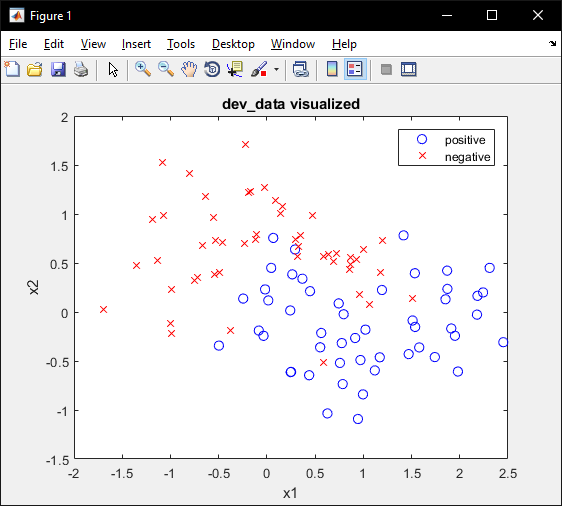
Machine Learning (Online) Übung 2

# Aufgabe 1: Logistische Regression für binäre Klassifizierung

## Entwicklung des Algorithmus mit dev\_data (Matlab script task1\_dev.m)

### Visualisierter Datensatz



### Vorbereiten der Daten

Zum testen und suchen von bestmöglichen Parametern teile ich die Daten zufällig in train, test und validation set ein (60%/20%/20%).

[Xtrain, Xval, Xtest, ytrain, yval, ytest] = splitDataset(X, y, .6, .2);

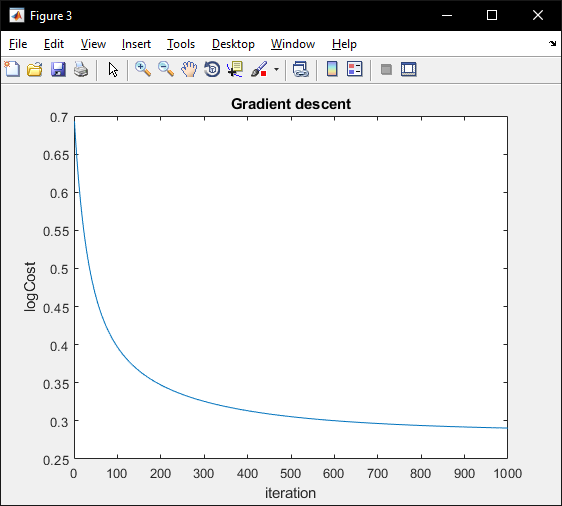
Ich verwende zudem nur die gegebenen 2 Features x1 und x2 (plus ein bias x0, der immer 1 ist) und keine künstlichen Features, da es ja vorerst nur darum geht, korrekte Algorithmen zu implementieren.

### Algorithmus (Gradient Descent)

alpha = .1; %learning rate

n\_iters = 1000;

lambda = 0; % regularization parameter

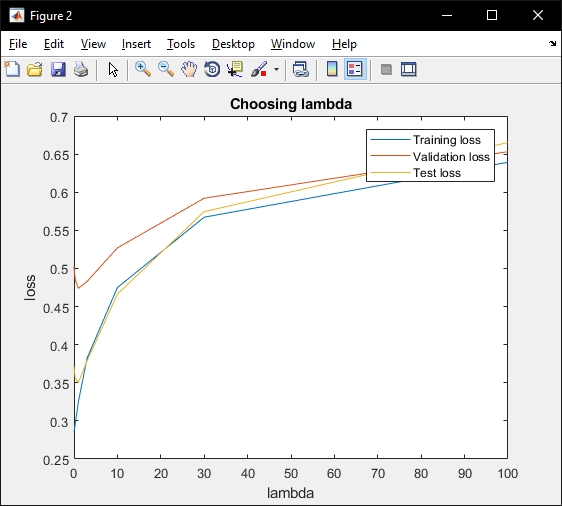


theta = 0.0541 1.3039 -3.8341

### Regularisierung

Zum Finden eines möglichst guten Regularisierungsparemeters habe ich verschiedene Werte ausprobiert. Es scheint, als wäre für dieses Datenset mit den 2 rohen features (keine polynomialen features, einfach nur x1 und x2) ein sehr kleiner Wert von ca. 0.03 am besten geeignet. Der Test loss ist hier natürlich zu ignorieren und nicht für die Bestimmung von lambda zu verweden. Ich habe die Werte nur als zusätzliche Information geplottet (das Gleiche gilt auch für den Rest der Aufgaben).

lambdaValues = [0 0.0001 0.0003 0.001 0.003 0.01 0.03 0.1 0.3 1 3 10 30 100];



### Klassifikation/Vorhersage

Zur Vorhersage wende ich die Sigmoid Funktion auf den berechneten Wert X \* theta' an und verwende 0.5 als Schwellenwert zur Bestimmung, ob eine Vorhersage positiv oder negativ ist.

function h = predict(X, theta, binary)

%PREDICT predict output values using features X and weights theta

h = sigmoid(X \* theta', false);

if binary

h = h >= .5;

end

end

Wenn wir das Modell mit den gelernten Parametern (theta) zur Vorhersage auf dem Testset verwenden, erhalten wir:

predict(Xtest, theta, true)' = 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 1 0

ytest’ = 1 1 1 1 1 0 1 1 0 1 1 0 0 1 0 1 1 0 1 0

### Kenngrössen

Die Auswertung des Modells zeigt, dass mit der gewählten Parametrisierung und dieser zufälligen Aufteilung der Daten die F-Score auf dem Test-set sogar höher ist, als auf dem Trainings-set! Das ist natürlich nicht immer so, da ich eine zufällige Unterteilung in Trainings- Valdiation- und Test-set vornehme. Ich habe das ganze mehrmals durchlaufen lassen. Die Werte sind immer sehr ähnlich, oft wird aber auf dem Trainings-set eine bessere F-Score erzielt als auf dem Test-set.

Evaluation on training set:

Confusion Matrix:

+-------------+

| 23 | 3 |

|-------------|

| 4 | 30 |

+-------------+

Accuracy: 0.883

Precision: 0.885

Recall: 0.852

F-Score: 0.868

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 6 | 2 |

|-------------|

| 4 | 8 |

+-------------+

Accuracy: 0.700

Precision: 0.750

Recall: 0.600

F-Score: 0.667

Evaluation on test set:

Confusion Matrix:

+-------------+

| 11 | 1 |

|-------------|

| 2 | 6 |

+-------------+

Accuracy: 0.850

Precision: 0.917

Recall: 0.846

F-Score: 0.880

### Fazit

Das Modell mit den zugehörigen Algorithmen, Funktionen inklusive Regularisierung funktioniert und kann jetzt weiterverwendet werden für das richtige Dataset (house\_data).

## Anwendung des Entwickelten Algorithmus zur Vorhersage von «Waterfront» (Matlab script task1.m)

### Analyse des Datasets

Eine erste Analyse zeigt, dass es nur sehr wenige Häuser mit einer Waterfront (positive Examples) gibt:

n\_pos\_train = 133

n\_neg\_train = 17251

n\_pos\_test = 30

n\_neg\_test = 4199

Es wäre daher möglicherweise auch sinnvoll, ein Anomaly Detection Modell zu entwickeln. Das ist aber hier nicht die Aufgabe.

### Vorbereiten der Daten

Ich verwende nach Auswahl der Features jeweils 10% der Trainingsdaten als Valdidierungs-set zur bestimmung optimaler Parameter.

[Xtrain, Xval, ~, ytrain, yval, ~] = splitDataset(X, y, .9, .1);

### Erstes Model

Als erste Variante nehme ich einfach alle (normalisierten) Features (mit einem bias term).

X = [

ones(size(features\_train, 1), 1), ...

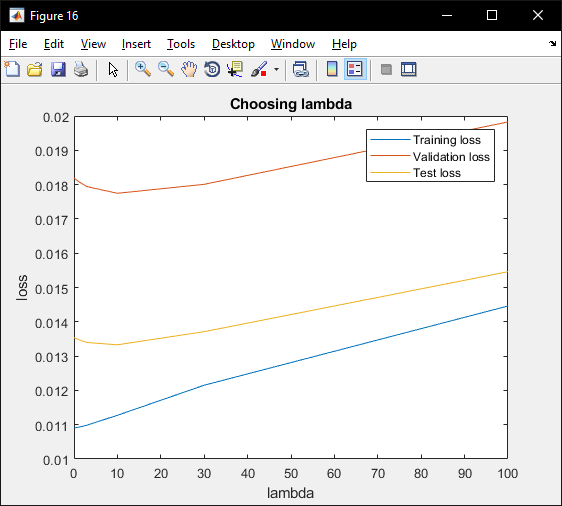
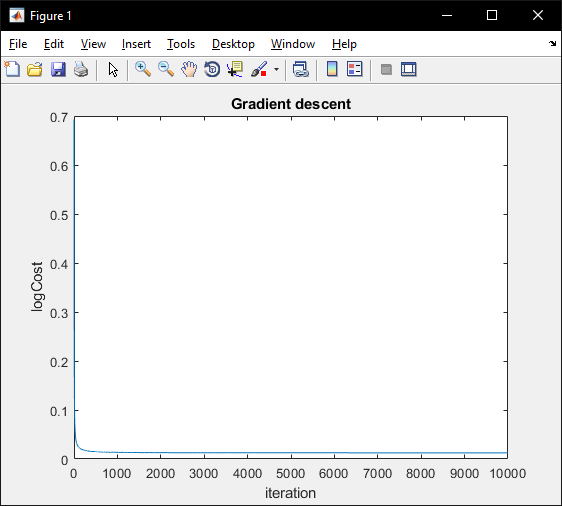
normalizeFeatures(features\_train)

];

alpha = 1; %learning rate

n\_iters = 10000;

Nach probieren verschiedener Werte für den Regularisierungs-Parameter lambda scheint **lambda=10** die beste Wahl zu sein (unter Beachtung des Validierungs-loss, Test-loss ist hier wiederum nur als Information aufgeführt). 10000 Iterationen scheint hier ein wenig übertrieben zu sein, aber da es nur so wenige positive Examples gibt erziele ich damit eine um einiges bessere F-Score als mit nur 1000 Iterationen (auch wenn theta sich wirklich nur minimal verändert).

Letzter Wert der Kostenfunktion (log cost) = 0.012913559329687

Das Modell erzielt eine sehr hohe Accuracy, was aber vorallem den Grund hat, dass der positive/negative ratio sehr klein ist. Die F-Score ist sprechender und zeigt, dass dieses Model wohl doch noch Optimierungspotential hat. Es ist zu bemerken, dass sich die F-Score mehr als verdoppelt hat, nachdem ich Gradient Descent für 10000 anstatt 1000 Iterationen laufen lassen habe und alpha von 0.1 auf 1 gesetzt habe!

Evaluation on training set:

Confusion Matrix:

+-------------+

| 65 | 19 |

|-------------|

| 51 |15510 |

+-------------+

Accuracy: 0.996

Precision: 0.774

Recall: 0.560

F-Score: 0.650

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 9 | 2 |

|-------------|

| 8 | 1719 |

+-------------+

Accuracy: 0.994

Precision: 0.818

Recall: 0.529

F-Score: 0.643

Evaluation on test set:

Confusion Matrix:

+-------------+

| 19 | 3 |

|-------------|

| 11 | 4196 |

+-------------+

Accuracy: 0.997

Precision: 0.864

Recall: 0.633

F-Score: 0.731

### Zweites Model

Zusätzlich zu den normalisierten Features verwende ich hier noch künstliche Features, und zwar die zweite und dritte Potenz der jeweiligen Features (jeweils normalisiert):

X = [

ones(size(features\_train, 1), 1), ...

normalizeFeatures(features\_train), ...

normalizeFeatures(features\_train.^2), ...

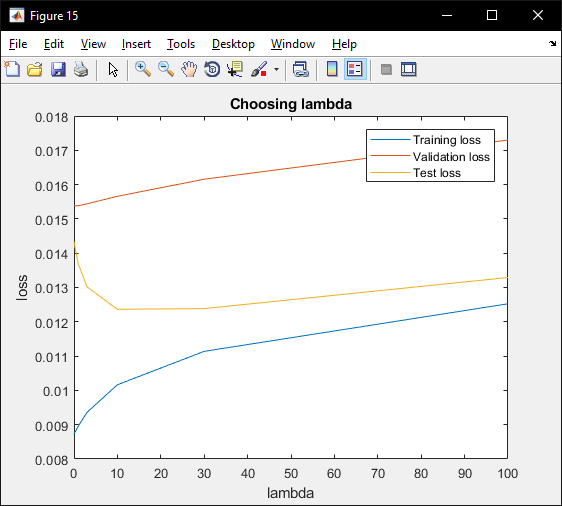
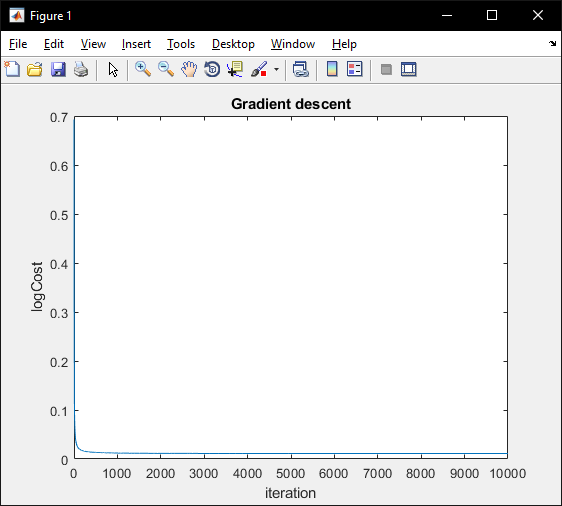
normalizeFeatures(features\_train.^3)

];

alpha = 1; %learning rate

n\_iters = 10000;

Nach erneuter Analyse verschiedener lambda Werte scheint gemäss Validation-set keine Regularisierung (**lambda=0**) am besten geeignet zu sein. Da diese Aufgabe aber möglicherweise an der F-Score auf diesem Test-set bewertet wird, begehe ich hier das Verbrechen und wähle trotzdem lambda=10, auch wenn man gemäss ML best practices das Test-set dazu nicht verwenden sollte.

Letzter Wert der Kostenfunktion (log cost) = 0.009573971772305

Dieses Modell performt leicht besser auf dem Trainings- und Validierungs-set, aber hat sich auf dem Test-set verschlechtert (und das ist, was zählt). Einfach noch die zweite und dritte Potenz der Features zu nehmen, hat also nicht geholfen.

Evaluation on training set:

Confusion Matrix:

+-------------+

| 77 | 16 |

|-------------|

| 38 |15514 |

+-------------+

Accuracy: 0.997

Precision: 0.828

Recall: 0.670

F-Score: 0.740

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 11 | 4 |

|-------------|

| 7 | 1716 |

+-------------+

Accuracy: 0.994

Precision: 0.733

Recall: 0.611

F-Score: 0.667

Evaluation on test set:

Confusion Matrix:

+-------------+

| 17 | 4 |

|-------------|

| 13 | 4195 |

+-------------+

Accuracy: 0.996

Precision: 0.810

Recall: 0.567

F-Score: 0.667

### Drittes Model

Das Feature «zipcode» macht rein numerisch eigentlich nicht viel Sinn, daher werde ich dieses hier one-hot encoden. Zusätzlich ist dies sicher ein aussagekräftiges Feature für die «waterfront» Vorhersage.

Weiter habe ich bemerkt, dass ich die Learning Rate alpha gut auf 20 stellen kann. Gradient Descent konvergiert so um einiges schneller.

X = [

ones(size(features\_train, 1), 1), ...

normalizeFeatures(features\_train(:, 1:13)), ...

normalizeFeatures(features\_train(:, 15:end)), ...

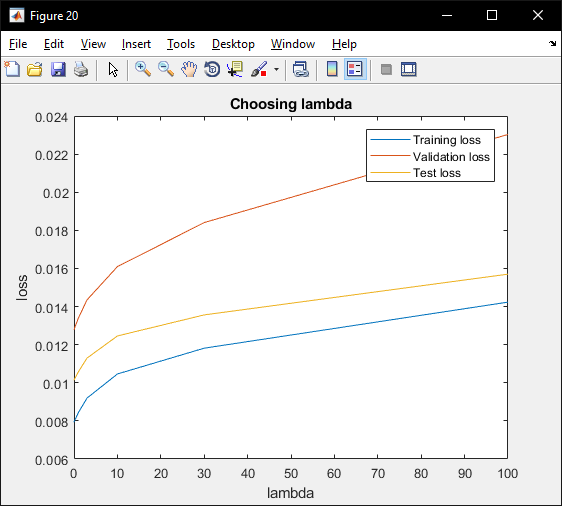
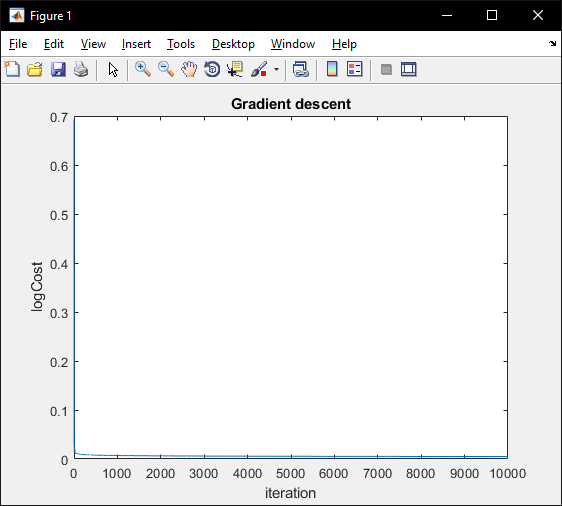
onehotEncode(features\_train(:, 14)) %zipcode

];

alpha = 20; %learning rate

n\_iters = 10000;

Hier liefert klar keine Regularisierung (**lambda=0**) das beste Ergebnis.

Letzter Wert der Kostenfunktion (log cost) = 0.005009312234119

Evaluation on training set:

Confusion Matrix:

+-------------+

| 90 | 12 |

|-------------|

| 23 |15520 |

+-------------+

Accuracy: 0.998

Precision: 0.882

Recall: 0.796

F-Score: 0.837

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 15 | 1 |

|-------------|

| 5 | 1717 |

+-------------+

Accuracy: 0.997

Precision: 0.938

Recall: 0.750

F-Score: 0.833

Evaluation on test set:

Confusion Matrix:

+-------------+

| 24 | 6 |

|-------------|

| 6 | 4193 |

+-------------+

Accuracy: 0.997

Precision: 0.800

Recall: 0.800

F-Score: 0.800

### Viertes Model

Gemäss Bauchgefühl haben hauptsächlich die örtlichen Features (zipcode, long, lat) sowie eventuell der Preis und die Bewertung des Hauses Einfluss darauf, ob es Blick aufs Wasser (waterfront) hat. Ich verwende darum hier genau diese Features.

X = [

ones(size(features\_train, 1), 1), ...

normalizeFeatures(features\_train(:, 1)), ... %price

normalizeFeatures(features\_train(:, 9)), ... %grade

normalizeFeatures(features\_train(:, 15:16)), ... %lat & long

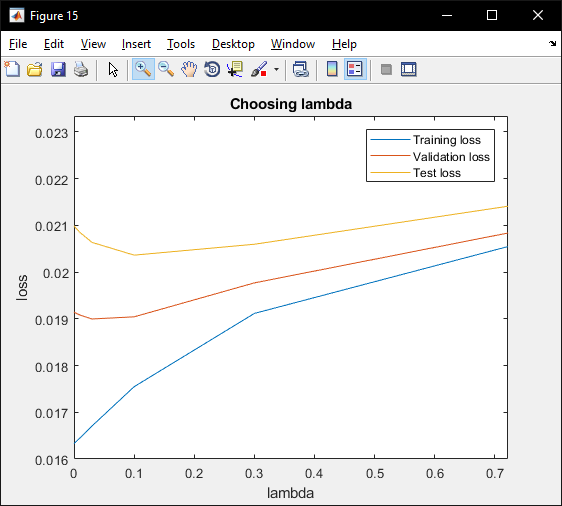
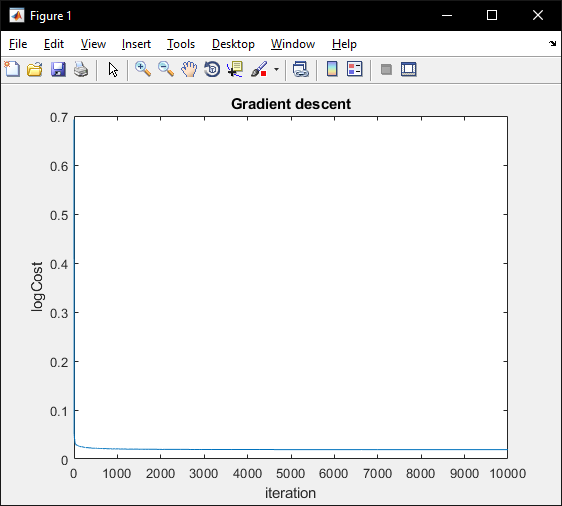
onehotEncode(features\_train(:, 14)) %zipcode

];

alpha = 20; %learning rate

n\_iters = 10000;

Hier ist interessant, dass ein ganz kleiner Lambda Wert von 0.1 sich sehr gut macht, im Gegensatz zu den vorherigen Versuchen (Plot ist herangezoomt, gegen rechts steigen alle Kurven).

Letzer Wert der Kostenfunktion (log cost) = 0.019132176146419

Dennoch schliesst das Modell insgesamt um einiges schlechter ab als das vorherige. Meine Intuition war also falsch, die anderen Features sagen sehr wohl auch etwas über «waterfront» aus.

Evaluation on training set:

Confusion Matrix:

+-------------+

| 45 | 10 |

|-------------|

| 79 |15511 |

+-------------+

Accuracy: 0.994

Precision: 0.818

Recall: 0.363

F-Score: 0.503

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 4 | 2 |

|-------------|

| 5 | 1727 |

+-------------+

Accuracy: 0.996

Precision: 0.667

Recall: 0.444

F-Score: 0.533

Evaluation on test set:

Confusion Matrix:

+-------------+

| 10 | 2 |

|-------------|

| 20 | 4197 |

+-------------+

Accuracy: 0.995

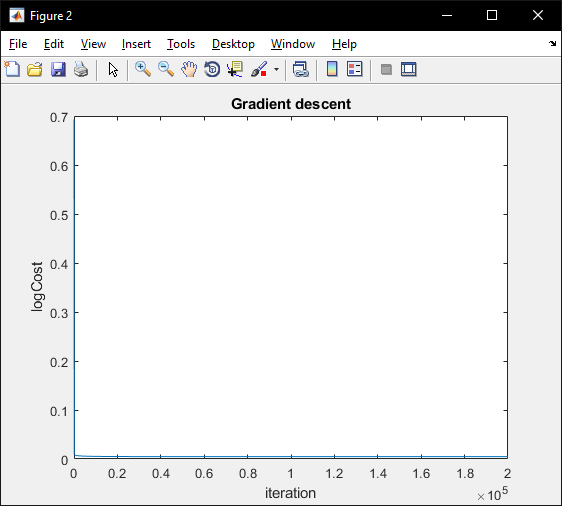
Precision: 0.833

Recall: 0.333

F-Score: 0.476

### Finales Modell & Fazit

Das dritte Model hat klar die besten Resultate geliefert (F-Score von 0.8). Hier nehme ich deshalb dieses Model und trainiere es für **200’000 Gradient Descent Iterationen** mit **alpha=20** (war die höchst mögliche learning rate, die die Kostenfunktion noch zur Konvergenz gebracht hat, mehr als 20 war zu viel) und wie gehabt **lambda=0**.



Letzer Wert der Kostenfunktion (log cost) = 0.004823863782946

Nach über 5 Minuten Trainingszeit auf den 8 cores meiner i7 CPU konnte ich das Resultat tatsächlich noch ein kleines bisschen verbessern auf eine **F-Score von 0.82** auf dem Test-set.

Evaluation on training set:

Confusion Matrix:

+-------------+

| 95 | 10 |

|-------------|

| 18 |15522 |

+-------------+

Accuracy: 0.998

Precision: 0.905

Recall: 0.841

F-Score: 0.872

Evaluation on validation set:

Confusion Matrix:

+-------------+

| 15 | 1 |

|-------------|

| 5 | 1717 |

+-------------+

Accuracy: 0.997

Precision: 0.938

Recall: 0.750

F-Score: 0.833

Evaluation on test set:

Confusion Matrix:

+-------------+

| 25 | 6 |

|-------------|

| 5 | 4193 |

+-------------+

Accuracy: 0.997

Precision: 0.806

Recall: 0.833

F-Score: 0.820