Как заставить работать бинарный классификатор чуточку лучше

* [Алгоритмы](https://habrahabr.ru/hub/algorithms/),
* [C++](https://habrahabr.ru/hub/cpp/)
* [Из песочницы](https://habrahabr.ru/sandbox/)

***Disclaimer:****пост написан по мотивам*[*данного*](http://habrahabr.ru/post/120194/)*. Я подозреваю, что большинство читателей прекрасно знает, как работает Наивный Байесовский классификатор, поэтому предлагаю лишь мельком хотя бы глянуть на то, о чём там говорится, перед тем как переходить под кат.*

Решение задач с помощью алгоритмов машинного обучения давно и прочно вошло в нашу жизнь. Это произошло по всем понятным и объективным причинам: дешевле, проще, быстрее, чем явно кодить алгоритм решения каждой отдельной задачи. До нас, обычно, доходят «черные ящики» классификаторов (вряд ли тот же ВК предложит вам свой корпус размеченных имен), что не позволяет ими управлять в полной мере.  
Здесь я бы хотел рассказать о том, как попробовать добиться «лучших» результатов работы бинарного классификатора, о том какие характеристики бинарный классификатор имеет, как их измерять, и как определить, что результат работы стал «лучше».

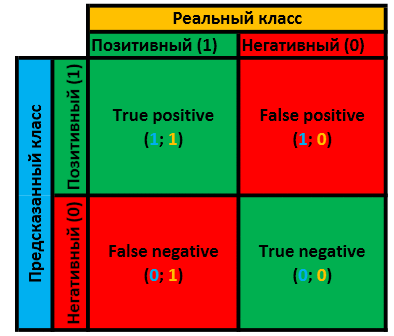
Теория. Краткий курс

1. Бинарная классификация

Пусть https://habrastorage.org/getpro/habr/post_images/19b/dd8/21e/19bdd821e3d32029e66bf63246c01108.png — множество объектов, https://habrastorage.org/getpro/habr/post_images/745/637/711/745637711a25d149d283b377e95ada16.png — конечное множество классов. **Классифицировать объект** — использовать отображение https://habrastorage.org/getpro/habr/post_images/64a/2da/d42/64a2dad425c6147b1af14e7ba0b98743.png. Когда https://habrastorage.org/getpro/habr/post_images/82a/cf8/116/82acf8116a2888dd50756ea870df59f3.png, такая классификация называется **бинарной**, потому что у нас всего 2 варианта на выходе. Для простоты дальше будем считать, что https://habrastorage.org/getpro/habr/post_images/ec0/33a/900/ec033a9003fd2a02aa0985ee0696a0cc.png, поскольку абсолютно любую задачу бинарной классификации можно к этому привести. **Обычно**, результатом отображения объекта в класс является действительное число, и, если оно выше заданного **порога**, то объект классифицируется, как **позитивный**, и его класс — 1.

2. Таблица сопряженности бинарного классификатора

Очевидно, что предсказанный класс объекта, полученный в результате отображения https://habrastorage.org/getpro/habr/post_images/6c9/154/701/6c91547016a6258eb18222c57bb15066.png, может либо совпасть в реальным классом, либо нет. Итого 4 варианта в сумме. Это очень наглядно демонстрируется данной табличкой:



Если мы знаем количественные значения каждой из оценок — мы знаем всё что можно о данном классификаторе и можем копать дальше.  
(Я намеренно не использую термины типа «ошибка 1го рода», потому что мне это кажется неочевидным и ненужным)  
Дальше будем использовать следующие обозначения:

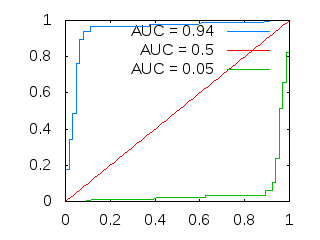
* https://habrastorage.org/getpro/habr/post_images/bec/0fe/57c/bec0fe57cc2fd896237996de84cc9c22.png — количество True Positive результатов на данной выборке.
* https://habrastorage.org/getpro/habr/post_images/d92/b32/a20/d92b32a205e3fa1974f342bd30ba8936.png — количество True Negative результатов на данной выборке.
* https://habrastorage.org/getpro/habr/post_images/481/5a7/12a/4815a712a20c85154840d73520e94da4.png — количество False Positive результатов на данной выборке.
* https://habrastorage.org/getpro/habr/post_images/cce/050/993/cce050993d18155925bdf641f4abad46.png — количество False Negative результатов на данной выборке.

3. Характеристики бинарного классификатора

**Точность (precision)** — показывает, сколько из предсказанных позитивных объектов, оказались действительно позитивными.  
https://habrastorage.org/getpro/habr/post_images/5fe/916/e53/5fe916e53c7df86ded4c773d3b69a455.png  
**Полнота (recall)** — показывает, сколько от общего числа реальных позитивных объектов, было предсказано, как позитивный класс.  
https://habrastorage.org/getpro/habr/post_images/09c/5d4/672/09c5d46721206811f90acc5e5c977557.png  
Эти две характеристики являются основными для любого бинарного классификатора. Какая из характеристик важнее — всё зависит от задачи. Например, если мы хотим создать «школьную поисковую систему», то в наших интересах будет убрать «недетский» контент из выдачи, и тут гораздо важнее полнота, чем точность. В случае же определения пола имени — нам скорее интересна точность, чем полнота.  
**F-мера (F-measure)** — характеристика, которая позволяет дать оценку одновременно по точности и полноте.  
https://habrastorage.org/getpro/habr/post_images/9c4/f96/816/9c4f968165afda9d658b5524de5ab678.png  
Коэффициент https://habrastorage.org/getpro/habr/post_images/dbf/78a/f1c/dbf78af1ca7b5f3f2d94600591367bdc.png задаёт соотношение весов точности и полноты. Когда https://habrastorage.org/getpro/habr/post_images/750/f8f/aa1/750f8faa10a90fa727f27d8d56becb08.png, F-мера придаёт одинаковый вес обеим характеристикам. Такая F-мера называется сбалансированной, или https://habrastorage.org/getpro/habr/post_images/a9f/148/c70/a9f148c7097aeed158e153846a815ec3.png  
https://habrastorage.org/getpro/habr/post_images/8de/037/c38/8de037c38cbc716bb6069930152ca898.png  
**False Positive Rate (https://habrastorage.org/getpro/habr/post_images/cde/9ca/616/cde9ca616f99d6529f31d87d6a1af776.png)** — показывает, сколько от общего числа реальных негативных объектов, оказались предсказанными неверно.  
https://habrastorage.org/getpro/habr/post_images/6dd/5b5/3cb/6dd5b53cb88f9140c3b34f90f31c73d6.png

4. ROC-кривая и её AUC

**ROC-кривая** — график, который позволяет оценить качество бинарной классификации. График показывает зависимость **TPR**(полноты) от **FPR** при варьировании порога. В точке (0,0) порог минимален, точно так же минимальны и **TPR** и **FPR**. Идеальным случаем для классификатора является проход графика через точку (0,1). Очевидно, что график данной функции всегда монотонно не убывает.



**AUC (Area Under Curve)** — данный термин (площадь под графиком) даёт количественную характеристику ROC-кривой: чем больше — тем лучше. AUC — эквивалентна вероятности, что классификатор присвоит большее значение случайно выбранному позитивному объекту, чем случайно выбранному негативному объекту. Именно поэтому ранее было сказано, что, **обычно**, позитивному классу ставится оценка выше, чем негативному.  
Когда **AUC = 0.5**, то данный классификатор равен случайному. Если **AUC < 0.5**, то можно просто перевернуть выдаваемые значения классификатором.

Кросс валидация

Методов кросс валидации (оценки качества бинарного классификатора) существует много, и это — тема для отдельной статьи. Здесь я хочу просто рассмотреть один из самых популярных методов, чтобы понять, как эта штука вообще работает, и зачем она нужна.  
Построить ROC-кривую, конечно же, можно по любой выборке. Однако, ROC-кривая, построенная по обучающей выборке, будет смещена влево-вверх из-за переобучения. Чтобы этого избежать и получить наиболее объективную оценку, используется кросс валидация.  
**K-fold cross validation** — пул разбивается на **k** фолдов, затем каждый фолд используется для теста, в то время как остальные **k-1** фолдов — для обучения. Значение параметра **k** может быть произвольным. В данном случае я использовал его равным 10. Для предоставленного классификатора пола имени получились следующие результаты AUC (get\_features\_simple — одна значимая буква, get\_features\_complex — 3 значимых буквы)

|  |  |  |
| --- | --- | --- |
| **fold** | **get\_features\_simple** | **get\_features\_complex** |
| 0 | 0.978011 | 0.962733 |
| 1 | 0.96791 | 0.944097 |
| 2 | 0.963462 | 0.966129 |
| 3 | 0.966339 | 0.948452 |
| 4 | 0.946586 | 0.945479 |
| 5 | 0.949849 | 0.989648 |
| 6 | 0.959984 | 0.943266 |
| 7 | 0.979036 | 0.958863 |
| 8 | 0.986469 | 0.951975 |
| 9 | 0.962057 | 0.980921 |
| **avg** | 0.9659703 | 0.9591563 |

Практика

1. Подготовка

Весь репозиторий [здесь](https://github.com/achigin/OptimalThresholdFinder).  
Я взял тот же размеченный файл и [заменил](https://raw.githubusercontent.com/achigin/OptimalThresholdFinder/master/names_learn_pool.txt) в нём «m» на 1, «f» — 0. Данный пул будем использовать для обучения, как и автор предыдущей статьи. Вооружившись первой страницей выдачи любимого поисковика и awk, я [наклепал](https://raw.githubusercontent.com/achigin/OptimalThresholdFinder/master/names_test_pool.txt) список имён, которых нет в изначальном. Данный пул будет использоваться для теста.  
Изменил функцию классификации так, чтобы она возвращала вероятности позитивного и негативного классов, а не логарифмические показатели.

Функция классификации

2. Реализация и использование

Как написано в заголовке, моей задачей было заставить работать бинарный классификатор лучше, чем он работает по умолчанию. В данном случае, мы хотим научиться определять пол имени, лучше чем по вероятности Наивного Байеса 0.5. Для этого и была написана эта простейшая утилита. Написана она на С++, потому что gcc есть везде. Сама реализация ничего интересного, как мне кажется, не представляет. С ключом **-?** или **--help** можно почитать справку, я постарался описать её максимально подробно.  
Ну а теперь, собственно то, к чему мы шли: оценка классификатора и его тюнинг. На выходе **nbc.py**создаёт простыню из файлов с результатами классификации, прямо их я и использую далее. Для наших целей, нам будет наглядно увидеть графики точности от порога, полноты от порога и F-меры от порога. Их можно построить следующим образом:

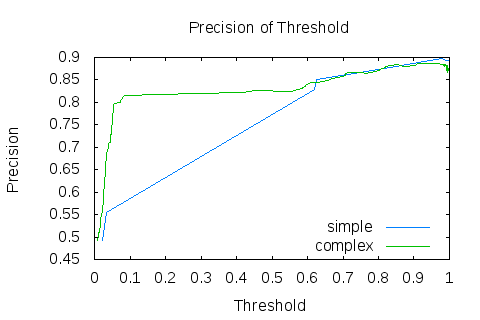
$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_simple -x thr -y prc -p plot\_test\_thr\_prc\_simple.txt

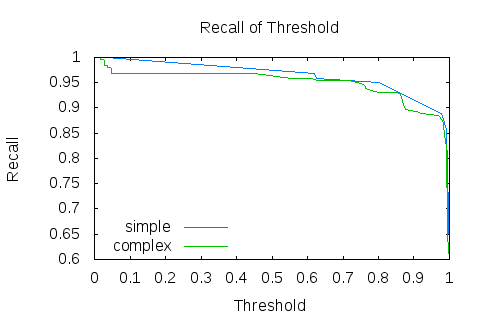
$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_simple -x thr -y tpr -p plot\_test\_thr\_tpr\_simple.txt

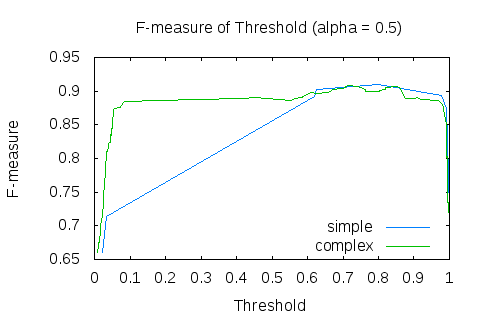
$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_simple -x thr -y fms -p plot\_test\_thr\_fms\_simple.txt

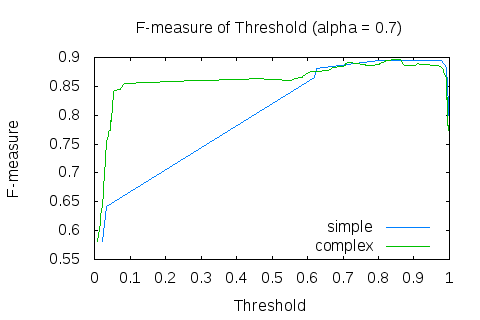
$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_simple -x thr -y fms -p plot\_test\_thr\_fms\_0.7\_simple.txt -a 0.7

Для образовательных целей, я сделал 2 графика F-меры от порога, с разными весами. Второй вес был выбран 0.7, потому что в нашей задаче нам больше интересна точность, чем полнота. График по умолчанию строится на основе 10000 различных точек, это очень много для таких простых данных, но это неинтересные тонкости оптимизации. Точно так же построив данные графиков для get\_features\_complex, получаем следующие результаты:









Из графиков становится очевидно, что классификатор показывает лучшие результаты отнюдь не при пороге 0.5. График F-меры явственно демонстрирует, что «сложная фича» даёт лучший результат на всём варьировании порога. Это довольно логично, учитывая, что на то она и «сложная». Получим значения порога, при которых F-мера достигает максимума:

$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_simple --target fms --argument thr --argval 0

Optimal threshold = 0.8 Target function = 0.911937 Argument = 0.8

$ ./OptimalThresholdFinder -A 3 -P 1 < names\_test\_pool.txt\_complex --target fms --argument thr --argval 0

Optimal threshold = 0.716068 Target function = 0.908738 Argument = 0.716068

Согласитесь, эти значения гораздо лучше тех, что оказались при пороге по умолчанию 0.5.

Заключение

Такими простыми манипуляциями, мы смогли подобрать оптимальный порог Наивного Байеса для определения пола имени, который работает лучше порога по умолчанию. Тут встаёт резонный вопрос, как мы определили, что он работает «лучше»? Я не раз упомянул о том, что в данной задаче нам важнее точность, чем полнота, однако вопрос о том насколько она важнее — очень и очень сложный, поэтому для оценки работы классификатора использовалась сбалансированная F-мера, которая в данном случае может являться объективным показателем качества.  
Что гораздо интереснее, результаты работы бинарного классификатора на основе «простой» и «сложной» фичи в итоге оказались примерно одинаковыми, отличаясь значением оптимального порога.