            print(test_res)
```

Оценка эффективности стекинга:

Для начала тесты будут проводиться над стекингом в самой простой конфигурации: 2 базовых алгоритма | 1 метаалгоритм. Для нивелирования эффекта случайных колебаний точности по-прежнему используется кросс-валидация.

Вывод: [точность ансамбля, точность алгоритма 1, точность алгоритма 2]

Наивный байесовский алгоритм в качестве метаалгоритма

2xNB | NB

```
st = Stacking([Naive_Bayes(),Naive_Bayes()],Naive_Bayes())
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 100)
print(arr.mean(axis = 0))
```

```
[0.6668428571428572 0.6430285714285716 0.6384571428571426]
```

Можно наблюдать, что точность метаалгоритма выше, чем точность каждого из базовых агоритмов. (примерно на 0.02)

KNN + NB | NB

```
st = Stacking([KNN(k = 1),Naive_Bayes()],Naive_Bayes())
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.7288571428571429 0.7213214285714284 0.6441071428571429]
```

Возрастание точности есть, но оно крайне мало в сравнении с точностью базового KNN

2xKNN | NB

```
st = Stacking([KNN(k = 1),KNN(k = 1)],Naive_Bayes())
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.7414285714285712 0.7247857142857144 0.72225]
```

Возрастание точности снова составляет ~ 0.02

KNN в качестве метаалгоритма

2xKNN | KNN

```
st = Stacking([KNN(k = 1),KNN(k = 1)],KNN(k = 1))
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.68175 0.7196785714285715 0.7271428571428572]
```

Наблюдается снижение точности метаалгоритма в сравнении с базовыми классификаторами.

```
st = Stacking([KNN(k = 1),KNN(k = 1)],KNN(k = 3))
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.7150357142857142 0.7193214285714286 0.7272857142857143]
```

От увеличения числа ближайших соседей в метаалгориме ситуация улучшается, но недостаточно, чтобы результат был сопостовим с результатом 2xKNN | NB.

Таким образом, среди конфигураций стекинга с двумя базовыми алгоритмами наиболее точной оказалась схема 2xKNN | NB.

Теперь тесты будут проводиться над конфигурациями стекинга, включающими в себя более, чем два базовых алгоритма.

3xKNN | NB

```
st = Stacking([KNN(k = 1),KNN(k = 1),KNN(k = 1)],Naive_Bayes())
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.7157857142857142 0.6881428571428574 0.6787142857142857
 0.6797857142857143]
```

Метаалгоритм теперь имеет точность примерно на ~0.03 выше, чем у базовых алгоритмов, но сама точность ниже, чем у конфигурации с двумя базовыми алгоритмами. Это происходит из-за того, что, чем на большее число частей будет разбита выборка, тем меньше достанется каждому классификатору для обучения.

При дальнейшем увеличении числа базовых алгоритмов, этот эффект становится все более выражен:

5xNB | NB

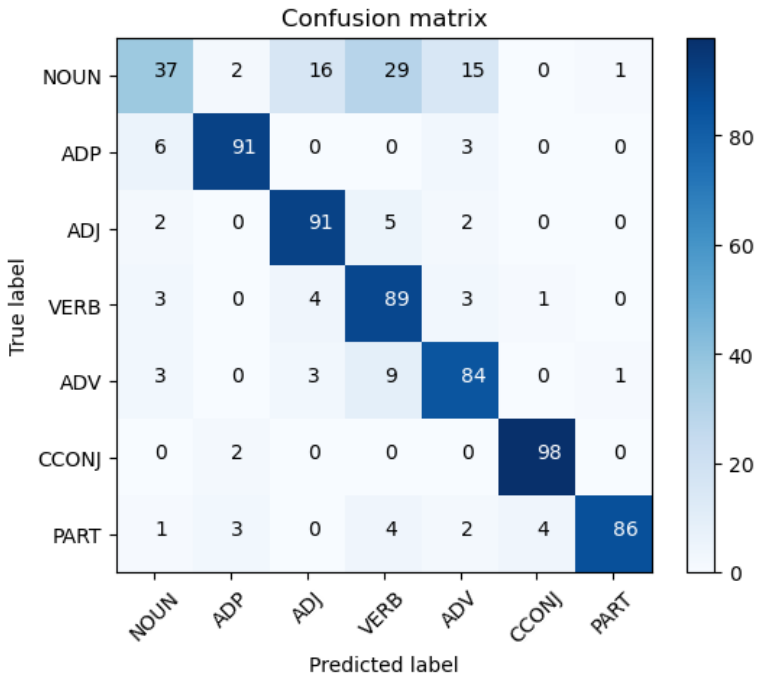
```
st = Stacking([Naive_Bayes(),Naive_Bayes(),Naive_Bayes(),
               Naive_Bayes(),Naive_Bayes()],Naive_Bayes())
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 40)
print(arr.mean(axis = 0))
```

```
[0.6091428571428571 0.5956071428571429 0.5928214285714286
 0.5860000000000001 0.5885 0.5813214285714287]
```

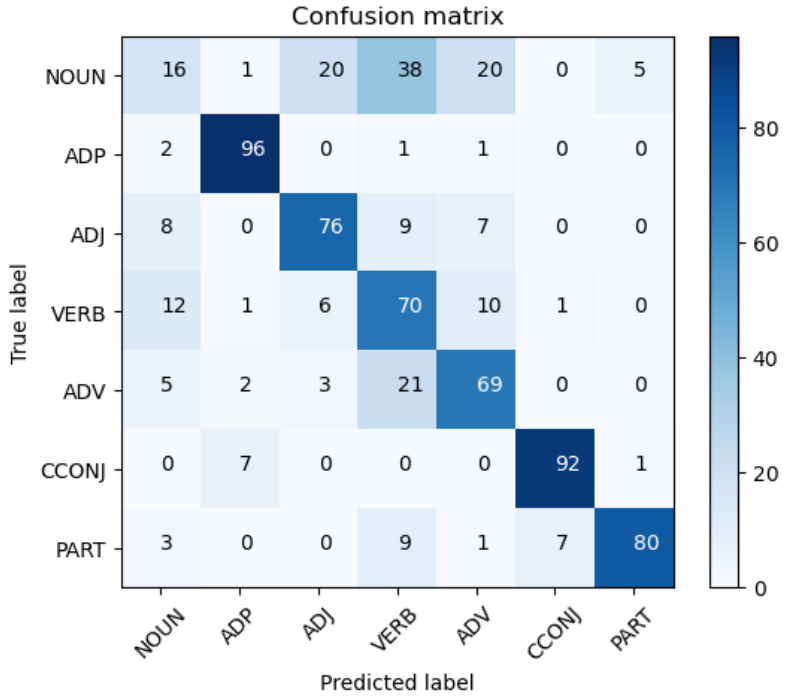
Было бы интересно посмотреть, что будет, если использовать в стекинге бутстрап.

Матрица спутывания для стекинга. 2XKNN | NB, биение на подвыборки осуществляется наивным образом.

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




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



4. Бустинг

При использовании бустинга базовые алгоритмы, кои однородны, подобно тому, как однородны они в бегинге и пастинге. Однако, в отличие от бегинга, выборка не разбивается, а подается каждому базовому алгоритму по очереди. Если классификатор после обучения допускает ошибку на некоем объекте выборки, вес этого объекта увеличивается при подаче выборки следующему алгоритму. При классификации установленный некоторым базовым алгоритмом класс учитывается в соответствии с присвоенным данному алгоритму весе, который зависит от точности, которую данный алгоритм показал на обучающей выборке.

Реализация бустинга:

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

    def __init__(self,model_exmpl,model_count):
        self.models = []
        for i in range(model_count):
            self.models.append(copy.deepcopy(model_exmpl))

        self.mod_cnt = model_count

    def fit(self,data_,labels_,verbal = False):
        data = np.array(copy.deepcopy(data_))
        labels = np.array(copy.deepcopy(labels_))
        wts = np.ones_like(labels)
        self.cl_cnt = len(np.unique(labels))

        err_X = None
        err_L = None
        self.wts = []
        num = 0
        for alg in self.models:
            num+=1

            errors_X = []
            errors_L = []

            alg.fit(data,labels,wts = wts)
            ln = len(data)

            for i in range(ln):
                if (alg.predict([data[i]]) != labels[i]):
                    errors_X.append(copy.deepcopy(data[i]))
                    errors_L.append(copy.deepcopy(labels[i]))

            err_part = 1 - alg.score(data_,labels_)

            if(verbal):
                print(len(err_X))
                print(len(data_))
                print("errors: {} ovl: {} err_part: {} trained on: {}".format(len(errors),ln,err_part,ln))

            cur_w = 0.5*np.log((1 - err_part)/err_part)
            if(err_part < 0.3):
                cur_w += 1

            #print("{} \t >{}".format(cur_w, 1 - err_part))

            self.wts.append(cur_w)

            for i in range(ln):
                if (alg.predict([data[i]]) != labels[i]):
                    wts[i] += num
```

```
    def predict(self,data):
        pred = np.ones(self.cl_cnt)
        for j in range(len(self.models)):
            pred[self.models[j].predict(data)] += self.wts[j]*100

        cl = np.argmax(pred)
        return cl

    def accuracy(self,data,labels,verbal = False):
        correct = 0
        overall = 0
        sep_acc = []
        for mod in self.models:
            sep_acc.append(mod.score(data,labels))

        for i in range(len(data)):
            overall += 1
            pred = np.ones_like(np.unique(labels))
            for j in range(len(self.models)):
                pred[self.models[j].predict([data[i]])] += self.wts[j]*100

            cl = np.argmax(pred)
            if(cl == labels[i]):
                correct += 1
            if(verbal):
                print(pred)
                print("{} {}".format(cl,labels[i]))

            res = [correct/overall]
            for i in sep_acc:
                res.append(i)
        return np.array(res)

    def score(self,data,labels,verbal = False):
        return self.accuracy(data,labels,verbal = verbal)
```

Базовые классификаторы, используемые с бустингом должны быть способны принимать выборки со взвешенными элементами и каким-либо образом имплементировать воздействие веса на процесс обучения. Для примера в данной лабораторной работе был модифицирован классификатор NB (наивный байесовский). В режиме работы со взвешенными выборками он учитывает каждый элемент так, будто он был встречен в выборке w раз, где w — его вес. Таким образом его признаки считаются более характерными для класса, к которому он относится.

Определение эффективности Бустинга с 20 наивными байесовскими классификаторами на выборке в 3500 слов (7 классов)

```
bt = boosting(Naive_Bayes(),20)
pipe = Pipeline([('transformer',transform()),('classifier',bt)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 20)
arr.mean(axis = 0)
```

```
array([0.7023571428571429, 0.684657142857143, 0.6885714285714285,
       0.6500571428571429, 0.5150857142857143, 0.4629142857142857,
       0.5947428571428572, 0.6487714285714287, 0.6263714285714286,
       0.6210142857142857, 0.6255428571428571, 0.5977571428571429,
       0.6219285714285715, 0.6339857142857144, 0.6382571428571429,
       0.6509714285714286, 0.6401, 0.6417714285714285, 0.6445428571
       0.6289285714285713, 0.6495857142857142], dtype=object)
```

Точность ансамбля ~0.70, точность составляющих ниже ~0.68

Обучение одного ансамбля из ста классификаторов NB:

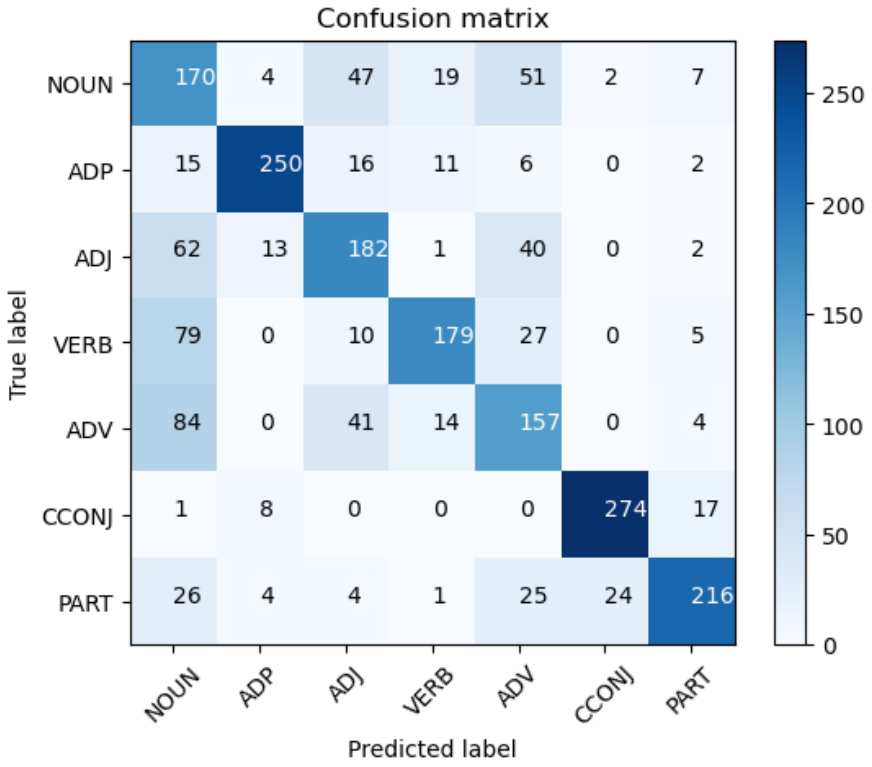
```
bt = boosting(Naive_Bayes(),100)
pipe = Pipeline([('transformer',transform()),('classifier',bt)])
pipe.fit(larrDC,larrLB)
print(pipe.score(larrDC,larrLB))
print(pipe.score(tarrDC,tarrLB))
```

```
[0.72190476 0.71333333 0.7        0.69857143 0.53809524 0.52047619
 0.53428571 0.68       0.64761905 0.70333333 0.62952381 0.65238095
 0.59571429 0.55       0.70666667 0.72190476 0.71285714 0.69190476
 0.59857143 0.65619048 0.6652381  0.58761905 0.71095238 0.71761905
 0.68333333 0.72857143 0.65285714 0.70142857 0.70666667 0.70571429
 0.6352381  0.57428571 0.69571429 0.7152381  0.64190476 0.72571429
 0.7147619  0.72238095 0.69095238 0.71380952 0.70380952 0.66238095
 0.69666667 0.56380952 0.68333333 0.64333333 0.71190476 0.69095238
 0.69666667 0.71285714 0.69047619 0.65761905 0.71047619 0.71857143
 0.58904762 0.73952381 0.66666667 0.71380952 0.70047619 0.7
 0.58857143 0.66809524 0.71428571 0.70380952 0.60952381 0.7247619
 0.68285714 0.69952381 0.73428571 0.67380952 0.71190476 0.66857143
 0.73190476 0.6952381  0.68571429 0.70285714 0.71380952 0.69285714
 0.49952381 0.69142857 0.70571429 0.7152381  0.68333333 0.70333333
 0.72428571 0.70952381 0.67809524 0.66714286 0.71333333 0.69333333
 0.57809524 0.70238095 0.71238095 0.61904762 0.71809524 0.69333333
 0.65285714 0.71238095 0.72333333 0.7152381  0.70285714]
[0.68       0.67571429 0.66285714 0.65857143 0.53571429 0.4952381
 0.49380952 0.64857143 0.61047619 0.6752381  0.60714286 0.62238095
 0.5552381  0.51666667 0.65428571 0.67904762 0.67380952 0.6547619
 0.56047619 0.60857143 0.62       0.55380952 0.67333333 0.67285714
 0.64952381 0.68       0.62285714 0.66       0.66809524 0.66142857
 0.58809524 0.5347619  0.64095238 0.67571429 0.60714286 0.67095238
 0.67190476 0.67714286 0.65619048 0.67571429 0.66285714 0.62714286
 0.65190476 0.51809524 0.63428571 0.59333333 0.66714286 0.64666667
 0.65428571 0.66619048 0.64857143 0.60904762 0.65952381 0.67238095
 0.55809524 0.69190476 0.63190476 0.65904762 0.66333333 0.65142857
 0.53857143 0.62333333 0.65857143 0.66238095 0.5747619  0.67857143
 0.63857143 0.64809524 0.69047619 0.62952381 0.66238095 0.63095238
 0.68619048 0.65666667 0.64571429 0.66619048 0.67380952 0.65
 0.45333333 0.64       0.65238095 0.66904762 0.64857143 0.65714286
 0.68285714 0.66714286 0.63666667 0.61761905 0.6647619  0.6447619
 0.53904762 0.66857143 0.66666667 0.57809524 0.66904762 0.65142857
 0.61142857 0.66047619 0.67904762 0.67190476 0.66571429]
```

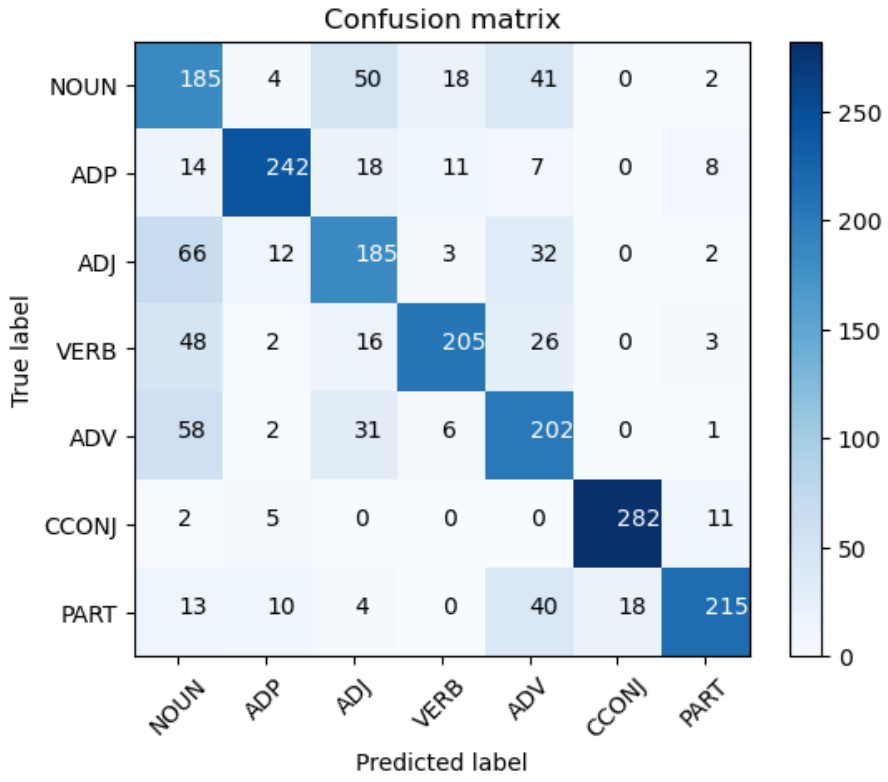
Наблюдается небольшой прирост точности, примерно как и у других ансамблей.

Матрица спутывания для бустинга 100xNB

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




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



5.1. Решающее дерево

Этот алгоритм основан на разбиениях пространства объектов классификации по значениям признаков. На каждом шаге обучения делается оптимальное разбиение, разбиения (признак, по которому оно выполняется и значение признака в точке разбиения) записываются, формируя древовидную структуру. Разбиения прекращаются, когда в оставшихся после разбиениях массивах остаются элементы одного класса или если достигнут лимит глубины дерева. При классификации выполняется проход по этой структуре до листа в соответствии со значениями признаков рассматриваемого элемента, которому присваивается класс, записанный в листе.

Для отыскания оптимального разбиения осуществляется полный перебор признаков и их значений. «Оптимальность» определяется функцией, которая измеряет однородность массивов после разбиения (то, насколько разнообразны там классы. Чем менее, тем лучше)

Вспомогательные функции для решающего дерева:

```
def entropy(n_cl,arr):
    c_cl = np.zeros(n_cl)
    for a in arr:
        c_cl[int(a)] += 1
    h = 0
    for a in c_cl:
        if(a != 0):
            h -= (a/len(arr)) * (np.log(a/len(arr)))

    return h

def split_by_feat(array,arr_lb,feat_n,split_val):
    lhs = []
    rhs = []
    lhs_lb = []
    rhs_lb = []
    for i in range(len(array)):
        elm = array[i]
        if(elm[feat_n] >= split_val):
            rhs.append(array[i])
            rhs_lb.append(arr_lb[i])
        else:
            lhs.append(array[i])
            lhs_lb.append(arr_lb[i])

    return np.array(lhs,dtype = object),np.array(lhs_lb,dtype = object),
           np.array(rhs,dtype = object),np.array(rhs_lb,dtype = object)

def domination(array,cl_num,required_part = 1,verbal = False):
    index = np.zeros(cl_num)
    for elm in array:
        index[elm] += 1

    amg = np.argmax(index)

    if(verbal):
        print("({} {} ({}))".format(int(index[amg]), int(np.sum(index)),
                                    index[amg]/np.sum(index)))

    if(index[amg]/np.sum(index) > required_part):
        return amg
    else:
        return None
```

Здесь для оценивания оптимальности разбиений используется энтропийный критерий. Функция domination() определяет, составляет ли один класс абсолютное большинство в массиве и возвращает его номер. Функция ниже рекурсивно делит выборку оптимальным образом.

```
def find_optimal_split(data,labels,depth,termination_depth = 10,verbal = False, dropout = 0):

    if(verbal):
        print("{}({}) | {}".format(depth,termination_depth,len(labels)))

    if(depth > termination_depth):
        return domination(labels,7,required_part = 0,verbal = verbal)

    dm = domination(labels,7,required_part = 1,verbal = verbal)
    if(dm != None):
        return dm

    if(len(labels) == 0):
        return 0

    if(len(data.shape) == 1):
        feats_cnt = 1
    else:
        feats_cnt = data.shape[1]
    class_cnt = len(np.unique(labels))

    class_cnt = (np.max(labels)) + 1

    feats_vocab = []
    for feat_ind in range(feats_cnt):
        cur_feat = []
        for elm in data:
            if(elm[feat_ind] not in cur_feat):
                cur_feat.append(elm[feat_ind])
        feats_vocab.append(np.sort(cur_feat))

    optimal_split_ind = -5,-5
    optimal_split_entropy = np.inf

    features_to_pass = int((1 - dropout)*len(feats_vocab))
    if(features_to_pass == 0):
        features_to_pass = 1

    feat_selection = np.arange(len(feats_vocab))
    feat_selected = []
    while(len(feat_selected) < features_to_pass):
        cf = random.choice(feat_selection)
        if(not cf in feat_selected):
            feat_selected.append(cf)
    for feat_ind in feat_selected:

        feat = feats_vocab[feat_ind]

        for val_ind in range(len(feat)):
            if(val_ind == 0):
                continue

            if(verbal):
                print(" --|{}|--".format(len(feat)))

            splits = split_by_feat(data,labels,feat_ind,feat[val_ind])

            cur_split_entropy = entropy(class_cnt,splits[1]) + entropy(class_cnt,splits[3])
            if(cur_split_entropy < optimal_split_entropy):
                optimal_split_ind = feat_ind,feat[val_ind]

                optimal_split_entropy = cur_split_entropy

    if(optimal_split_entropy == np.inf):
        return domination(labels,7,required_part = 0,verbal = verbal)

    optimal_split = split_by_feat(data,labels,optimal_split_ind[0],optimal_split_ind[1])

    if(verbal):
        print("|| {}  - {} ||".format(optimal_split[1],optimal_split[3]))
        print("| {} |".format(optimal_split_ind))
        print("_____________")

    spl_nxt_1 = find_optimal_split(optimal_split[0],optimal_split[1],depth + 1,
                                   termination_depth = termination_depth,
                                   verbal = verbal,dropout = dropout)
    spl_nxt_2 = find_optimal_split(optimal_split[2],optimal_split[3],depth + 1,
                                   termination_depth = termination_depth,
                                   verbal = verbal,dropout = dropout)
    return (optimal_split_ind, spl_nxt_1,spl_nxt_2)
```

Само решающее дерево:

```
class solver_tree(skl.base.ClassifierMixin):
    def __init__(self,termination_depth = 50,verbal = False,dropout = 0):
        self.termination_depth = termination_depth
        self.verbal = verbal
        self.dropout = dropout

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

    def predict(self,elm):
        data = elm[0]
        solver = copy.deepcopy(self.solver)
        if(self.verbal):
            print(data)
        while np.shape(solver) != ():
            if(self.verbal):
                print(solver[0])
            if(data[solver[0][0]] >= solver[0][1]):
                if(self.verbal):
                    print(" --> {}".format(solver[2]))
                if(np.shape(solver[2]) == ()):
                    return solver[2]
                else:
                    solver = copy.deepcopy(solver[2])
            else:
                if(self.verbal):
                    print(" --> {}".format(solver[1]))
                if(np.shape(solver[1]) == ()):
                    return solver[1]
                else:
                    solver = copy.deepcopy(solver[1])

    def fit(self,data_,labels_,verbal = False):
        data = np.array(copy.deepcopy(data_),dtype= object)
        labels = np.array(copy.deepcopy(labels_),dtype= object)
        pairs = np.array([[data[i],labels[i]]
                          for i in range(len(labels))],dtype = object)

        if(len(data.shape) == 1):
            feats_cnt = 1
        else:
            feats_cnt = data.shape[1]
        class_cnt = len(np.unique(labels))

        feats_vocab = []
        for feat_ind in range(feats_cnt):
            cur_feat = []
            for elm in data:
                if(elm[feat_ind] not in cur_feat):
                    cur_feat.append(elm[feat_ind])
            feats_vocab.append(np.sort(cur_feat))

        optimal_split_ind = -1,-1
        optimal_split_entropy = np.inf

        spl_res = find_optimal_split(data,labels,0,termination_depth =
                                     self.termination_depth,
                                     verbal = self.verbal,dropout = self.dropout)

        if(self.verbal):
            print(spl_res)

        self.solver = spl_res
```

(оно должно называться decision tree, да?..)

Оценка эффективности решающего дерева.

Кросс-валидация дерева с предельной глубиной ветвления 50 на выборке из 1400 слов (7 классов)

```
st = solver_tree(termination_depth = 50)
pipe = Pipeline([('transformer',transform()),('classifier',st)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 10)
```

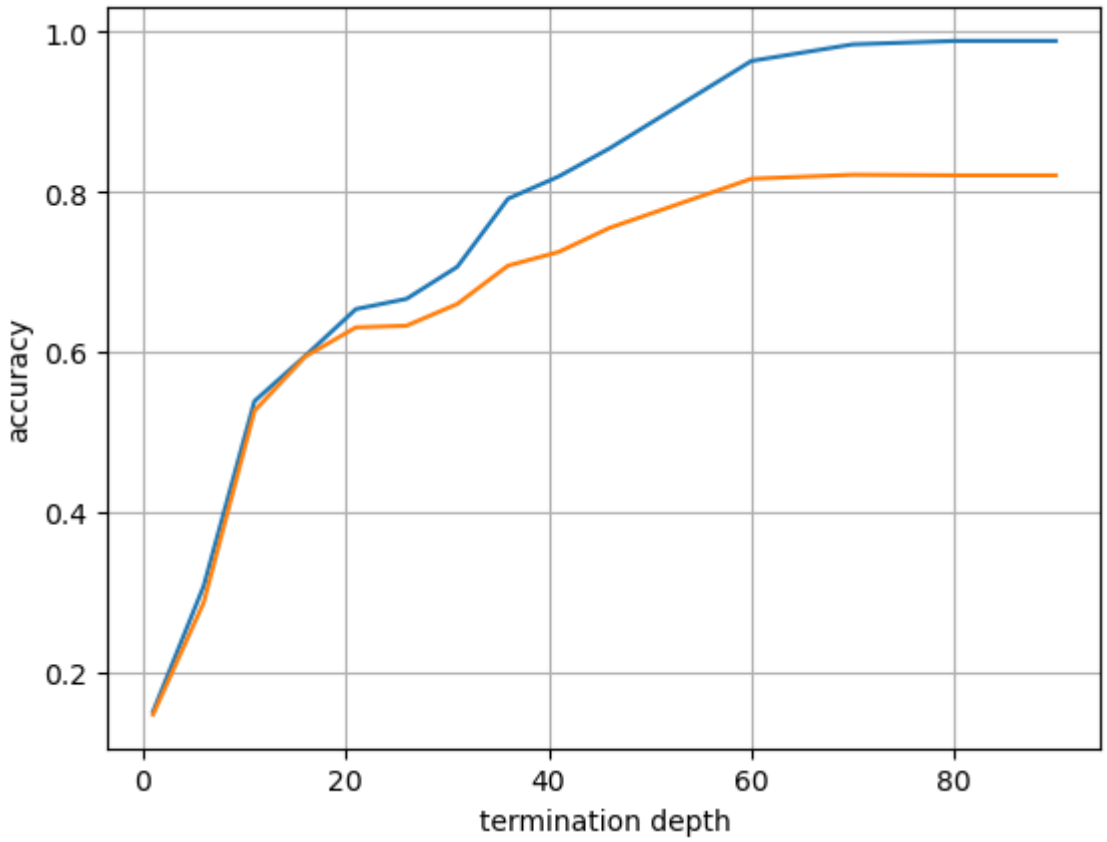
```
print("accuracy:")
print(arr.mean(axis = 0))
```

```
accuracy:
0.8420714285714286
```

Наблюдается точность 0.84 на тестовом датасете.

Кривая максимальная глубина / точность:

Синим показана точность на обучающей выборке, желтым — на тестовой



Наблюдается заметное переобучение.

5.2. Случайный лес

Данный алгоритм представляет из себя ансамбль из решающих деревьев. Для разделения выборки используется бутстрап. Для того чтобы деревья были более разнообразны по своим структурам и не представляли из себя копии друг друга, на каждом шаге выбор оптимального разделения производится не по всем признакам, а по нескольким случайно выбраным (оттого лес и случайный). Реализация случайного выбора признаков осуществляется параметром dropout, который передается в функцию поиска оптимальных разбиений и определяет, какая часть признаков будет исключена.

```
class random_forest(skl.base.ClassifierMixin):
    def __init__(self,trees_count,max_depth = 50,sel_size = None,dropout = 0.5):
        self.sel_size = sel_size
        self.models = []
        for i in range(trees_count):
            self.models.append(solver_tree(termination_depth = max_depth,dropout = dropout))

        self.mod_cnt = trees_count

    def fit(self,data,labels):
        selections = bootstrap(data,labels,num_sel = self.mod_cnt,sel_size = self.sel_size)

        for i in range(self.mod_cnt):
            self.models[i].fit(selections[i,:,0],selections[i,:,1])

    def predict(self,data):
        pred = []
        for k in self.models:
            pred.append(k.predict(data))
        uc = np.unique(pred, return_counts=True)
        cl = uc[0][np.argmax(uc[1])]
        return cl

    def accuracy(self,data,labels,verbal = False):
        correct = 0
        overall = 0
        sep_acc = []
        for mod in self.models:
            sep_acc.append(mod.score(data,labels))

        for i in range(len(data)):
            overall += 1
            pred = []
            for k in self.models:
                pred.append(k.predict([data[i]]))
            uc = np.unique(pred, return_counts=True)
            cl = uc[0][np.argmax(uc[1])]
            if(cl == labels[i]):
                correct += 1
            if(verbal):
                print(pred)
                print(cl)
            res = [correct/overall]
            for i in sep_acc:
                res.append(i)
        return np.array(res)

    def score(self,data,labels,verbal = False):
        return self.accuracy(data,labels,verbal = verbal)
```

По умолчанию dropout = 0.5.

Оценка эффективности случайного леса

Кросс-валидация ансамбля. *10 деревьев с максимальной глубиной в 40 и размерами подвыборок в 300 элементов.*

```
rf = random_forest(trees_count = 10,max_depth = 40,sel_size = 300)
pipe = Pipeline([('transformer',transform()),('classifier',rf)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 10)
```

```
arr.mean(axis = 0)
```

```
array([0.8539285714285715, 0.7373571428571428, 0.7332142857142857,
       0.7340714285714286, 0.7317857142857143, 0.7289285714285715,
       0.7410714285714285, 0.7247857142857143, 0.738642857142857,
       0.7183571428571429, 0.7365714285714284], dtype=object)
```

Точность ансамбля ~0.85, точность классификаторов ~0.73

Наблюдается существенный точности ансамбля в сравнении с точностью отдельных классификаторов. Также это значение выше, чем средняя точность на тестовой выборке одного решающего дерева (~0.84)

30 деревьев, предельная глубина 50, размер подвыборок 400, основной выборки 1400:

```
rf = random_forest(trees_count = 30,max_depth = 50,sel_size = 400)
pipe = Pipeline([('transformer',transform()),('classifier',rf)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 10,ratio = 0.95,verbal = True)
```

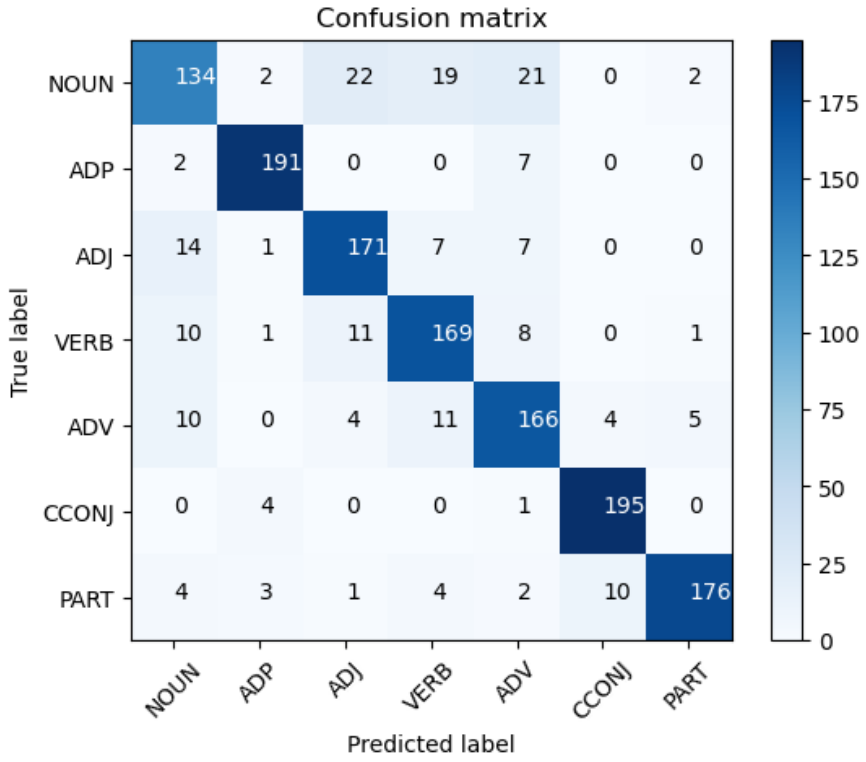
```
arr.mean(axis = 0)
```

```
array([0.902857142857143, 0.7271428571428572, 0.7335714285714287,
       0.7114285714285714, 0.7378571428571429, 0.7407142857142857,
       0.7542857142857142, 0.7471428571428571, 0.7785714285714286,
       0.7692857142857142, 0.7392857142857143, 0.7457142857142858,
       0.757142857142857, 0.75, 0.7621428571428571, 0.7564285714285715,
       0.74, 0.7435714285714285, 0.7614285714285713, 0.7428571428571429,
       0.75, 0.7442857142857142, 0.7671428571428571, 0.7578571428571428,
       0.7507142857142857, 0.745, 0.7649999999999999, 0.7621428571428571,
       0.7478571428571428, 0.7414285714285714, 0.7521428571428571],
      dtype=object)
```

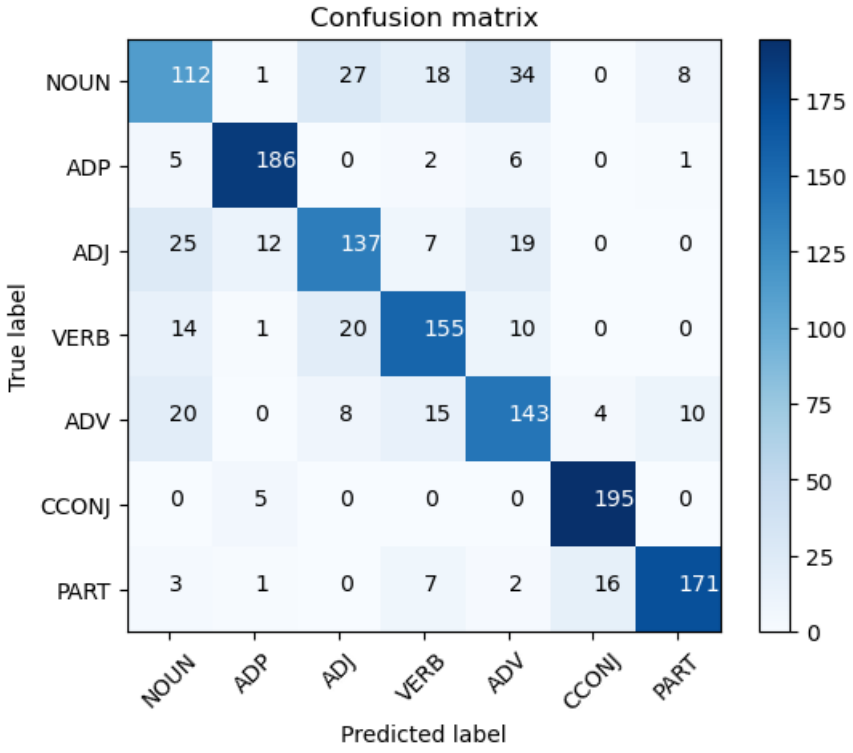
Видно, что, если подобрать оптимальные гиперпараметры, то возможно достижение точности, существенно превосходящий точность одного дерево.

Матрица спутывания случайного леса, 10 деревьев, предельная глубина = 40, размер подвыборок = 300 элементов, размер выборки = 1400 элементов.

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



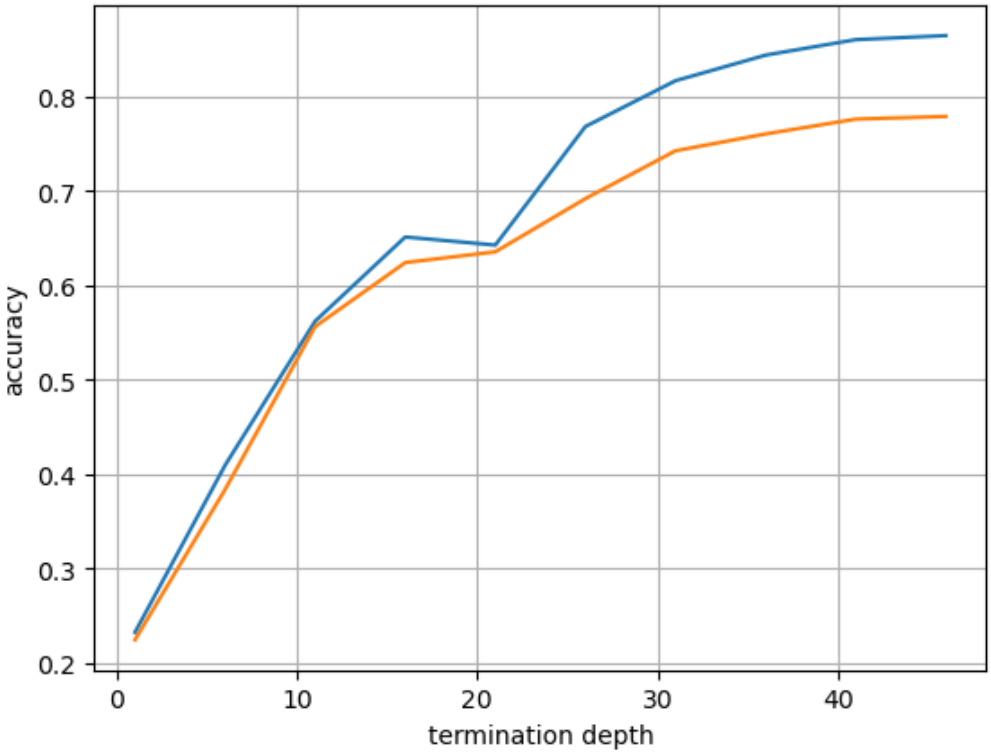

```
test set:
Confusion matrix, without normalization
```



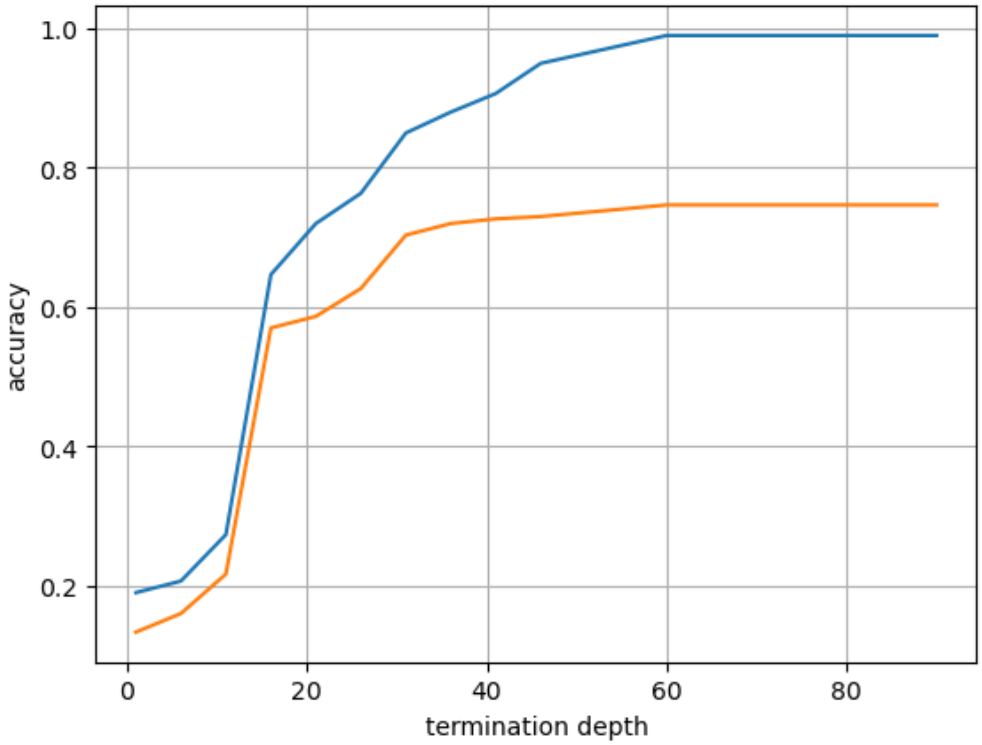
Сравнение случайного леса и одного решающего дерева

Производится сравнение кривых «Максимальная глубина / точность классификации»

Случайный лес из 10 деревьев, подвыборки длиной 300 элементов.

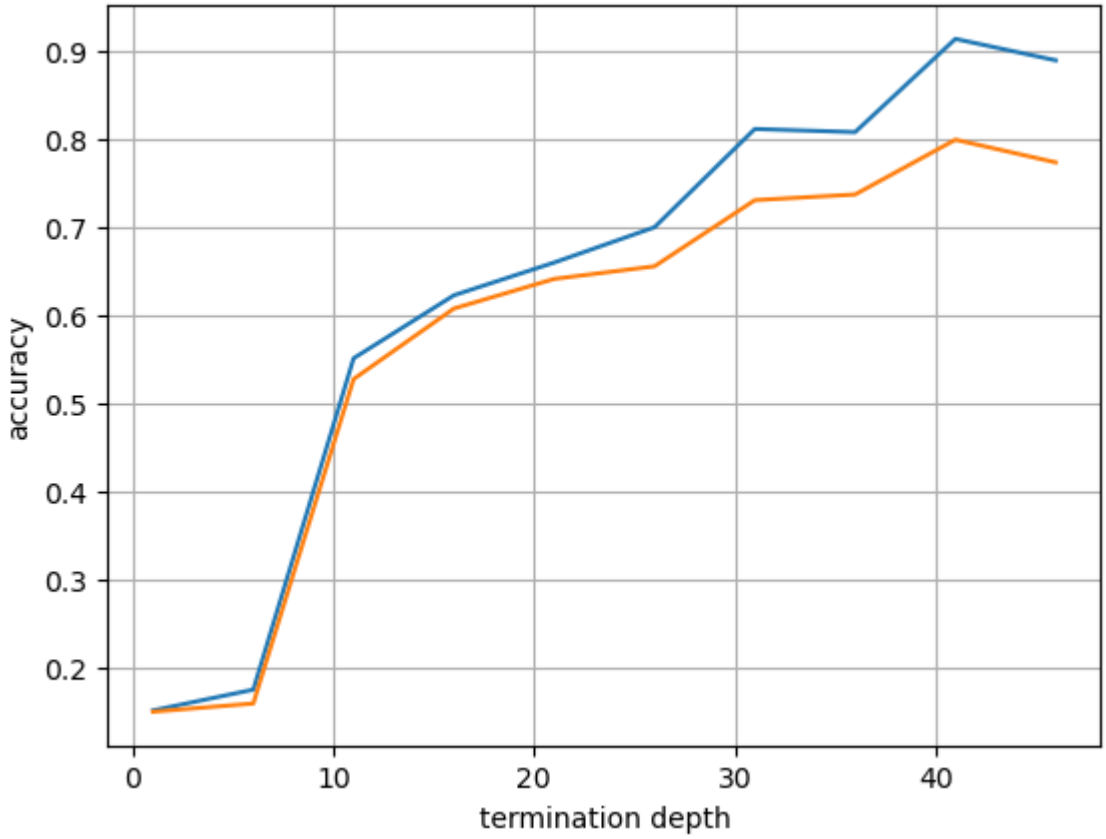


1 дерево, выборка за 300 элементов.

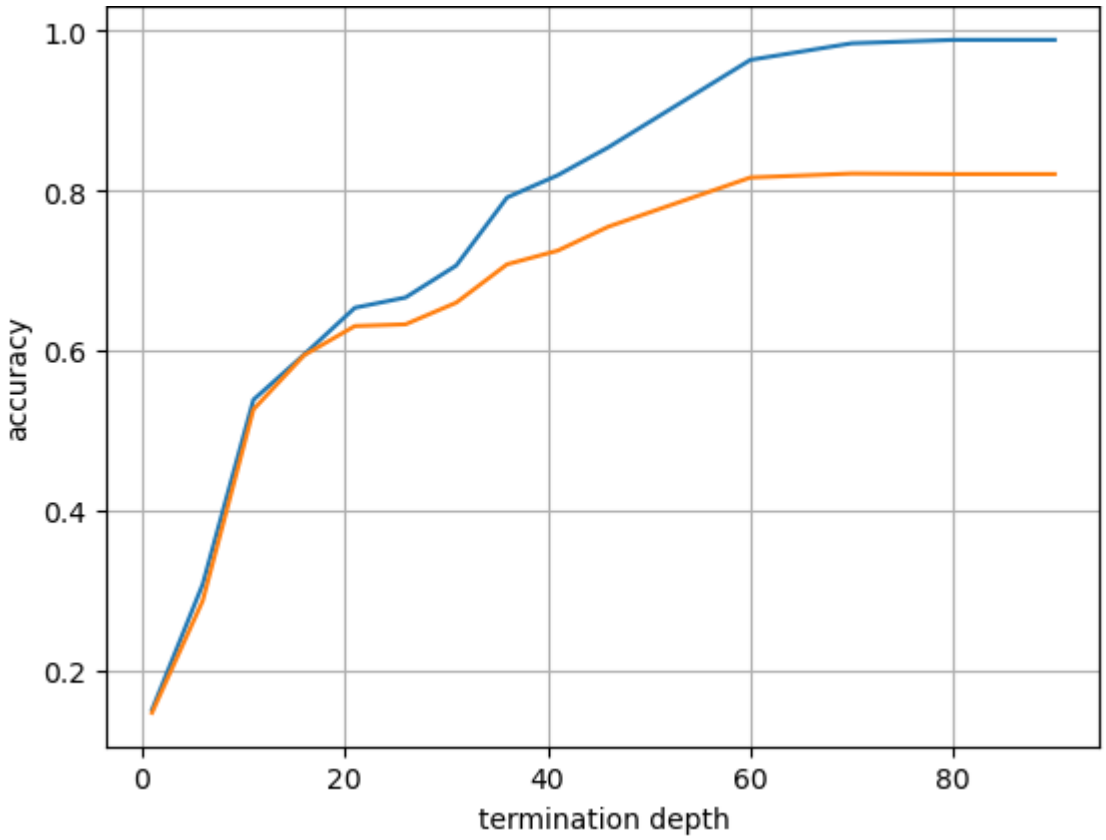


Невооруженным глазом видно, что переобучение случайного леса существенно ниже, чем переобучение одного решающего дерева.

Случайный лес из 10 деревьев, подвыборки длиной с длину основной выборки, длина основной выборки 1400 элементов.



1 дерево, выборка за 1400 элементов.



Также видно, что переобучение одного дерева сильно ниже, чем ансамбля из 10.

6. Коробочная реализация градиентного бустинга

Используется skl.ensemble.GradientBoostingClassifier.

Эта функция реализует градиентный бустинг решающих деревьев. По умолчанию используется 100 деревьев предельной глубины 3.

Пример обучения

```
gb_skl = ensemble.GradientBoostingClassifier()
pipe = Pipeline([('transformer',transform()),('classifier',gb_skl)])
pipe.fit(larrDC,larrLB)

print(pipe.score(larrDC,larrLB))
print(pipe.score(tarrDC,tarrLB))
```

```
0.9414285714285714
0.8357142857142857
```

Кросс-валидация коробочного градиентного бустинга:

```
gb_skl = ensemble.GradientBoostingClassifier()
pipe = Pipeline([('transformer',transform()),('classifier',gb_skl)])
CrossValidataion = RCV([larrDC,larrLB],[tarrDC,tarrLB],pipe = pipe)
arr = CrossValidataion.validate(i = 50)
```

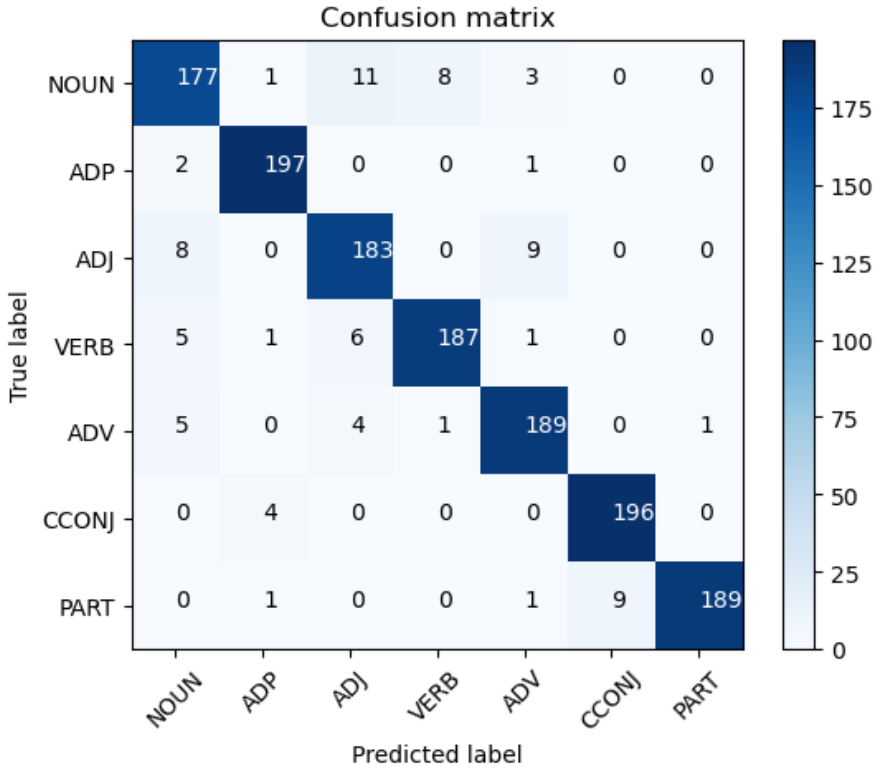
```
arr.mean(axis = 0)
```

```
0.9028285714285714
```

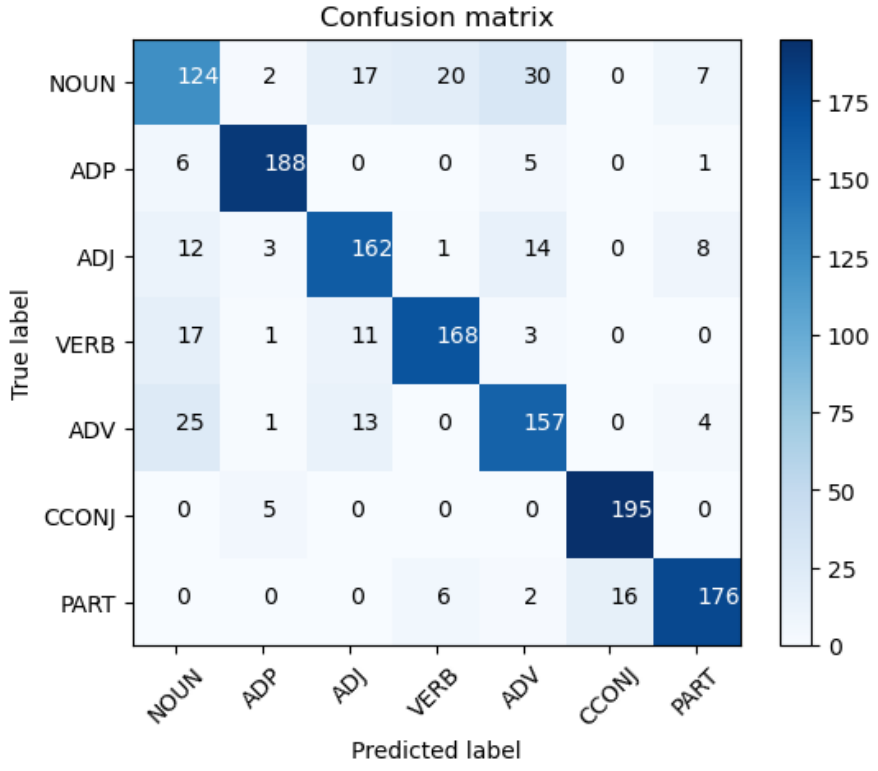
Средняя точность на тестовом сете 0.90.

Матрица спутывания для коробочного градиентного бустинга:

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




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



3. Выводы

В ходе выполнения этой лабораторной работы я реализовал ансамбли классификаторов: бегинг, пастинг, стекинг, бкстинг и случайный лес. Также я реализовал классификатор решающее дерево, который был необходим в качестве составной части случайного леса. Я оценил эффективность этих ансамблей при решении задачи классификации слов русского языка по частям речи, используя в качестве базовых классификаторов показавшие себя наиболее подходящими для решения этой задачи в предыдущей лабораторной работе KNN и NB. Также я сравнил их с коробочной реализацией градиентного бустинга. (skl.ensemble.GradientBoostingClassifier).

Я наглядно продемонстрировал то, насколько случайный лес более защищен от переобучения, чем одиночное решающее дерево, а также проиллюстрировал разницу между бегингом и пастингом путем демонстрации того, что при запуске обоих с длиной подвыборок равной длине основной выборки пастинг вырождается, а бегинг по-прежнему дает выигрыш в эффективности.

Используя кросс-валидацию случайными разбиениями, я установил средние значения точности вышеуказанных ансамблей и сравнил их с точностью базовых классификаторов. Все ансамбли дали повышение точности в худшем случае на несколько процентов в сравнении с ними, в некоторых конфигурациях выигрыш был в десятки процентов, но, чем ближе точность ансамбля была к предельно достижимой, тем меньше была разница в точности между ансамблем и базовыми классификаторами.

Так как ансамбли работают существенно медленнее отдельных классификаторов, чаще всего мне приходилось прибегать к использованию меньших по размеру выборок, а именно 700 для всех ансамблей, включавших KNN, 1400 для ансамблей включавших мою реализацию решающего дерева и 3500 для ансамблей исключительно из NB. Сопоставимых с полученной в первой работе 0.90 + точности на коробочном KNN удалось достичь с использованием коробочного градиентного бустинга, давшего 0.903 ровно, согласно кросс-валидации в 50 тестов на случайных разбиениях по выборке из 7000 слов, а также случайного леса из 30 деревьев предельной глубиной 50 и размером выборки в 1400 слов (подвыборки в 400), точность равна 0.902.

Решающее дерево оказалось алгоритмом хорошо подходящим для классификации слов русского языка: стабильная точность 0.84 на отдельном дереве, 0.85 у случайного леса и 0.90 у градиентного бустинга на решающих деревьях (коробочном из skl).

Теория о том, что совмещение разнородных классификаторов даст существенный прирост точности не подтвердилась, пример со стекингом показал, что два KNN дают большую точность, чем комбинация KNN + NB.