Московский Государственный
Университет

факультет Вычислительной Математики
и Кибернетики

Отчёт о выполнении задания №3

студент III курса Н. В. Байтеков

29 декабря 2016 г.

Содержание

1 Постановка общей задачи

В данном задании требовалось реализовать следующие методы для решения задачи SVM:

1. Метод внутренней точки для решения прямой задачи. Рекомендуется использовать библиотеку **cvxopt**, метод **cvxopt.solvers.qp**.

2. Метод внутренней точки для решения двойственной задачи. Рекомендуется использовать библиотеку **cvxopt**, метод **cvxopt.solvers.qp**.

3. Метод субградиентного спуска для решения прямой задачи, а также его стохастический вариант. Рассмотреть критерий останова как по значению целевой функции, так и по норме аргумента. Реализовать полностью самостоятельно. Для этого потребуется вывести формулу для субградиента функционала в прямой задаче SVM без ограничений, вывод вставить в отчет.

4. Метод, используемый в библиотеке **liblinear**. Рекомендуется использовать биндинги из библиотеки **scikit-learn**, класс **sklearn.svm.LinearSVC**.

5. Метод, используемый в библиотеке **libsvm**. Рекомендуется использовать биндинги из библиотеки **scikit-learn**, класс **sklearn.svm.SVC**.

Также в задании присутствовала исследовательская часть, в ходе выполнения которой требовалось провести ряд экспериментов:

1. Исследовать зависимость времени работы реализованных методов для решения задачи линейного SVM от размерности признакового пространства и числа объектов в обучающей выборке. Исследовать скорость сходимости методов. Сравнить методы по полученным значениям целевой функции.

2. Провести эти исследования для случая SVM с RBF ядром для тех методов, где возможен ядровой переход.

3. Реализовать процедуру поиска оптимального значения параметра C и ширины RBF ядра с помощью кросс-валидации (можно воспользоваться библиотекой scikit-learn). Исследовать зависимость ошибки на валидационной выборке от значений этих параметров. Рассмотреть случаи хорошо и трудно разделимых выборок.

4. Сравнить (по скорости сходимости и точности решения) несколько стратегий выбора шага α_t в методе субградиентого спуска: $\alpha, \frac{\alpha}{t}, \frac{\alpha}{\beta^t}$, где α, β — некоторые константы, t — номер итерации.

5. Исследовать, как размер подвыборки, по которой считается субградиент, в методе стохастического суб- градиентного спуска влияет на скорость сходимости метода и на точность решения. В этом и предыдущем пунктах за точное решение можно взять решение, полученное с помощью одного из методов внутренней точки.

6. Для двумерного случая:

   - Провести визуализацию выборки
   - Для линейного SVM и для SVM с RBF ядром провести визуализацию разделяющей поверхности
   - Отобразить объекты, соответствующие опорным векторам.

2 Написание модуля

Дальше приводятся листинги тел функций соответственно реализуемому методу

2.1 Метод внутренней точки для решения прямой задачи

Комментарий к коду: Приводим исходную задачу SVM к задаче квадратичного программирования, используем функцию, возвращаем по соглашению результаты замеров эффективности

```
if self.method == 'primal':
    solvers.options['show_progress'] = verbose
    solvers.options['maxiters'] = max_iter
    solvers.options['feastol'] = tol
    P = np.zeros((D+1+N, D+1+N))
    P[:D, :D] = np.eye(D)

    q = np.zeros(D+1+N)
    q[D+1:] = self.C

    G = np.zeros((2*N, D+1+N))
    G[:N, D+1:] = -np.eye(N)
    G[N:, D+1:] = -np.eye(N)
    G[N:, :D] = -y.reshape((N, 1))*X
    G[N:, D] = -y
    h = np.zeros(2*N)
    h[N:] = -1

    P = matrix(P, tc='d')
    q = matrix(q, tc='d')
    G = matrix(G, tc='d')
    h = matrix(h, tc='d')
    start_time = timer.time()
    sol_obj = solvers.qp(P, q, G, h)
    fit_time = timer.time()-start_time

    solution = np.array(sol_obj['x'])
    status = 0 if sol_obj['status'] == 'optimal' else 1

    self.w = solution[:D].reshape((D, 1))
    self.w0 = solution[D]
    return {'status': status,
            'time': fit_time}
```

Листинг 2.1: Реализация метода внутренней точки для прямой задачи через cvxopt

2.2 Метод внутренней точки для решения обратной задачи

Комментарий к коду: Приводим исходную задачу SVM к задаче квадратичного программирования, используем функцию, возвращаем по соглашению результаты замеров эффективности

```
elif self.method == 'dual':
    solvers.options['show_progress'] = verbose
    solvers.options['maxiters'] = max_iter
    solvers.options['feastol'] = tol
    y = y.reshape((N, 1))
    P = None
    if self.kernel == 'rbf':
        P = self.kmfunc_rbf(X, X)
    elif self.kernel == 'linear':
        P = self.kmfunc_linear(X, X)
    P = matrix(P * y * y.T, tc='d')
    q = matrix(-np.ones(N), tc='d')

    G = np.zeros((2*N, N))
    G[:N] = -np.eye(N)
    G[N:] = np.eye(N)
    h = np.zeros(2*N)
    h[N:] = self.C

    A = matrix(y.T, tc='d')
    b = matrix(0, tc='d')
    G = matrix(G, tc='d')
    h = matrix(h, tc='d')

    start_time = timer.time()
    sol_obj = solvers.qp(P, q, G, h, A, b)
    fit_time = timer.time()-start_time

    solution = np.array(sol_obj['x'])
    status = 0 if sol_obj['status'] == 'optimal' else 1
    self.A = solution.reshape((N, 1))
    self.X_train = X.copy()
    self.y_train = y.copy().reshape((N, 1))

    sv_inds = np.where((self.A > 0.0) * (self.A < self.C))[0]
    self.sv = list(zip(X[sv_inds], y[sv_inds]))
    return {'time': fit_time,
            'status': status}
```

Листинг 2.2: Реализация метода внутренней точки для обратной задачи через cvxopt

2.3 Метод субградиентного спуска для решения прямой задачи

Комментарий к коду: для того, чтобы использовать метод субградиентного спуска, нам нужно найти как минимум формулу субградиента и успешно применить её до сходимости. Формула легко находится путём векторного дифференцирования формулы прямой задачи SVM без ограничений:

$$\nabla Q(\omega) = \omega - C\sum_{n=1}^{N} \mathbb{I}(1 - y_n(\omega^{\mathbf{T}}\mathbf{x_n} + \omega_0) > 0)y_n x_n$$

$$\nabla Q(\omega_0) = -C\sum_{n=1}^{N} \mathbb{I}(1 - y_n(\omega^{\mathbf{T}}\mathbf{x_n} + \omega_0) > 0)y_n$$

Таким образом применяя алгоритм субградиентного спуска, мы получаем итерационно решение задачи:

1. Задаём начальные условия - ω, ω_0

2. Рассчитываем следущее приближение по формуле:

$$\omega^{(n+1)} = \omega^{(n)} - \eta * \nabla Q(\omega^{(n)})$$

$$\omega_0^{(n+1)} = \omega_0^{(n)} - \eta * \nabla Q(\omega_0^{(n)})$$

3. Проверяем условие останова (либо по норме аргумента, либо по значению целевой функции). В зависимости от результатов проверки либо возвращаемся ко второму шагу, либо выдём текущее приближение $\omega^{(n+1)}$ как ответ

Листинг на следующей странице.

2.4 Метод стохастического субградиентного спуска для решения прямой задачи

Комментарий к коду: для того, чтобы использовать стохастический градиентный спуск нужно просто сходиться не по всем признакам, а каждый раз лишь по некоторой подвыборке. Сходиться будет медленнее, зато вычислительная нагрузка в разы меньше. Формула субградиента, очевидно, будет такой же. Листинг на следующей странице.

```python
elif self.method == 'subgradient':
    # initialising all vars
    self.w = np.zeros(D)
    prev_w = self.w.copy()
    self.w0 = np.zeros(1)
    prev_w0 = self.w0.copy()
    func_err = self.compute_primal_objective(X, y)

    graph_data = []
    iter_count = 0
    status = 1
    start_time = timer.time()
    for iter_n in range(1, max_iter+1):
        # computing one more step of gradient
        iter_count += 1
        graph_data.append(self.compute_primal_objective(X, y))
        eta_coef = alpha/(iter_n**beta)

        self.w -= eta_coef * prev_w
        for n in range(N):
            if 1.0 - y[n]*(X[n] @ prev_w + prev_w0) > 0.0:
                self.w += eta_coef * self.C * y[n] * X[n]  # doubled -
                self.w0 += eta_coef * self.C * y[n]

        # checking stop criterion
        if stop_criterion == 'argument':
            if norm(np.hstack((self.w, self.w0)) -
                    np.hstack((prev_w, prev_w0))) <= tol:
                status = 0
                break

        elif stop_criterion == 'objective':
            new_err = self.compute_primal_objective(X, y)
            if abs(func_err-new_err) <= tol:
                status = 0
                break

            func_err = new_err
        else:
            raise ValueError('Unknown stop criterion')

        prev_w = self.w.copy()
        prev_w0 = self.w0.copy()
    fit_time = timer.time()-start_time
    self.w = self.w.reshape((D, 1))
    return {'objective_curve': graph_data,
            'status': status,
            'iteration': iter_count,
            'time': fit_time}
```


Листинг 2.3: Реализация метода субградиентного спуска для прямой задачи

```python
elif self.method == 'stoch_subgradient':
    self.w = np.zeros(D)
    prev_w = self.w.copy()
    self.w0 = np.random.rand(1)
    prev_w0 = self.w0.copy()

    func_err = self.compute_primal_objective(X, y)
    graph_data = []
    iter_count = 0
    status = 1
    start_time = timer.time()
    for iter_n in range(1, max_iter+1):
        # computing one more step of gradient
        eta_coef = alpha/(iter_n**beta)
        iter_count += 1
        graph_data.append(self.compute_primal_objective(X, y))
        # preparing batch indices
        batch_indices = np.arange(N)
        np.random.shuffle(batch_indices)
        batch_indices = batch_indices[:batch_size]

        self.w -= eta_coef * prev_w
        for n in batch_indices:
            if 1.0-y[n]*(X[n] @ prev_w + prev_w0) >= 0.0:
                self.w += eta_coef * self.C * y[n] * X[n]
                self.w0 += eta_coef * self.C * y[n]

        # checking stop criterion
        if stop_criterion == 'argument':
            if norm(np.hstack((self.w, self.w0)) -
                    np.hstack((prev_w, prev_w0))) <= tol:
                status = 0
                break
        elif stop_criterion == 'objective':
            func_err = (1-lamb)*func_err + \
                       lamb*self.compute_primal_objective(X, y)
            if func_err <= tol:
                status = 0
                break
        else:
            raise ValueError('Unknown stop criterion')
        prev_w = self.w.copy()
        prev_w0 = self.w0.copy()

    fit_time = timer.time() - start_time
    self.w = self.w.reshape((D, 1))
    return {'objective_curve': graph_data,
            'status': status,
            'iteration': iter_count,
            'time': fit_time}
```


Листинг 2.4: Реализация метода стохастического субградиентного спуска для прямой

2.5 Метод из liblinear

```
elif self.method == 'liblinear':
    self.inner_svm = sk_svm.LinearSVC(C=self.C, tol=tol,
                                      verbose=verbose,
                                      max_iter=max_iter)
    start_time = timer.time()
    self.inner_svm.fit(X, y)
    fit_time = timer.time() - start_time
    self.w = self.inner_svm.coef_.T
    self.w0 = self.inner_svm.intercept_
    return {'time': fit_time}
```

Листинг 2.5: Реализация метода решения прямой задачи из liblinear

2.6 Метод из libsvm

```
elif self.method == 'libsvm':
    self.A = np.zeros((X.shape[0], 1))
    self.inner_svm = sk_svm.SVC(self.C, self.kernel, gamma=self.gamma,
                                tol=tol, verbose=verbose,
                                max_iter=max_iter, probability=True)
    start_time = timer.time()
    self.inner_svm.fit(X, y)
    fit_time = timer.time()-start_time
    self.sv = list(zip(self.inner_svm.support_vectors_,
                       y[self.inner_svm.support_]))
    self.A[self.inner_svm.support_] = self.inner_svm.dual_coef_.T
    self.X_train = X.copy()
    self.y_train = y.copy()
    return {'time': fit_time}
```

Листинг 2.6: Реализация метода решения прямой/обратной задачи из libsvm

3 Исследовательская часть

3.1 Генерация выборки для дальнейших тестов

Для начала была сгенерирована линейно неразделимая 100-мерная выборка из 600 объектов, любой из которых по каждому измерению распределён по одному из двух нормальных распределений, смещённых относительно друг друга. Когда требовалась выборка размерности N < 100, то брались лишь первые N измерений.

```
X_fst = 7*np.random.randn(300, 100)
signs = 2*(np.random.randint(0, 2, 100)-0.5)
X_sec = 7*np.random.randn(300, 100)+((np.random.randn(100)+10)*signs)
X_train = np.vstack((X_fst, X_sec))
y_train = np.hstack((np.ones(300), np.zeros(300)-1))
X_train, y_train = skutils.shuffle(X_train, y_train)
```

Листинг 3.1: Код генерации выборки

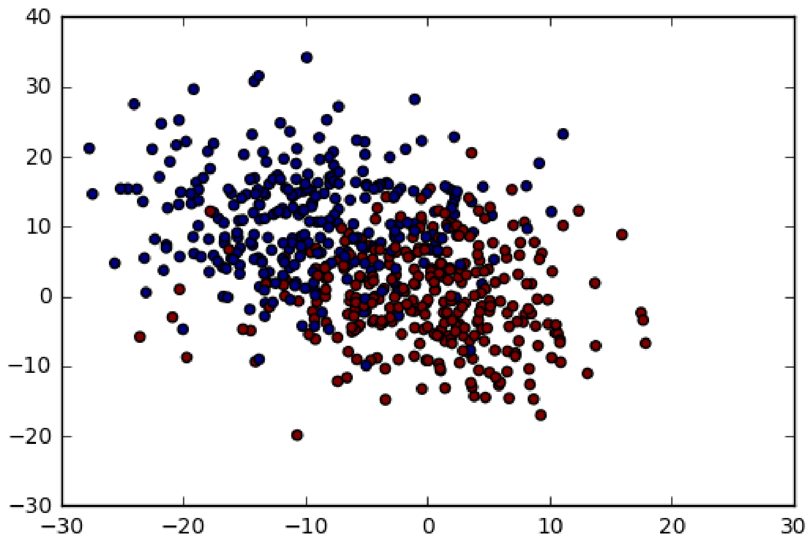

Рис. 1: Отображение выборки по первым двум измерениям

3.2 Исследование зависимости времени работы метода от размерности данных на линейном ядре

В данном пункте мы будем замерять время настройки метода в зависимости либо от изменения числа объектов в выборке, либо от размерности данной выборки. Наши подопытные, подпадающие под критерии: метод liblinear, метод решения прямой задачи cvxopt (и обратной с линейным ядром), а также субградиентный и стохастический субградиентные спуски. Графики приведены ниже. Считаем что в D-мерной выборке N объектов.

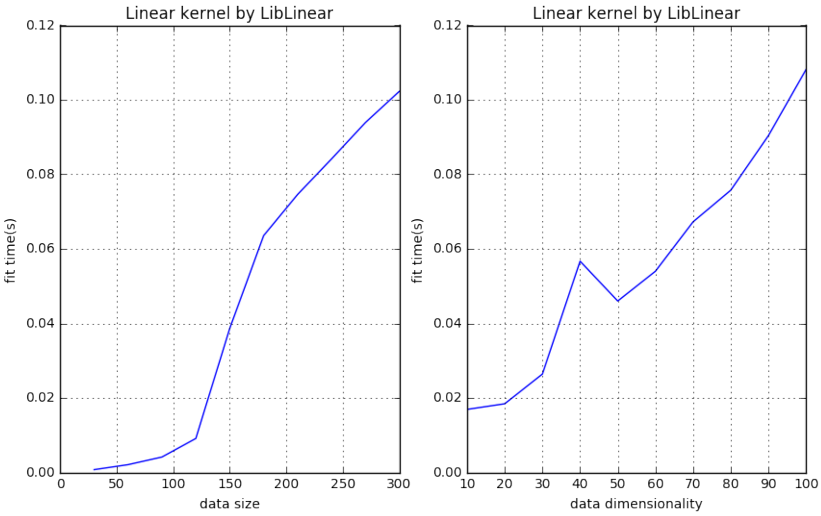

Рис. 2: Графики для метода liblinear

Метод liblinear показывает сложность $O(ND^2)$, этот вывод можно сделать, посмотрев на график.

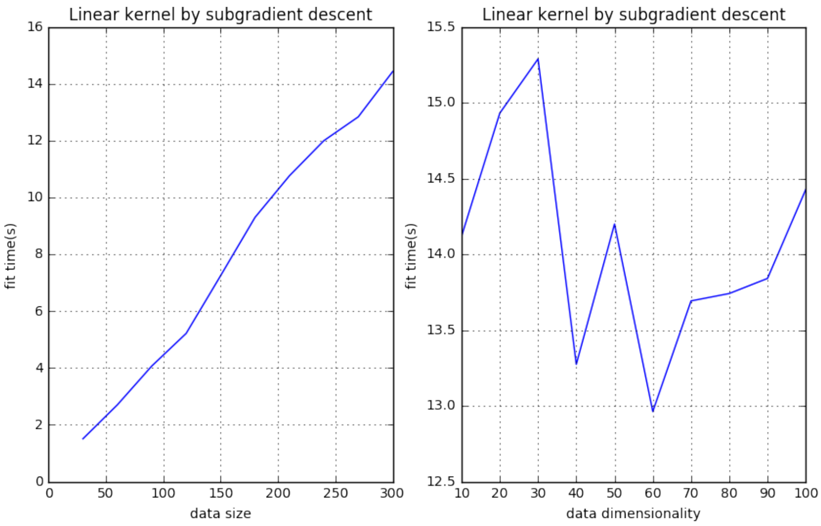

Рис. 3: Графики для субградиентного спуска

Метод субградиентного спуска показывает сложность $O(ND^2)$, этот вывод можно сделать, посмотрев на график (касаемо правого графика - спорный вопрос, но я специально строил ещё один график на выборке с гигантской размерностью, и там это было явно видно)

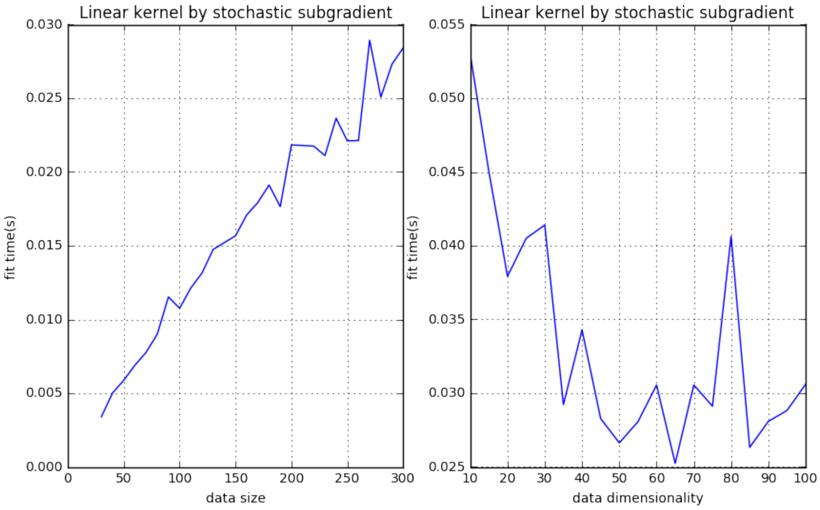

Рис. 4: Графики для стохастического субградиентного спуска

Аналогично предыдущему. Единственное, что хотелось бы отметить - график получается более неравномерным, чем у просто субградиентного спуска. Это связано со случайным выбором признаков в подвыборку для итерирования.

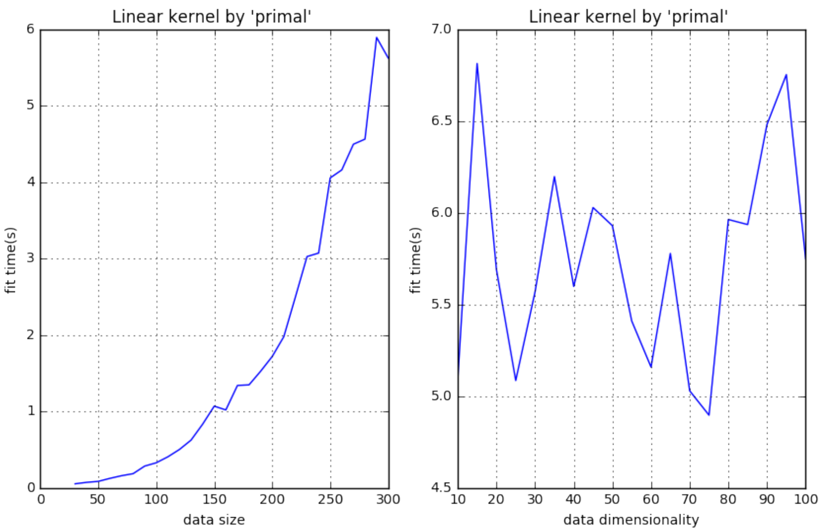

Рис. 5: Графики для метода primal

Метод имеет сложность $O(N^2D)$. Правый график, опять же, пришлось достраивать вручную на выборке большей размерности. Изначально выборку большей размерности не смог взять по причине большого количества необходимых расчётов, особенно при субградиентном спуске.

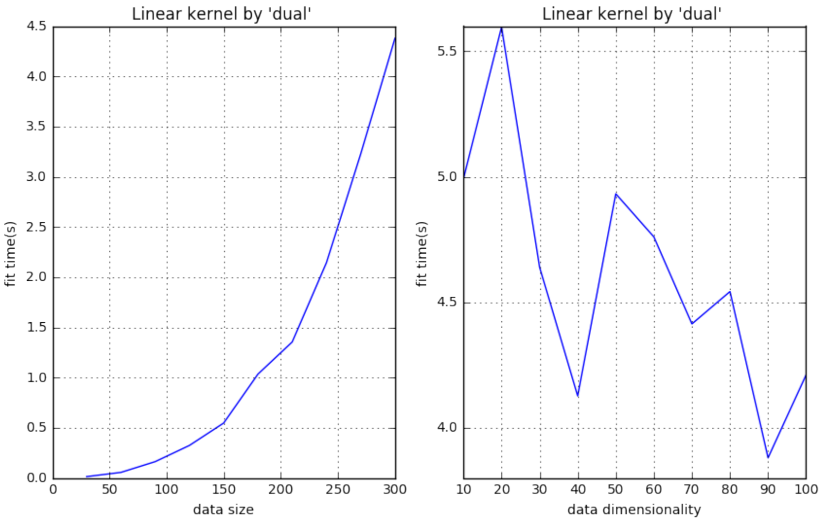

Рис. 6: Графики для метода dual с линейным ядром

Аналогичная сложность $O(N^2D)$. Чем лучше - по размерности график смещён вправо - можно использовать для данных небольшой размерности, чтобы получить выигрыш в скорости.

3.2.1 Сравнение значений целевой функции

```python
my_svm = svm.SVM(C=1.0, method='liblinear')
my_svm.fit(X_train, y_train, verbose=False)
print("LibLinear:", my_svm.compute_primal_objective(X_train, y_train))

my_svm = svm.SVM(C=1.0, method='primal')
my_svm.fit(X_train, y_train, verbose=False)
print("Primal:", my_svm.compute_primal_objective(X_train, y_train))

my_svm = svm.SVM(C=1.0, method='subgradient')
my_svm.fit(X_train, y_train, tol=1e-5, max_iter = 1000, verbose=False, \
           stop_criterion="objective", alpha=0.3, beta=1.0)
print("Subgradient:", my_svm.compute_primal_objective(X_train, y_train))

my_svm = svm.SVM(C=0.5, method='stoch_subgradient')
my_svm.fit(X_train, y_train, tol=1e-5, max_iter = 10000, verbose=False, \
           stop_criterion="objective", alpha=0.5, beta=1.0)
print("Stochastic subgradient:",my_svm.compute_primal_objective(X_train,\
                                                                 y_train))

my_svm = svm.SVM(C=0.5, method='dual', kernel='linear')
my_svm.fit(X_train, y_train, verbose=False)
print("Dual linear:", my_svm.compute_dual_objective(X_train, y_train))
```

Листинг 3.2: Вывод значений целевой функции

```
LibLinear: [ 1.75562456]
Primal: 0.841667944618
Subgradient: [ 642591.70264089]
Stochastic subgradient: [ 17.2000971]
Dual linear: 0.00125797236425
```

Рис. 7: Вывод значений целевой функции для методов с линейным ядром

По этим значениям можно сделать вывод, что лучше всего решают задачу методы primal и dual с линейным ядром (так как они её решают в чистом виде) Так же неплохо справляются и остальные методы. Отдельно хотелось бы выделить метод субградиентного спуска, который имеет необычно большое значение целевой функции. Однако, после суток потраченного времени выяснилось, что это связано с размерностью данных нашей выборки, что субградиент при этом прекрасно себя чувствует и работает ничуть не хуже остальных методов.

3.3 Исследование зависимости времени работы метода от размерности данных на ядре RBF

Теперь мы сделаем аналогичное исследование относительно методов libsvm и dual с ядром RBF. Графики будем строить аналогично.

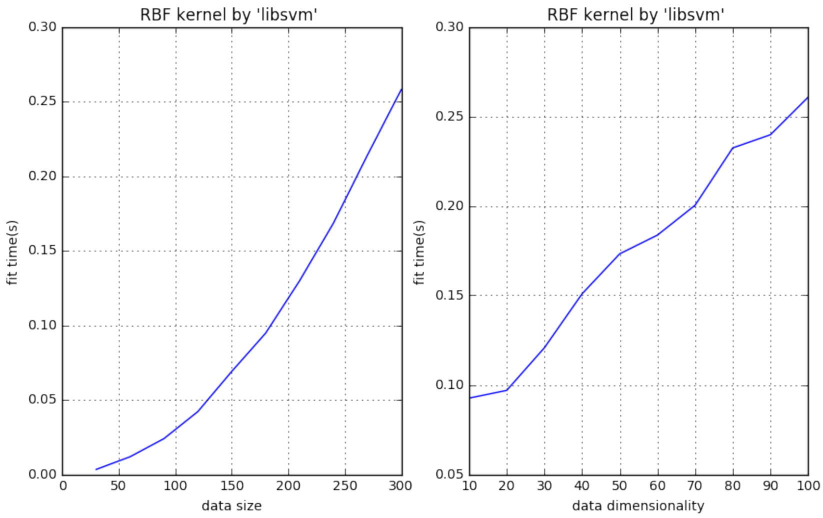

Рис. 8: Графики для метода libsvm

По графикам становится очевидна сложность метода - $O(N^2D)$

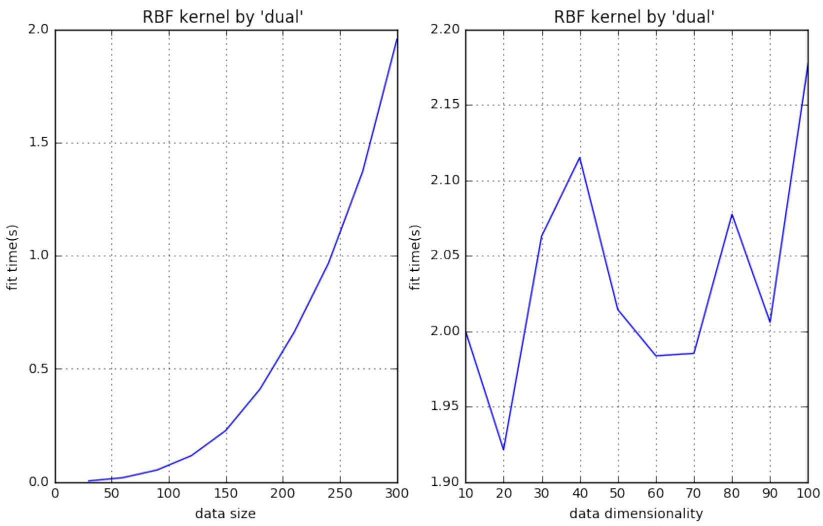

Рис. 9: Графики для метода dual с RBF ядром

Сложность данного метода - $O(N^2D^2)$

```python
my_svm = svm.SVM(C=0.5, method='libsvm', gamma=1.0)
my_svm.fit(X_train, y_train, verbose=False)
print("Lib SVM:", my_svm.compute_dual_objective(X_train, y_train))

my_svm = svm.SVM(C=0.5, method='dual', kernel='rbf', gamma=1.0)
my_svm.fit(X_train, y_train, verbose=False)
print("Dual RBF:", my_svm.compute_dual_objective(X_train, y_train))
```

Листинг 3.3: Вывод значений целевой функции

```
Lib SVM: 0.0
Dual RBF: 299.99996162
```

Рис. 10: Вывод значений целевой функции для методов с RBF ядром

По этим значениям можно сделать вывод, что хоть dual и решает задачу (и на практике следует использовать именно его),но libsvm всё же как-то удаётся получить нулевой функционал. Подобные результаты были получены на различных выборках, так что это не случайность, а правило. Возможно тут закралась какая-то ошибка, но я не знаю, где.

3.4 Поиск оптимальных значений C и gamma

Найдём оптимальные значения параметров C и gamma с помощью кросс-валидации. Сначала сделаем отдельно исследования для нашей обычной выборки, что нелинейно разделима, а затем запустим на линейно разделимом варианте.

```
from sklearn.model_selection import KFold
kfold_obj = KFold(n_splits=5, shuffle=True)

C_pg = []
for i in range(-2, 3):
    C_pg.extend([10**i, 2*10**i, 5*10**i])
gamma_pg = C_pg

errors = []
for curr_C in C_pg:
    for curr_gamma in gamma_pg:
        my_svm = svm.SVM(C=curr_C, method='dual', kernel='rbf',\
                         gamma=curr_gamma)
        error = []
        for train_inds, test_inds in kfold_obj.split(X_train):
            my_svm.fit(X_train[train_inds, :5], y_train[train_inds],\
                       verbose=False)
            y_pred = my_svm.predict(X_train[test_inds, :5], \
                                    return_classes=True)
            error.append(get_accuracy(y_pred,\
                                      y_train[test_inds]))
        errors.append({'C': curr_C,
                       'gamma': curr_gamma,
                       'score': min(error)})
```

Листинг 3.4: Поиск оптимальных C и gamma по логарифмической шкале

Сопоставляя эти два графика можно сделать вывод, что максимальная точность достигается при очень маленьком gamma и, возможно, в нескольких C. Поэтому стоит брать gamma поменьше, а затем уже искать самое оптимальное значение C. Линейно разделимая выборка с некоторого момента делится с абсолютной точностью, а значит, C и gamma оптимальны уже в некотором интервале (тут уже идёт задача SVM, и просто точности становится недостаточно)

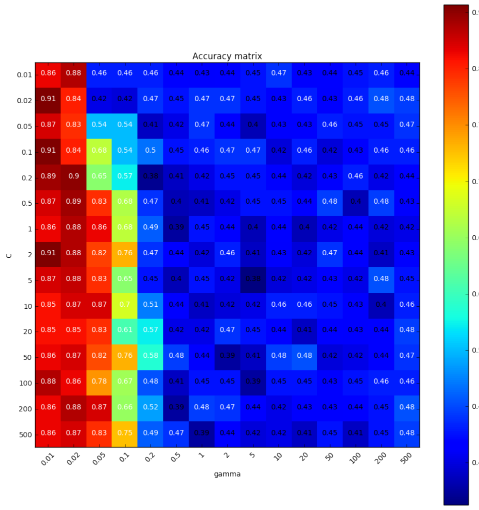

Рис. 11: Матрица точности от C и gamma(линейно неразделима)

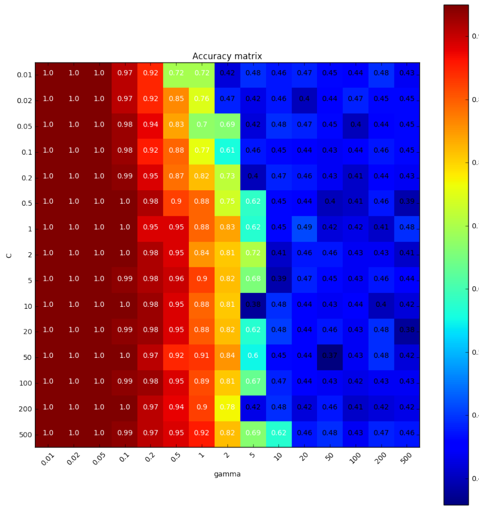

Рис. 12: Матрица точности от C и gamma(линейно разделима)

3.5 Сравнение стратегий выбора шага субградиентного спуска

Воспользуемся предыдущей стратегией и построим матрицу точности от коэффициентов alpha и beta. Код практически тот же самый.

По этому графику можно сделать вывод, что alpha-beta дают лучший результат в некоторой пропорции, и сразу несколько пар значений отвечают лучшей точности (0.05-0.0, 1 - 0.001, 0.001 - 0.001 и т.д.) Поэтому нужно придерживаться определённого соотношения и не брать один коэффициент сильно больше другого.

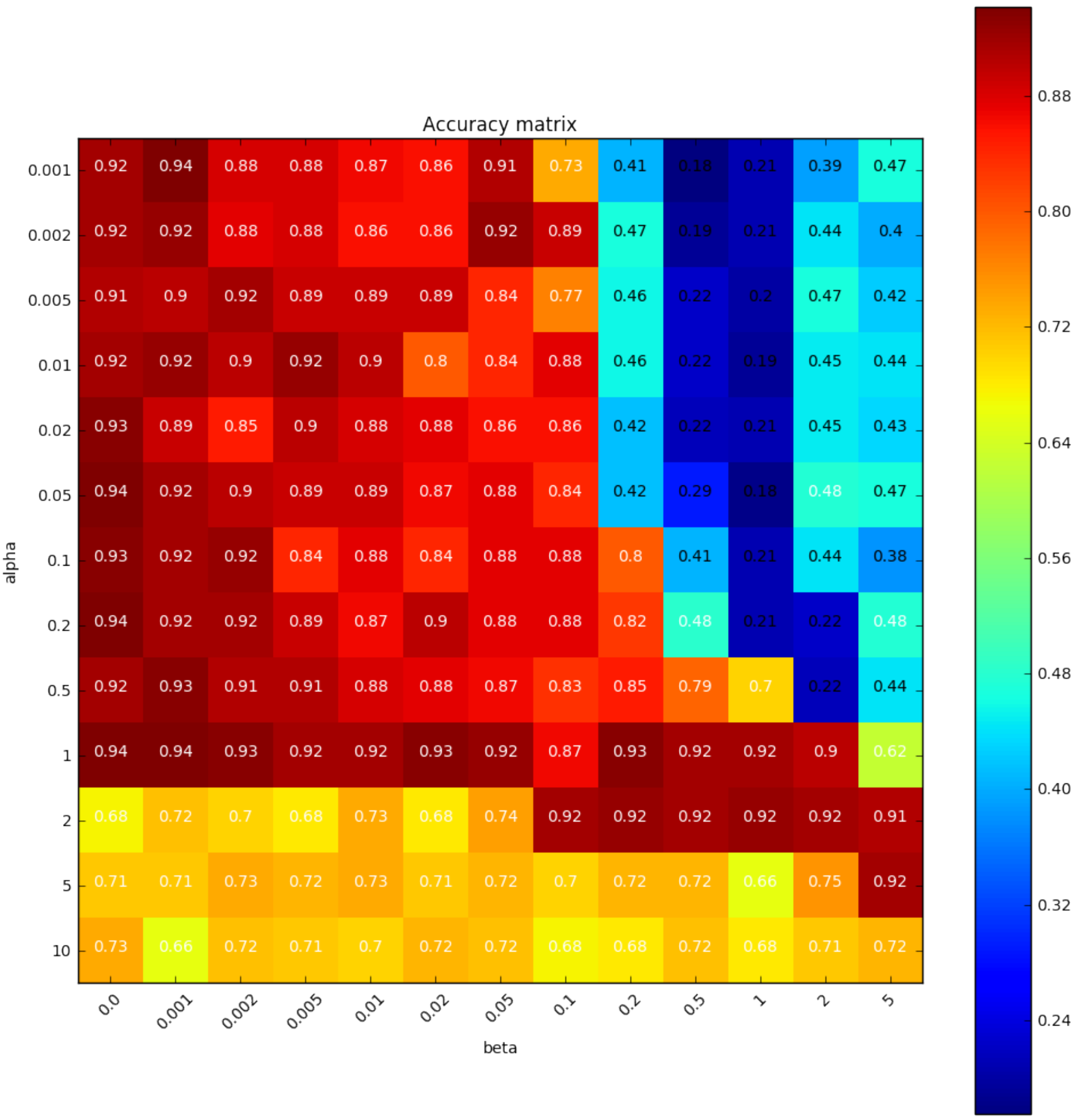

Рис. 13: Матрица точности от alpha и beta (субградиентный спуск)

3.6 Исследование влияния размера подвыборки на скорость сходимости и точность решения метода стохастического субградиентного спуска

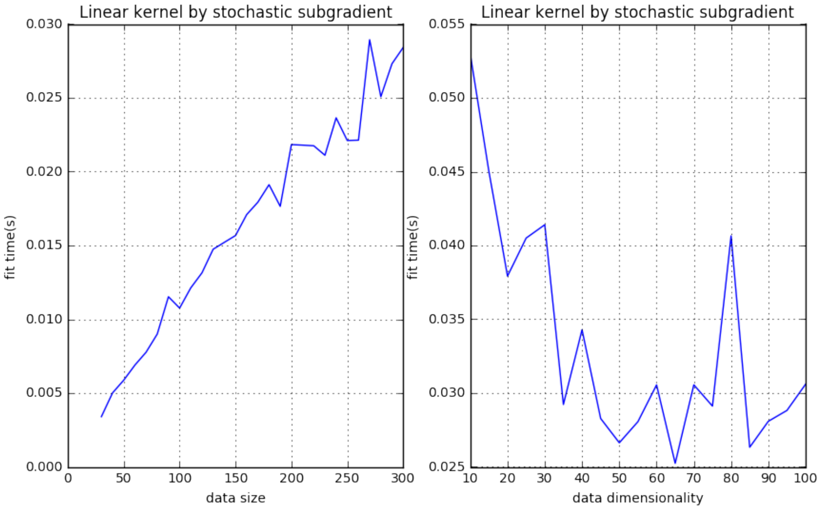

Рис. 14: График влияния размера выборки на скорость сходимости

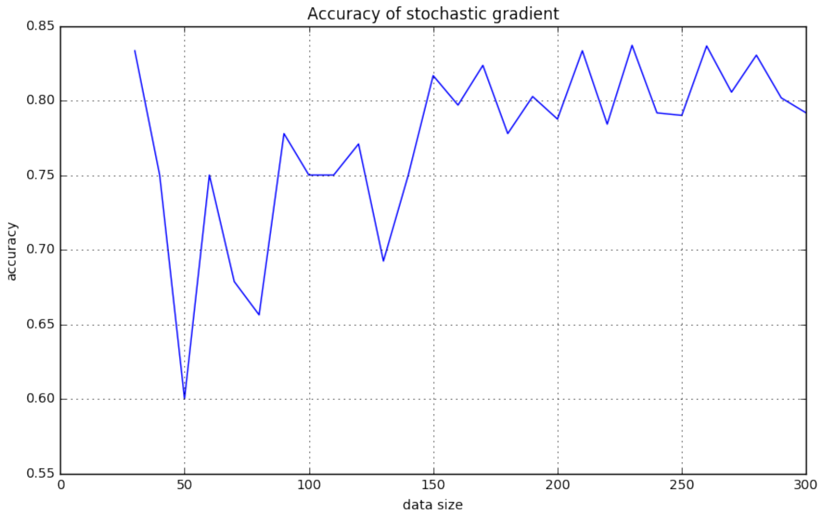

Рис. 15: График влияния размера выборки на точность решения

Вывод: выборка должна быть среднего размера, так как либо на ней нельзя будет добиться хорошего качества из-за недостатка объектов, либо модель будет обучаться слишком долгое время, либо просто переобучится.

3.7 Визуализация двумерной выборки для линейного и RBF ядра

Ниже приведён код визуализации разделения выборки. Опорные вектора отмечены звёздочками, классы соответственно синий и красный.

```
def visualize(X, y, alg_svm, show_vectors=False):
    N, D = X.shape

    if D != 2:
        raise ValueError('''Can't visualize multidimensional data''')

    # following code was inspired by SVM visualiser example:
    # http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html

    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    y_pred = alg_svm.predict(np.c_[xx.ravel(), yy.ravel()], True)
    y_pred = y_pred.reshape(xx.shape)
    plt.contourf(xx, yy, y_pred, cmap=plt.cm.coolwarm, alpha=0.8)

    if show_vectors is True:
        supp_inds = ~(alg_svm.A == 0).ravel()
        colors = []
        for y_elem in y[supp_inds]:
            if y_elem == 1:
                colors.append('c')
            else:
                colors.append('y')
        area = np.pi * (np.ones(len(colors))*4.5)**2
        plt.scatter(X[~supp_inds, 0], X[~supp_inds, 1], c=y[~supp_inds])
        plt.scatter(X[supp_inds, 0], X[supp_inds, 1], s=area, c=y[supp_inds],
                    marker="*", alpha=1.0)
    else:
        plt.scatter(X[:, 0], X[:, 1], c=y)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.xticks(())
    plt.yticks(())

    plt.show()
```

Листинг 3.5: Код функции визуализации

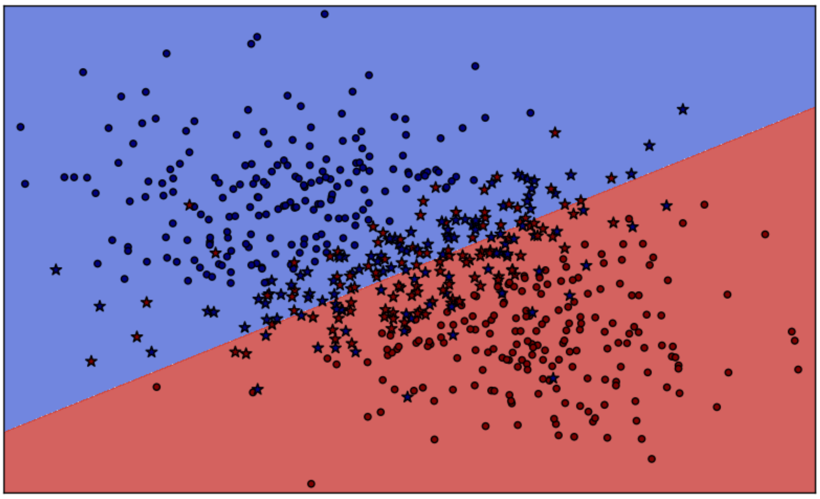

Рис. 16: Визуализация разделения выборки с помощью метода с линейным ядром

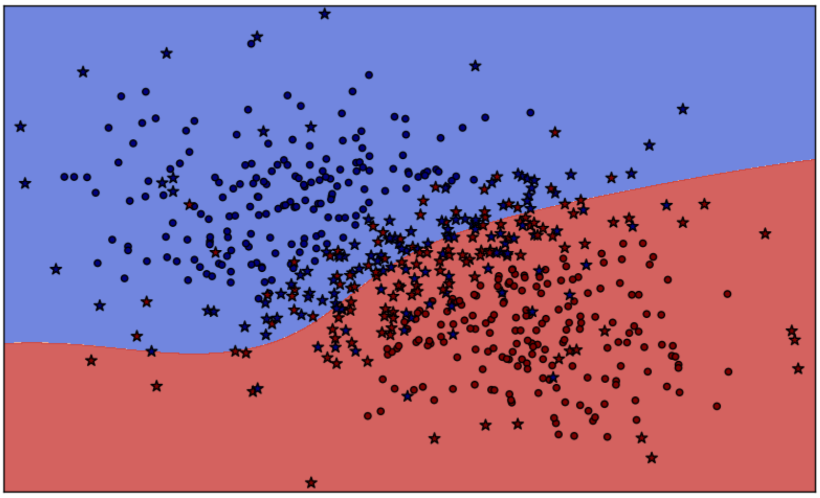

Рис. 17: Визуализация разделения выборки с помощью метода с RBF ядром

4 Выводы

После исследования соотношения эффективности реализаций различных задач можно заключить, что:

- Стохастический субградиентный бустинг тратит меньше ресурсов, чем обычный субградиентный
- Задача SVM настолько разрешима, что её можно просто посчитать. Задача квадратичного программирования + cvxopt в этом помогут.
- Иногда стоит обращать внимание на сторонние библиотеки - так liblinear показал себя относительно неплохо
- Визуализация даёт большую обратную связь, чем большинство других методов отладки

Список литературы

[1] Воронцов К.В. LaTeX2e в примерах