SVM (Метод опорных векторов для линейно неразделимого случая)

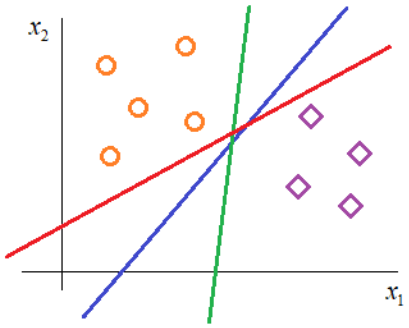
1 SVM - это

Метод опорных векторов (SVM) — это набор методов обучения с учителем, используемых для классификации, регрессии и выявления выбросов.

В данной презентации мы рассмотрим метод опорных векторов для задачи классификации. Будет представлена основная идея алгоритма, вывод настройки его весов и разобрана реализация SVM для линейного случая своими руками на C++. На примере датасета WinesQuality будет продемонстрирована работа написанного алгоритма с линейно разделимыми/неразделимыми данными в пространстве и визуализация обучения/прогноза. Дополнительно будут озвучены плюсы и минусы алгоритма, его модификации.

2 Основная идея

Рассмотрим задачу бинарной классификации образов с позиции разделяющей гиперплоскости. Для простоты рассмотрения и реализации в рамках презентации положим, что у нас двумерное признаковое пространство и значит гиперплоскость будет линией. Тогда каждый образ класса можно представить точкой на плоскости. Пусть образы обучающей выборки распределяются, следующим образом как показано на рисунке



Как можно заметить, для нашего примера можно построить множество различных разделяющих линий (гиперплоскостей) так, что каждая из них будет корректно отделять один класс от другого. На нашем рисунке красная «полагает», что образы распределены несколько горизонтально, а зеленая, что образы идут более вертикально. Синяя линия в этом случае — наилучший вариант, потому что она проходит ровно посередине между группами, ни к одной не наклоняясь. Она не делает лишних предположений о том, как точки могли бы располагаться, а просто максимально их разделяет.

Вот эта идея проведения разделяющей гиперплоскости, которая бы ориентировалась только на распределение обучающей выборки и по возможности не делала бы дополнительных предположений о распределении образов в классах, положена в основу метода опорных векторов.

3 Математическая модель

Начнём с самой модели. В задаче бинарной классификации мы ищем гиперплоскость, которая разделяет данные на два класса. Уравнение гиперплоскости задаётся так:

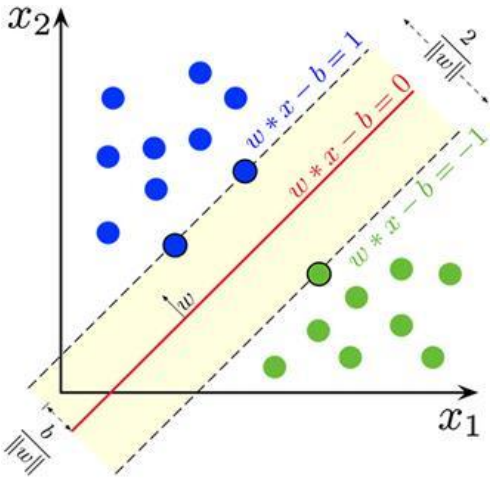
$$f(x) = \langle \omega, x \rangle + b$$

где

- (omega) — вектор весов (нормаль к гиперплоскости),
- (b) — смещение,
- (omega, x) — скалярное произведение весов и признаков.

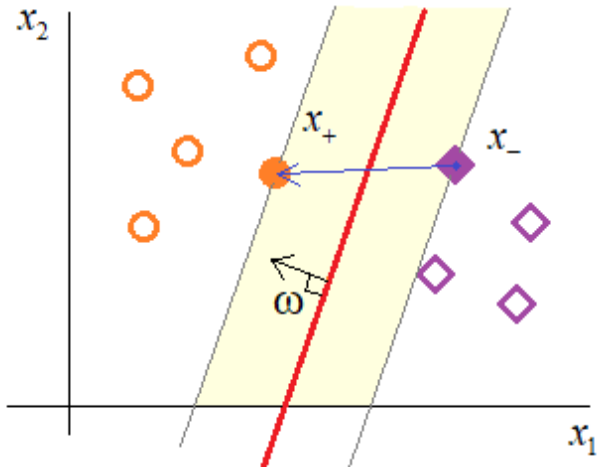
Знак функции (f(x)) определяет класс объекта.

Теперь предположим, что данные линейно разделимы (затем, мы этот случай обобщим на линейно неразделимый). Тогда мы можем построить не просто разделяющую прямую/плоскость, а полосу между двумя параллельными гиперплоскостями, в которую не попадает ни один объект выборки. Эта полоса называется **разделяющей полосой** (или «margin»).



Ширина полосы будет определяться расположением граничных векторов x в признаковом пространстве. Если взять любые два образа разных классов, расположенные ближе всего к разделяющей границе (то есть, лежащие на границе полосы), то ширину полосы можно вычислить как проекцию вектора $x_+ - x_-$ на вектор ω:

$$\langle \omega, x_+ - x_- \rangle = \omega^T \cdot (x_+ - x_-) = |\omega| \cdot \underbrace{|x_+ - x_-| \cdot \cos\alpha}_{\text{ширина полосы}}$$



То есть, получаем ширину, умноженную на длину вектора коэффициентов ω. Поэтому, окончательно, имеем:

$$L = |x_+ - x_-| \cdot \cos\alpha = \frac{\langle \omega, x_+ - x_- \rangle}{|\omega|} \to \max$$

И эту величину нужно максимизировать. Получаем выражение для максимизации ширины:

$$L = \frac{\langle \omega, x_+ - x_- \rangle}{\|\omega\|^2} \to \max$$

Наконец, сделаем еще одно, последнее упрощение. В задачах бинарной классификации вводится понятие отступа (margin):

$$M_i = y_i \cdot a(x_i) = y_i \cdot (\langle \omega, x_i \rangle - b), \quad i = 1, 2, \ldots, l$$

Эта величина характеризует расстояние от разделяющей гиперплоскости до выбранного образа. Причем:

$$\begin{cases} M_i > 0 & \text{при верной классификации} \\ M_i < 0 & \text{при неверной классификации} \end{cases}$$

Но, так как мы рассматриваем случай линейно-разделимых образов, то заведомо существуют такие значения ω и b, что:

$$M_i = y_i \cdot (\langle \omega, x_i \rangle - b) > 0, \quad i = 1, 2, \ldots, l$$

Мы можем умножить эту величину на некоторое число:

$$\alpha > 0$$

В результате, получим:

$$\alpha M_i = y_i \cdot (\langle \alpha\omega, x_i \rangle - \alpha b), \qquad i = 1, 2, \ldots, l$$

Теперь можем подобрать такое значение α,
чтобы для любых образов, лежащих на границе полосы, величина:

$$M_i(x_+) = 1; \quad M_i(x_-) = 1$$

Очевидно, что для линейно-разделимого случая мы всегда можем это сделать. Но тогда автоматически получается, что:

$$\langle \omega, x_+ - x_- \rangle = \langle \omega, x+ \rangle - \langle \omega, x_- \rangle = 1 - (-1) = 2$$

и ширина полосы будет определяться выражением:

$$L = \frac{2}{\|\omega\|^2} \to \max$$

То есть, чем меньше длина вектора весов, тем шире разделяющая полоса.

Цель метода опорных векторов (SVM) — **максимизировать ширину разделяющей полосы**. Это эквивалентно задаче минимизации нормы весов:

$$\min \frac{1}{2}\|\omega\|^2$$

при условиях:

$$y_i(\langle \omega, x_i \rangle + b) \geq 1, \quad \forall i$$

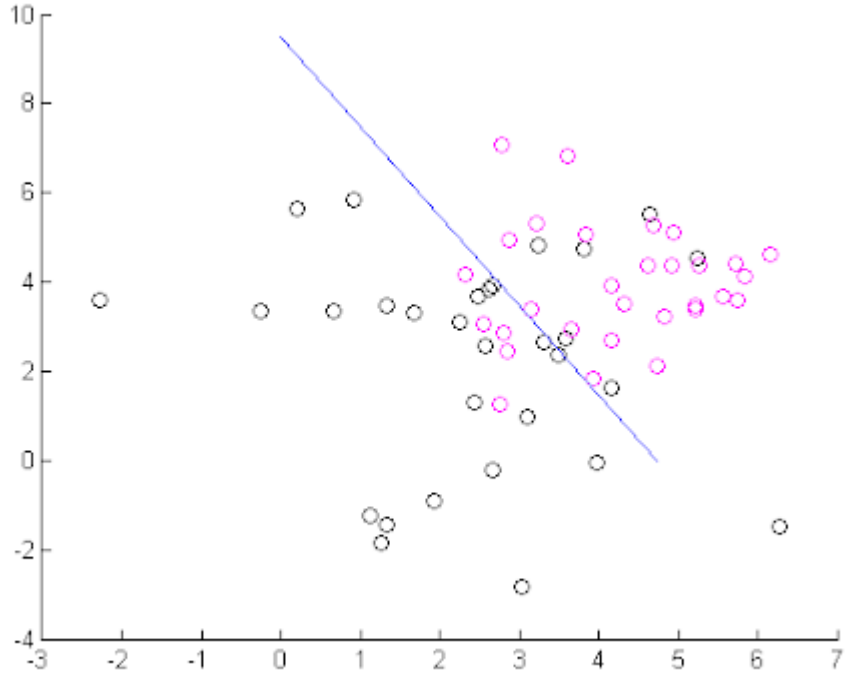
где ($y_i \in \{-1, +1\}$) — метка класса.

Эти ограничения означают, что каждый объект находится либо «снаружи» разделяющей полосы, либо точно на её границе (в этом случае отступ равен 1).

Таким образом, задача SVM формулируется как **задача квадратичного программирования**:

- минимизируем квадратичную норму весов $\frac{1}{2}(\|\omega\|^2)$,
- при линейных ограничениях в виде неравенств

4 Метод опорных векторов для линейно неразделимого случая
Однако в общем случае образы в обучающих выборках редко бывают линейно разделимыми



Поэтому при линейно неразделимой выборке мы не сможем найти параметры ω и b, которые бы удовлетворяли линейным ограничениям на отступы:

$$M_i(\omega, b) \geq 1, \quad i = 1, 2, \ldots, l$$

Для этого придумали следующую эвристику. Разрешим классификатору ошибаться на некоторую величину (slack variables):

$$\xi_i \geq 0, \quad i = 1, 2, \ldots, l$$

для каждого i-го образа:

$$M_i(\omega, b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, l$$

Величины slack variables еще можно воспринимать как некоторый штраф за нарушение исходного неравенства. Величина этой ошибки должна быть как можно меньше, то есть, нам нужно найти такие ω и b, чтобы

$$\xi_i \to 0, \quad i = 1, 2, \ldots, l$$

Это условие мы можем учесть в алгоритме минимизации, записав его, следующим образом:

$$\begin{cases} \frac{1}{2}\|\omega\|^2 + C \cdot \sum_{i=1}^{l} \xi_i \to \min\limits_{\omega, b, \xi} \\ M_i(\omega, b) \geq 1 - \xi_i, \quad i = 1, 2, \ldots, l \\ \xi_i \geq 0, \quad i = 1, 2, \ldots, l \end{cases}$$

где C – гиперпараметр, определяющий степень минимизации величин $\{\xi_i\}$. Чем меньше C, тем “мягче” разделение.

5 Реализация упрощенного SVM

Нашей основной практической задачей было реализовать SVM таким образом, что бы отличие от библиотечной sklearn было минимальным

немного про код. Код был реализован на языке C++, потому что при тестировании кода использовался объемный датасет, обработка которого заняло бы время. Так же C++ позволяет подробно рассмотреть механизмы работы алгоритма и имеет более точную настройку

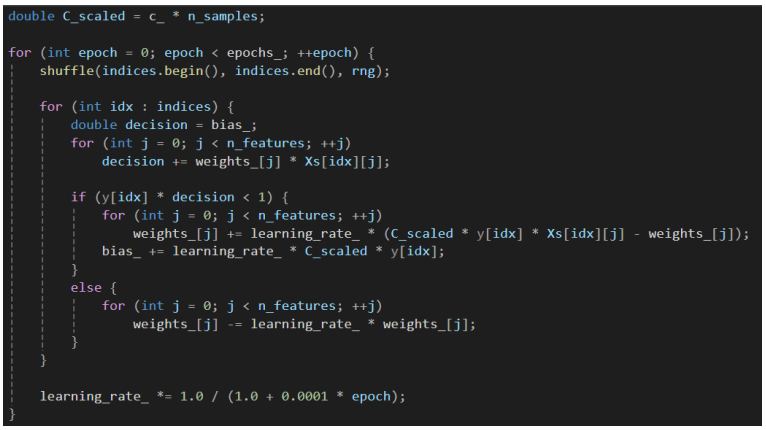
```
int main() {
    cout << "Starting SVM C++\n";
    cout << "Collecting data\n";
    WineDataProcessor train_processor;
    vector<Wine> wines = train_processor.GetWines();
    auto [X_train, y_train] = PrepareData(wines);

    double c = 1.0;
    double lr = 0.001;
    int epochs = 5000;
    cout << "Launching SVM\n";
    LinearSVM svm(c, lr, epochs);

    cout << "Trainig...\n";
    svm.Fit(X_train, y_train);

    cout << "Finding support vectors\n";
    cout << "Saving data\n";
    SaveWeights(svm.GetWeights(), svm.GetBias(), "weights.csv");

    RunPythonScript();
}
```

в main функции мы готовим датасет, разбиваем на данные для обучения и запускаем алгоритм. Дальше идут сохранения полученных данных в файл и запуск скрипта Python для отрисовки графиков

Рассмотрим главную функцию Fit, которая обучает нашу модель

Функция Fit стандартизирует признаки, затем обучает линейную SVM-модель методом стохастического градиентного спуска на hinge-loss с L2-регуляризацией, применяя «шаги» для каждого образца, уменьшая learning rate по эпохам.

```
void Fit(const vector<vector<double>>& X, const vector<int>& y) {
    int n_samples = X.size();
    int n_features = X[0].size();

    mean_.assign(n_features, 0.0);
    std_.assign(n_features, 0.0);
    for (int j = 0; j < n_features; ++j) {
        for (int i = 0; i < n_samples; ++i)
            mean_[j] += X[i][j];
        mean_[j] /= n_samples;

        double var = 0.0;
        for (int i = 0; i < n_samples; ++i)
            var += pow(X[i][j] - mean_[j], 2);
        std_[j] = sqrt(var / n_samples);
    }

    vector<vector<double>> Xs(n_samples, vector<double>(n_features));
    for (int i = 0; i < n_samples; ++i)
        Xs[i] = Standardize(X[i]);

    weights_.assign(n_features, 0.0);
    bias_ = 0.0;

    vector<int> indices(n_samples);
    iota(indices.begin(), indices.end(), 0);
    mt19937 rng(42);
```



```
double C_scaled = c_ * n_samples;

for (int epoch = 0; epoch < epochs_; ++epoch) {
    shuffle(indices.begin(), indices.end(), rng);

    for (int idx : indices) {
        double decision = bias_;
        for (int j = 0; j < n_features; ++j)
            decision += weights_[j] * Xs[idx][j];

        if (y[idx] * decision < 1) {
            for (int j = 0; j < n_features; ++j)
                weights_[j] += learning_rate_ * (C_scaled * y[idx] * Xs[idx][j] - weights_[j]);
            bias_ += learning_rate_ * C_scaled * y[idx];
        }
        else {
            for (int j = 0; j < n_features; ++j)
                weights_[j] -= learning_rate_ * weights_[j];
        }
    }

    learning_rate_ *= 1.0 / (1.0 + 0.0001 * epoch);
}
```

в начале мы готовим данные и собираем размеры
int n_samples = X.size();
int n_features = X[0].size();

Вычисляем среднее и стандартное отклонение (mean_, std_)

```
mean_.assign(n_features, 0.0);
std_.assign(n_features, 0.0);
for (int j = 0; j < n_features; ++j) {
    for (int i = 0; i < n_samples; ++i)
        mean_[j] += X[i][j];
    mean_[j] /= n_samples;

    double var = 0.0;
    for (int i = 0; i < n_samples; ++i)
        var += pow(X[i][j] - mean_[j], 2);
    std_[j] = sqrt(var / n_samples);
}
```

Стандартизация признаков (вычитание среднего и деление на std) делает признаки сопоставимыми по масштабу. Это важно для SVM/SGD: без этого признаки с большим масштабом «доминируют» в градиенте, обучение медленнее и неустойчиво.

---

3) Стандартизация всех образцов

```
vector<vector<double>> Xs(n_samples, vector<double>(n_features));
for (int i = 0; i < n_samples; ++i)
    Xs[i] = Standardize(X[i]);
```

Standardize(x) реализует: x_std[j] = (x[j] - mean_[j]) / std_[j] (если std_[j] очень маленькое — делим на 1, чтобы не на 0).
Что даёт: все строки в Xs — в стандартизованном (z-score) пространстве. Дальше обучение идёт на этих Xs.

---

4) Инициализация параметров

```
weights_.assign(n_features, 0.0);
```

bias_ = 0.0;
weights_ — вектор коэффициентов w (длина = число признаков), начинается с нуля.
bias_ — смещение b, тоже с 0.

Почему так: простая, детерминированная инициализация; с нуля удобно для объяснений и воспроизводимости.

---

5) Подготовка порядка прохода по данным (shuffle)
vector<int> indices(n_samples);
iota(indices.begin(), indices.end(), 0);
mt19937 rng(42);
indices = [0,1,2,...,n-1];
rng(42) — фиксированный seed (42) для детерминированного поведения shuffle.

Зачем: в SGD порядок примеров влияет на траекторию оптимизации; мы перемешиваем и хотим детерминированный результат при повторных запусках.

---

6) Масштабирование C
double C_scaled = c_ * n_samples;

Пояснение:
В класической SVM-формулировке мы минимизируем 1/2 ||w||^2 + C * Σ hinge_i.

Когда реализуем SGD по одному примеру, мы часто масштабируем C либо не масштабируем — разные реализации используют разные конвенции. Здесь вы умножаете C на n_samples, чтобы сила штрафа при одиночном шаге соответствовала масштабу суммарной суммы по всем примерам. Это попытка согласовать C с тем, как он интерпретируется в другой реализации (sklearn).

---

7) Основной цикл обучения — эпохи и SGD

```
for (int epoch = 0; epoch < epochs_; ++epoch) {
    shuffle(indices.begin(), indices.end(), rng);

    for (int idx : indices) {
        double decision = bias_;
        for (int j = 0; j < n_features; ++j)
            decision += weights_[j] * Xs[idx][j];

        if (y[idx] * decision < 1) {
            for (int j = 0; j < n_features; ++j)
                weights_[j] += learning_rate_ * (C_scaled * y[idx] * Xs[idx][j] - weights_[j]);
            bias_ += learning_rate_ * C_scaled * y[idx];
        }
        else {
            for (int j = 0; j < n_features; ++j)
                weights_[j] -= learning_rate_ * weights_[j];
        }
    }
```

```
    learning_rate_ *= 1.0 / (1.0 + 0.0001 * epoch);
}
```

Разберём это по частям.

7.1. decision — значение линейной функции
decision = b + w^T x_std (в коде: сначала bias_, затем добавляем weights_[j] * Xs[idx][j]).
Это то, что машинная модель использует для прогнозирования: если decision >= 0 — модель даёт метку +1, иначе −1.
y[idx] * decision — это margin: положительный, если классификация правильная, и большая по величине, если уверенна.
7.2. Условие if (y * decision < 1) — где действует hinge-loss
Hinge loss для одного примера: L = max(0, 1 - y * decision).
Если y * decision >= 1 — loss = 0 (точка за пределами или на границе маржи).
Если y * decision < 1 — точка либо внутри маржи, либо ошибочно классифицирована.
Таким образом, условие (y*decision < 1) означает «эта точка вносит вклад в loss и надо обновить параметры так, чтобы исправить это».

7.3. Обновление при нарушении маржи (внутри margin или ошибка)
Код:
weights_[j] += learning_rate_ * (C_scaled * y[idx] * Xs[idx][j] - weights_[j]);
bias_ += learning_rate_ * C_scaled * y[idx];
Математика (пошагово):
Целевая функция для одного примера (вклад в суммарную цель): f(w) = 1/2 ||w||^2 + C * max(0, 1 - y (w^T x + b)).
Если пример нарушает маржу, градиент по w равен: ∂f/∂w = w - C * y * x.
(первое слагаемое — производная 1/2||w||^2, второе — производная C*(1 - y·(w·x+b)) = -C*y*x).
Шаг градиентного спуска (SGD): w := w - lr * ∂f/∂w = w - lr*(w - C*y*x) = w + lr*(C*y*x - w).
Это ровно то, что код делает (weights_[j] += learning_rate_* (C_scaled*y*X - weights_[j])) — только с C_scaled.
Для смещения b: градиент ∂f/∂b = -C*y (при нарушении), шаг b := b - lr * (-C*y) = b + lr * C * y. Код делает bias_ += learning_rate_ * C_scaled * y.
Важно: это — корректный SGD-шаг для hinge + L2 регуляризации.

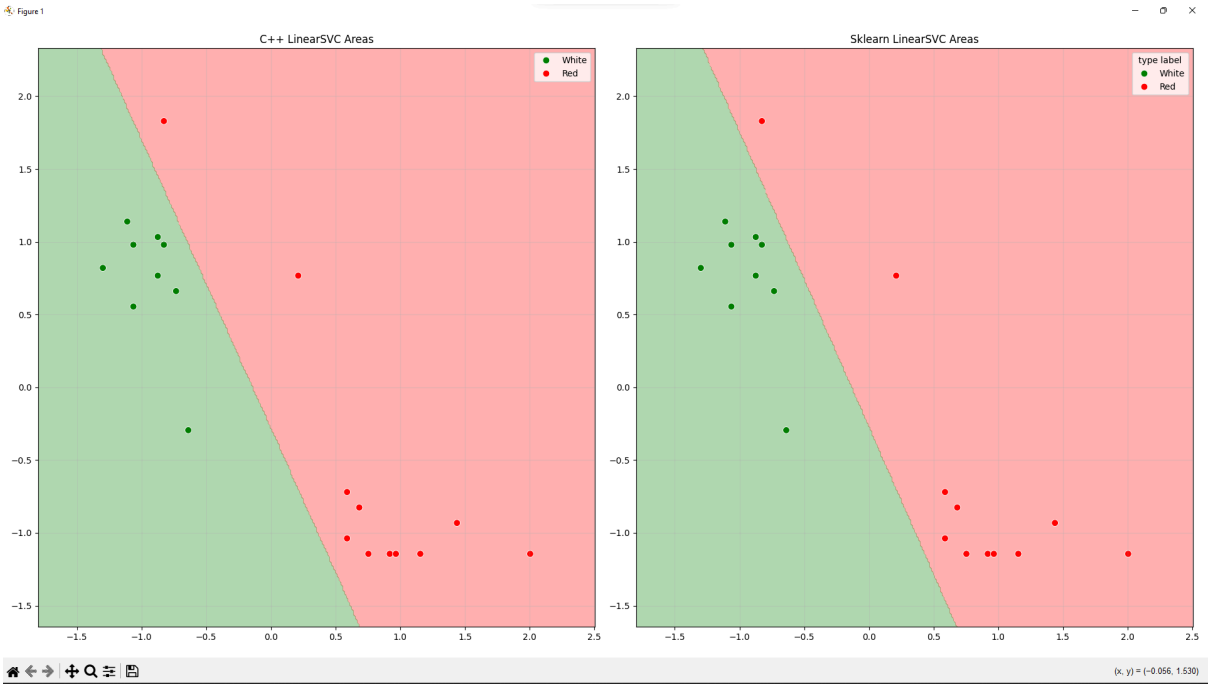
7.4. Обновление, когда маржа соблюдена
Код:
weights_[j] -= learning_rate_ * weights_[j];
Это эквивалентно w := w - lr * w = (1 - lr) * w — механизм весовой «усадки» или weight decay (L2 only term, когда loss на данном примере = 0). Это реализует регуляризатор 1/2||w||^2 даже при отсутствии hinge-градиента.
Примечание: эффект — веса постепенно сжимаются к нулю, что предотвращает их разрастание.

7.5. Снижение шага обучения (learning rate decay)
learning_rate_ *= 1.0 / (1.0 + 0.0001 * epoch);

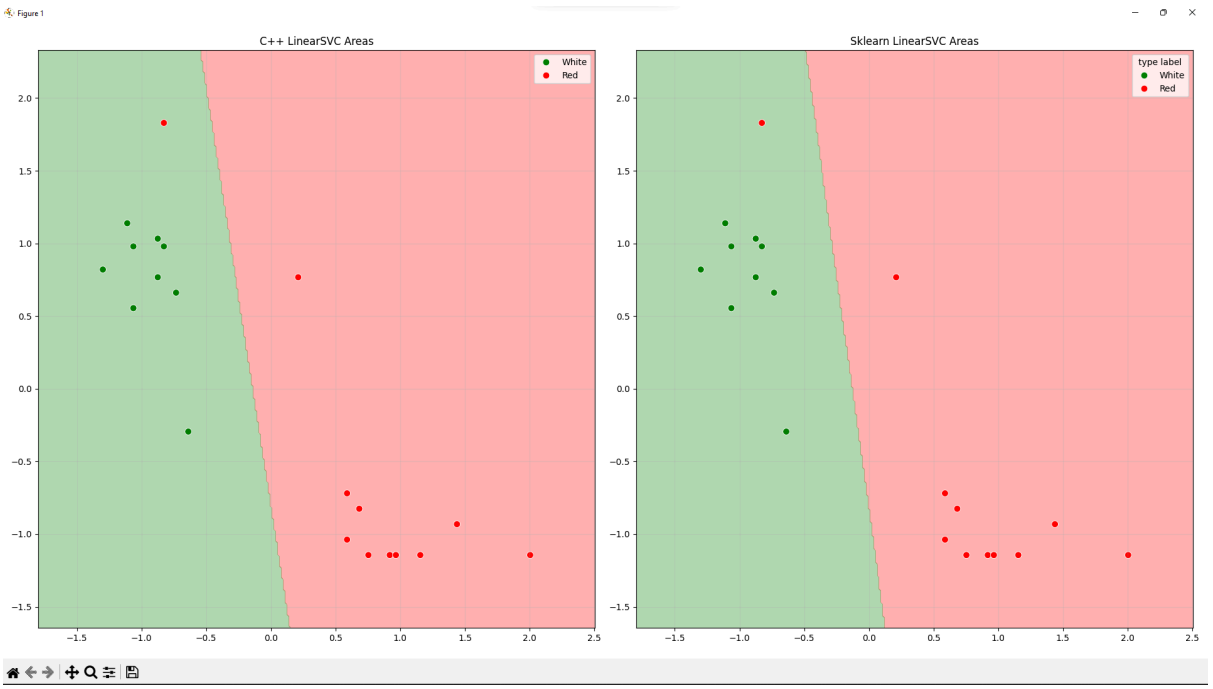
Это делает шаг обучения меньше по мере роста числа эпох, помогает сходимости (меньшие колебания при приближении к минимуму).
Формула: lr_new = lr_old / (1 + 0.0001 * epoch). Она уменьшает lr медленно.

Стандартизация нужна, чтобы признаки не «соревновали» по масштабу; это делает обучение более стабильным.
SGD даёт простой итеративный способ приближенно минимизировать цель; по одному примеру — экономно и быстро для больших данных.
Шаг слагается из двух частей: с одной стороны регуляризатор w тянет вектора к нулю; с другой — слагаемое C*y*x тянет веса так, чтобы правильно классифицировать текущий пример.
Мешание (shuffle) нужно, чтобы SGD не застревал в циклических паттернах и получал «разнообразные» градиенты.
Decay lr — стандартная практика улучшения сходимости: сначала делаем лестный шаг, потом уменьшаем, чтобы «дотянуться» до минимума.

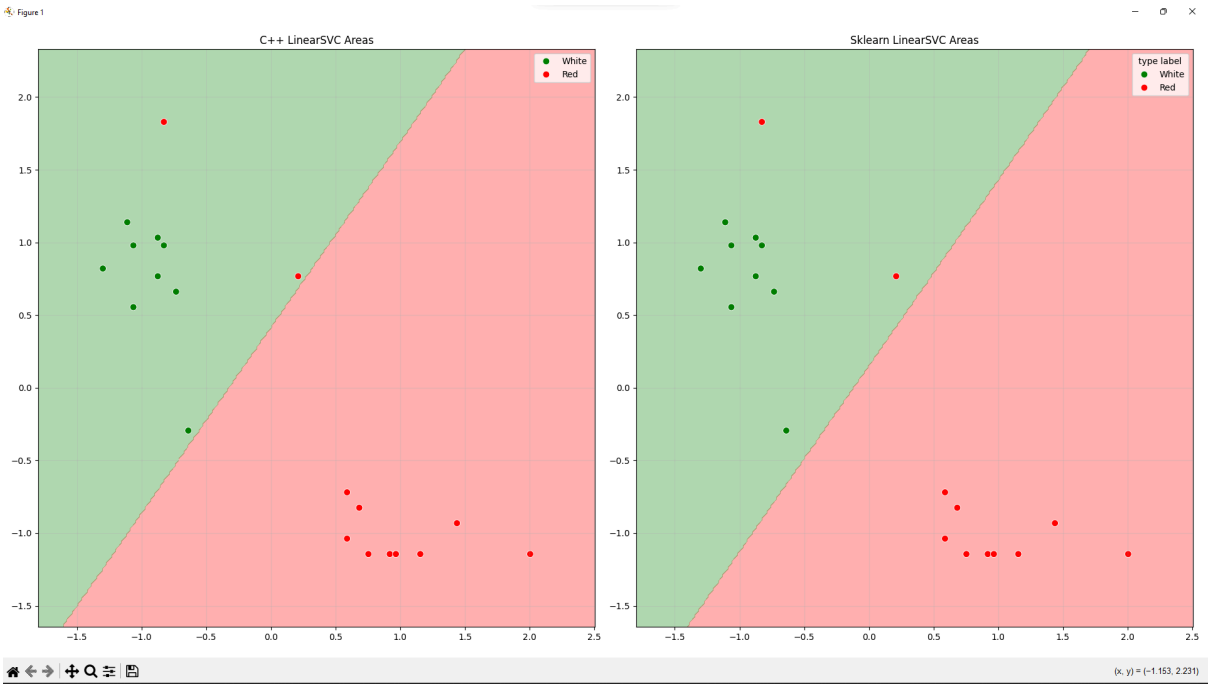
Результаты работы программы для линейно разделимого случая
C=100
LR=0.001
epoch=5000
wines_count=26



C=1
LR=0.001
epoch=5000
wines_count=26



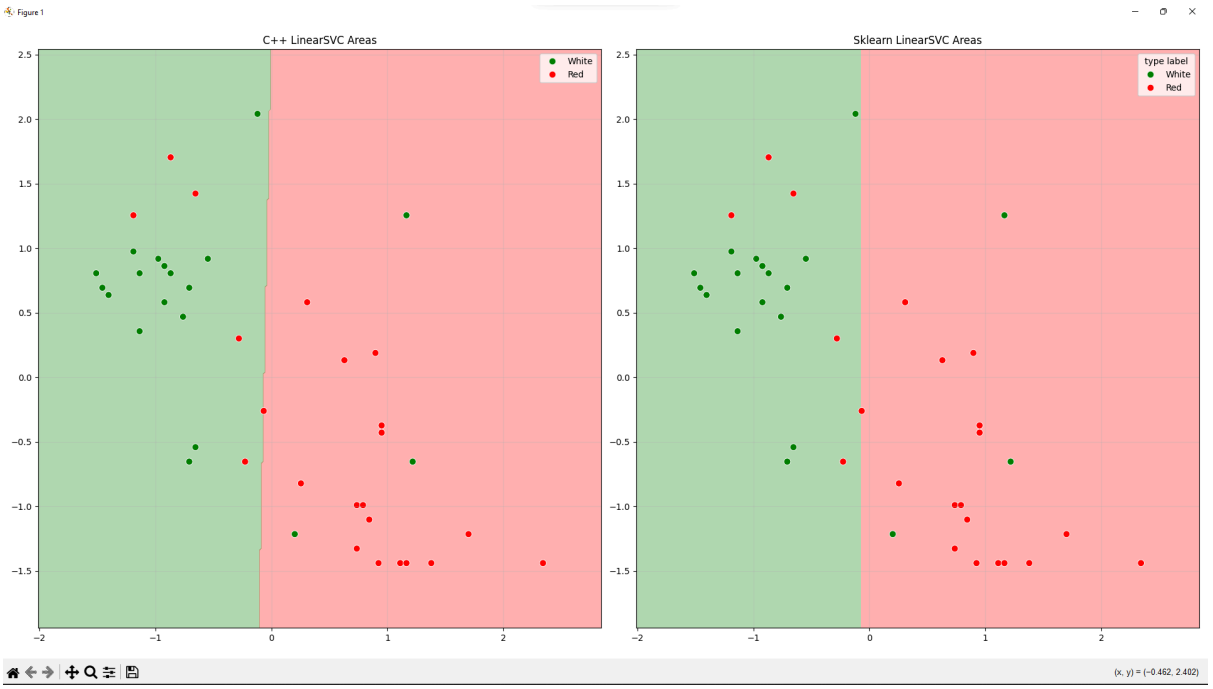
C=0.01
LR=0.001
epoch=5000
wines_count=26



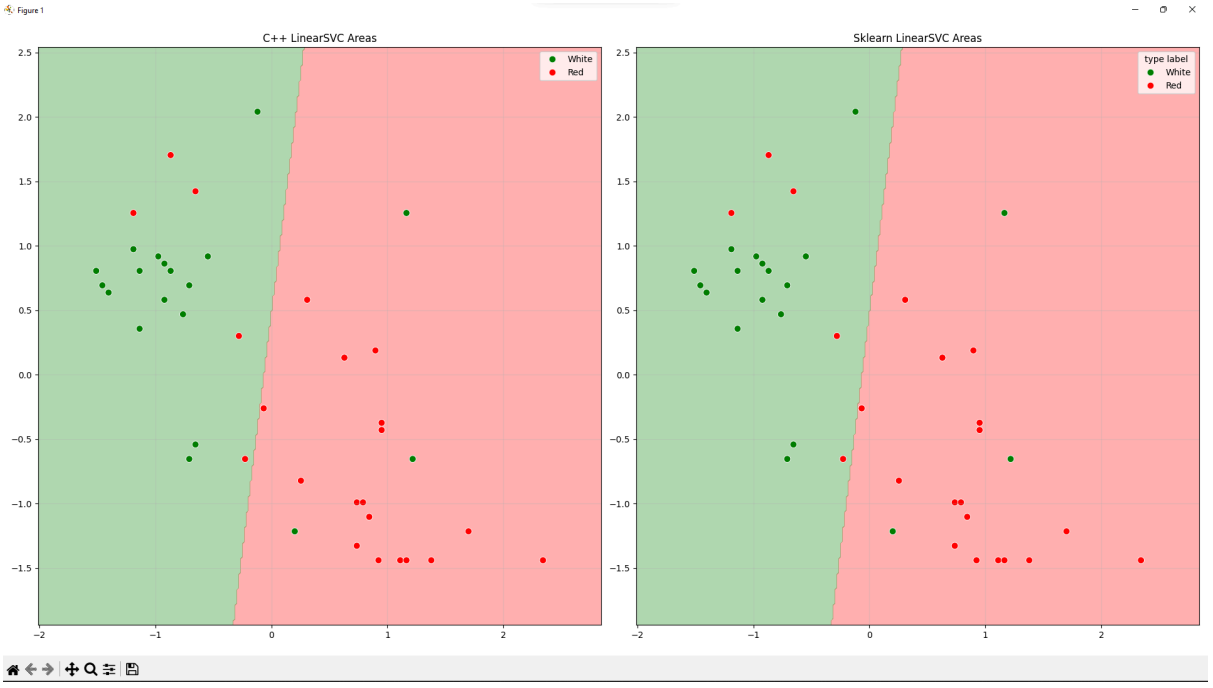
о чем нам говорят данные графики
с уменьшением C, увеличивается отступ от линии разделения, соответственно модель классифицирует данные, увеличивая допуск для опорных векторов

Результаты работы программы для линейно неразделимого случая
C=0.01
LR=0.001
epoch=5000

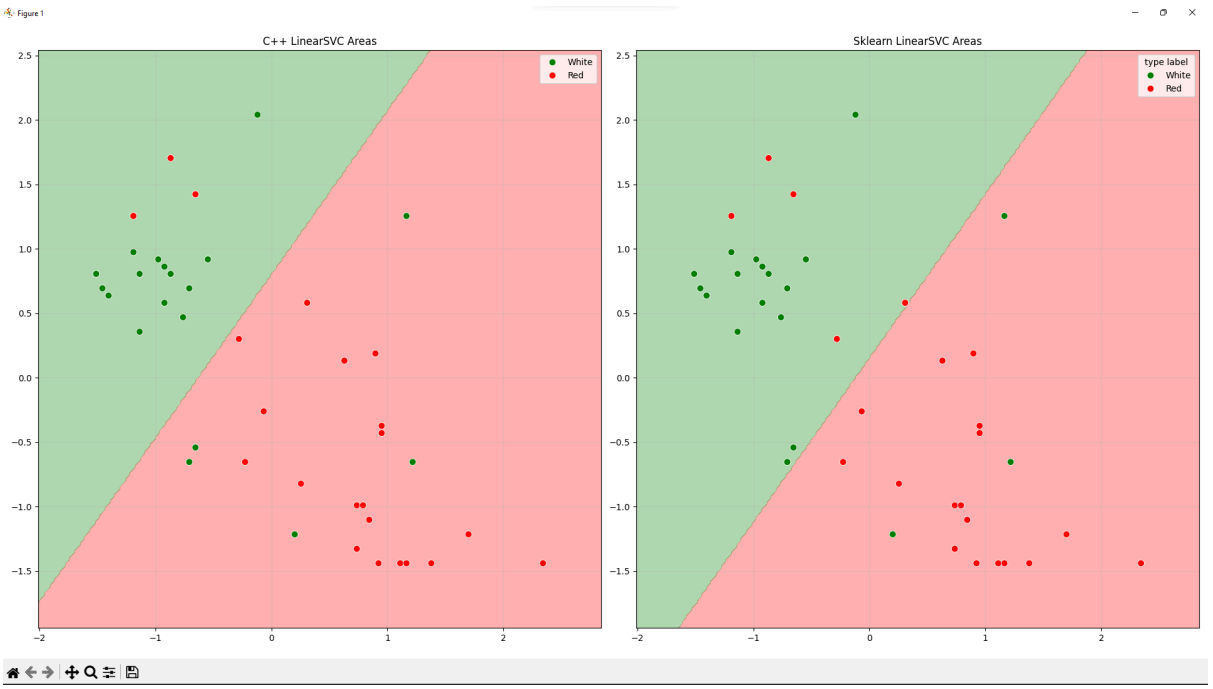
wines_count=50



C=1.0
LR=0.001
epoch=5000
wines_count=50



C=0.01
LR=0.001
epoch=5000
wines_count=50



для линейно неразделимого случая видно, что модель может чаще ошибаться и с уменьшением параметра C, расхождения двух почти одинаковых моделей все больше

7) Преимущества и недостатки метода опорных векторов:

Преимущества метода опорных векторов

Эффективность в высокомерных пространствах

- SVM работает даже когда количество признаков значительно превышает количество (p >> n)
- В отличие от многих других алгоритмов, не страдает от "проклятия размерности" в той же степени

Теоретическая обоснованность и глобальный оптимум

- Задача SVM сводится к задаче выпуклой квадратичной оптимизации
- Гарантированно находим глобальный минимум (в отличие от нейросетей, где возможны локальные минимумы)

Универсальность благодаря ядерному трюку (Kernel Trick)

- Линейно неразделимые данные можно разделять в пространствах более высокой размерности
- Популярные ядра: линейное, полиномиальное, RBF (радиально-базисное), сигмоидальное
- Можно создавать собственные ядерные функции для специфических задач

Экономичность использования памяти

- В финальной модели хранятся только опорные векторы (обычно небольшая часть данных)
- Для прогноза нужны только эти векторы, а не вся обучающая выборка

Робастность к переобучению в высокомерных пространствах

- Регуляризация через параметр C позволяет контролировать компромисс между точностью и сложностью модели
- Максимизация зазора (margin) естественным образом предотвращает переобучение

Четкие геометрические интерпретации

- Интуитивно понятная концепция разделяющей гиперплоскости и зазора
- Легко визуализировать для 2D и 3D случаев

Недостатки и ограничения SVM

- Вычислительная сложность для больших наборов данных
- Время обучения растет квадратично $O(n^2)$ или кубически $O(n^3)$ от размера выборки
- Для наборов > 100,000 требуется значительные вычислительные ресурсы

Чувствительность к шуму и выбросам

- Выбросы могут сильно влиять на положение разделяющей гиперплоскости
- При малом C модель слишком чувствительна к шуму, при большом C - теряет обобщающую способность

Сложность интерпретации модели

- Веса модели в исходном пространстве не всегда имеют ясную интерпретацию
- При использовании нелинейных ядер модель становится "черным ящиком"

Отсутствие прямых вероятностных оценок

- SVM выдает только метки классов или расстояния до гиперплоскости
- Для получения вероятностей требуется дополнительная калибровка (platt scaling) с k-fold cross-validation

Сложность выбора гиперпараметров

- Для RBF-ядра нужно подбирать C и γ (gamma)
- Требуется тщательная настройка сеточным поиском или байесовской оптимизацией

Неэффективность для очень больших наборов данных

- Требует хранения матрицы ядер размера n×n в памяти
- Для данных > 1GB возникают проблемы с памятью

Плохая работа с несбалансированными классами

- SVM стремится максимизировать общий зазор, игнорируя дисбаланс классов
- Требует дополнительной балансировки весов или семплирования

ссылки

https://proproprogs.ru/ml/ml-vvedenie-v-metod-opornyh-vektorov
https://blog.skillfactory.ru/svm-metod-opornyh-vektorov/
https://scikit-learn.org/stable/modules/svm.html
https://habr.com/ru/companies/ods/articles/484148/