Data Mining und Maschinelles Lernen

Prof. Kristian Kersting
Steven Lang
Johannes Czech

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2021
10. Juni 2021
Übungsblatt 8

Diese Übung wird am 17.06.2021 um 13:30 Uhr besprochen und nicht bewertet.

Benötigte Dateien
Alle benötigten Datensätze und Skriptvorlagen finden Sie in unserem Moodle-Kurs:
https://moodle.informatik.tu-darmstadt.de/course/view.php?id=1058

8.1 Supportvektormaschinen

Supportvektormaschinen (engl. support vector machines) wurden 1995 von Cortes und Vapnik [1] eingeführt, obwohl ihre Idee schon früher diskutiert wurde. Nachfolgend werden ihre Prinzipien kurz erläutert. Um einfach zu beginnen, wird angenommen, dass die Eingabedaten aus zwei Klassen mit $y_i \in \{-1, 1\}$ bestehen und linear trennbar sind. Jeder Datenpunkt wird als ein Vektor im Eingabefeature-Raum gesehen. Die SVM-Methode zielt darauf ab, die Parameter $\mathbf{w}$ und b für die lineare Entscheidungsfunktion zu finden

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

mit der Hyperebene $\mathbf{w}^T\mathbf{x} + b = 0$, die die Datenpunkte trennt:

$$\mathbf{w}^T\mathbf{x}_i^+ + b \geq 1 \tag{1}$$
$$\mathbf{w}^T\mathbf{x}_i^- + b \leq -1\,, \tag{2}$$

sodass alle positiven Beispiele $\mathbf{x}_i^+$ auf der einen Seite der Hyperebene liegen, während alle negativen Punkte $\mathbf{x}_i^-$ auf der anderen Seite liegen. Der Abstand eines Punktes $\mathbf{x}_i$ zur Hyperebene ist gegeben durch

$$\frac{y_i y(\mathbf{x}_i)}{||\mathbf{w}||} = \frac{y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right)}{||\mathbf{w}||} \tag{3}$$

und die Spanne wird durch den orthogonalen Abstand zum nächstgelegenen Punkt x_i definiert. Das Ziel ist es, $\mathbf{w}$ und b so zu optimieren, dass die Spanne maximiert wird. Dies kann durch Lösen folgenden Optimierungsproblems erreicht werden

$$\text{argmax}_{\mathbf{w},b}\{\frac{1}{||\mathbf{w}||}\min_i[y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right)]\} \tag{4}$$

w.r.t. $y_i\left(\mathbf{w}^T\mathbf{x}_i + b\right) \geq 1$.

a) **Datensatzvisualisierung**

Beginnen wir mit einem einfachen Datensatz, der aus zwei Klassen-Clustern besteht.

Visualisieren Sie den Datensatz mittels eines Scatterplotts in der Methode `plot_data`.

```python
def plot_data(X, y):
    """Plots the data and label assignment as a scatter plot."""
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired, edgecolors="k", zorder=10)
    plt.xticks([])
    plt.yticks([])
    plt.axis("tight")
    plt.show()
```

b) **Lineare Kernel**

Verwenden Sie das Modul `sklearn.svm.SVC` in der Funktion `fit_linear_svm`, um die Stützvektoren zu finden, die die Hyperebene in den Daten definieren, und die Spanne zu den nächstgelegenen Datenpunkten maximiert.

Verwenden Sie dabei keine Regularisierung ($C = 10^{10}$), sodass alle Stützvektoren so definiert, dass sie die Datenpunkte sind, die auf dem Rand der Spanne liegen und den exakt gleichen orthogonalen Abstand zur trennenden Hyperebene (den Randwert) haben.

```python
def fit_linear_svm(X, y):
    """Fits a svm with a linear kernel and no regularization to the data."""
    clf = svm.SVC(kernel="linear", C=1e10, max_iter=10000)
    clf.fit(X, y)
    return clf
```

c) **Kernel-Trick**

Wie Sie sehen können, trennt die Hyperebene die beiden Klassen perfekt voneinander. Wenn wir eine SVM mit linearem Kernel jedoch auf einem neuen Datensatz zweier konzentrischer Kreisen anwenden ist eine lineare Hyperebene unzureichend.

SVMs erlauben die Verwendung des sogenannten *Kernel-Tricks* für Probleme, die nicht wie oben angenommen linear trennbar sind, d.h. die Berechnung des Vektorprodukts in einem Merkmalsraum, in welchem die Daten erneut linear seperarierbar sind. Bekannte Kernel sind:

1.) der lineare Kernel:

$$k_{linear}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \mathbf{x}_i^T \mathbf{x}_j \,, \tag{5}$$

2.) der d-Grad-Polynom-Kernel:

$$k_{poly}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \left(\mathbf{x}_i^T \mathbf{x}_j + c\right)^d \,, \tag{6}$$

3.) der radiale Basisfunktionskernel (RBF):

$$k_{rbf}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}\right) = \exp\left(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2\right) \,. \tag{7}$$

Wir müssen daher eine Hyperebene in einem anderen Kernelraum finden, in dem die Daten wieder trennbar werden. Verwenden wir nun den radialen Basisfunktion Kernel $k_{rbf}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{2\sigma^2}\right)$ in der Methode `fit_rbf_svm` um unseren Merkmalsraum zu transferieren.

Geben Sie dabei $\frac{1}{n\_features}$ als Kernel Koeffizient γ an.

```python
def fit_rbf_svm(X, y):
    """Fits a svm with a rbf kernel and no regularization to the data."""
    clf = svm.SVC(kernel="rbf", C=1e10, gamma="auto")
    clf.fit(X, y)
    return clf
```

d) **Regularisierung**

Ein weiterer wichtiger Hyperparameter für die SVM ist die Regularisierung. Die SVM-Optimierung versucht, Datenpunkte zu finden, die die Spanne definieren. Dies ist nicht immer die beste Lösung, da die Daten verrauscht sein können und die trennende Hyperebene das Rauschen überlagert. Daher ist es üblich, einige Datenpunkte **in** der Spanne zu erlauben, die als *Slack Variablen* bezeichnet werden. Diese Regularisierung wird mit dem `C`

Parameter in `svm.SVC(...)` implementiert. Das sind die Kosten, die entstehen, wenn man zulässt, dass Supportvektoren innerhalb des Spannraums liegen. Die Zuweisung hoher Kosten (z.B. `C=1e10`) wird dazu führen, dass keine Datenpunkte innerhalb der Spanne liegen, während die Zuweisung niedriger Kosten (z.B. `C=1e-1`) zu einer weniger strengen Spanne führt.

Visualisieren Sie die Entscheidungsregionen und Support-Vektoren für $C = [2^{-6}, 2^{-3}, 1, 2^3, 2^6, 2^{12}]$ in der Methode `plot_svm_C`. Sie können dabei auf die Funktion `utils.plot_support_vectors` zurückgreifen.

```python
def plot_svm_C(X, y):
    """Plot the decision boundaries and support vectors for regularization constants:
    C=[2^-6, 2^-3, 1, 2^3, 2^6, 2^12]."""
    # min = -6, max = 12
    for C in range(-6, 13, 3):
        # Create an SVC object with the rbf kernel parameter
        clf = svm.SVC(kernel="rbf", C=2 ** C, max_iter=10000, gamma="auto")

        # Fit the hyperplane to the data
        clf.fit(X, y)

        # Plot data with hyperplane and support vectors
        utils.plot_support_vectors(X, y, clf, title="$C=2^{%s}$" % C)
```

8.2 SVM Multi-Label Klassifikation

Wir werden den Zifferndatensatz verwenden, um die Genauigkeit der SVM in Bezug auf die verschiedenen Hyperparameter-Setups zu messen.

a) **SVM Gridsearch**

Support-Vektor-Maschinen sind leistungsfähige Klassifikatoren. Es ist wichtig, eine Menge von Hyperparametern zu finden, die gut zu den Daten passen. Dazu ist es üblich, eine Hyperparametersuche z.B. über eine Gridsearch durchzuführen, das sich über mögliche Werte für verschiedene Parameter erstreckt. Für den RBF-Kernel können wir z.B. ein Gridsearch mit den Parametern `C` und γ abdecken, was effektiv jede Kombination beider Parameterbereiche aufbaut.

Führen Sie eine Gridsearch in der Methode `run_grid_search` für die Bereiche $2^{-13} \leq \gamma \leq 2^{-4}$, $2^{-3} \leq C \leq 2^3$, $1 \leq d \leq 6$ durch. Sie können dabei die Funktion `sklearn.model_selection.GridSearchCV` verwenden.

```python
def run_grid_search(X_digits, y_digits, C_range, degree_range, gamma_range):
    """Defines and runs a grid search over the given hyperparameter ranges for
    rbf and polynomial kernel. Each run is evaluated by using a 5-fold-cross-validation."""
    # Define grid over parameters for rbf and poly kernel
    rbf_grid = {"kernel": ["rbf"], "C": C_range, "gamma": gamma_range}
    poly_grid = {"kernel": ["poly"], "C": C_range, "degree": degree_range}
    # GridSearch arguments
    args = dict(
        estimator=svm.SVC(), scoring=make_scorer(accuracy_score), cv=5, verbose=1, n_jobs=-1
    )
    # Run gridsearch on the RBF grid
    rbf_gs = GridSearchCV(param_grid=rbf_grid, **args).fit(X_digits, y_digits)
    poly_gs = GridSearchCV(param_grid=poly_grid, **args).fit(X_digits, y_digits)
    return poly_gs, rbf_gs
```

8.3 K-Means Clustering

Der K-Means-Algorithmus gruppiert Daten, indem er versucht, Proben in n Gruppen gleicher Varianz einzuteilen, wodurch ein Kriterium minimiert wird, bekannt als die Trägheit (engl. inertia) oder die innere Cluster-Quadratsumme (engl. *within-cluster sum-of-squares criterion*). Dieser Algorithmus erfordert eine gegebene Anzahl von Clustern. Er lässt sich gut auf eine große Anzahl von Proben skalieren und wurde über eine große Bandbreite von Anwendungsbereichen in vielen verschiedenen Bereichen, wie z.B. Suchmaschinen, Dokumenten-Clustering, Kundensegmentierung, Sensorvernetzung und vielem mehr angewendet.

K-Means teilt einen Satz N von Datenpunkten X in K disjunkte Cluster C, jedes beschrieben durch den Mittelwert μ_j der Proben im Cluster. Die Mittelwerte sind gemeinhin als *Zentroide* bezeichnet werden; beachten Sie, dass sie es im Allgemeinen nicht Punkte aus X sind, obwohl sie sich im gleichen Raum befinden.

Der K-Means-Algorithmus zielt darauf ab, Zentroide zu wählen, die die *Trägheiten* minimieren:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} \left(||x_i - u_j||^2 \right).$$

Trägheit kann als ein Maß dafür erkannt werden, wie intern kohärent Cluster sind. Diese Metrik leidet jedoch unter verschiedenen Nachteilen:

- Die Trägheit trifft die Annahme, dass Cluster konvex und isotrop sind, was nicht immer der Fall ist. Es reagiert schlecht auf langgestreckte Cluster oder Krümmungen mit unregelmäßigen Formen.
- Trägheit ist keine normalisierte Metrik: wir wissen einfach, dass niedrigere Werte besser sind und Null optimal. Aber in sehr hochdimensionalen Räumen neigen die euklidischen Abstände dazu, aufgeblasen zu sein (dies ist ein Beispiel für den so genannten "Fluch der Dimensionalität"). Ausführen eines Dimensionalitätsreduktionsalgorithmus wie z.B. *Principal Component Analysis* (PCA) vor der K-Means-Clusterbildung kann dieses Problem lindern und beschleunigen die Berechnungen.

a) **Datensatzvisualisierung**

Beginnen wir mit einem Beispieldatensatz, der aus drei getrennte Cluster besteht. Visualisieren Sie den Datensatz (`plot_cluster_data`) und wenden Sie K-Means Clustering an (`fit_kmeans`). Stellen Sie anschließend die gefundenen Cluster in der Methode `plot_clustering` dar.

```python
def plot_cluster_data(X):
    """Creates a scatter plot for the given data."""
    plt.figure()
    plt.scatter(X[:, 0], X[:, 1])
    plt.axis("off")
    plt.show()
```

```python
def fit_kmeans(X, k, init="k-means++", n_init=10, random_state=0) -> [np.ndarray, np.ndarray]:
    """Runs K-Means clustering and returns the label assignments and cluster centroids."""
    # Build K-Means model
    model = KMeans(n_clusters=k, n_init=n_init, init=init, random_state=random_state)
    # Fit model
    model.fit(X)
    # Predict cluster assignment for each datapoint
    labels = model.predict(X)
    # Get cluster centers
    centers = model.cluster_centers_
    return labels, centers
```

```python
def plot_clustering(X, labels, centers=None, title="", subplot=None):
    """Helper function to plot the clustering"""
    # Plot in given subplot
    if subplot:
        plt.subplot(subplot)

    # Plot data with labels as color
    plt.scatter(X[:, 0], X[:, 1], c=labels)

    # Plot centers if given
    if centers is not None:
        plt.scatter(
            centers[:, 0], centers[:, 1], c="red", s=150, alpha=0.9, label="Centers"
        )

    # Set title
    plt.title(title)
    plt.axis("off")
```

b) **Iterationen**

K-Means hat erfolgreich drei separate Cluster gefunden.

Nun möchten wir die Bewegung der Cluster-Zentroiden über die Funktion *plot_kmeans_iterations* nach jeder Iteration visualisieren.

```python
def plot_kmeans_iterations(X, guess_center, initial_labels):
    """Plots the cluster assignments after iteration 1 to 6."""
    plt.figure(figsize=(2 * 3, 3 * 3))
    plot_clustering(
        X, initial_labels, centers=guess_center, title="Iteration 0", subplot=321
    )
    for i in range(1, 6):
        # Stop K-Means after i iterations
        model = KMeans(n_clusters=3, init=guess_center, n_init=1, max_iter=i).fit(X)
        labels = model.predict(X)
        centers = model.cluster_centers_

        # Plot clustering for current iteration
        plot_clustering(X, labels, centers, title="Iteration " + str(i), subplot=321 + i)
    plt.show()
```

c) **Problemepunkte**

Bei der Anwendung von K-Means gibt es einige Problemstellen wie unten aufgeführt.

Erläutern Sie jeden dieser Aspekte:

1.) Initialisierung

Der K-Means-Algorithmus beginnt mit dem anfänglichen Cluster-Zentroid. Es gibt mehrere Initialisierungsstrategien:

*- **Educated Guess**: Sie haben eine Ahnung, wo der Schwerpunkt sein könnte.*

*- **Zufällig**: Der Zentroid wird zufällig in der gültigen Datenregion initialisiert.*

*- **k-Means++**: Die Zentroide werden (im Allgemeinen) gleich weit voneinander entfernt sein, was zu nachweislich besseren Ergebnissen führt als eine zufällige Initialisierung.*

Eine zufällige Initialisierung von Zentroiden kann zu unterschiedlichen Clustering-Ergebnissen führen, da K-Means nur in einem lokalen Optimum konvergiert. Wir können dies zeigen, indem wir einen anderen Datensatz erzeugen, bei dem drei Cluster nahe beieinander liegen und ein Cluster weit entfernt ist.

Wie in den Darstellungen zu sehen, haben beide K-Means-Läufe unterschiedliche Cluster-Zuordnungen gefunden. Um das Problem der unterschiedlichen Ergebnisse in Abhängigkeit von der Initialisierung zu umgehen, ist es üblich, den K-Means-Algorithmus wiederholt mit unterschiedlichen Initialisierungen auszuführen. Die endgültige Clusterzuordnung wird dann auf der Grundlage der Minimierung eines Trägheitskriteriums ausgewählt, wie z.B. der Summe der quadrierten Abstände zum nächstgelegenen Schwerpunkt für alle Beobachtungen im Trainingssatz.

2.) Korrekte Anzahl von Clustern

Für den K-Means-Algorithmus ist es notwendig, die Zahl der Cluster vorher zu kennen. Dies ist möglicherweise nicht immer der Fall und führt zu falschen Gruppierungen. Sehen wir uns an, wie K-Means bei unserem ersten Datensatz mit unterschiedlicher Anzahl von Clustern abschneidet.

3.) Nicht-sphärische Cluster

Ein weiteres Problem bei K-Means ist, dass aufgrund des Zieles den quadrierten Abstands zwischen Datenpunkten und dem nächstgelegenen Cluster zu minimieren, nur sphärische Cluster gefunden werden. Ein gutes Beispiel dafür ist z.B. der Monddatensatz oder ein Datensatz, der aus Kreisen besteht, die andere Kreise enthalten.

4.) Cluster mit ungleicher Varianz

Wenn verschiedene Cluster einen signifikanten Unterschied in ihrer Varianz aufweisen wird K-Means falsche Cluster finden, da es die Annahme triffte, dass alle Cluster eine ähnliche Varianz aufweisen.

5.) Erzwungene Clusterbildung

Eine Sache, die man im Auge behalten sollte, ist, dass K-Means ***immer*** K*-Cluster findet, egal wie die Daten aussehen, ob es mehr oder weniger als* K *wahrer Cluster gibt, und selbst wenn es gar keine Cluster gibt. Dies kann man in einem Beispiel zeigen, bei dem wir gleichmäßig verteilte 2D-Daten in* $[0, 1]$ *erzeugen.*

Literatur

[1] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.