Московский физико-технический
институт(Государственный университет)

Факультет управления и прикладной
математики

Сравнительный анализ линейных классификаторов

**Выпускная квалификационная работа на степень бакалавра
студента 376 группы Змеева Максима Владимировича**

Научный руководитель: доцент Рязанов В.В.

Содержание

1 Введение

В наше время машинное обучение завоевало сильно положительную репутацию. В данных часто есть закономерности, которые не удаётся отследить невооруженным глазом. Для этого существует много математических моделей, которые находят зависимости между данными и позволяют делать предсказания. Предсказательные модели можно использовать практически в любой сфере. Самые простые предсказания строятся на основе линейных моделей, которые отличаются от других высокой скоростью работы и, особенно, обучения, но меньшей предсказательной силой. Особенно удобно использовать линейные модели для анализа текстов, если используется подход, ставящий в соответствие тексту так называемый мешок слов, потому что из-за огромного количества признаков использование, например, случайных лесов является сильно невыгодным. В данной работе рассмотрены основные известные виды линейных классификаторов, к ним добавлены два менее известных подхода(релаксационный метод и метод сведения к линейному программированию). Оба подхода решают задачу поиска максимальной совместной подсистемы системы линейных неравенств. Представляет интерес сравнить их между собой и с уже проявившими себя методами. Каждый классификатор реализован в Python 3.6 с максимальным использованием библиотеки numpy для ускорения вычислений. Все классификаторы сравнивались как на модельных данных, так и на настоящих данных, взятых из известных датасетов. Сравнение производится по доле правильных ответов, для классификаторов, делающих выбор между двумя классами, на настоящих данных также приводится сравнение по метрике ROC AUC. Также были измерены времена работы классификаторов.

2 Релаксационный метод

2.1 Постановка задачи

Есть N элементов, каждый из которых принадлежит одному из двух классов. Каждый объект представляет из себя вектор размерности D. Для N_{train} элементов их классы известны:

$$y_k = \begin{cases} 1, & \text{если объект } \boldsymbol{x_k} \text{ принадлежит классу 1} \\ 0, & \text{если объект } \boldsymbol{x_k} \text{ принадлежит классу 0} \end{cases}.$$

Для остальных $N_{test} = N - N_{train}$ элементов(тестовой выборки) их классы нужно предсказать для тестовых N_{test} объектов.

2.2 Предлагаемое решение

Предлагается пытаться отделять объекты классов друг от друга гиперплоскосью в D-мерном пространстве. Гиперплоскость задаётся (D+1) -мерным вектором $\boldsymbol{w}$. Объект $\boldsymbol{x_k}$ считается объектом класса 1, если $w_0 + \sum_{j=1}^{D} x_{ij} \cdot w_j > 0$ и объектом класса 0, если $w_0 + \sum_{j=1}^{D} x_{ij} \cdot w_j < 0$(вероятность, что непрерывно распределенная случайная величина будет равна ровно нулю в такой линейной комбинации, равна нулю, поэтому качество не будет ухудшено вне зависимости от того, будет ли присваиваться нулевой

или первый класс объектам, на которых оба неравенства не выполнятся). Все векторы $\boldsymbol{x_k}$ для удобства преобразуются в (D+1)-мерные вектора вида $(1, x_{k1}, x_{k2}, .., x_{kD})$, чтобы можно было более кратко писать $\boldsymbol{x_k^T w}$. Для нахождения верктора $\boldsymbol{w}$ минимизируется функционал:

$$L = \frac{1}{N_{train}} \sum_{k=1}^{N_{train}} ([\boldsymbol{x_k^T w} >= 0] \cdot (1 - y_k) + [\boldsymbol{x_k^T w} < 0] \cdot y_k) \tag{1}$$

То есть долю ошибок при классификации обучающей выборки.

Составляется система неравенств:

$$\left\{ \boldsymbol{x_k^T w} \cdot (-1)^{y_k} >= 0, \quad k = \overline{1, N_{train}} \right. \tag{2}$$

Теперь получается, что нужно найти такой вектор w, чтобы максимальное число неравенств из (2) выполнялось. Задача сводится к поиску максимальной совместной подсистемы системы неравенств.

Далее под $\boldsymbol{x_k}$ подразумевается $\boldsymbol{x_k} \cdot (-1)^{y_k}$ как коэффициент одного уравнения системы (2). Допустим, система совместна. Искать её решение предлагается, строя последовательность $\boldsymbol{w_1}, \boldsymbol{w_2}...\boldsymbol{w_k}$.... Введено обозначение $I(\boldsymbol{w_k}) = \{k | \boldsymbol{x_k^T w} < 0\}$. Далее идут разные варианты построения последовательности $\{\boldsymbol{w_k}\}_{k=1}^{K}$, а точнее выбор направления $\boldsymbol{d_w}$ в $\boldsymbol{w_{k+1}} = \boldsymbol{w_k} + \theta_{k+1} \cdot \boldsymbol{d_w}$:

- $$\boldsymbol{d_w} = \boldsymbol{x_q} \cdot \frac{\boldsymbol{x_q^T w_k}}{||\boldsymbol{x_q}||^2}, \text{ где } q = \arg \max_{j \in I(\boldsymbol{w_k})} |\boldsymbol{x_j^T w_k}| \tag{3}$$

- $$\boldsymbol{d_w} = \boldsymbol{z} \cdot \frac{\sum_{j \in I(\boldsymbol{w_k})} (\boldsymbol{x_j^T w_k})^2}{||\boldsymbol{z}||^2}, \text{ где } \boldsymbol{z} = \sum_{j \in I(\boldsymbol{w_k})} \boldsymbol{x_j} \cdot |\boldsymbol{x_j^T w_k}| \tag{4}$$

- $$\boldsymbol{d_w} = \boldsymbol{z} \cdot \frac{\sum_{j \in I(\boldsymbol{w_k})} \lambda_j \cdot |\boldsymbol{x_j^T w_k}|}{||\boldsymbol{z}||^2}, \text{ где } \boldsymbol{z} = \sum_{j \in I(\boldsymbol{w_k})} \boldsymbol{x_j} \cdot \lambda_j; \lambda_j \geq 0 \tag{5}$$

  λ_j находятся как решения задачи максимизации числа неравенств, которые сменяют свой знак с отрицательного на положительный.

Для таких случаев при $0 \leq \theta \leq 2$ и, если система совместна, доказано[1]:

$$\rho(\boldsymbol{w_{k+1}}, Q) \leq \sigma \cdot \rho(\boldsymbol{w_k}, Q), 0 < \sigma < 1, \text{где } Q \text{ - множество решений}$$

Вернёмся к случаю, когда система не обязательно совместна. Ясно, что тогда не будет найдено решение полной системы, потому что не абсолютно любые объекты отделяются гиперплоскостью, также всегда можно встретить выбросы. То есть нужно отбрасывать некоторые из неравенств. Предлагается это делать через каждые несколько итераций, отбрасывая те неравенства, которые чаще всего не выполняются. Таким образом, наступит момент, когда система станет совместна, и решение будет найдено.

Для увеличения скорости сходимости рекомендуется решать систему:

$$\left\{ \boldsymbol{x_k^T w} \cdot (-1)^{y_k} >= -\varepsilon, \quad k = \overline{1, N_{train}} \right. \tag{6}$$

Итак, данный классификатор, имеет 3 гиперпараметра:

- ε - поправка к неравенствам;
- k - число шагов, после которого отбрасываются неравенства, которые чаще всего не выполнялись;
- θ - скорость обучения.

Данное число, конечно, можно расширить засчёт того, что скорость обучения можно менять по мере приближения к множеству решений, одновременно с этим можно менять и число шагов, которое делается до отбрасывания неравенств по мере обучения.

2.3 Реализация

Везде далее np - библиотека numpy, ускоряющая и упрощающая многие вычисления. Итак, приступим к реализации: создаётся свой класс, который называется **one_weight_linear_model**, имеющий вектор весов w, число объектов **N**, число признаков **D**(включая константу), число шагов перед исключением неравенств **steps_to_exclude**, параметр θ, который называется **theta**, параметр ε : **eps** и долю исключающихся неравенств **part_to_exclude**, потому что, если исключать по одному неравенству на большом наборе данных, то слишком долго будет находиться решение. Вектор весов сначала пустой, так как размерность данных изначально неизвестна.

```
class one_weight_linear_model():
   def __init()__(self, method, theta, eps, steps_to_exclude,
       part_to_exclude):
      self.w = None
      self.D = None
      self.N = None
      self.theta = theta
      self.eps = eps
      self.part_to_exclude = part_to_exclude
      self.steps_to_exclude = steps_to_exclude
```

Далее нужно описать метод **fit(self, X, y)**, принимающий на вход матрицу объектов и вектор номеров классов. Но для начала преобразовывается **X** так, что для объектов класса 0 добавляется ещё один константный признак, равный единице, а объекты класса 1 также получают константный признак, а затем домножаются на -1. Таким образом матрица X - матрица коэффициентов системы линейных неравенств, для которой все неравенства должны быть больше нуля. Так же инициализируются веса модели **w**. Воспользовавшись простыми соображениями о том, что примерно половина весов будет больше нуля, половина - меньше нуля, **w** инициализируются случайными числами из нормального распределения. Чтобы отклик не был слишком большим, для большего числа размерности устанавливается меньшая дисперсия, пропорционально $\sqrt{k}$. Эти подготовительные действия помещаются в отдельный метод **prepare_data(self, X, y)**

```
def prepare_data(self, X, y):
   X = np.array(X)
```

```
    self.N = X.shape[0]
    X = np.concatenate((X, np.ones([self.N, 1])), axis=1)
    self.D = X.shape[1]
    self.w = np.random.normal(0, 2. / self.D, self.D)
    X = np.apply_along_axis(lambda a: a * (-1) ** (y + 1), 0, X)
    return X
```

Вызов этого метода делается первым в методе **fit**. Далее нужно приготовить словарь номеров неравенств, которые не выполняются. Для этого предлагается хранить вектор **unfulfilled**(изначально из нулей) размера n. Если неравенство под номером i не было выполнено, то i-тый элемент вектора **unfulfilled** под номером i будет увеличен на единицу. Теперь нужно выбрать, какой шаг будет делаться. Здесь реализован второй вариант. Написана функция **calc_dw(self, X)**, которая будет высчитывать направление шага, которое будет умножаться на **theta**:

```
def calc_dw(self, X):
    negatives = X[X.dot(self.w) <= 0]
    scalar_mults = np.abs(negatives.dot(self.w))
    dw = np.sum(np.apply_along_axis(lambda a: a * scalar_mults, 0,
        negatives), axis=0)
    dw *= np.sum(scalar_mults ** 2) / np.sum(dw ** 2)
    return dw
```

Здесь **negatives** - те объекты, для которых неравенства не выполняются, **scalar_mults** - это матрица попарных произведений $\boldsymbol{x_{ji}w_{ki}}$, где j - номер объекта, i - номер признака.

Есть один ньюанс: может случиться так, что dw окажется очень маленьким по норме. для таких случаев написано дополнительное условие, которое бы не позволяло делить на 0 в вычислении согласно формуле (4):

```
if np.any(np.abs(dw) > 1e-10):
    dw *= np.sum(scalar_mults ** 2) / np.sum(dw ** 2)
else:
    dw *= np.sum(scalar_mults ** 2) / 1e-11
```

Здесь, если видно, что по модулю **dw** очень мало, то делится не на 0, а на очень маленькую величину.

Теперь для обучения нужно отбрасывать те неравенства, которые не выполняются чаще всего. Для этого пишется метод **delete_worst(self, X, unfulfilled)**, который на вход получает **X** - матрицу объекты-признаки и вектор **unfulfilled**, содержащий в себе частоту невыполнения каждого из неравенств:

```
def delete_worst(self, X, unfulfilled):
    for i in range(max(1, int(self.N * self.part_to_exclude))):
        num_to_del = np.argmax(unfulfilled)
        X = np.delete(X, num_to_del, 0)
        unfulfilled = np.delete(unfulfilled, num_to_del, 0)
    return X, unfulfilled
```

Видно, что исключается по крайней мере одно неравенство, которое ча-

ще всего не выполняется.

Итак, всё готово для написания метода **fit**. Последовательность действий довольно проста: подготавливается матрица объекты-признаки методом **prepare_data**, затем идёт цикл, где на каждом шаге делается один шаг на **dw**. Каждые **steps_to_exclude** шагов отбрасывается одно или часть **part_to_exclude** неравенств, пока не будут выполнены все неравенства оставшейся системы.

```
def fit_relax(self, X, y):
   if self.method == 'relax':
      X = self.prepare_data(X, y)
      unfulfilled = np.zeros(self.N)
      step_number = 0
      while np.any(X.dot(self.w) < self.eps) > 0:
         dw = self.calc_dw(X)
         self.w += self.theta * dw
         unfulfilled += X.dot(self.w) < self.eps
         step_number += 1
         if step_number % self.steps_to_exclude == 0:
            X, unfulfilled = self.delete_worst(X, unfulfilled)
```

Также нужно написать метод, который будет предсказывать классы для объектов после обучения **predict(self, X)**:

```
def predict(self, X):
   X = np.array(X)
   return (X.dot(self.w[: -1]) + self.w[-1] >= 0).astype(np.int)
```

3 Сведение к линейному программированию

3.1 Постановка задачи

Постановка та же, что для релаксационного метода.

3.2 Предлагаемое решение

Задача снова сводится к поиску максимальной совместной подсистемы системы линейных неравенств. Снова те неравенства, чьи объекты принадлежат к классу 0, умножаются на -1. Введится дополнительная переменная ε, которую нужно максимизировать в следующей задаче:

$$\begin{cases} \varepsilon \to \max \\ \boldsymbol{x_k^T w} \cdot (-1)^{y_k} > \varepsilon, \quad k = \overline{1, N_{train}} \end{cases} \tag{7}$$

Если система несовместна, то очевидно, что решение будет $\varepsilon = \epsilon \leq 0$. Далее предлагается отбрасывать неравенства. Отбрасывается то, для кото-

рого достигается максимум величины

$$\frac{\frac{1}{|I_0|} \cdot (\sum\limits_{\substack{i \in I_0 \\ i \neq j}} w_i^T) w_j}{||\sum\limits_{\substack{i \in I_0 \\ i \neq j}} w_i|| \cdot ||w_j||}$$

Такая процедура выполняется, пока ε не станет больше нуля.
Итак, получили классификатор с 2 гиперпараметрами:

- k - число шагов, после которого мы отбрасываем неравенства, самые отклюняющиеся от остальных невыполненных;
- *part_to_exclude* - доля неравенств, которую мы будем исключать из системы.

3.3 Реализация

Для реализации данного метода нам нужно уметь решать задачи линейного программирования. Для этого будем использовать библиотеку python-а **scipy.optimize**, в которой реализована нужная нам функция **linprog**. Переписывать конструктор класса нам не нужно, потому что все нужные нам гиперпараметры уже присутствуют в том, который мы написали ранее.

Так как мы максимизируем ε, а метод **linprog** минимизирует линейную функцию $\boldsymbol{c}^T\boldsymbol{w}$, а нам хочется максимизировать ε, что равносильно минимизации $-\varepsilon$, то в нашем случае $\boldsymbol{c} = (\underbrace{0, 0, ...,}_{\text{D + 1 нулей}} -1)^T$. Условия выглядят следующим образом:

$$A_{ub}\boldsymbol{w} \leq \boldsymbol{b_{ub}}. \tag{8}$$

Нам нужно их свести к

$$X\boldsymbol{w} \geq \boldsymbol{\varepsilon}, \text{где } \boldsymbol{\varepsilon} = (\underbrace{\varepsilon, \varepsilon, ..., \varepsilon}_{N_{train} \text{ штук}})^T, \tag{9}$$

что равносильно $X\boldsymbol{w} - \boldsymbol{\varepsilon} \geq \mathbf{0}$ или $-X\boldsymbol{w} + \boldsymbol{\varepsilon} \leq \mathbf{0}$. Для упрощения записи введём новый вектор $\tilde{\boldsymbol{w}} = (\boldsymbol{w}, \varepsilon)^T$ и новую матрицу объекты-признаки $\tilde{X} = \left(\begin{array}{c|c} -X & \mathbf{1} \end{array} \right)$. Тогда (9) эквивалентно $\tilde{X}\tilde{\boldsymbol{w}} \leq \mathbf{0}$.

Таким образом, в роли A_{ub} будет $\tilde{X}$, в роли $\boldsymbol{b_{ub}}$ будет $(\underbrace{0, 0, ..., 0}_{N_{train} \text{ штук}})^T$.

Итак, напишем метод **prepare_for_linprog(self, X, y)** для подгтовки матрицы объекты признаки $\tilde{X}$, вектора **c** и границ **bounds**. Начальный вид для $\tilde{X}$ и **c** описаны выше. Пойдём, как нужно задать границы для переменных. Ясно, что вектор весов $\boldsymbol{w}$ не должен быть ограничен какими-то конкретными числами(хотя желательно, чтобы по модулю они не были слишком велики). Но если мы не ограничим ε сверху, то, решение вовсе не будет ограничено, а значит, не будет найдено. Причина этому проста: допустим, что у нас есть какое-то решение $\tilde{\boldsymbol{w}}^*$ такое, что последняя его переменная, то есть ε, больше нуля. Тогда, так как система (8) однородна, есть решение $\tilde{\boldsymbol{w}}' = 2 \cdot \tilde{\boldsymbol{w}}^*$, у которого ε в 2 раза больше. Поэтому нам нужно ограничить ε сверху любым положительным числом, например, единицей. Ниже приведена реализация метода:

```
def prepare_for_linprog(self, X, y):
   X = self.prepare_data(X, y)
   X = np.concatenate((- X, np.ones([self.N, 1])), axis=1)
   c = np.concatenate([np.zeros(self.D), [-1]])
   bounds = [[None, None] for i in np.arange(self.D + 1)]
   bounds[-1][1] = 1
   return X, c, bounds
```

Далее нужно написать метод **exclude_with_largest_angles**, который будет исключать неравенства, вектор $\boldsymbol{x_k}$ которых имеет наибольший косинус угла со средним вектором остальных невыпоненных:

```
def exclude_with_largest_angles(self, X, res):
   unfulfilled = X[:, :-1].dot(res[:-1]) <= 0
   sumed = np.sum(X[unfulfilled, :-1], axis=0)
   cosines = np.ones(X.shape[0])
   for j in np.arange(len(unfulfilled)):
      if not unfulfilled[j]:
           continue
      rest_of_sum = sumed - X[j, :-1]
      cosines[j] = X[j, :-1].dot(rest_of_sum) / \
      rest_of_sum.dot(rest_of_sum) / X[j, :-1].dot(X[j, :-1])
      to_exclude = []
   for i in np.arange(max(1, self.part_to_exclude * self.N)):
        argmin = np.argmin(cosines)
      cosines[argmin] = 1
      to_exclude.append(argmin)
   X = np.delete(X, to_exclude, axis=0)
   return X
```

В первом цикле мы находим косинусы для каждого невыполненного неравенства. В следующем цикле сохраняем номера тех, которые собираемся удалять. Далее производим удаление. На число невыполненных неравенств делить не обязательно, потому что это общий коэффициент для всех косинусов, поэтому лишнее деление отброшено.

Остаётся собрать всё вместе и написать метод **fit_using_linprog(self, X, y)** и дописать метод **fit(self, X, y)**:

```
def fit_using_linprog(self, X, y):
   X, c, bounds = self.prepare_for_linprog(X, y)
   res = np.nan
   while True:
      b_ub = np.zeros(X.shape[0])
      res = op.linprog(c, A_ub=X, b_ub=b_ub, bounds=bounds).x
      if res[-1] > 0:
         break
      X = self.exclude_with_largest_angles(X, res)
      if res is not np.nan:
         self.w = res[:-1]

def fit(self, X, y):
   if self.method == 'relax':
```

```
    self.fit_using_relaxation(X, y)
elif self.method == 'linprog':
    self.fit_using_linprog(X, y)
```

4 Логистическая регрессия

4.1 Постановка задачи

Совпадает с постановкой в двух преддущих случаях.

4.2 Предлагаемое решение

Теперь хочется хотим ускорить процесс обучения. Для этого нужно сделать функцию потерь L непрерывно дифференцируемой. Хорошо подходит сигмоида:

$$\sigma(\boldsymbol{x_k}) = \frac{1}{1 + \exp^{-\boldsymbol{x_k^T w} \cdot (-1)^{y_k}}} \tag{10}$$

Если объект k классифицировали как объект класса 1, то значение $\boldsymbol{x_k^T w} > 0$. Если решение принято верно(true positive), то степень экспоненты будет положительной, значит вся дробь будет стремиться к нулю при увеличении значения $\boldsymbol{x_k^T w}$. Если решение ошибочное(false positive), то степень экспоненты будет отрицательной, а значит вся дровь будет стремиться к единице при отдалении $\boldsymbol{x_k^T w}$ от единицы вправо. Аналогично, если $\boldsymbol{x_k^T w} < 0$ и выбор правильный(true negative), то вся дробь будет тем ближе к нулю, чем левее от нуля значение $\boldsymbol{x_k^T w}$, если выбор неправильный(false negative), то вся дробь будет тем ближе к единице, чем больше значение $\boldsymbol{x_k^T w}$. То есть при правильных ответах значение (10) близко к единице, при неправильных - к нулю.

Функция потерь - это сумма таких сигмоид для каждого объекта из обучающей выборки:

$$\boldsymbol{L} = \sum_{k=1}^{N_{train}} \sigma(\boldsymbol{x_k}) \tag{11}$$

Подсчёт градиента сигмоиды:

$$\frac{\partial \sigma(\boldsymbol{x_k})}{\partial w_j} =$$
$$= -\frac{1}{(1 + \exp(-\boldsymbol{x_k^T w}(-1)^{y_k}))^2} \cdot \exp(-\boldsymbol{x_k^T w}(-1)^{y_k}) \cdot (-x_{kj} \cdot (-1)^{y_k}) =$$
$$= \frac{1}{1 + \exp(-\boldsymbol{x_k^T w}(-1)^{y_k})} \cdot \frac{\exp(-\boldsymbol{x_k^T w}(-1)^{y_k})}{1 + \exp(-\boldsymbol{x_k^T w}(-1)^{y_k})} \cdot (-1)^{y_k+1} \cdot x_{kj} =$$
$$= \sigma(\boldsymbol{x_k^T w}) \cdot (1 - \sigma(\boldsymbol{x_k^T w})) \cdot x_{kj} \cdot (-1)^{y_k+1}, \text{ при } k = \overline{1, N_{train}}, j = \overline{1, D+1}$$

Видно, что градиент сигмоиды легко выражается через саму сигмоиду, здесь же нужно добавить производную показателя сигмоиды.

Итак, эту функцию потерь минимизируют итерационным методом, использующим аналитический подсчёт градиента, например, градиентным спуском и тем самым находят решение.

4.3 Реализация

Для данной реализации также понадобится библиотека **scipy.optimize**, из которой мы будем использовать функцию **minimize**.

Для подготовки данных просто воспользуемся написанной **prepare_data**. Тогда все наши объекты можно будет считать за объекты класса 1, тогда перед скалярным произведением в показателе экспоненты будет просто единичный

$$\sigma(\boldsymbol{x_k}^T\boldsymbol{w}) = \frac{1}{1+\exp(\boldsymbol{x_k}^T\boldsymbol{w}}) \\ \frac{\partial\sigma(\boldsymbol{x_k}^T\boldsymbol{w})}{\partial w_j} = -\sigma(\boldsymbol{x_k}^T\boldsymbol{w}) \cdot (1-\sigma(\boldsymbol{x_k}^T\boldsymbol{w})) \cdot x_{kj}$$

Остаётся написать методы для вычисления самой функции штрафов L и для вычисления её градиента. Назовём их **calc_sigmoid_L(self, X, w)** и **calc_grad_L(self, X, w)**:

```
def calc_sigmoid_L(self, X, w):
   return (1 / (1 + np.exp(X.dot(w)))).sum()
def calc_grad_L(self, X, w):
   s = (1 / (1 + np.exp(X.dot(w))))
   return - X.T.dot(s * (1 - s))
```

И остаётся просто собрать всё в методе **fit_logistic(self, X, y)**:

```
def fit_logistic(self, X, y):
   X = self.prepare_data(X, y)
   self.w = op.minimize(lambda w: self.calc_sigmoid_L(X, w),
       self.w,method='BFGS', jac=lambda w: self.calc_grad_L(X, w)).x
```

Метод **fit** теперь выглядит следующим образом:

```
def fit(self, X, y):
   if self.method == 'relax':
      self.fit_using_relaxation(X, y)
   elif self.method == 'linprog':
      self.fit_using_linprog(X, y)
   elif self.method == 'logistic':
      self.fit_logistic(X, y)
   else:
      print('No such method', self.method)
```

5 Support Vector Machine (SVM)

5.1 Постановка задачи

Имеется N объектов, каждый из которых принадлежит одному из M классов. Каждый объект имеет D числовых признаков. Для N_{train} объек-

тов(обучающей выборки) их классы известны.

$$y_k = \begin{cases} 0, & \text{если объект } \boldsymbol{x_k} \text{ принадлежит классу 0} \\ 1, & \text{если объект } \boldsymbol{x_k} \text{ принадлежит классу 1} \\ ... & \\ M, & \text{если объект } \boldsymbol{x_k} \text{ принадлежит классу M} \end{cases}.$$

Для остальных $N_{test} = N - N_{train}$ элементов(тестовой выборки) их классы нужно предсказать для тестовых N_{test} объектов.

5.2 Предлагаемое решение

Теперь имеется более, чем два класса, поэтому бля принятия решения недостаточно одной разделяющей гиперплоскости. Для каждого класса заводится свой вектор весов $\boldsymbol{w_k}, k = \overline{1, M}$. В итоге имеется матрица весов $\boldsymbol{W} = ||\boldsymbol{w_1}, \boldsymbol{w_2}, ..., \boldsymbol{w_M}||$. Для определения класса объекта $\boldsymbol{x}$ вычисляется произведение $\boldsymbol{W}^T\boldsymbol{x}$ - вектор оценок. Выбирается тот класс, чья оценка самая высокая.

Функция потерь L_i на одном объекте выглядит следующим образом:

$$\boldsymbol{L_k} = \sum_{j \neq y_k} \max\left(0, \boldsymbol{x_k^T w_k} - \boldsymbol{x_k^T w_{y_k}} + \Delta\right) \tag{12}$$

Также есть смысл использовать регуляризацию

$$R(W) = \sum_{i=1}^{D}\sum_{j=1}^{M} W_{i,j}^2 \tag{13}$$

Итоговая функция потерь может быть записана как

$$\boldsymbol{L} = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W)$$

Или более полно:

$$L = \frac{1}{N}\sum_{k=1}^{N}\sum_{j \neq y_k} \max\left(0, \boldsymbol{x_k^T w_k} - \boldsymbol{x_k^T w_{y_k}} + \Delta\right) + \lambda \sum_i \sum_j W_{i,j}^2 \tag{14}$$

Остаётся найти минимум по весам данной функции. Оптимизация проводится градиентным спуском. Производная функции потерь по весу верного класса выражается в виде:

$$\nabla_{\boldsymbol{w_{y_k}}} L_k = -\left(\sum_{j \neq y_k} \mathbf{1}(\boldsymbol{x_k^T w_j} - \boldsymbol{x_k^T w_{y_k}} + \Delta > 0)\right)\boldsymbol{x_k} \tag{15}$$

где $\mathbf{1}$ - функция-индикатор, равная единице, если условие верно, и нулю, если условие не верно. По остальным весам ($j \neq y_k$):

$$\nabla_{\boldsymbol{w_j}} L_k = \mathbf{1}(\boldsymbol{x_k^T w_j} - \boldsymbol{x_k^T w_{y_k}} + \Delta > 0)\boldsymbol{x_k} \tag{16}$$

Данный классификатор имеет 2 гиперпараметра:

1. Δ - сдвиг в функции потерь;
2. λ - коэффициент перед весами. Чем он больше, тем сильнее регуляризируются веса.

5.3 Реализация

Создан отдельный класс **many_weight_linear_model**. Для SVM требуется реализовать подсчёт функции потерь и её градиента. Снова нужна подготовка данных. Реализация **prepare_svm(self, X, y)**:

```
def prepare_data(self, X, y):
   X = np.array(X)
   y = np.array(y).astype(int)
   X = np.concatenate((X, np.ones([X.shape[0], 1])), axis=1)
   self.N = X.shape[0]
   self.D = X.shape[1]
   self.M = np.unique(y).shape[0]
   self.W = np.random.randn(X.shape[1], np.max(y) + 1) /
       np.sqrt(X.shape[1] / 2)
   self.t = 0
   self.m = np.zeros(self.W.shape)
   self.v = np.zeros(self.W.shape)
   return X, y
```

В этом классификаторе и следующем для минимизации градиентного спуска используется метод минимизации *adam*. Для его реализации нужно уметь подсчитывать градиент функции потерь. Функция, вычисляющая градиент и сами потери, названа **loss_and_grad_svm(self, X, y)**.

```
def loss_and_grad_svm(self, X, y):
   scores = X.dot(self.W)
   right_scores = scores[np.arange(self.N), y].reshape([self.N, 1])
   margins = np.maximum(0, scores - right_scores + self.delta)
   margins[np.arange(self.N), y] = 0
   loss = np.sum(margins) / self.N + self.lmda * np.sum(self.W * self.W)

   margins = margins > 0
   to_minus = np.sum(margins, axis=1)
   coefs_0 = ((margins[:, 0] > 0) - to_minus * (y ==
       0)).reshape([self.N, 1])
   derivatives = np.sum(X * coefs_0, axis=0)
   derivatives = derivatives.reshape([1, self.D])
   for i in np.arange(1, self.M):
      coefs_i = ((margins[:, i] > 0) - to_minus * (y ==
          i)).reshape([self.N, 1])
      derivatives = np.concatenate((derivatives, [np.sum(X * coefs_i,
          axis=0)]))
   dW = derivatives.T / self.N + 2 * self.lmda * self.W

   return loss, dW
```

Отдельно реализован метод **adam(self, X, y**.

```
def adam(self, X, y):
   loss, dW = self.loss_and_grad(X, y)
   self.t += 1
   self.m = self.m * self.beta_1 + (1 - self.beta_1) * dW
   self.v = self.v * self.beta_2 + (1 - self.beta_2) * dW * dW
```

```
    m = self.m / (1 - self.beta_1 ** self.t)
    v = self.v / (1 - self.beta_2 ** self.t)
    self.W -= self.learning_rate * self.m / (np.sqrt(self.v) + self.eps)
```

Метод **loss_and_grad** - обобщение того же для svm.

```
def loss_and_grad(self, X, y):
    if self.method == 'softmax':
        return self.loss_and_grad_softmax(X, y)
    else:
        return self.loss_and_grad_svm(X, y)
```

Для простоты проводится просто 1000 итераций сдвига весов без анализа изменения функции потерь.

```
def fit(self, X, y):
    X, y = self.prepare_data(X, y)
    for step in range(1000):
        self.adam(X, y)
```

Функция **predict** принимает новый вид:

```
def predict(self, X):
    X = np.concatenate((X, np.ones([X.shape[0],1])), axis=1)
    scores = X.dot(self.W)
    return np.argmax(scores, axis=1)
```

6 Softmax

6.1 Постановка задачи

Та же, что в SVM.

6.2 Предлагаемое решение

Пусть имеется некая оценочкая функция $f_j(\boldsymbol{x_k}, \boldsymbol{W})$, в данном случае $f_j(\boldsymbol{x_k}, \boldsymbol{W}) = \boldsymbol{W}^T \boldsymbol{x_k}$, дающая оценку принадлежности объекта $\boldsymbol{x_k}$ классу j. Функция потерь в таком случае выглядит следующим образом:

$$L_k = -\log\left(\frac{e^{f_{y_k}(\boldsymbol{x_k},\boldsymbol{W})}}{\sum_j e^{f_j(\boldsymbol{x_k},\boldsymbol{W})}}\right) = -f_{y_k} + \log\sum_j e^{f_j(\boldsymbol{x_k},\boldsymbol{W})} \tag{17}$$

Кросс-энтропия между оценённым распределением q и настоящим p даётся формулой:

$$H(p,q) = -\sum_x p(x)\log q(x) \tag{18}$$

В нашем случае $q = \frac{e^{\boldsymbol{f_{y_k}}(\boldsymbol{x_k},\boldsymbol{W})}}{\sum_j e^{\boldsymbol{f_j}(\boldsymbol{x_k},\boldsymbol{W})}}$, $p = (0,0,0,...,1,...,0)$, где единице стоит на y_k месте. То есть данная функция потерь минимизирует кросс-энтропию между оценённым и истинным распределением.

Также такой выбор функции потерь можно интерпретировать как желание найти максимум логарифма функции правдоподобия для данного рапределения.

Итоговая функция потерь выглядит следующим образом:

$$L = \frac{1}{N} \sum_k \boldsymbol{W}^T \boldsymbol{x_k} + R(W) \tag{19}$$

Практический совет: из-за использования экспонент в функции потерь могут возникнуть проблемы с численными подсчётами, потому что значения $e^{f_j(\boldsymbol{x_k}, \boldsymbol{W})}$ могут оказаться очень большими. Для избежания этих проблем стоит сделать поправку на константу:

$$\frac{e^{f_{y_k}(\boldsymbol{x_k}, \boldsymbol{W})}}{\sum_j e^{f_j(\boldsymbol{x_k}, \boldsymbol{W})}} = \frac{Ce^{f_{y_k}(\boldsymbol{x_k}, \boldsymbol{W})}}{C\sum_j e^{f_j(\boldsymbol{x_k}, \boldsymbol{W})}} = \frac{e^{f_{y_k}(\boldsymbol{x_k}, \boldsymbol{W}) + \log C}}{\sum_j e^{f_j(\boldsymbol{x_k}, \boldsymbol{W}) + \log C}} \tag{20}$$

Для уверенности можно выбрать $\log(C) = -\max_j f_j$. Тогда все экспоненты будут давать значение меньше единицы, что даст уверенность в точности вычислений.

Остаётся подсчитать градиент функции потерь:

$$p_k = \frac{e^{f_{y_k}}}{\sum_j e^{f_j}} \qquad\qquad L_k = -\log p_k$$

$$\frac{\partial L_k}{\partial f_j} = p_j - \mathbf{1}(k = y_k) \tag{21}$$

$$\frac{\partial L_k}{\partial \boldsymbol{w_j}} = (p_j - \mathbf{1}(k = y_k))\, \boldsymbol{x_k} \tag{22}$$

6.3 Реализация

Всё, что нужно для реализации - написать метод **loss_and_grad_softmax(self, X, y)**. Вся остальная часть заложена в уже реализованных **fit** и **adam**.

```
def loss_and_grad_softmax(self, X, y):
   scores = X.dot(self.W)
   scores -= np.max(scores, axis=1, keepdims=True)
   exp_scores = np.exp(scores)
   probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

   correct_logprobs = - np.log(probs[np.arange(X.shape[0]), y])
   loss = np.sum(correct_logprobs) + self.lmda * np.sum(self.W * self.W)

   dscores = probs
   dscores[np.arange(X.shape[0]), y] -= 1
   dscores /= X.shape[0]
   dW = np.dot(X.T, dscores) + self.lmda * 2 * self.W

   return loss, dW
```

7 Сравнение на модельных данных

Для начала стоит сравнить классификаторы на модельных данных. Были сгенерированы разные виды выборок:

- Выборка, имеющая 20 признаков. Для каждого признака $\mathbb{E}[x^1] = \Delta + \mathbb{E}[x^2]$, где верхний индекс - номер выборки. Объекты сгенерированы из нормального распределения, где $\mathbb{E}[x^1] = -0.5, \mathbb{E}[x^2] = 0.5, \mathbb{D}[x^i] = \sigma^2$, значение дисперсии изменяется в разных экспериментах, $\Delta = 1$.
- Датасет аналогичен первому, но есть три признака, данные для которого сгенерированы из одного распределения. Здесь тест покажет, могут ли классификаторы определять значимость признаков.
- Выборка, имеющая 2000 признаков. Объекты сгенерированы из нормального распределения, где $\mathbb{E}[x^1] = -0.5, \mathbb{E}[x^2] = 0.5, \mathbb{D}[x^i] = \sigma^2$, значение дисперсии изменяется в разных экспериментах, $\Delta = 1$.
- **The 20 newsgroups text dataset** - датасет из газетных статей на 20 разных тем. Чтобы сравнить сразу все классификаторы между собой, брались только статьи на 2 темы. Близость тем менялась.
- **breast cancer wisconsin dataset** - датасет по раковым больным в Висконсине.

Значение σ менялось от 0.5 до 2, приложен график количества правильных ответов для каждого из классификаторов для разных дисперсий.

7.1 Первая выборка

В обучающей выборке - 400 объектор, в тестовой - 200. Проводилось по 5 тестов на одну дисперсию. Дисперсия менялась в диапазоне от 0.7 до 2. Параметры для тестирования:

```
sigmas = np.linspace(0.7, 2, 14)
m1 = -0.5
m2 = 0.5
train_size = 400
test_size = 200
tests_num = 5
results = np.zeros([len(sigmas), 5])

lin_m = mm.one_weight_linear_model(method='linprog')
rel_m = mm.one_weight_linear_model(method='relax')
log_m = mm.one_weight_linear_model(method='logistic')
soft_m = mm.many_weight_linear_model(method='softmax')
svn_m = mm.many_weight_linear_model(method='svm')
models = [lin_m, rel_m, log_m, soft_m, svn_m]
```

Приведён график, на котором видно, как менялась доля верных предсказаний для каждого классификатора.

На тестах оказалось, что при увеличении числа признаков довольно сильно начинает проигрывать по веремни работы классификатор, отнованный на решении задачи линейного программирования и логистическая регрессия.

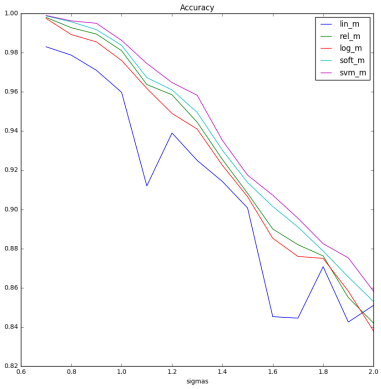

Рис. 1: График доли верных предсказаний для классификатров на 1 выборке

Также его точно сильно нестабильна. Отбросим его для следующих датасетов с большим числом признаков.

7.2 Третья выборка

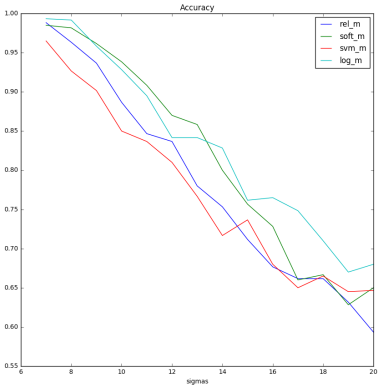

Рис. 2: График доли верных предсказаний для классификатров на 3 выборке

8 Используемая литература

Список литературы

[1] The relaxation method for linear inequalities, Canad. J. Math. 6(1954), 382-392.

[2] CS231n Convolutional Neural Networks for Visual Recognition http://cs231n.github.io/linear-classify/