**Определение победителей матчей регулярного чемпионата КХЛ**

Всем привет! Давно хотел применить методы машинного обучения в области спортивной индустрии, что обусловлено двумя факторами: интерес к самому спорту и к тому, насколько хорошо математические модели могут предсказывать исходы различных спортивных событий. Возможность реализации задуманного представилась на выпускном проекте курса Machine Learning. Professional в Otus. Можно было взять любую интересующую тему, и я выбрал определение победителей матчей регулярного чемпионата КХЛ. Так как курс был по ML, для решения задачи рекомендовалось применять классические методы без использования нейросетевых моделей. Дав волю своему экспериментаторскому началу, я принялся за дело.

1. **Исходные данные**

Исходные данные для обучения и тестирования моделей парсились с сайта khl.ru. В расчёт бралась информация по 4 последним завершенным сезонам – 2018/2019, 2019/2020, 2020/2021 и 2021/2022.

Парсинг состоял из двух частей:

1. сбор ссылок на элементы информации (резюме, протокол, текстовую трансляцию и видео ключевых моментов) по каждому матчу;

# ссылки страниц с информацией по каждому сезону

page\_links = ['https://www.khl.ru/calendar/1097/00/', 'https://www.khl.ru/calendar/1045/00/', 'https://www.khl.ru/calendar/851/00/', 'https://www.khl.ru/calendar/671/00/']

section\_names = []

for link in page\_links:

    response = requests.get(link, headers={'User-Agent': UserAgent().chrome})

    soup = BeautifulSoup(response.content, 'html.parser')

    sections = soup.findAll('a', attrs={'class':'card-game\_\_hover-link\_small'})

    for i in sections:

        section\_names.append(i.attrs['href'])

# выбираем необходимые ссылки для сбора статистики матчей и добавляем к ним доменное имя

links = ['https://khl.ru' + element for element in section\_names if 'protocol' in element]

1. парсинг интересующей информации по ссылкам на резюме и протоколы матчей.

# функция подключения к веб-странице

def connect(link):

    response = requests.get(link, headers={'User-Agent': UserAgent().chrome})

    if not response.ok:

        return response.status\_code

    soup = BeautifulSoup(response.content, 'html.parser')

    return soup

# функция очистки содержимого веб-страниц от пустых строк и пробелов

def stats\_cleaner(data):

    data = [x.strip('\n').strip(' ') for x in data]

    return data

all\_stats, matches\_missing = [], []

for link in tqdm\_notebook(links):

    match\_stats = []

    # часть статистики матча в его протоколе

    soup\_protocol = connect(link)

    # находим необходимую статистику матчей из протокола

    statistics\_protocol = soup\_protocol.findAll('div', attrs={'class':'fineTable-totalTable d-none\_768'})

    # часть матчей не игралась, но ссылки на пустые протоколы есть - выбираем не пустые протоколы

    if statistics\_protocol != []:

        goals = soup\_protocol.findAll('p', attrs={'class':'preview-frame\_\_center-score roboto-condensed roboto-bold color-white title-xl'})[0].contents

        # часть статистики матча в его резюме

        soup\_resume = connect(link.replace('protocol', 'resume'))

        # находим необходимую статистику матчей из резюме

        statistics\_resume = soup\_resume.findAll('p', attrs={'class':'roboto-condensed'})[1:21]

        # номер матча для его идентификации

        match\_number\_resume = soup\_resume.findAll('h2', attrs={'class':'roboto-condensed roboto-bold title-md color-dark title-btns\_\_title'})

        # находим даты проведения матчей

        soup\_preview = connect(link.replace('protocol', 'preview'))

        date\_preview = soup\_preview.findAll('div', attrs={'class':'card-infos\_\_item-info'})

        # добавляем в список статистики дату, время проведения и номер матча

        try:

            match\_stats.append(stats\_cleaner(date\_preview[0].contents[1].contents)[0])

        except IndexError:

            match\_stats.append(np.NaN)

        match\_stats.append(match\_number\_resume[0].contents[0].split('№')[1].replace(' ', ''))

        # добавляем в список статистики основную информацию

        for stat in statistics\_resume:

            clean\_stat = stats\_cleaner(stat.contents)

            if clean\_stat != [''] and clean\_stat != ['n/a']:

                match\_stats.append(clean\_stat[0])

            else: pass

        # добавляем в список статистики дополнительную информацию

        if len(statistics\_protocol[len(statistics\_protocol)-2].contents) == 11 and 'Всего' in str(statistics\_protocol[1].contents[9].contents[7].contents[1]):

            for stat in statistics\_protocol[1].contents[9].contents:

                if stat != '\n':

                    clean\_stat = stats\_cleaner(stat.contents[1].contents)

                    match\_stats.append(clean\_stat[0])

                else: pass

        elif len(statistics\_protocol[len(statistics\_protocol)-2].contents) == 11 and 'овертайм' in str(statistics\_protocol[1].contents[9].contents[7].contents[1]):

            for stat in statistics\_protocol[2].contents[9].contents:

                if stat != '\n':

                    clean\_stat = stats\_cleaner(stat.contents[1].contents)

                    match\_stats.append(clean\_stat[0])

                else: pass

        elif len(statistics\_protocol) == 3 and len(statistics\_protocol[len(statistics\_protocol)-2].contents) == 13:

            for stat in statistics\_protocol[1].contents[11].contents:

                if stat != '\n':

                    clean\_stat = stats\_cleaner(stat.contents[1].contents)

                    match\_stats.append(clean\_stat[0])

                else: pass

        else:

            for stat in statistics\_protocol[2].contents[11].contents:

                if stat != '\n':

                    clean\_stat = stats\_cleaner(stat.contents[1].contents)

                    match\_stats.append(clean\_stat[0])

                else: pass

        # для идентификации овертаймов

        if len(statistics\_protocol[len(statistics\_protocol)-2].contents) == 11 and ('овертайм' not in statistics\_protocol[0].contents[9].contents[7].contents[1].contents[0].replace(' ', '').replace('\n', '')):

            match\_stats.append('main\_time')

        else: match\_stats.append('add\_time')

        # добавляем информацию о кол-ве заброшенных шайб с учетом буллитов

        if goals[0] != '\n':

            try:

                goal\_h = goals[0].replace(' ', '').replace('\n', '')

            except TypeError:

                goal\_h = goals[0].contents[0]

            try:

                goal\_a = goals[3].replace(' ', '').replace('\n', '')

            except IndexError:

                pass

            except TypeError:

                goal\_a = goals[3].contents[0]

        else:

            goal\_h = goals[1].contents[0]

            goal\_a = goals[4].replace(' ', '').replace('\n', '').replace('Б', '').replace('OT', '')

        match\_stats.append(goal\_h)

        match\_stats.append(goal\_a)

        # проверка наличия данных о пройденной дистанции и времени владения шайбой (есть не во всех матчах, удаляется)

        if len(match\_stats) == 32:

            del match\_stats[15:19]

        else: pass

        all\_stats.append(match\_stats)

    else: matches\_missing.append(link)

time.sleep(0.5)

За 4 рассматриваемых сезона в рамках регулярного чемпионата КХЛ было сыграно 2789 матчей.

Из спарсенных данных построен датасет, фрагмент которого представлен в таблице ниже.

data = pd.DataFrame(all\_stats, columns=['date', 'match', 'team\_h', 'score\_h', 'ppp\_h', 'ppp\_a', 'ppk\_h', 'ppk\_a', 'numa\_h', 'numa\_a', 'wt\_h', 'wt\_a', 'pt\_h', 'pt\_a', 'sog\_h', 'sog\_a', 'team\_a', 'score\_a', 'bs\_h', 'pm\_h', 'at\_h', 'tot', 'bs\_a', 'pm\_a', 'at\_a', 'end', 'final\_score\_h', 'final\_score\_a'])

# сортируем признаки по принадлежности к командам

data\_sorted = data.iloc[:, [0,1,2,16,3,17,4,6,8,10,12,14,18,19,20,5,7,9,11,13,15,22,23,24,25,26,27,21]]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **date** | **match** | **team\_h** | **score\_h** | **ppp\_h** | **ppp\_a** | **ppk\_h** | **ppk\_a** | **numa\_h** | **numa\_a** | **wt\_h** | **wt\_a** | **pt\_h** | **pt\_a** | **sog\_h** | **sog\_a** | **team\_a** | **score\_a** | **bs\_h** | **pm\_h** | **at\_h** | **tot** | **bs\_a** | **pm\_a** | **at\_a** | **end** | **final\_score\_h** | **final\_score\_a** |
| **0** | 14.01.2022 12:30 | 589 | Адмирал | 3 | 1 | 1 | 0 | 0 | 4 | 5 | 29 | 29 | 10 | 8 | 16 | 32 | Сибирь | 2 | 29 | 7 | 10:35 | Всего | 10 | 16 | 15:06 | main\_time | 3 | 2 |
| **1** | 13.01.2022 17:00 | 582 | Автомобилист | 5 | 2 | 0 | 0 | 0 | 5 | 4 | 37 | 20 | 8 | 10 | 40 | 33 | Северсталь | 2 | 12 | 15 | 16:34 | Всего | 16 | 5 | 14:13 | main\_time | 5 | 2 |
| **2** | 13.01.2022 19:30 | 588 | СКА | 2 | 0 | 0 | 0 | 0 | 4 | 2 | 31 | 19 | 4 | 8 | 33 | 21 | ХК Сочи | 1 | 10 | 6 | 21:07 | Всего | 22 | 10 | 09:13 | main\_time | 2 | 1 |
| **3** | 12.01.2022 12:30 | 578 | Адмирал | 2 | 0 | 0 | 0 | 1 | 2 | 1 | 24 | 35 | 4 | 6 | 30 | 27 | Сибирь | 1 | 27 | 10 | 12:54 | Всего | 22 | 17 | 13:11 | main\_time | 2 | 1 |
| **4** | 12.01.2022 19:00 | 580 | Нефтехимик | 4 | 0 | 0 | 0 | 0 | 2 | 1 | 27 | 31 | 2 | 4 | 26 | 24 | Трактор | 2 | 23 | 13 | 10:02 | Всего | 9 | 13 | 16:40 | main\_time | 4 | 2 |

Собрана следующая информация о проведенных матчах:

* дата и время проведения матча (**date**);
* номер матча на сайте (**match**);
* название команды (**team**);
* кол-во забитых командой шайб без учета буллитов (**score**);
* кол-во шайб, забитых в большинстве (**ppp**);
* кол-во шайб, забитых в меньшинстве (**ppk**);
* численные преимущества (**numa**);
* выигранные вбрасывания (**wt**);
* штрафное время, минут (**pt**);
* броски по воротам (**sog**);
* блокированные броски (**bs**);
* силовые приемы (**pm**);
* время в атаке, минут (**at**);
* признак для проверки корректности работы парсинга (**tot**);
* вариант окончания матча - в основное или дополнительное время (**end**);
* кол-во забитых командой шайб с учетом буллитов (**final\_score**).

Статистика домашней команды обозначалась суффиксом «**\_h**», гостевой – суффиксом «**\_a**». Индексом является порядковый номер строки исходного датасета.

1. **Подготовка данных**

Проверена корректность работы парсинга по признаку **tot**, в котором было одно уникальное значение «Всего» – своего рода метка на веб-странице, соответствующая окончанию блока с необходимой для работы информацией.

В исходных данных содержалась ненужная информация о матчах всех звезд, которая в ходе препроцессинга была удалена.

Имелись пропуски по ряду матчей в блокированных бросках и силовых приемах у гостевых команд, которые заполнялись значениями аналогичных показателей домашних команд.

Также добавлен целевой признак, которым являлась победа домашней команды (**home\_win**). Соответственно, значение 1 в данном признаке – победа домашней команды, 0 – победа гостевой команды. Таким образом, данная работа сводилась к решению задачи бинарной классификации.

data\_sorted['home\_win'] = np.where(data\_sorted.final\_score\_h > data\_sorted.final\_score\_a, 1, 0)

Помимо целевого в датасет добавлены следующие признаки:

* порядковый номер сезона (**season**);

data\_sorted['season'] = np.where(data\_sorted['date'] < pd.Timestamp('2019-06-01 00:00:00'), 1, np.NaN)

data\_sorted['season'] = np.where((data\_sorted['date'] > pd.Timestamp('2019-08-01 00:00:00')) & (data\_sorted['date'] < pd.Timestamp('2020-06-01 00:00:00')), 2, data\_sorted['season'])

data\_sorted['season'] = np.where((data\_sorted['date'] > pd.Timestamp('2020-08-01 00:00:00')) & (data\_sorted['date'] < pd.Timestamp('2021-06-01 00:00:00')), 3, data\_sorted['season'])

data\_sorted['season'] = np.where((data\_sorted['date'] > pd.Timestamp('2021-08-01 00:00:00')), 4, data\_sorted['season'])

* кол-во очков у каждой команды (**tp**);

# добавляем очки домашней команды по результатам матча

data\_sorted['points\_h'] = np.where(data\_sorted.home\_win == 1, 2, np.NaN)

data\_sorted['points\_h'] = np.where((data\_sorted.home\_win == 0) & (data\_sorted.end == 'add\_time'), 1, data\_sorted['points\_h'])

data\_sorted['points\_h'] = np.where((data\_sorted.home\_win == 0) & (data\_sorted.end == 'main\_time'), 0, data\_sorted['points\_h'])

# добавляем очки гостевой команды по результатам матча

data\_sorted['points\_a'] = np.where(data\_sorted.home\_win == 0, 2, np.NaN)

data\_sorted['points\_a'] = np.where((data\_sorted.home\_win == 1) &  (data\_sorted.end == 'add\_time'), 1, data\_sorted['points\_a'])

data\_sorted['points\_a'] = np.where((data\_sorted.home\_win == 1) &  (data\_sorted.end == 'main\_time'), 0, data\_sorted['points\_a'])

data\_sorted[['tp\_h', 'tp\_a']] = np.NaN

# создаем колонки для подсчета очков индивидульно по каждой команде

for team in data\_sorted.team\_h.unique():

    data\_sorted['p\_{}'.format(team)] = np.NaN

# создаем пустой датафрейм для добавления очков по каждому сезону

d = {}

for i in range(len(data\_sorted.dtypes)):

    d[data\_sorted.dtypes.index[i]] = pd.Series(dtype=data\_sorted.dtypes.values[i])

data\_sorted\_v1 = pd.DataFrame(d)

for season in data\_sorted['season'].unique():

    df = data\_sorted[data\_sorted['season'] == season]

    for team in data\_sorted.team\_h.unique():

        df['p\_{}'.format(team)] = np.where(df.team\_h == team, pd.Series(np.where(df.team\_h == team, df.points\_h, np.where(df.team\_a == team, df.points\_a, 0)).cumsum()).shift().fillna(0), df['p\_{}'.format(team)])

df['p\_{}'.format(team)] = np.where(df.team\_a == team, pd.Series(np.where(df.team\_a == team, df.points\_a, np.where(df.team\_h == team, df.points\_h, 0)).cumsum()).shift().fillna(0), df['p\_{}'.format(team)])

df['tp\_h'] = np.where(df.team\_h == team, df['p\_{}'.format(team)], df['tp\_h'])

df['tp\_a'] = np.where(df.team\_a == team, df['p\_{}'.format(team)], df['tp\_a'])

data\_sorted\_v1 = pd.concat([data\_sorted\_v1, df])

data\_sorted\_v1.drop(columns=data\_sorted\_v1.columns[-29:-27].to\_list() + data\_sorted\_v1.columns[-25:].to\_list(), inplace=True)

* число, месяц, день недели проведения матча и является ли день выходным (**day, month, day\_of\_week, weekend**);

data\_sorted\_v1['day'] = data\_sorted\_v1.date.dt.day

data\_sorted\_v1['month'] = data\_sorted\_v1.date.dt.month

data\_sorted\_v1['day\_of\_week'] = data\_sorted\_v1.date.dt.day\_of\_week

data\_sorted\_v1['weekend'] = np.where((data\_sorted\_v1['day\_of\_week'] == 5) | (data\_sorted\_v1['day\_of\_week'] == 6), 1, 0)

* наличие переезда у команд перед матчем (**reloc**);

teams = data\_sorted\_v1.team\_h.unique().tolist()

# список городов, в которых играют команды

cities = ['Казань', 'Челябинск', 'Магнитогорск', 'Екатеринбург',

          'Москва', 'Москва', 'Череповец', 'Ярославль',

          'Хельсинки', 'Нижний Новгород', 'Нижнекамск', 'Сочи',

          'Санкт-Петербург', 'Рига', 'Братислава', 'Омск',

          'Минск', 'Подольск', 'Новосибирск', 'Москва',

          'Хабаровск', 'Нур-Султан', 'Уфа', 'Пекин', 'Владивосток']

d\_cities = dict(zip(teams, cities))

data\_sorted\_v1['city\_h'] = data\_sorted\_v1['team\_h'].map(d\_cities)

data\_sorted\_v1[['reloc\_h', 'reloc\_a', 'rest\_h', 'rest\_a', 'intense\_h', 'intense\_a']] = np.NaN

for team in data\_sorted\_v1.team\_h.unique():

    index = data\_sorted\_v1[(data\_sorted\_v1['team\_h'] == team) | (data\_sorted\_v1['team\_a'] == team)].index

# определяем наличие переезда у домашней команды

    data\_sorted\_v1.loc[index, 'reloc\_h'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_h'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        0,

        np.where(data\_sorted\_v1.loc[index, 'city\_h'] == data\_sorted\_v1.loc[index, 'city\_h'].shift(),

        0,

        1)),

        data\_sorted\_v1.loc[index, 'reloc\_h'])

# определяем наличие переезда у гостевой команды

    data\_sorted\_v1.loc[index, 'reloc\_a'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_a'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        1,

        np.where(data\_sorted\_v1.loc[index, 'city\_h'] == data\_sorted\_v1.loc[index, 'city\_h'].shift(),

        0,

        1)),

        data\_sorted\_v1.loc[index, 'reloc\_a'])

* кол-во дней отдыха у команд между играми (**rest**);

# определяем макс. кол-во дней отдыха среди команд в рамках одного сезона (исключаем команду Слован, т.к. она играла только в 1м сезоне, и команду Адмирал - не играла в 3м сезоне)

max\_rest = []

for season in data\_sorted\_v1['season'].unique():

    for team in data\_sorted\_v1.team\_h.unique()[np.where((data\_sorted\_v1.team\_h.unique() != 'Слован') & (data\_sorted\_v1.team\_h.unique() != 'Адмирал'))]:

        df = data\_sorted\_v1[((data\_sorted\_v1.team\_h == team) | (data\_sorted\_v1.team\_a == team)) & (data\_sorted\_v1.season == season)]

        max\_rest.append(round((df.date[1:] - df.date.shift()[1:]).max().days + (df.date[1:] - df.date.shift()[1:]).max().seconds/86400))

# определяем кол-во дней отдыха у домашней команды

    data\_sorted\_v1.loc[index, 'rest\_h'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_h'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        max(max\_rest), # для первых матчей проставляем макс. кол-во дней отдыха в течение сезона

        (data\_sorted\_v1.loc[index, 'date'] - data\_sorted\_v1.loc[index, 'date'].shift()).dt.total\_seconds()/86400),

        data\_sorted\_v1.loc[index, 'rest\_h'])

# определяем кол-во дней отдыха у гостевой команды

    data\_sorted\_v1.loc[index, 'rest\_a'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_a'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        max(max\_rest), # для первых матчей проставляем макс. кол-во дней отдыха в течение сезона

        (data\_sorted\_v1.loc[index, 'date'] - data\_sorted\_v1.loc[index, 'date'].shift()).dt.total\_seconds()/86400),

        data\_sorted\_v1.loc[index, 'rest\_a'])

* наличие предыдущего напряженного матча – завершившегося в овертайме или по буллитам (**intense**);

# определяем наличие напряженного матча у домашней команды

    data\_sorted\_v1.loc[index, 'intense\_h'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_h'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        0,

        np.where(data\_sorted\_v1.loc[index, 'end'].shift() == 'add\_time',

        1,

        0)),

        data\_sorted\_v1.loc[index, 'intense\_h'])

# определяем наличие напряженного матча у гостевой команды

    data\_sorted\_v1.loc[index, 'intense\_a'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_a'] == team,

        np.where((data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index, 'season'] - data\_sorted\_v1.loc[index, 'season'].shift() == 1),

        0,

        np.where(data\_sorted\_v1.loc[index, 'end'].shift() == 'add\_time',

        1,

        0)),

        data\_sorted\_v1.loc[index, 'intense\_a'])

data\_sorted\_v1.drop(columns=['city\_h'], inplace=True)

* кол-во побед дома у домашней команды и кол-во побед в гостях у гостевой команды на момент проведения матча (**win**);

data\_sorted\_v1[['win\_h', 'win\_a']] = np.NaN

for season in data\_sorted\_v1.season.unique():

    for team in data\_sorted\_v1.team\_h.unique():

        index\_h = data\_sorted\_v1[(data\_sorted\_v1['team\_h'] == team) & (data\_sorted\_v1['season'] == season)].index

        index\_a = data\_sorted\_v1[(data\_sorted\_v1['team\_a'] == team) & (data\_sorted\_v1['season'] == season)].index

# кол-во побед дома у домашней команды

        data\_sorted\_v1.loc[index\_h, 'win\_h'] = np.where((data\_sorted\_v1.loc[index\_h, 'season'] - data\_sorted\_v1.loc[index\_h, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index\_h, 'season'] - data\_sorted\_v1.loc[index\_h, 'season'].shift() == 1), 0, data\_sorted\_v1.loc[index\_h, 'home\_win'].shift().cumsum())

# кол-во побед в гостях у гостевой команды

        data\_sorted\_v1.loc[index\_a, 'win\_a'] = np.where((data\_sorted\_v1.loc[index\_a, 'season'] - data\_sorted\_v1.loc[index\_a, 'season'].shift()).isna() | (data\_sorted\_v1.loc[index\_a, 'season'] - data\_sorted\_v1.loc[index\_a, 'season'].shift() == 1), 0, (1 - data\_sorted\_v1.loc[index\_a, 'home\_win'].shift()).cumsum())

* кол-во побед команд в очных противостояниях, начиная с 1 рассматриваемого сезона (**ftf**);

data\_sorted\_v1[['ftf\_h', 'ftf\_a']] = np.NaN

for team\_h in data\_sorted\_v1.team\_h.unique():

    for team\_a in data\_sorted\_v1.team\_a.unique():

        index = data\_sorted\_v1[((data\_sorted\_v1.team\_h == team\_h) & (data\_sorted\_v1.team\_a == team\_a)) | ((data\_sorted\_v1.team\_h == team\_a) & (data\_sorted\_v1.team\_a == team\_h))].index

# кол-во очных побед дома у домашней команды

        data\_sorted\_v1.loc[index, 'ftf\_h'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_h'] == team\_h,

        pd.Series(np.where(data\_sorted\_v1.loc[index, 'team\_h'] == team\_h, data\_sorted\_v1.loc[index, 'home\_win'], (1-data\_sorted\_v1.loc[index, 'home\_win'])).cumsum()).shift().fillna(0),

        pd.Series(np.where(data\_sorted\_v1.loc[index, 'team\_h'] == team\_a, data\_sorted\_v1.loc[index, 'home\_win'], (1-data\_sorted\_v1.loc[index, 'home\_win'])).cumsum()).shift().fillna(0))

# кол-во очных побед дома у гостевой команды

        data\_sorted\_v1.loc[index, 'ftf\_a'] = np.where(

        data\_sorted\_v1.loc[index, 'team\_a'] == team\_a,

        pd.Series(np.where(data\_sorted\_v1.loc[index, 'team\_a'] == team\_a, (1-data\_sorted\_v1.loc[index, 'home\_win']), data\_sorted\_v1.loc[index, 'home\_win']).cumsum()).shift().fillna(0),

        pd.Series(np.where(data\_sorted\_v1.loc[index, 'team\_a'] == team\_h, (1-data\_sorted\_v1.loc[index, 'home\_win']), data\_sorted\_v1.loc[index, 'home\_win']).cumsum()).shift().fillna(0))

* среднее кол-во очков оппонентов, против которых играли команды до рассматриваемого матча (**mop**);

data\_sorted\_v1[['mop\_h', 'mop\_a']] = np.NaN

for season in data\_sorted\_v1.season.unique():

    for team in data\_sorted\_v1.team\_h.unique():

        index\_h = data\_sorted\_v1[(data\_sorted\_v1['team\_h'] == team) & (data\_sorted\_v1['season'] == season)].index

        index\_a = data\_sorted\_v1[(data\_sorted\_v1['team\_a'] == team) & (data\_sorted\_v1['season'] == season)].index

# создаем массив из очков гостевых команд при игре дома и очков домашних команд при игре в гостях

        tp = pd.concat([data\_sorted\_v1.loc[index\_a, 'tp\_h'], data\_sorted\_v1.loc[index\_h, 'tp\_a']]).sort\_index()

# создаем датафрейм из кол-ва проведенных игр

        games\_count = pd.DataFrame(index=tp.index, columns=['number\_of\_games'], data=np.ones(len(tp.index)))

# среднее кол-во очков оппонентов домашней команды

        data\_sorted\_v1.loc[index\_h, 'mop\_h'] = np.where(((data\_sorted\_v1.loc[tp.index, 'season'] - data\_sorted\_v1.loc[tp.index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[tp.index, 'season'] - data\_sorted\_v1.loc[tp.index, 'season'].shift() == 1)).loc[index\_h], 0, tp.shift().cumsum()[index\_h] / games\_count.shift().cumsum().loc[index\_h, 'number\_of\_games'].values)

# среднее кол-во очков оппонентов гостевой команды

        data\_sorted\_v1.loc[index\_a, 'mop\_a'] = np.where(((data\_sorted\_v1.loc[tp.index, 'season'] - data\_sorted\_v1.loc[tp.index, 'season'].shift()).isna() | (data\_sorted\_v1.loc[tp.index, 'season'] - data\_sorted\_v1.loc[tp.index, 'season'].shift() == 1)).loc[index\_a], 0, tp.shift().cumsum()[index\_a] / games\_count.shift().cumsum().loc[index\_a, 'number\_of\_games'].values)

Также проведена проверка на соответствие количества матчей, по которым спарсена информация, нумерации матчей на сайте (признак **match**).

def empty\_matches\_count (season, matches\_missing):

    if season == 1:

        text = '/671/'

    elif season == 2:

        text = '/851/'

    elif season == 3:

        text = '/1045/'

    else:

        text = '/1097/'

    return len([x for x in matches\_missing if text in x])

for season in range(1, 5):

    if data\_sorted\_v1[data\_sorted\_v1.season == season].match.max() != data\_sorted\_v1[data\_sorted\_v1.season == season].match.shape[0] + empty\_matches\_count(season, matches\_missing):

        print('По {} сезону есть несоответствие между кол-вом матчей, по которым спарсена информация, и фактической нумерацией матчей'.format(season))

    else:

        pass

По 4 сезону было несоответствие, обусловленное пропусками в нумерации матчей на сайте, при этом информация по всем фактически проведенным играм спарсена корректно.

Удалены не нужные для дальнейшей работы признаки (**tot,** **date, match, score, end**).

Проверено наличие сильных корреляций между признаками, результаты проверки представлены в таблице ниже.

features\_correlation = pd.DataFrame(columns=['feature\_1', 'feature\_2', 'correlation\_pearson', 'correlation\_spearman'])

index = 0

for i in range(3, len(data\_sorted\_v1.columns)):

    for a in range (i+1, len(data\_sorted\_v1.columns)):

        features\_correlation.loc[index, 'feature\_1'] = data\_sorted\_v1.columns[i]

        features\_correlation.loc[index, 'feature\_2'] = data\_sorted\_v1.columns[a]

        features\_correlation.loc[index, 'correlation\_pearson'] = data\_sorted\_v1[data\_sorted\_v1.columns[i]].corr(data\_sorted\_v1[data\_sorted\_v1.columns[a]], method='pearson')

        features\_correlation.loc[index, 'correlation\_spearman'] = data\_sorted\_v1[data\_sorted\_v1.columns[i]].corr(data\_sorted\_v1[data\_sorted\_v1.columns[a]], method='spearman')

        index += 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **feature\_1** | **feature\_2** | **correlation\_pearson** | **correlation\_spearman** |
| 609 | tp\_h | tp\_a | 0.79079 | 0.8321 |
| 621 | tp\_h | win\_h | 0.96445 | 0.9656 |
| 622 | tp\_h | win\_a | 0.70637 | 0.7568 |
| 625 | tp\_h | mop\_h | 0.87699 | 0.89812 |
| 626 | tp\_h | mop\_a | 0.87744 | 0.89833 |
| 638 | tp\_a | win\_h | 0.76446 | 0.80477 |
| 639 | tp\_a | win\_a | 0.94854 | 0.95138 |
| 642 | tp\_a | mop\_h | 0.88491 | 0.90452 |
| 643 | tp\_a | mop\_a | 0.88395 | 0.90257 |
| 689 | day\_of\_week | weekend | 0.79452 | 0.78122 |
| 768 | win\_h | mop\_h | 0.84281 | 0.86428 |
| 769 | win\_h | mop\_a | 0.84571 | 0.86598 |
| 772 | win\_a | mop\_h | 0.78235 | 0.81468 |
| 773 | win\_a | mop\_a | 0.783 | 0.81395 |
| 779 | mop\_h | mop\_a | 0.98748 | 0.9894 |

Зависимости между очками играющих команд нет (признаки **tp\_h, tp\_a**) - сильная корреляция объясняется набором очков по ходу сезона. Тем же объясняется сильная корреляция в комбинациях признаков **tp, win** и **mop**. Указанные признаки оставлены без изменений.

Между днем недели и наличием выходного (признаки **day\_of\_week, weekend**) связь очевидна - признак **weekend** удален.

Проведена проверка матчей на наличие выбросов, аномалий, ошибок и обнаружены следующие:

for column in data\_sorted\_v1.columns[-(len(data\_sorted\_v1.columns) - 3):]:

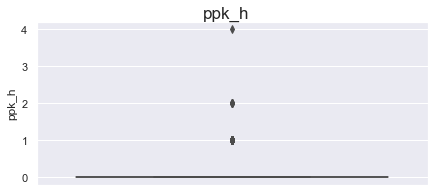
    sns.boxplot(y=data\_sorted\_v1[column])

    sns.set(rc={'figure.figsize':(7, 3)})

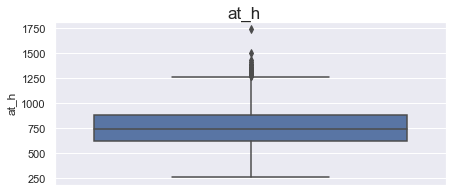
    plt.suptitle(column, fontsize=17, y=0.95)

    plt.show()

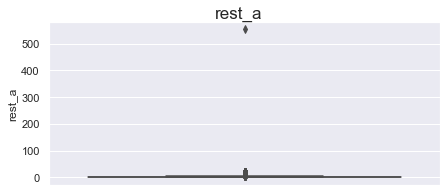
* в матче с **ppk\_h** равным 4 - ошибка, все шайбы заброшены в равных составах;



* в матче с **at\_h** равным 1740 - ошибка, во втором периоде у домашней команды время владения шайбой в атаке составило 19:52 и является нереальным значением. Макс время в атаке за весь матч у игрока данной команды составляет 17:22, т.е. вероятное время в атаке команды во 2м периоде составляет 9:52 - значение **at\_h** скорректировано с учетом данной информации;



* в матче с **rest\_a** равным 554 - значение появилось из-за отсутствия команды «Адмирал» в 3 сезоне. Значение заменено на 20 (макс. кол-во дней отдыха).



1. **Генерация датасетов**

Частью данной работы являлось исследование влияния вариантов формирования датасетов (с учетом метода масштабирования, т.к. значения разных признаков находились в разных диапазонах) на качество обучаемых моделей. В качестве данных для обучения моделей была информация по предшествующим матчам играющих команд. Т.к. по ходу сезона форма команд меняется, брать в расчет большое кол-во предшествующих игр виделось нецелесообразным – в итоге их кол-во составило 1, 3, и 5.

При 1 матче формировалось 3 исходных датасета без масштабирования:

* из статистики каждой команды в предшествующем матче (**df\_1**);
* из разности статистики рассматриваемой команды и её оппонента в предшествующем матче (**df\_2**);
* из разницы разностей статистики играющих команд, полученной из датасета **df\_2** (**df\_3**).

При 3 матчах формировалось 6 исходных датасетов без масштабирования:

* из средней и медианной статистики каждой команды в 3 предшествующих матчах (**df\_4, df\_5**);
* из средней и медианной разности статистики рассматриваемой команды и её оппонентов в 3 предшествующих матчах (**df\_6, df\_7**);
* из разницы средней и медианной разностей статистики в 3 предшествующих матчах, полученной из датасетов **df\_6, df\_7** (**df\_8, df\_9**).

При 5 матчах формировалось 6 исходных датасетов без масштабирования:

* из средней и медианной статистики каждой команды в 5 предшествующих матчах (**df\_10, df\_11**);
* из средней и медианной разности статистики рассматриваемой команды и её оппонентов в 5 предшествующих матчах (**df\_12, df\_13**);
* из разницы средней и медианной разностей статистики в 5 предшествующих матчах, полученной из датасетов **df\_12, df\_13** (**df\_14, df\_15**).

Для формирования 15 исходных датасетов (точнее групп датасетов с учетом обучающих и тестовых наборов признаков) написана функция **df\_creator** с масштабированием циклических временных значений (признаки **day, month, day\_of\_week**) через sin и cos для приведения их к сопоставимой значимости, разделением на обучающую, тестовую выборки и делением их на обучающие и целевой признаки. Деление на обучающую и тестовую выборки выполнялось по сезонам – первые 3 шли в обучение, 4 – на тест.

# df                    - исходный датасет

# match\_count           - кол-во матчей, которые берутся в расчёт

# type\_of\_calculation   - тип расчета (None - в расчёт берется исходная статистика, 'diff' - разность статистики для каждой команды по предыдущим матчам, 'diff\_diff' - разница разностей статистики для каждой команды по предыдущим матчам)

# mean\_median           - опция расчета среднего или медианы по нескольким матчам (доступна для кол-ва матчей 3 и 5)

def df\_creator(df, match\_count, type\_of\_calculation=None, mean\_median=None):

    # делаем копию исходного датасета для трансформации

    df\_main = df.copy()

    if type\_of\_calculation == None:

        # определяем столбцы для сдвига и вставки у домашней команды

        columns\_to\_shift\_paste\_h = ['ppp\_h', 'ppk\_h', 'numa\_h', 'wt\_h', 'pt\_h', 'sog\_h', 'bs\_h', 'pm\_h', 'at\_h', 'final\_score\_h']

        # определяем столбцы для сдвига и вставки у гостевой команды

        columns\_to\_shift\_paste\_a = ['ppp\_a', 'ppk\_a', 'numa\_a', 'wt\_a', 'pt\_a', 'sog\_a', 'bs\_a', 'pm\_a', 'at\_a', 'final\_score\_a']

    elif type\_of\_calculation == 'diff' or type\_of\_calculation == 'diff\_diff':

        column\_prefix = ['ppp', 'ppk', 'numa', 'wt', 'pt', 'sog', 'bs', 'pm', 'at', 'final\_score', 'tp', 'reloc', 'rest', 'intense', 'win', 'ftf', 'mop']

        columns\_to\_shift\_paste\_h, columns\_to\_shift\_paste\_a = [], []

        # считаем разности статистик, которые будут сдвигаться

        for pref in column\_prefix[:10]:

            df\_main['d\_' + pref + '\_h'] = df\_main[pref + '\_h'] - df\_main[pref + '\_a']

            df\_main['d\_' + pref + '\_a'] = df\_main[pref + '\_a'] - df\_main[pref + '\_h']

            df\_main.drop(columns=[pref + '\_h', pref + '\_a'], inplace=True)

            # определяем столбцы для сдвига и вставки у домашней команды

            columns\_to\_shift\_paste\_h.append('d\_' + pref + '\_h')

            # определяем столбцы для сдвига и вставки у гостевой команды

            columns\_to\_shift\_paste\_a.append('d\_' + pref + '\_a')

    else:

        raise TypeError(f'Type of calculation "{type\_of\_calculation}" does not exist')

    def stat\_shifter(team, df\_team, columns\_to\_shift\_paste):

        # определяем группу статистик (домашняя или гостевая), которые будут рассчитываться для команды

        if 'ppp\_h' in columns\_to\_shift\_paste or 'd\_ppp\_h' in columns\_to\_shift\_paste:

            sec\_columns\_set = columns\_to\_shift\_paste\_a

            home\_away\_team = 'team\_h'

        else:

            sec\_columns\_set = columns\_to\_shift\_paste\_h

            home\_away\_team = 'team\_a'

        if match\_count == 1 and mean\_median == None:

            df\_main.loc[df\_team.index, columns\_to\_shift\_paste] = np.where((df\_main.loc[df\_team.index, [home\_away\_team]\*10] == team), df\_team.drop(columns=['season']).shift(), df\_main.loc[df\_team.index, columns\_to\_shift\_paste])

        elif (match\_count == 3 or match\_count == 5) and mean\_median == 'mean':

            df\_main.loc[df\_team.index, columns\_to\_shift\_paste] = np.where((df\_main.loc[df\_team.index, [home\_away\_team]\*10] == team), df\_team.drop(columns=['season']).rolling(window=match\_count).mean().shift(), df\_main.loc[df\_team.index, columns\_to\_shift\_paste])

        elif (match\_count == 3 or match\_count == 5) and mean\_median == 'median':

            df\_main.loc[df\_team.index, columns\_to\_shift\_paste] = np.where((df\_main.loc[df\_team.index, [home\_away\_team]\*10] == team), df\_team.drop(columns=['season']).rolling(window=match\_count).median().shift(), df\_main.loc[df\_team.index, columns\_to\_shift\_paste])

        else:

            raise TypeError(f'Parameter mean\_median with value "{mean\_median}" does not exist, match count ({match\_count}) is not set correctly (must be 1, 3 or 5) or their combination is set incorrectly')

    for team in df\_main.team\_h.unique():

        # создаем отдельный датасет с матчами рассматриваемой команды

        df\_team = df\_main[(df\_main['team\_h'] == team) | (df\_main['team\_a'] == team)]

        # оставляем только статистику рассматриваемой команды

        df\_team.drop(columns=['home\_win', 'tp\_h', 'tp\_a', 'delta\_p', 'day', 'month', 'day\_of\_week', 'reloc\_h', 'reloc\_a', 'rest\_h', 'rest\_a', 'intense\_h', 'intense\_a', 'win\_h', 'win\_a', 'ftf\_h', 'ftf\_a', 'mop\_h', 'mop\_a'], inplace=True)

        columns\_h = ['ppp\_h', 'ppk\_h', 'numa\_h', 'wt\_h', 'pt\_h', 'sog\_h', 'bs\_h', 'pm\_h', 'at\_h', 'final\_score\_h']

        columns\_a = ['ppp\_a', 'ppk\_a', 'numa\_a', 'wt\_a', 'pt\_a', 'sog\_a', 'bs\_a', 'pm\_a', 'at\_a', 'final\_score\_a']

        df\_team[columns\_to\_shift\_paste\_h] = np.where(df\_team[['team\_a'] \* 10] == team, df\_team[columns\_to\_shift\_paste\_a], df\_team[columns\_to\_shift\_paste\_h])

        df\_team.drop(columns=columns\_to\_shift\_paste\_a + ['team\_h', 'team\_a'], inplace=True)

        stat\_shifter(team, df\_team, columns\_to\_shift\_paste\_h)

        stat\_shifter(team, df\_team, columns\_to\_shift\_paste\_a)

        # метим кол-во матчей, на которое производится сдвиг, по каждому сезону у всех команд для последующего удаления

        index\_first\_games = df\_team[((df\_team.season - df\_team.season.shift(match\_count)).isna()) | ((df\_team.season - df\_team.season.shift(match\_count)) > 0)].index

        for i in index\_first\_games:

                if df\_main.loc[i, 'team\_h'] == team:

                        df\_main.loc[i, 'team\_h'] = np.nan

                else: df\_main.loc[i, 'team\_a'] = np.nan

    # удаляем первые матчи команд

    df\_main.dropna(subset=['team\_h', 'team\_a'], inplace=True)

    # считаем разницу разностей между статистиками играющих команд для 3 типа датасетов

    if type\_of\_calculation == 'diff\_diff':

        for pref in column\_prefix[:10]:

            df\_main['dd\_' + pref + '\_ha'] = df\_main['d\_' + pref + '\_h'] - df\_main['d\_' + pref + '\_a']

            df\_main.drop(columns=['d\_' + pref + '\_h', 'd\_' + pref + '\_a'], inplace=True)

        for pref in column\_prefix[-7:]:

            df\_main['dd\_' + pref + '\_ha'] = df\_main[pref + '\_h'] - df\_main[pref + '\_a']

            df\_main.drop(columns=[pref + '\_h', pref + '\_a'], inplace=True)

        df\_main.drop(columns=['delta\_p'], inplace=True)

    else:

        pass

    # приводим к сопоставимой значимости циклические временные значения ('day', 'month', 'day\_of\_week') с помощью тригонометрических функций. Используем sin и cos, чтобы они уравновешивали друг друга и не было перекоса в данных

    df\_main['day\_of\_week\_cos'], df\_main['day\_of\_week\_sin'] = np.cos(2 \* np.pi \* df\_main['day\_of\_week'] / 6), np.sin(2 \* np.pi \* df\_main['day\_of\_week'] / 6)

    df\_main['month\_cos'], df\_main['month\_sin'] = np.cos(2 \* np.pi \* df\_main['month'] / 12), np.sin(2 \* np.pi \* df\_main['month'] / 12)

    df\_main['day\_cos'], df\_main['day\_sin'] = np.cos(2 \* np.pi \* df\_main['day'] / 31), np.sin(2 \* np.pi \* df\_main['day'] / 31)

    # удаляем более не нужные столбцы

    df\_main.drop(columns=['team\_h', 'team\_a', 'day', 'day\_of\_week', 'month'], inplace=True)

    # делим датасет на обучающую и тестовую выборки

    df\_train, df\_test = df\_main[df\_main.season < 4], df\_main[df\_main.season == 4]

    # удаляем ненужный столбец season

    df\_train.drop(columns=['season'], inplace=True)

    df\_test.drop(columns=['season'], inplace=True)

    x\_train, y\_train = df\_train.drop(columns=['home\_win']), df\_train['home\_win']

    x\_test, y\_test = df\_test.drop(columns=['home\_win']), df\_test['home\_win']

    return x\_train, y\_train, x\_test, y\_test

Сформировав 15 групп датасетов и проверив качество разделения на обучающие и тестовые выборки (для одно- и трехматчевых вариантов в обучающие выборки попали 74% наблюдений, для пятиматчевых – 75%), было необходимо смасштабировать данные, не относящиеся к категориальным признакам. Масштабирование, также как и формирование датасетов, было вариативным для оценки наилучшего варианта. Для каждого датасета применено 3 варианта масштабирования:

* минимакс (MinMaxScaler);
* стандартизация (StandardScaler);
* нормализация (Normalizer).

Масштабирование исходных 15 групп датасетов выполнялось с помощью функции **scaler**.

def scaler(df\_train, df\_test):

    cat\_columns = ['reloc\_h', 'reloc\_a', 'intense\_h', 'intense\_a']

    time\_columns = ['day\_of\_week\_cos', 'day\_of\_week\_sin', 'month\_cos', 'month\_sin', 'day\_cos', 'day\_sin']

    # убираем из масштабируемых датасетов категориальные признаки (присутствуют не во всех датасетах) и смаштабированные временные значения

    try:

        df\_train\_wo\_cat\_mm, df\_train\_wo\_cat\_st, df\_train\_wo\_cat\_norm = df\_train.drop(columns=cat\_columns+time\_columns), df\_train.drop(columns=cat\_columns+time\_columns), df\_train.drop(columns=cat\_columns+time\_columns)

        df\_test\_wo\_cat\_mm, df\_test\_wo\_cat\_st, df\_test\_wo\_cat\_norm = df\_test.drop(columns=cat\_columns+time\_columns), df\_test.drop(columns=cat\_columns+time\_columns), df\_test.drop(columns=cat\_columns+time\_columns)

        i = 0

    except KeyError:

        df\_train\_wo\_cat\_mm, df\_train\_wo\_cat\_st, df\_train\_wo\_cat\_norm = df\_train.drop(columns=time\_columns), df\_train.drop(columns=time\_columns), df\_train.drop(columns=time\_columns)

        df\_test\_wo\_cat\_mm, df\_test\_wo\_cat\_st, df\_test\_wo\_cat\_norm = df\_test.drop(columns=time\_columns), df\_test.drop(columns=time\_columns), df\_test.drop(columns=time\_columns)

        i = 1

    mm\_scaler = MinMaxScaler()

    df\_train\_wo\_cat\_mm.loc[:, :] = mm\_scaler.fit\_transform(df\_train\_wo\_cat\_mm)

    df\_test\_wo\_cat\_mm.loc[:, :] = mm\_scaler.transform(df\_test\_wo\_cat\_mm)

    st\_scaler = StandardScaler()

    df\_train\_wo\_cat\_st.loc[:, :] = st\_scaler.fit\_transform(df\_train\_wo\_cat\_st)

    df\_test\_wo\_cat\_st.loc[:, :] = st\_scaler.transform(df\_test\_wo\_cat\_st)

    # особенностью работы нормалайзера является масштабирование по строкам, поэтому сначала объединяем train и test, транспонируем, совместно нормализуем, потом снова транспонируем и разбиваем на train и test

    norm\_scaler = Normalizer()

    index = min(df\_test.index)

    df\_train\_test\_wo\_cat\_norm = pd.concat([df\_train\_wo\_cat\_norm, df\_test\_wo\_cat\_norm])

    df\_train\_test\_wo\_cat\_norm = df\_train\_test\_wo\_cat\_norm.transpose()

    df\_train\_test\_wo\_cat\_norm.loc[:, :] = norm\_scaler.fit\_transform(df\_train\_test\_wo\_cat\_norm)

    df\_train\_test\_wo\_cat\_norm = df\_train\_test\_wo\_cat\_norm.transpose()

    df\_train\_wo\_cat\_norm = df\_train\_test\_wo\_cat\_norm[df\_train\_test\_wo\_cat\_norm.index < index]

    df\_test\_wo\_cat\_norm = df\_train\_test\_wo\_cat\_norm[df\_train\_test\_wo\_cat\_norm.index >= index]

    # добавляем к смасштабированным данным категориальные признаки (если имеются) и смаштабированные временные значения

    if i == 0:

        df\_train\_mm = pd.merge(df\_train\_wo\_cat\_mm, df\_train[cat\_columns+time\_columns], left\_index=True, right\_index=True)

        df\_test\_mm = pd.merge(df\_test\_wo\_cat\_mm, df\_test[cat\_columns+time\_columns], left\_index=True, right\_index=True)

        df\_train\_st = pd.merge(df\_train\_wo\_cat\_st, df\_train[cat\_columns+time\_columns], left\_index=True, right\_index=True)

        df\_test\_st = pd.merge(df\_test\_wo\_cat\_st, df\_test[cat\_columns+time\_columns], left\_index=True, right\_index=True)

        df\_train\_norm = pd.merge(df\_train\_wo\_cat\_norm, df\_train[cat\_columns+time\_columns], left\_index=True, right\_index=True)

        df\_test\_norm = pd.merge(df\_test\_wo\_cat\_norm, df\_test[cat\_columns+time\_columns], left\_index=True, right\_index=True)

    else:

        df\_train\_mm = pd.merge(df\_train\_wo\_cat\_mm, df\_train[time\_columns], left\_index=True, right\_index=True)

        df\_test\_mm = pd.merge(df\_test\_wo\_cat\_mm, df\_test[time\_columns], left\_index=True, right\_index=True)

        df\_train\_st = pd.merge(df\_train\_wo\_cat\_st, df\_train[time\_columns], left\_index=True, right\_index=True)

        df\_test\_st = pd.merge(df\_test\_wo\_cat\_st, df\_test[time\_columns], left\_index=True, right\_index=True)

        df\_train\_norm = pd.merge(df\_train\_wo\_cat\_norm, df\_train[time\_columns], left\_index=True, right\_index=True)

        df\_test\_norm = pd.merge(df\_test\_wo\_cat\_norm, df\_test[time\_columns], left\_index=True, right\_index=True)

    return df\_train\_mm, df\_train\_st, df\_train\_norm, df\_test\_mm, df\_test\_st, df\_test\_norm

После масштабирования получено 45 наборов датасетов, которые были собраны в одном словаре для удобства дальнейшей работы с ними.

total\_df\_dict = {'1':[[df\_1\_x\_train\_mm, df\_1\_y\_train, df\_1\_x\_test\_mm, df\_1\_y\_test], [df\_1\_x\_train\_st, df\_1\_x\_test\_st], [df\_1\_x\_train\_norm, df\_1\_x\_test\_norm]],

                '2':[[df\_2\_x\_train\_mm, df\_2\_y\_train, df\_2\_x\_test\_mm, df\_2\_y\_test], [df\_2\_x\_train\_st, df\_2\_x\_test\_st], [df\_2\_x\_train\_norm, df\_2\_x\_test\_norm]],

                '3':[[df\_3\_x\_train\_mm, df\_3\_y\_train, df\_3\_x\_test\_mm, df\_3\_y\_test], [df\_3\_x\_train\_st, df\_3\_x\_test\_st], [df\_3\_x\_train\_norm, df\_3\_x\_test\_norm]],

                '4':[[df\_4\_x\_train\_mm, df\_4\_y\_train, df\_4\_x\_test\_mm, df\_4\_y\_test], [df\_4\_x\_train\_st, df\_4\_x\_test\_st], [df\_4\_x\_train\_norm, df\_4\_x\_test\_norm]],

                '5':[[df\_5\_x\_train\_mm, df\_5\_y\_train, df\_5\_x\_test\_mm, df\_5\_y\_test], [df\_5\_x\_train\_st, df\_5\_x\_test\_st], [df\_5\_x\_train\_norm, df\_5\_x\_test\_norm]],

                '6':[[df\_6\_x\_train\_mm, df\_6\_y\_train, df\_6\_x\_test\_mm, df\_6\_y\_test], [df\_6\_x\_train\_st, df\_6\_x\_test\_st], [df\_6\_x\_train\_norm, df\_6\_x\_test\_norm]],

                '7':[[df\_7\_x\_train\_mm, df\_7\_y\_train, df\_7\_x\_test\_mm, df\_7\_y\_test], [df\_7\_x\_train\_st, df\_7\_x\_test\_st], [df\_7\_x\_train\_norm, df\_7\_x\_test\_norm]],

                '8':[[df\_8\_x\_train\_mm, df\_8\_y\_train, df\_8\_x\_test\_mm, df\_8\_y\_test], [df\_8\_x\_train\_st, df\_8\_x\_test\_st], [df\_8\_x\_train\_norm, df\_8\_x\_test\_norm]],

                '9':[[df\_9\_x\_train\_mm, df\_9\_y\_train, df\_9\_x\_test\_mm, df\_9\_y\_test], [df\_9\_x\_train\_st, df\_9\_x\_test\_st], [df\_9\_x\_train\_norm, df\_9\_x\_test\_norm]],

                '10':[[df\_10\_x\_train\_mm, df\_10\_y\_train, df\_10\_x\_test\_mm, df\_10\_y\_test], [df\_10\_x\_train\_st, df\_10\_x\_test\_st], [df\_10\_x\_train\_norm, df\_10\_x\_test\_norm]],

                '11':[[df\_11\_x\_train\_mm, df\_11\_y\_train, df\_11\_x\_test\_mm, df\_11\_y\_test], [df\_11\_x\_train\_st, df\_11\_x\_test\_st], [df\_11\_x\_train\_norm, df\_11\_x\_test\_norm]],

                '12':[[df\_12\_x\_train\_mm, df\_12\_y\_train, df\_12\_x\_test\_mm, df\_12\_y\_test], [df\_12\_x\_train\_st, df\_12\_x\_test\_st], [df\_12\_x\_train\_norm, df\_12\_x\_test\_norm]],

                '13':[[df\_13\_x\_train\_mm, df\_13\_y\_train, df\_13\_x\_test\_mm, df\_13\_y\_test], [df\_13\_x\_train\_st, df\_13\_x\_test\_st], [df\_13\_x\_train\_norm, df\_13\_x\_test\_norm]],

                '14':[[df\_14\_x\_train\_mm, df\_14\_y\_train, df\_14\_x\_test\_mm, df\_14\_y\_test], [df\_14\_x\_train\_st, df\_14\_x\_test\_st], [df\_14\_x\_train\_norm, df\_14\_x\_test\_norm]],

                '15':[[df\_15\_x\_train\_mm, df\_15\_y\_train, df\_15\_x\_test\_mm, df\_15\_y\_test], [df\_15\_x\_train\_st, df\_15\_x\_test\_st], [df\_15\_x\_train\_norm, df\_15\_x\_test\_norm]]}

Сильного дисбаланса классов в целевом признаке не было – для датасетов с 1 матчем было 55% наблюдений с положительным классом, для датасетов с 3 и 5 матчами – 54%. Метрикой качества выбрана ROC AUC, не требующая подбора порога для разделения прогнозируемых моделью вероятностей.

Для оценки качества обучаемых моделей определено значение метрики базовой модели, прогнозирующей победу домашней команды во всех матчах – для всех вариантов датасетов оно составило 0.50.

1. **Моделирование**

Для обучения выбрано 5 моделей – LogisticRegression, RandomForestClassifier, XGBClassifier, LGBMClassifier, CatBoostClassifier, каждая из которых обучалась с перебором гиперпараметров.

# делаем 10 фолдов, т.к. обучающих данных немного

folds = KFold(n\_splits=10, shuffle=True, random\_state=800)

# LogisticRegression

lr\_param = [{'solver': ['newton-cg'], 'penalty': ['l2', 'none'], 'C': [1e-5, 1e-3, 1e-1, 1, 10], 'max\_iter': [100, 500, 1000]},

            {'solver': ['lbfgs'], 'penalty': ['l2', 'none'], 'C': [1e-5, 1e-3, 1e-1, 1, 10], 'max\_iter': [100, 500, 1000]},

            {'solver': ['liblinear'], 'penalty': ['l1', 'l2'], 'C': [1e-5, 1e-3, 1e-1, 1, 10], 'max\_iter': [100, 500, 1000], 'random\_state': [800]},

            {'solver': ['sag'], 'penalty': ['l2', 'none'], 'C': [1e-5, 1e-3, 1e-1, 1, 10], 'max\_iter': [100, 500, 1000], 'random\_state': [800]},

            {'solver': ['saga'], 'penalty': ['l1', 'l2', 'elasticnet', 'none'], 'C': [1e-5, 1e-3, 1e-1, 1, 10], 'max\_iter': [100, 500, 1000], 'random\_state': [800]}]

lr\_model = LogisticRegression()

lr\_model\_grid = GridSearchCV(estimator=lr\_model,

                            param\_grid=lr\_param,

                            scoring='roc\_auc',

                            n\_jobs=-1,

                            cv=folds,

                            verbose=1,

                            return\_train\_score=True)

# RandomForestClassifier

rfc\_param = {'n\_estimators': [200, 500],

            'max\_depth': [5, 10],

            'min\_samples\_split': [2, 5],

            'min\_samples\_leaf': [1, 3],

            'max\_features': [0.5, 0.7],

            'max\_samples': [0.5, 0.7]}

rfc\_model = RandomForestClassifier(criterion='log\_loss', bootstrap=True, random\_state=800)

rfc\_model\_grid = GridSearchCV(estimator=rfc\_model,

                            param\_grid=rfc\_param,

                            scoring='roc\_auc',

                            n\_jobs=-1,

                            cv=folds,

                            verbose=1,

                            return\_train\_score=True)

# XGBClassifier

xgb\_param = {'eta': [0.001, 0.01],

            'n\_estimators': [200, 500],

            'max\_depth': [5, 10],

            'subsample': [0.5, 0.7],

            'min\_child\_weight': [2, 5],

            'colsample\_bytree': [0.5, 0.7]}

xgb\_model = XGBClassifier(objective='binary:logistic', eval\_metric='auc', seed=800)

xgb\_model\_grid = GridSearchCV(estimator=xgb\_model,

                            param\_grid=xgb\_param,

                            scoring='roc\_auc',

                            n\_jobs=-1,

                            cv=folds,

                            verbose=1,

                            return\_train\_score=True)

# LGBMClassifier

lgbm\_param ={'n\_estimators': [200, 500],

            'max\_depth': [5, 10],

            'min\_data\_in\_leaf': [5, 10],

            'learning\_rate': [0.001, 0.01],

            'subsample': [0.5, 0.7],

            'colsample\_bytree': [0.5, 0.7]}

lgbm\_model = LGBMClassifier(objective='binary', metric='auc', random\_state=800)

lgbm\_model\_grid = GridSearchCV(estimator=lgbm\_model,

                                param\_grid=lgbm\_param,

                                scoring='roc\_auc',

                                n\_jobs=-1,

                                cv=folds,

                                verbose=1,

                                return\_train\_score=True)

# CatBoostClassifier

cbc\_param = {'iterations': [200, 500],

            'learning\_rate': [0.001, 0.01],

            'subsample': [0.5, 0.7],

            'rsm': [0.5, 0.7],

            'depth': [5, 10],

            'min\_data\_in\_leaf': [1, 3]}

cbc\_model = CatBoostClassifier(loss\_function='Logloss', eval\_metric='AUC:hints=skip\_train~false', random\_seed=800, verbose=0)

cbc\_model\_grid = GridSearchCV(estimator=cbc\_model,

                                param\_grid=cbc\_param,

                                scoring='roc\_auc',

                                n\_jobs=-1,

                                cv=folds,

                                verbose=1,

                                return\_train\_score=True)

Для каждого из 45 наборов датасетов определялась модель с наибольшим ROC AUC.

df\_scores = pd.DataFrame(index=[index.replace('\_x\_train', '') for index in df\_info.index if '\_x\_train' in index ], columns=['best\_model\_mm', 'best\_score\_mm', 'best\_model\_st', 'best\_score\_st','best\_model\_norm', 'best\_score\_norm'])

models = [lr\_model\_grid, rfc\_model\_grid, xgb\_model\_grid, lgbm\_model\_grid, cbc\_model\_grid]

# функция возвращает строку с лучшей моделью и лучшим качеством для одного датасета

def training(x, y):

    df = pd.DataFrame(columns=['best\_model', 'best\_score'])

    for model in models:

        model.fit(x, y)

        df = pd.concat([df, pd.DataFrame(columns=['best\_model', 'best\_score'], data=[[model.best\_estimator\_, model.best\_score\_]])], axis=0)

    df.reset\_index(drop=True, inplace=True)

    df.sort\_values(by='best\_score', ascending=False, inplace=True)

    df.drop(index=df.index[1:], inplace=True)

    return df

for k, v in total\_df\_dict.items():

    # определяем лучшую модель для mm-датасета

    df\_scores.iloc[int(k) - 1, :2] = training(v[0][0], v[0][1])

    # определяем лучшую модель для st-датасета

    df\_scores.iloc[int(k) - 1, 2:4] = training(v[1][0], v[0][1])

    # определяем лучшую модель для st-датасета

    df\_scores.iloc[int(k) - 1, 4:6] = training(v[2][0], v[0][1])

Полученные результаты представлены в таблице ниже.

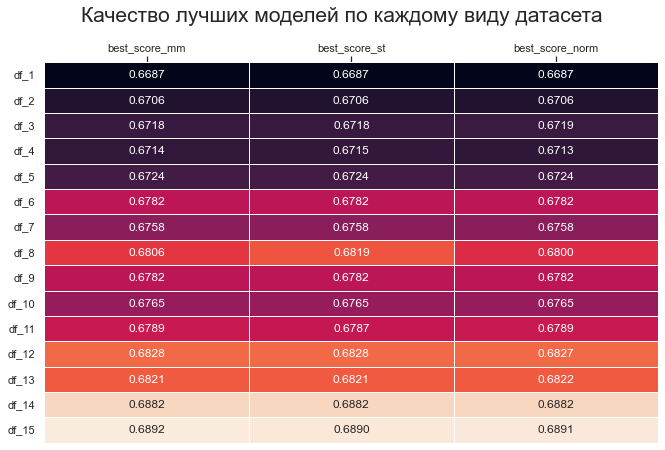
plt.figure(figsize=(11, 7))

plt.suptitle('Качество лучших моделей по каждому виду датасета', fontsize=21, y=0.99)

ax = sns.heatmap(data=df\_scores[['best\_score\_mm', 'best\_score\_st', 'best\_score\_norm']].astype('float64'), annot=True, cbar=False, linewidths=.5, fmt= '.4f')

ax.xaxis.tick\_top()

plt.show()



Лучшей по качеству моделью стала:

print('Лучшей по качеству моделью стала: \n\n {}'.format(df\_scores[df\_scores.best\_score\_norm == df\_scores.best\_score\_norm.max()]['best\_model\_norm'].values.tolist()[0]))

RandomForestClassifier (criterion='log\_loss', max\_depth=5, max\_features=0.7, max\_samples=0.5, min\_samples\_leaf=3, n\_estimators=200, random\_state=800).

Таким образом, максимальный полученный ROC AUC составил 0.689 у модели RandomForest на датаcете с смасштабированными минимаксом данными, состоящими из разницы медианной разности статистики рассматриваемой команды и её оппонентов в 5 предшествующих матчах. Неожиданный результат, учитывая, что в соперниках у случайного леса были 3 наиболее сильные модели градиентного бустинга. В целом же имеется тенденция к повышению качества моделей при увеличении количества предшествующих матчей, которые берутся в расчёт.

Качество лучшей модели на тестовых данных составило 0.637, что говорит о её переобученности – будет чем заняться на досуге. При этом данный показатель выше качества базовой модели на 27%, что уже весьма неплохо.

best\_model = df\_scores[df\_scores.best\_score\_norm == df\_scores.best\_score\_norm.max()]['best\_model\_norm'].values.tolist()[0].fit(df\_14\_x\_train\_norm, df\_14\_y\_train)

# считаем вероятности для классов

probs = best\_model.predict\_proba(df\_14\_x\_test\_norm)

# сохраняем вероятности только для положительного класса

probs = probs[:, 1]

# считаем FPR и TPR, необходимые для построения графика

fpr, tpr, treshold = roc\_curve(df\_14\_y\_test, probs)

roc\_auc = auc(fpr, tpr)

# строим график

plt.figure(figsize=(11, 7))

plt.plot(fpr, tpr, color='darkorange', label='ROC AUC = %0.3f' % roc\_auc)

plt.plot([0, 1], [0, 1], color='navy', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.0])

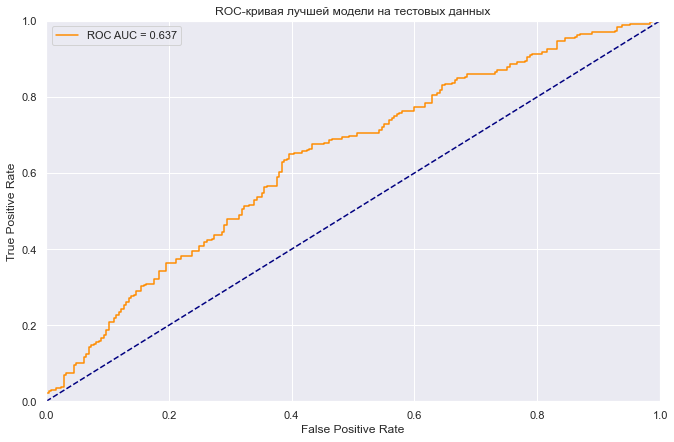
plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

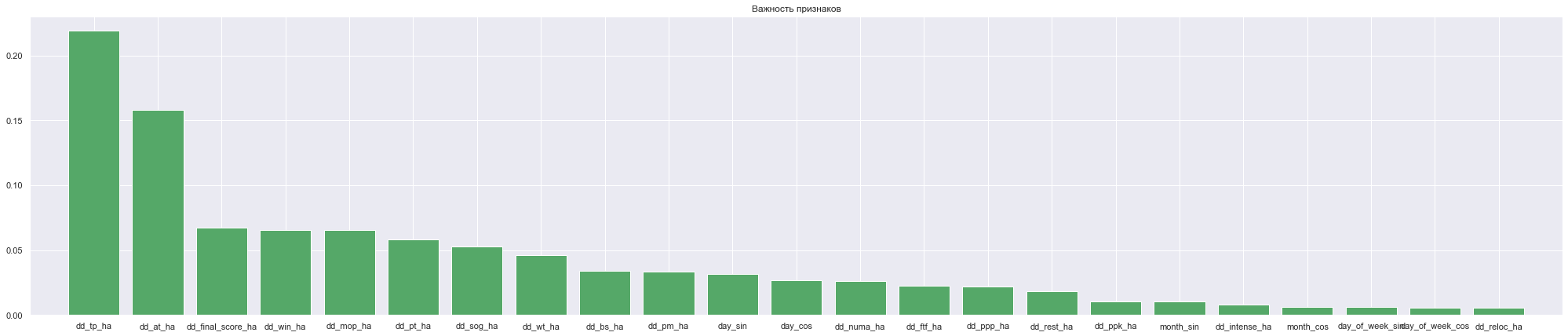
plt.title('ROC-кривая лучшей модели на тестовых данных')

plt.legend(loc="upper left")

plt.show()



Наиболее значимыми признаками для лучшей модели оказались разность очков играющих команд и разность времени, проведенного ими в атаке.



**Заключение**

Поставленная для данной работы задача выполнена – лучшая из обученных моделей показала рост качества на 27% по метрике ROC AUC в сравнении с базовой моделью (константная модель с победой домашней команды), качество которой было на случайном уровне (ROC AUC равен 0.50).

Лучшей стала модель случайного леса, обученная на датаcете с смасштабированными минимаксом данными, состоящими из разницы медианной разности статистики рассматриваемой команды и её оппонентов в 5 предшествующих матчах. На кроссвалидации качество данной модели составило 0.689, на тесте – 0.637, что свидетельствует о её переобучении. Очевидно, что требуется её более тонкая настройка, а также увеличение диапазона гиперпараметров моделей градиентного бустинга, перебираемых при обучении. При этом с помощью нейросетевых моделей можно получить ещё более высокое качество.

Помимо обучения не рассмотренных в данной работе моделей, можно спарсить и/или сгенерировать дополнительные признаки, взять в расчет большее кол-во предшествующих матчей и оценить влияние данных действий на качество модели. В данной же работе наиболее значимыми для лучшей модели оказались разность очков играющих команд (сгенерированный признак) и разность времени, проведенного ими в атаке (спарсенный признак).

На этом моё повествование заканчивается – рад поделиться с вами результатами своей проектной работы, надеюсь было интересно.

Весь код, использованный в работе, доступен в репозитории github (https://github.com/Giproman/DS/blob/main/khl\_\_predictor.ipynb).