Отчет по третьему лабораторному заданию «Метод опорных векторов»

Алескин Александр

21 ноября 2015 г.

Содержание

1 Введение

В рамках данного задания были выполнены все требуемые пункты. Некоторые из них были изменены, так как некоторые вопросы были не совсем корректно поставлены. Был создан класс со всем необходимым набором функций для решения поставленных задач. В целях краткости были предоставлены только основные данные.

В данном отчете:

- предоставлено описание созданного класса
- описаны особенности реализации функций класса
- изложены главные пункты исследовательской работы

2 Описание Класса

Для удобства в классе хранятся все необходимые переменные:

- X_ - матрица признаков обучаемой выборки
- y_ - ответы обучаемой выборки
- C_ - параметр минимизации
- A_ - матрица коэффициентов двойственной задачи
- w_ - вектор весов
- b_ - смещение
- ksi_ - коэффициенты(переменные) прямой задачи
- sup_vec_ - опорные вектора
- gamma_ - параметр rbf ядра
- obj_curve_ - значение целевой функции на последней итерации(или вектор при проверки в стохастическом варианте)

Переменные инициализируется при вызове функции fit, и обновляются при каждом вызове метода решения задачи SVM. В классе имеются следующие функции:

- fit(X, y, C) — функция инициализации переменных класса. X - матрица признаков, y - ответы, C - константа
- svm_qp_primal_solver(tol= 1e-6, max_iter= 100, verbose= False)
- svm_qp_dual_solver(tol=1e-6, max_iter=100, verbose=False, gamma=0)
- svm_liblinear_solver(tol= 1e-6, max_iter= 100, verbose= False)
- svm_libsvm_solver(tol = 1e-6, max_iter = 100,verbose = False, gamma = 0)

- compute_w()
- compute_support_vectors()
- predict(X) − классифицирует объекты по матрицы признаков X
- visualize()
- compute_primal_objective()
- compute_dual_objective()
- svm_subgradient_solver(sw_start, w0 = 0, step = _const_, a = 0.01, b = 0.1, max_iter= 1000 , verbose= False, tol= 1e-6, rule = 'f', stochastic = False, func = 0, norma = 0, sub = 0.5)[1]

3 Особенности реализации функций

Метод внутренней точки сводится к задаче, решаемой функцией cvxopt.solvers.qp:

$$\begin{cases} q^T x + \frac{1}{2} x^T P x \to \min_x \\ Gx \le h \\ Ax = b \end{cases}$$

Для прямой задачи нужно совершить следующий переход:

$$x_{(D+1+N)} = (w, w_0, \xi)$$

$$P_{(D+1+N;D+1+N)} = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}, q_{(D+1+N;1)} = \begin{bmatrix} 0 \\ C \end{bmatrix},$$

$$G_{(2*N;D+1+N)} = \begin{bmatrix} 0 & 0 & -I \\ -y * X & -y & -I \end{bmatrix}, h_{(2*N;1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

где D — размерность пространства, а N — число элементов в выборке, y — целевая функция, X — матрица признаков.

Для обратной задачи нужно преобразовать переменные следующий образом:

$$P_{(N;N)} = \left(y_i y_j K(x_i, x_j)\right), q_{(N;1)} = \left(-1\right),$$

$$G_{(2*N;N)} = \begin{bmatrix} I \\ -I \end{bmatrix}, h_{(2*N;1)} = \begin{bmatrix} C * 1 \\ 0 \end{bmatrix},$$

$$A_{(N,1)} = y^T, b = 0,$$

где $K(x_i, x_j)$ — функция ядра.

Для реализации метода субградиентного спуска, необходимо вычислить направление итерационного шага. Выпишем минимизируемую функции из прямой задачи SVM без ограничений:

$$f(w, w_0) = \frac{1}{2}||w||^2 + C \sum_{i=1}^{N} max\{0, 1 - y_n(w^T x_n + w_0)\}$$

[1]Более подробное описание этой функции можно найти в ноутбуке с исследованиями

Вычислим градиент этой функции по переменным w и w_0:

$$\bigtriangledown f = (w, 0) - C \sum_{i=1}^{N} \mathbf{I}[y_n(w^T x_n + w_0) < 1] y_n(x_n, 1),$$

где $\mathbf{I}$ — функция индикатор. Полученное значение $\bigtriangledown f$ и будет направление итерационного шага. Стоит отметить, что направление выбирается не только по w, но и по w_0. Длину же будем выбирать экспериментально.

Кратко опишем принцип работы следующих функций класса:

- svm_liblinear_solver() — вызывает функцию sklearn.svm.SVC()
- svm_libsvm_solver() — вызывает функцию sklearn.svm.LinearSVC()
- visualize() — для визуализации работы алгоритма функция вычисляет значение целевую функцию в точках на плоскости, содержащей выборку, и генерирует изображения при помощи matplotlib.pyplot.contourf()
- compute_w() — переходит от двойственных переменных к прямым по следующей формуле $w = X^T \times B$, где $B_{(1,N)} = (a_i y_i)$
- compute_support_vectors() — определяет является ли i элемент опорным вектором по следующим формулам:
  - в прямой задаче — $y_i(w^T x_i + w_0) = 1 - \xi_i$
  - в двойственной — $A_i > 0$.

4 Пункт 1 исследований

Задача: выяснить временные характеристики методов обучения линейного SVM, а так же их точность.

Скорость обучения алгоритмов зависит как от размерности пространства, так и от количества элементов. Будем замерять время выполнения алгоритмов, а так же их точность(всего 6 методов, в том числе стохастический метод) при нескольких значениях размерности пространства и количества элементов. Для каждого значения будем генерировать выборки из мультинормального распределений с такими параметрами, что разница между математическими ожиданиями классов равна дисперсии классов.

На рисунках 1 - 3 приведены результаты исследования в при постоянном значении одного из аргументов. Замирение времени в секундах, а не в итерациях лучше, так это ключевой параметр алгоритма. А прямой зависимости времени от количества итераций нет: для субградиентного метода нужно больше итераций, но при этом каждая итерации выполняется быстрее, чем у любого другого алгоритма(из представленных).

Как видно из рисунков, реализованный субградиентый спуск работает на пол порядка медленнее функций scikit learn. Несмотря на то, что стохастический метод должен работать быстрее, он работает медленнее. Можно выделить основные 2 причины этому. Во-первых, на Python требуется в рамках субградиентного спуска требуется больше команд. На C++ эта добавка кода была бы незначительной, и выигрыш по скорости был бы заметнее. Во-вторых, при стохастическом спуске лучше в качестве критерия останова брать норму аргумента(но

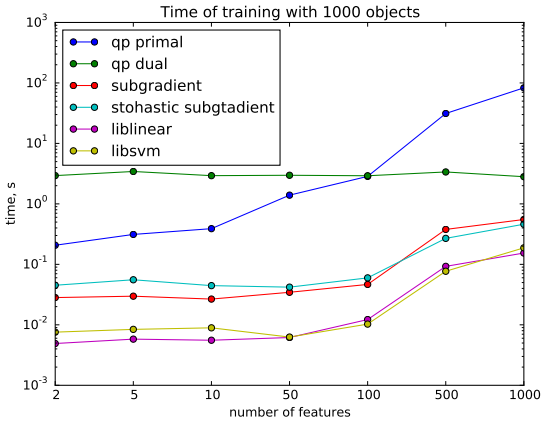

Рис. 1: Время работы при 1000 объектах

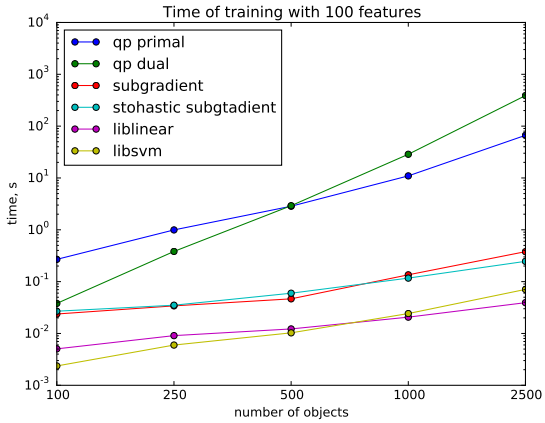

Рис. 2: Время работы при 100 признаках

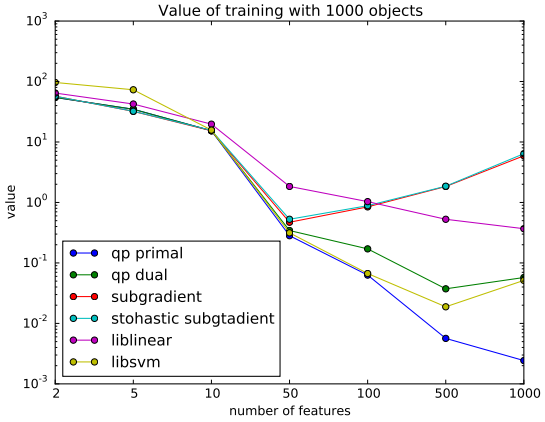

Рис. 3: Значение целевой функции при 1000 объектах

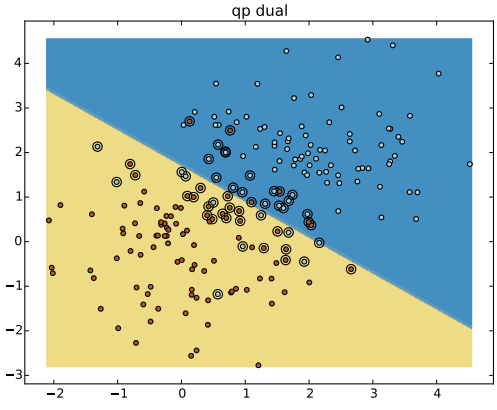

Рис. 4: Метод внутренней точки двойственная задача

на питоне опять же не получилось значительного ускорения), тогда не нужно вычислять все "опорные вектора"на каждом шаге.

Хотя функция cvxopt.solvers.qp() решает задачу точнее всех, использовать ее на больших данных не лучший вариант, так как время работы зависит экспоненциально от размерности матрицы P (3). Вычисление ядра в двойственной задачи функции svm_qp_dual_solver() в сравнении по скорости с работой с функцией qp() весьма не значителен, что демонстрируется на рисунке 2. На рис.3 уменьшение оптимального значения функционала с повышением размерности объясняется тем, что выборка становится если не линейно разделяемой, то почти линейно разделяемой [2]. Плохая сходимость субградиентного метода при большом количестве признаков(и не значительном количестве объектов) является следствие независимости констант a и b от размерности и выборки.

на рис.4 приведен пример работы SVM алгоритма в двумерном признаковом пространстве.

[2] имеется ввиду можно провести гиперплоскость

5 Пункт 2 исследований

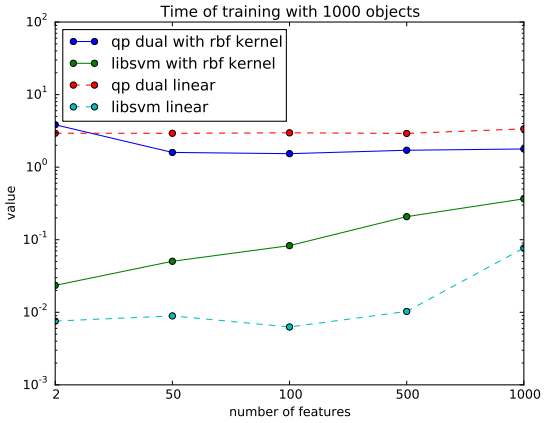

Рис. 5: Время работы алгоритмов, с

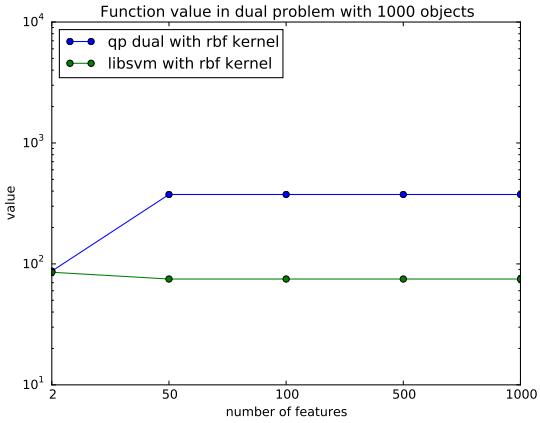

Рис. 6: Значение целевой функции при 1000 объектах

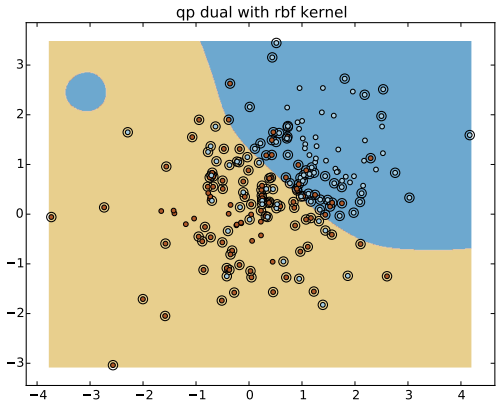

Рис. 7: Метод внутренней точки с rbf ядром

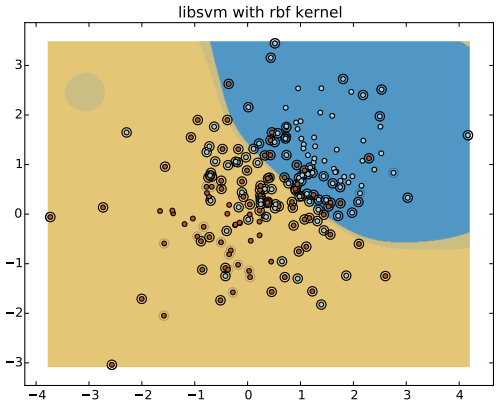

Рис. 8: Метод библиотеки libsvm с rbf ядром

Задача: Провести те же исследования для случая SVM с RBF ядром для тех методов, где возможен ядровой переход.

Проведем эксперименты по той же схеме. На графиках 5-6 приведены результаты исследования только при 1000 элементах. Несмотря на то, что время работы метода libsvm увеличилось на порядок, метод двойственной задачи ускорился в 2 раза. Вероятная причина это более удобное перераспределение выборки. Так как в двойственной задачи мы максимизируем функционал, но метод внутренней точки лучше справился с этой задачи. На рис.7-8 изображены решения методов. Для наглядности на втором рисунке неявно очерчены контуры решения метода svm_qp_qual_solver().

6 Пункт 3 исследований

Задача: Реализовать процедуру поиска оптимального значения параметра C и ширины RBF ядра. Исследовать зависимость ошибки на валидационной выборке от значений этих параметров.

Разобьем выборку на 4 фолда и по ним будем считать оптимальные значения C и γ с точностью до 0.1 на интервалах $[0.1, 1]$ и $[0.1, 2]$ соответственно. Генерируемые простые и сложные выборки приведены на рис.11-12. В простой выборке разделяемая границы классов проходит по окружности. В сложной выборке центры мультинормальных распределений находятся не далеко друг от друга.

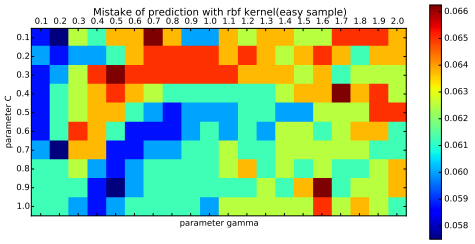

Рис. 9: Функция ошибки при простой выборке

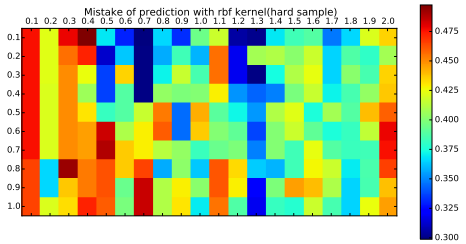

Рис. 10: Функция ошибки при сложной выборке

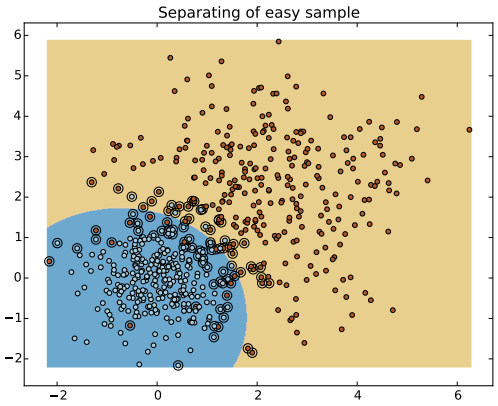

Рис. 11: Простая выборка

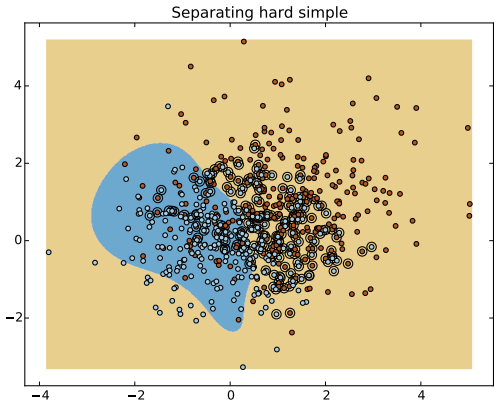

Рис. 12: Сложно разделимая выборка

Анализируя результаты эксперимента с хорошо разделяемой выборкой рис.9[3], можно выделить 2 области на которых достигаются наилучшие значения:$\gamma \leq 0.2$

- при $\gamma \leq 0.2$ и $C \leq 0.5$
- при $0.5 \leq \gamma \leq 1$ и $C \geq 0.5$

[3]по оси y параметр C, по оси x ширина ядра γ

Хотя градация ошибки при простой выборки не значительна в сравнении со сложно разделяемой выборкой рис10. Здесь наилучшая область при $0.5 \leq \gamma \leq 1$ и $C \leq 0.2$. Выявленную закономерность сложно интерпретировать.

7 Пункт 4 исследований

Задача: 4. Сравнить (по скорости сходимости и точности решения) несколько стратегий выбора шага ? t в методе субградиентого спуска: $a, \frac{a}{it}, \frac{a}{it^b}$.

Будем исследовать на сходимость при $10^{-5} \leq \gamma \leq 10$ и $0 \leq \gamma \leq 2$. Заметим, что при $b = 0$ имеем первый случай, а при $b = 1$ случай 2. Поскольку субградиентный метод требует больше итераций, поставим, максимальное число итераций 10000. Результаты эксперимента на рис.13(при $a = 10$ и малых значениях b функция выдает Nan и досрочно выходит). Наилучшее сходимость при $a = 0.01$ и $b = 0.1$. Как видим, параметр b должен быть достаточно малым, чтобы алгоритм хорошо сходился, то есть слабая зависимость от итераций. Параметр a показывает, что длина итерационного шага должна быть не слишком большой, иначе мы проскакиваем оптимальное значение.

В среднем в поставленных опытах(при 200, 1000, 2000 объектах) достаточно 300 итераций, чтобы метод сходился. За точное значение функции бралось значение, посчитанное функцией svm_qp_primal_solver() с большой точностью.

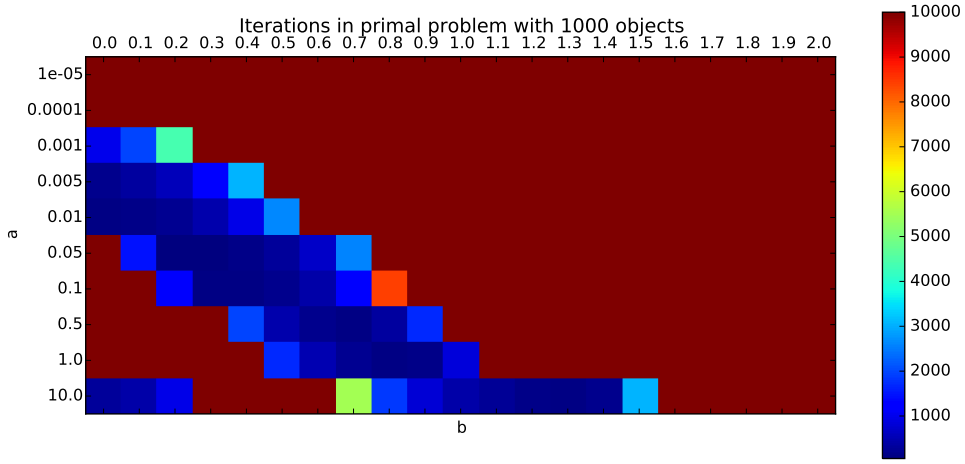

Рис. 13: Количество итераций в зависимости от параметров a и b

8 Пункт 5 исследований

Задача: Исследовать, как размер подвыборки, по которой считается субградиент, в методе стохастического субградиентного спуска влияет на скорость сходимости метода и на точность решения.

Как и в предыдущем пункте считаем, что если метод остановился, то он достиг точного значения. Замерим количество итераций, необходимое для сходимости. По скольку, величина

здесь вероятностная, то будем проводить серию экспериментов и усреднять результат. На рис.14 приведены результаты эксперимента. Как видно, количество итераций обратно пропорционально доли выборки (если брать 50% от решающей выборки, то необходимо почти в 2 раза больше итераций).

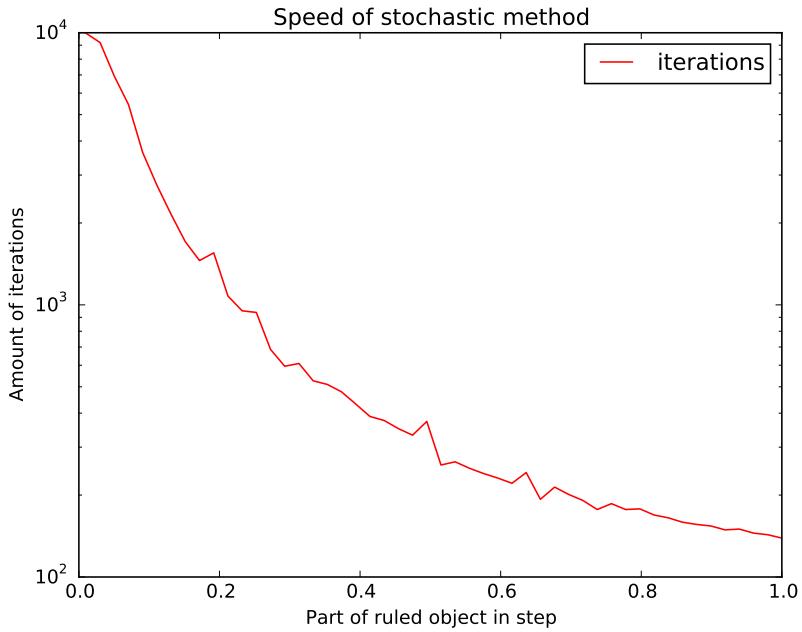

Рис. 14: Количество итераций в зависимости от размера подвыборки

9 Заключение

В рамках исследовательской части были получены следующие выводы:

- При больших значениях cvxopt.solvers.qp() очень медленно работает
- Для сходимости субградиентного метода нужно не менее 200 итераций
- Сходимость субградиетного метода ухудшается прямо пропорционально взятой доли выборки(при доли в 50% нужно в 2 раза больше итераций)
- Функция sklearn.svm.SVC() при хорошей скорости работы дает вполне точное решение.
- Оптимальные значения C и γ для нелийного методя с rbf ядром зависят от выборке

  В работе не была исследована скорость сходимости субградиентного метода от размерности пространства.