Final Course Assignment

Mark Niehues, Stefaan Hessmann
Mathematical Aspects in Machine Learning

July 14, 2017

1 Introduction

In the past course we dealt with the broad mathematical foundations of machine learning. To get an idea of what the consequences of those mathematical theorems and approaches are and to get in touch with the standard Python tools, we have evaluated an comparatively easy data science example found on kaggle.com. Since this was our first machine learning project, we decided to deal with an rather simple problem. The example dataset [**Kaggle2017**] consists of the historic passenger records of the disastrous Titanic maiden voyage in 1912. The goal in this Challenge was to predict if a passenger survived the accident based on informations like for example age, sex and payed ticket fare. Therefore it is in terms of machine learning a *classification problem* with two classes: survived and not survived.

Inspired by sample solutions from the website, we first took a deeper look on the dataset and tried to select the most significant influences by reviewing the statistical properties of the dataset. In the following we implemented an naive Sequential Minimal Optimization (SMO) algorithm and ran a few tests with them in order to finally compare the results with other machine learning algorithms.

2 Applying Machine Learning Methods on the Titanic Disaster

2.1 Dataset

The given dataset consists of a CSV-file containing data of 891 passengers. The dataset contains an ID for every passenger, a label if the passenger has survived the disaster and the features that are described in table 1. It can be noticed that some of the features are incomplete.

After loading the dataset it is necessary to process the data for our learning machine. Therefore the different features will be investigated to select meaningful features and the missing data needs to be handled.

2.2 Feature: Sex

The sex-feature divides the passengers into the categories ’female’ and ’male’. Figure 1 shows the probability of survival for male and female passengers. It is obvious that females had a much higher probability to survive than the male passengers. ’Sex’ seems to be a useful feature for the learning machine.

Table 1: Features and their amount of missing data.

PassengerId	Unique ID for every passenger	0.0 %
Survived	Survived (1) or died (0)	0.0 %
Pclass	Passenger's class	0.0 %
Name	Passenger's name	0.0 %
Sex	Passenger's sex	0.0 %
Age	Passenger's age	**19.87 %**
SibSp	Number of siblings/spouses aboard	0.0 %
Parch	Number of parents/children aboard	0.0 %
Ticket	Ticket number	0.0 %
Fare	Ticket-price	0.0 %
Cabin	Number of the passenger's cabin	**77.10 %**
Embarked	Port of embarkation	**0.22 %**

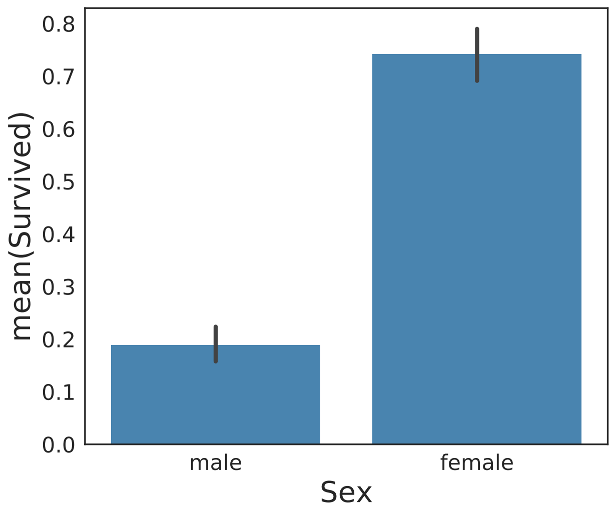


Figure 1: Survival probability depending on the passenger's sex.

2.3 Feature: Age

The impact of a passengers' age on their probability to survive categorized by their sex is pointed out

3 Implementation of an easy SMO Algorithm

3.1 Brief Introduction

To get a better understanding of what a Support Vector Machine does we decided to implement one on our own using several publications. Most of them were based on the important paper of Platt [**platt**] where he introduced a new approach for the calculation of the Support Vectors that improves the performance a lot. This algorithm is called Sequential Minimal Optimization. Performance was not the highest priority for us but instead understandability and the costs of implementation. Therefore we implemented a less complex version of the algorithm presented in Platt's paper.

As the mathematical background of the SVMs has been explained in the lecture and might be considered a standard solution for machine learning, the following introduction focuses on the main equations.

The initial problem is a linear separable dataset with the labels $y_i \in \{-1, 1\}$. The classifier that the SVM is supposed to compute will have the form

$$f(x) =< \omega, x > +b \tag{1}$$

Now suppose we have a separating hyperplane and w is perpendicular. The main task of the SVM is to maximise the closest perpendicular distance between the hyperplane and the two classes. This is down by the following constraints

$$f(x) \geq 1 \text{ for } y_i = +1 \tag{2}$$

$$f(x) \leq 1 \text{ for } y_i = -1 \tag{3}$$

Consequently do points that lie on the hyperplane satisfy $f(x) = 0$.

From these set of equations follows that the minimal distance from the hyperplane to one of the datapoints is $d = \frac{1}{|\omega|}$ which shall be maximized. Introducing an additional factor that allows but penalizes non separable noise and reformulating the problem with Lagrange multipliers (the α_i) we get the following problem:

$$\max_\alpha W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i < x^{(i)}, x^{(j)} > \tag{4}$$

$$\text{subject to } 0 \leq \alpha_i \leq C, i = 1, ..., m \tag{5}$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \tag{6}$$

For the presented problem the Kuhn Tucker conditions define the α_i that represent an optimal solution. The KKT conditions are

$$\alpha_i = 0 \implies y^{(i)}(< \omega, x^{(i)} > +b) \geq 1 \quad (7)$$

$$\alpha_i = C \implies y^{(i)}(< \omega, x^{(i)} > +b) \leq 1 \quad (8)$$

$$0 \geq \alpha_i \geq C \implies y^{(i)}(< \omega, x^{(i)} > +b) = 1 \quad (9)$$

To deal with linearly non separable data, the scalar products can be replaced by kernel functions $kernel(x_i, x_j)$.

3.2 Description of the Implementation

Instead of trying to maximize the whole set of α the SMO algorithm exploits that the maximum will be reached when pairs α_i, α_j fulfil the KKT conditions (while it needs to be at least a pair, since the conditions imply linearity of two α values). Thus the SMO algorithm selects two α parameters (that do not meet the KKT conditions) and optimizes them. Afterwards the b value gets adjusted according to the new values.

A big part of the actual publication from Platt deals with the heuristic of how two choose the α_i and α_j since this is a critical factor for the pace of its convergence as the number of possible pairs in a setup with m features is $m(m-1)$. Accordingly the amount of time it takes to find the *critical* values is decisive for the algorithms performance.

Nevertheless, in this assignment a very simple heuristic is implemented in order to keep the code simple and understandable: the pairs are just purely randomly selected.

Fig. 5 in the Appendix shows the pseudo code of our implementation while Listing. 1 in the Appendix shows the actual implementation in Python using Numpy. Basically the algorithm consists of an outer and inner loop. The inner loop iterates through the $alpha_i$ and checks weather it violates the KKT conditions. If this is the case, randomly a second parameter α_j is selected and will be adjusted using that the optimal α_j is given by

$$\alpha_j^{'} = \alpha_j - \frac{y^{(j)}(E_i - E_j)}{\eta} \quad (10)$$

where η can be interpreted as the second derivative of the Loss function function $W(\alpha)$ [**smo**]. E_i is the current error on the sample x_i. After that the new parameter is cropped to boundaries that follow from equation 9 and 6. After the opposing parameter is calculated exploiting the linearity the threshold can be updated by using the classifier function $f(x)$ and either one $alpha$ that lays within $(0, C)$ or if this is true for both, their arithmetic mean.

The outer loop counts how often the inner loop fails to find a partner for optimization or that yields to no significant (significance is defined by the user) changes. The algorithm terminates when a certain, user defined number of passes is reached.

3.3 Comparison with SciKit SVM

The implementation does not make a lot use of Numpy's vectorization skills and therefore performs poor even considering that it is Python code. Still it reaches satisfying scores with the titanic train set in comparison to other SciKit algorithms as can be seen in fig. 2.

On the other hand and less surprisingly, the performance is quite poor compared to the SVM Module from the SciKit framework[1]. One can deduce from figure 3 that the prefactor as well as the exponential behaviour is significantly worse.

It turned out that the quality of the result the implemented algorithm yields heavily depends on the parameter that determines after how many *changeless* runs it terminates. As you can see in figure 4, the algorithm yields

[1] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

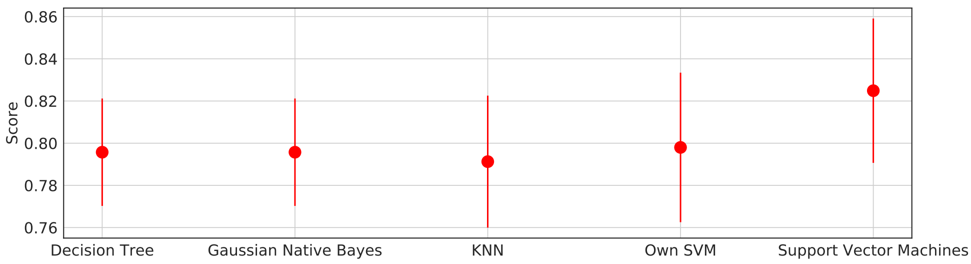


Figure 2: Comparison of different ML Algorithms and their score using the Titanic training set and K-Fold validation.

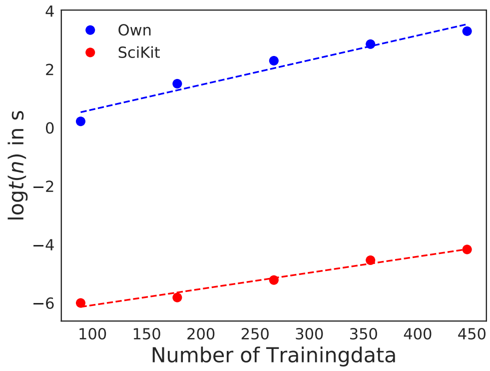
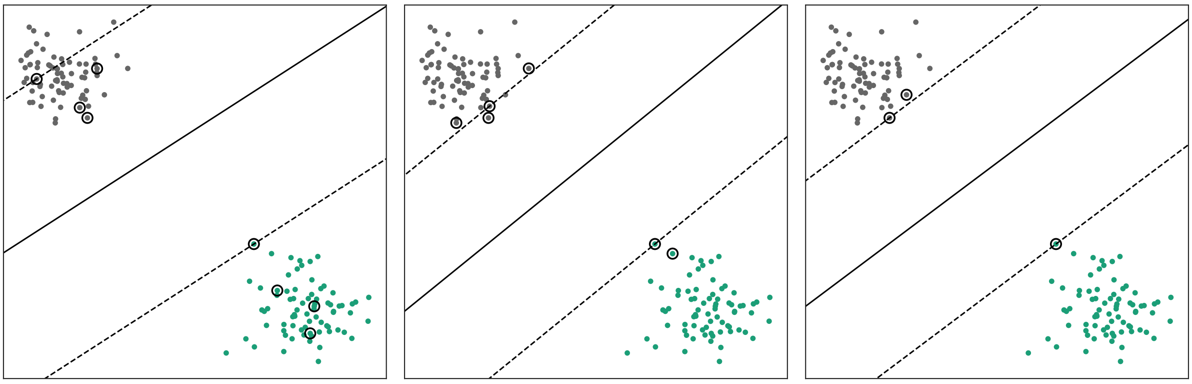


Figure 3: Comparison the self made implementation and the optimized SciKit implementation. The different intercepts (δ 6) as well as the different slopes (factor 1.6) express the smaller prefactor and exponent of runtime of the SciKit's implementation.

a lot support vectors that do not lie within the margin when it stops after one iteration causes no change. The quality of the solution increases when the number of passes is larger. This behaviour results from the property of the SVM which is that it only optimizes data points that lay within the margin. Since this "quickly" becomes a small number of data points it is unlikely that the random selection finds a pair to optimize.



(a) `max_passes=1` (b) `max_passes=15` (c) `max_passes=30`

Figure 4: Result of our SVM depending on number of loops without any chaning α that terminate the alogrithm.

Appendix

4 Pseudo Code of the SMO algorithm

5 Code

Listing 1: The main procedure of the SMO algorithm

```
    def fit(self, X_train, y_train, max_passes=10, tol=1e-8, kernel="rbf"):
        """
        Fits alpha values and the threshold b with given Training data

        Parameters
        ----------
        X_train: Numpy Array or Pandas Data Set
            Training Data Set
        y_train: numpy.ndarray Array oder Pandas Series
            Labels for Training
        max_passes: int
            Maximal Number of runs without any change in the alpha values that
            determines the end of fitting
        tol: float
            Tolerance on estimated Error
        """
        # Convert arguments to numpy arrays if they are in pandas datastructures
        if type(X_train) == pd.DataFrame:
            self.X_train = X_train.as_matrix()
        if type(y_train) == pd.DataFrame or type(y_train) == pd.Series:
            self.y_train = y_train.as_matrix()

        self.n_test_samples = len(y_train)

        # Set Kernel
        self.kernel = self.kernel_set.get_kernel(kernel)
        if kernel == "rbf" and self.kernel_set.gamma is None:
            self.kernel_set.gamma = 1 / self.n_test_samples

        # QUICK AND DIRTY
        # Detect if the labels are [1, 0] instead of [1, -1] and correct them
        if np.min(self.y_train) == 0:
            self.min_label = 0
```

Input:

C: regularization parameter
tol: numerical tolerance
max_passes: max # of times to iterate over α's without changing
$(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$: training data

Output:

$\alpha \in \mathbb{R}^m$: Lagrange multipliers for solution
$b \in \mathbb{R}$: threshold for solution

- Initialize $\alpha_i = 0, \forall i, \quad b = 0$.
- Initialize $passes = 0$.
- **while** $(passes < max\_passes)$
  - $num\_changed\_alphas = 0$.
  - **for** $i = 1, \ldots m$,
    - Calculate $E_i = f(x^{(i)}) - y^{(i)}$ using (2).
    - **if** $((y^{(i)} E_i < -tol \quad \&\& \quad \alpha_i < C) \mid\mid (y^{(i)} E_i > tol \quad \&\& \quad \alpha_i > 0))$
      - Select $j \neq i$ randomly.
      - Calculate $E_j = f(x^{(j)}) - y^{(j)}$ using (2).
      - Save old α's: $\alpha_i^{(\mathrm{old})} = \alpha_i$, $\alpha_j^{(\mathrm{old})} = \alpha_j$.
      - Compute L and H by (10) or (11).
      - **if** $(L == H)$
        continue to next i.
      - Compute η by (14).
      - **if** $(\eta >= 0)$
        continue to next i.
      - Compute and clip new value for α_j using (12) and (15).
      - **if** $(|\alpha_j - \alpha_j^{(\mathrm{old})}| < 10^{-5})$
        continue to next i.
      - Determine value for α_i using (16).
      - Compute b_1 and b_2 using (17) and (18) respectively.
      - Compute b by (19).
      - $num\_changed\_alphas := num\_changed\_alphas + 1$.
    - **end if**
  - **end for**
  - **if** $(num\_changed\_alphas == 0)$
    $passes := passes + 1$
  - **else**
    $passes := 0$
- **end while**

Figure 5: Pseudo Code of the implemented SMO algorithm. Taken from[**smo**]

```
            self.y_train = self.y_train * 2 - 1

        # Create array for storing the alpha values
        self.alpha = np.zeros(self.n_test_samples)

        passes = 0  # Counting runs without changing a value
        while passes < max_passes:
```

```python
        changed_alpha = False

        for i in range(self.n_test_samples):
            # Calculate the error with the current alpha
            y_i = self.y_train[i]
            E_i = self.dec_func(self.X_train[i]) - y_i

            # If accuracy is not satisfying yet
            if (y_i * E_i < -tol and self.alpha[i] < self.C) or \
                    (y_i * E_i > tol and self.alpha[i] > 0):

                # Randomly choose another alpha to pair
                j = randint(0, self.n_test_samples - 1)
                y_j = self.y_train[j]

                # Saving old alphas
                a_i_old = self.alpha[i]
                a_j_old = self.alpha[j]

                # Calculate the error of the other alpha
                E_j = self.dec_func(self.X_train[j]) - y_j

                # Calculate the valid limits that are a consequence of the linear dependence
                L, H = self.calc_limits(i, j)
                if L == H:
                    continue

                # Evaluate the second derivative of the Loss function for optimizing
                # Eta should be negative to make shore, what we are approaching a maximum
                kernel_i_i = self.kernel_ind(i, i)
                kernel_j_j = self.kernel_ind(j, j)
                kernel_i_j = self.kernel_ind(i, j)
                eta = 2 * kernel_i_j - kernel_i_i - kernel_j_j
                if eta >= 0:
                    continue

                a_j = a_j_old - y_j * (E_i - E_j) / eta

                # Clip the new alpha to the limits
                if a_j > H:
                    a_j = H
                elif a_j < L:
                    a_j = L

                # Check if the change is not negligible
                if np.abs(a_j - a_j_old) < tol:
                    continue

                # Calculate the new value for a_i from the new value of a_j
                a_i = a_i_old + y_i * y_j * (a_j_old - a_j)

                # Apply tolerance
                if a_i < tol:
                    a_i = 0
                elif a_i > self.C - tol:
                    a_i = self.C

                # Calculate new threshold
                d_a_i = y_i * (a_i - a_i_old)
                d_a_j = y_j * (a_j - a_j_old)
                b_1 = self.b - E_i - d_a_i * kernel_i_i - d_a_j * kernel_i_j
                b_2 = self.b - E_j - d_a_i * kernel_i_j - d_a_j * kernel_j_j

                if self.C > a_i > 0.:
                    self.b = b_1
                elif self.C > a_j > 0:
```

```
                    self.b = b_2
                else:
                    self.b = (b_1 + b_2)/2

                # Replace the alpha values
                self.alpha[i] = a_i
                self.alpha[j] = a_j
                changed_alpha = True

        passes += 1 if changed_alpha else 0
```