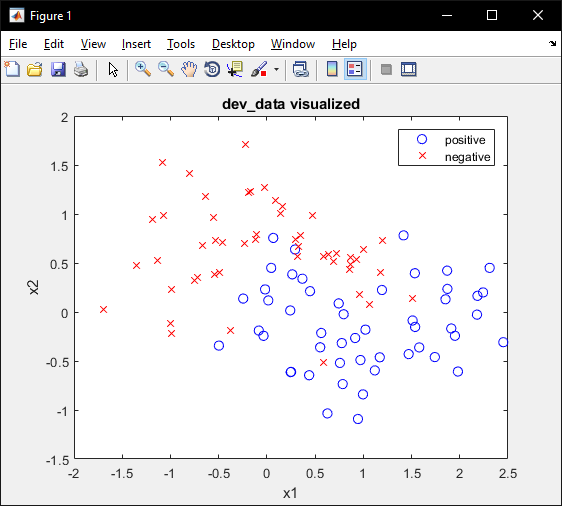
Machine Learning (online) Übung 2

# Aufgabe: Logistische Regression für binäre Klassifizierung

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

### Visualisierter Datensatz

Dieser Plot zeigt die positiven und negativen Examples aus dem Datensatz gemäss den 2 Features.



### Vorbereiten der Daten

Auch wenn es in dieser Aufgabe nicht unbedingt nötig ist: 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);

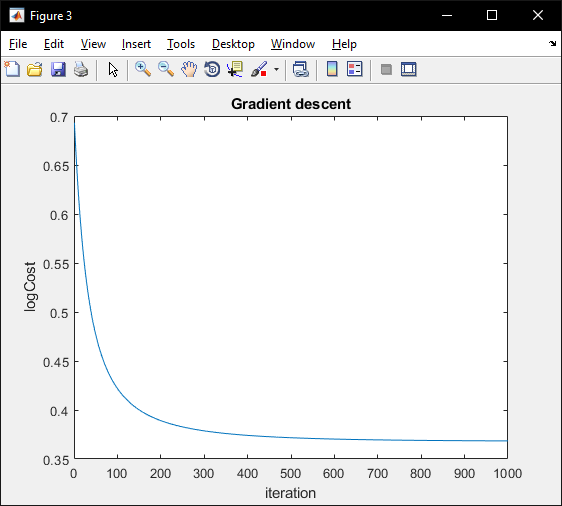
Ich verwende zudem nur die gegebenen 2 Features x1 und x2 (plus ein bias x0, der immer 1 ist) und keine künstlichen Features, da es ja vorerst nur darum geht, korrekte Algorithmen zu implementieren.

### Algorithmus (Gradient Descent, logistische Regression)

alpha = .1; %learning rate

n\_iters = 1000;

lambda = 0; % regularization parameter



Letzter Wert der Kostenfunktion (log cost) = 0.3684

Letzte Gradientenlänge (norm) = 0.0431

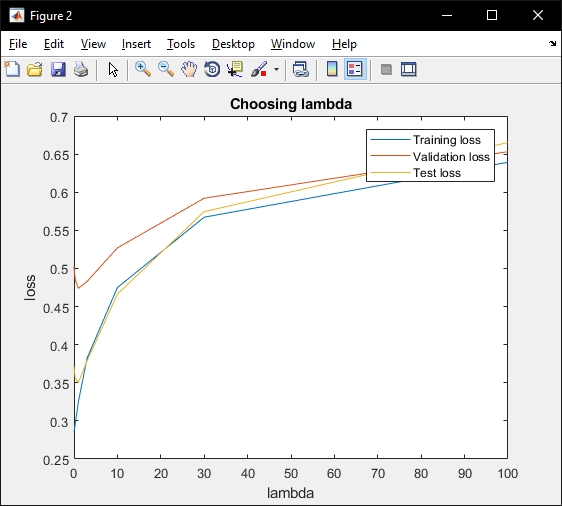
theta = -0.3979 1.4817 -2.6294

Abbruchgrund: Konvergiert, reicht zum Testen

### Regularisierung

Zum Finden eines möglichst guten Regularisierungsparemeters habe ich verschiedene Werte ausprobiert. Es scheint, als wäre für dieses Dataset 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];



Dieser Plot zeigt die Auswirkung von Lambda auf Kostenfunktion des fertig trainierten Models.

### 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 Test-Set 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. Die confusion matrices zeigen, dass der Grossteil korrekt vorhergesagt wird.

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 Validierungs-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), ...

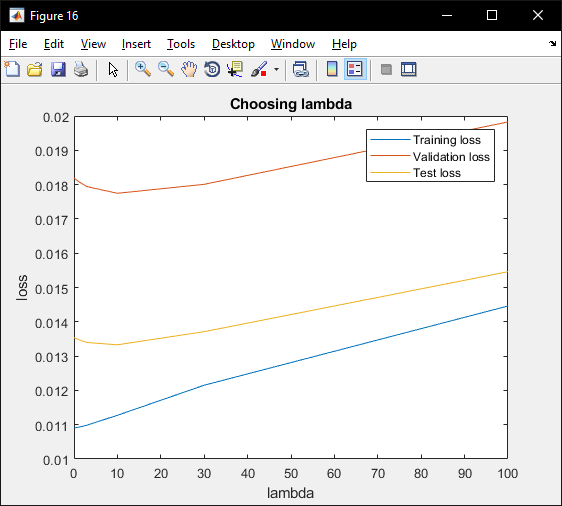
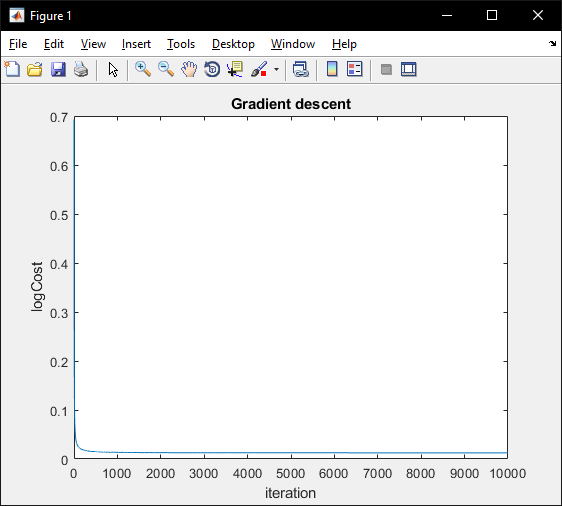
normalizeFeatures(features\_train)

];

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 gegen Ende wirklich nur minimal verändert).

Letzter Wert der Kostenfunktion (log cost) = 0.012913559329687  
Letzte Gradientenlänge (norm) = 0.0277  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

Das Modell erzielt eine sehr hohe Accuracy, was aber vor allem 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!  
Die Auswertung zeigt, dass es nur sehr wenige false negatives gibt, hingegen noch verhältnismässig false positives.

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), ...

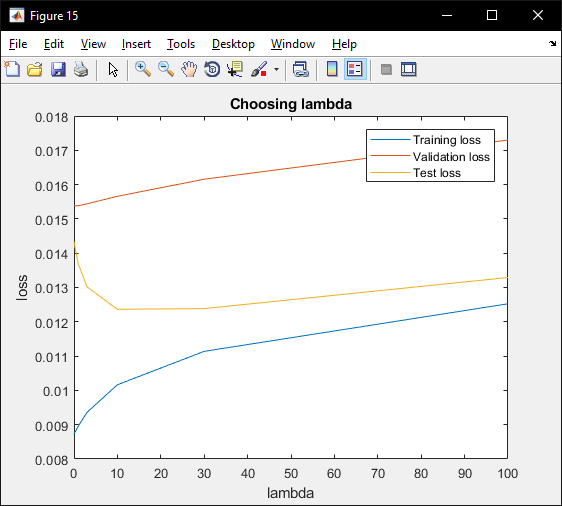
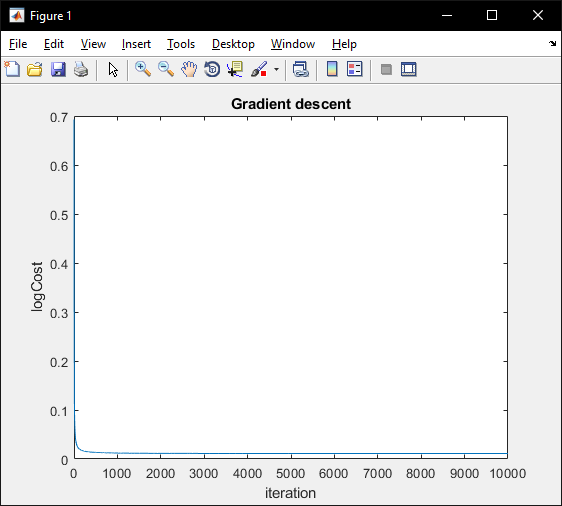
normalizeFeatures(features\_train.^3)

];

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, wähle ich 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  
Letzte Gradientenlänge (norm) = 2.5730e-04  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

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 diesen hier one-hot encoden. Zusätzlich ist dies sicher ein aussagekräftiges Feature für die «waterfront» Vorhersage.

Weiter wird hier mit alpha=20 eine schnellere Konvergenz erreicht.

X = [

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

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

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

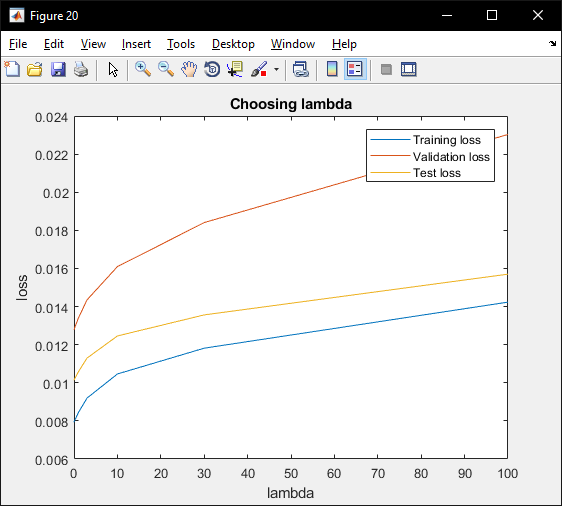
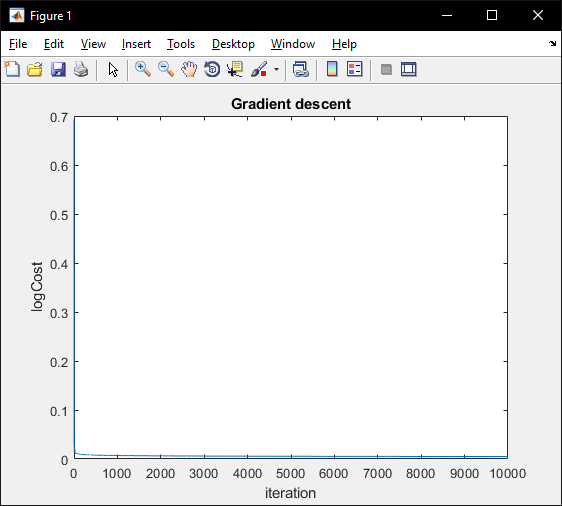
onehotEncode(features\_train(:, 14)) %zipcode

];

alpha = 20; %learning rate

n\_iters = 10000;

Hier liefert klar keine Regularisierung (**lambda=0**) das beste Ergebnis, wie man gut im folgenden linken Plot sieht.

Letzter Wert der Kostenfunktion (log cost) = 0.005009312234119  
Letzte Gradientenlänge (norm) = 5.6618e-05  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

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

Intuitiv 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

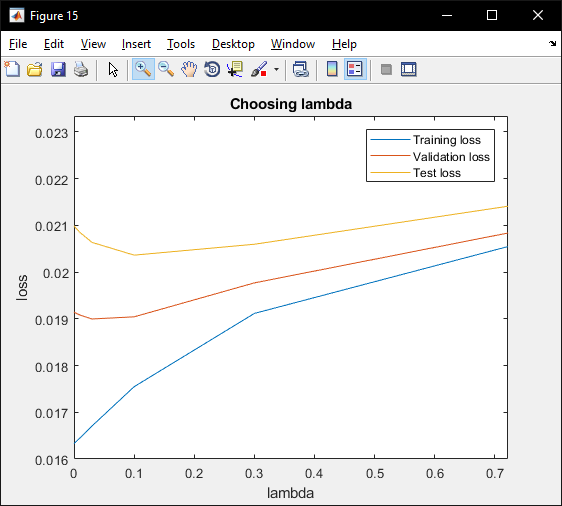
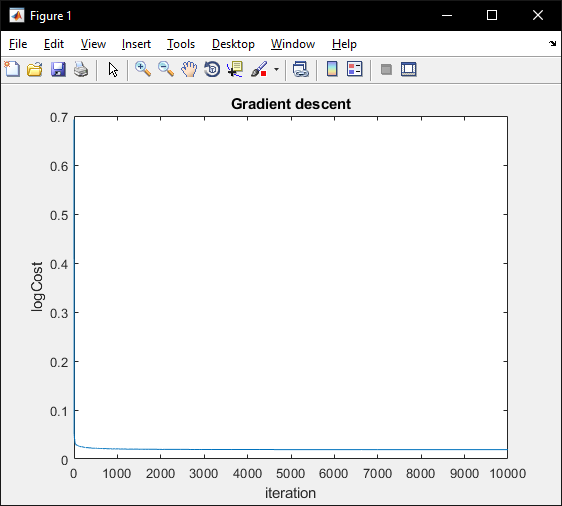
onehotEncode(features\_train(:, 14)) %zipcode

];

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

Letzte Gradientenlänge (norm)= 1.9327e-05  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

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

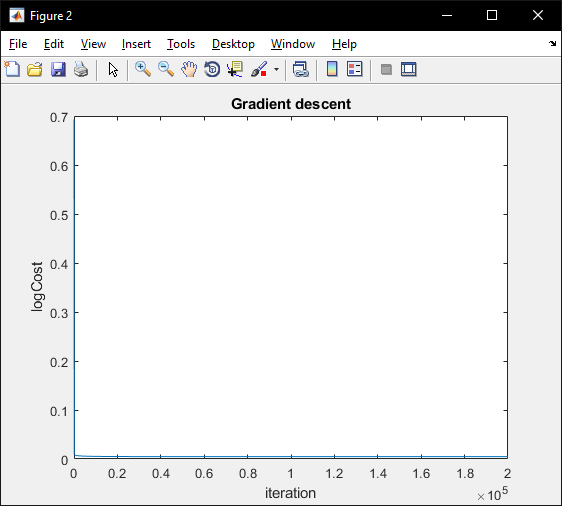
Precision: 0.833

Recall: 0.333

F-Score: 0.476

### Finales Modell & Interpretation der Resultate

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**.



Letzter Wert der Kostenfunktion (log cost) = 0.004823863782946  
Letzte Gradientenlänge (norm) = 1.5048e-06  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

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, was nicht schlecht ist. Gäbe es aber mehr positive Training Examples, könnte wohl noch eine um einiges bessere Klassifizierung vorgenommen werden (die verhältnismässig vielen false negatives könnten verbessert werden).

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

# Aufgabe: Neuronales Netz zur Multi-class Klassifizierung

## Entwicklung des Algorithmus mit dev\_data (Matlab script task2\_dev.m)

### Vorbereiten der Daten

Gleich wie bei der ersten Aufgabe, Aufteilung in Train-, Validation- und Test-set (60%/20%/20%).

### Algorithmus (Gradient Descent, neuronales Netzwerk)

Ich habe eine Klasse NeuralNetwork geschrieben, welche auf einfache Art und Weise erlaubt, ein sequentielles Netz aufzubauen und nötige Operationen darauf auszuführen. Dadurch lässt sich einfach mit verschiedenen Architekturen und Parameter herumspielen. Der zweite Parameter für den Layer-Konstrukor gibt an, ob ein bias-Neuron verwendet werden soll.

%% setup neural network

inputLayer = Layer(size(X, 2), true, @linear);

outputLayer = Layer(1, false, @sigmoid);

nn = NeuralNetwork(inputLayer, outputLayer);

nn = nn.addLayer(Layer(size(X, 2), true, @sigmoid));

nn = nn.initWeights(-1, 1);

%% set parameters and train network

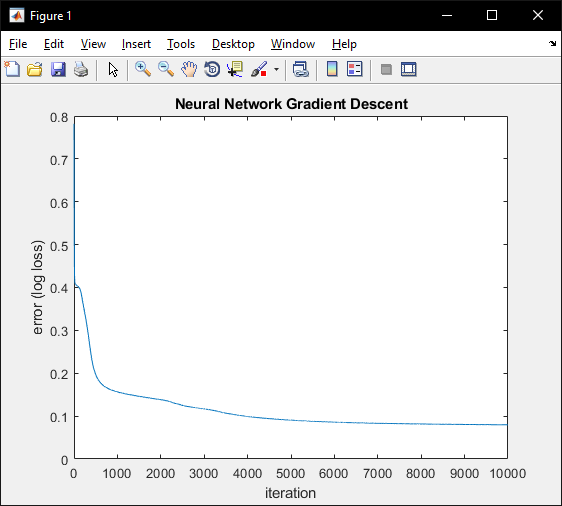
alpha = 3; %learning rate

lambda = 0; %regularization parameter

n\_iters = 10000;

nn = nn.train(Xtrain, ytrain, alpha, lambda, n\_iters);

Je nach Parameter und Initialisierung der Gewichte sieht die Kurve ein wenig anders aus, die Kostenfunktion wird aber wie gewünscht minimiert.

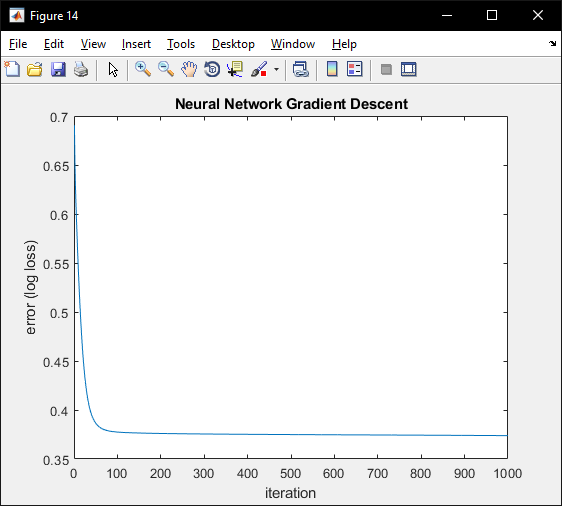
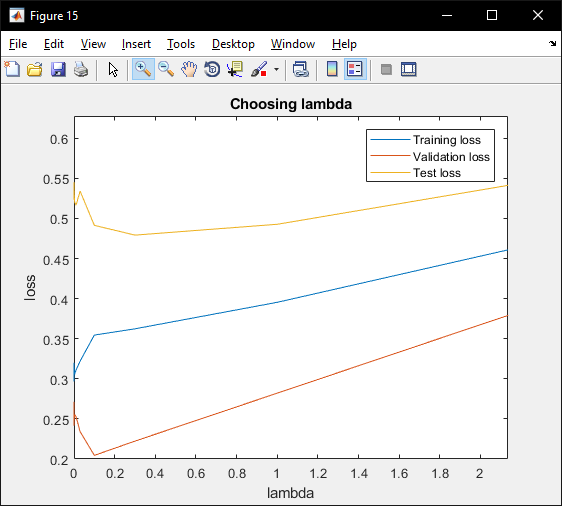


Letzter Wert der Kostenfunktion = 0.0912

Letzte Gradientenlänge (norm) = 0.0108  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

### Regularisierung

Auf diesem Testset eignet sich **lambda=0.1** offenbar am besten (linker Plot herangezoomt, gegen rechts nehmen alle Kurven zu).



Letzter Wert der Kostenfunktion = 0.3798

Letzte Gradientenlänge (norm) = 0.0031

Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

### Klassifizierung/Vorhersage

nn.predict(Xtest, true)' = 1 0 0 0 1 0 1 1 0 0 1 0 1 0 0 1 0 0 1 1

ytest' = 1 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 0 1 1

### Kenngrössen

Confusion Matrix:

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

| 6 | 3 |

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

| 2 | 9 |

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

Accuracy: 0.750

Precision: 0.667

Recall: 0.750

F-Score: 0.706

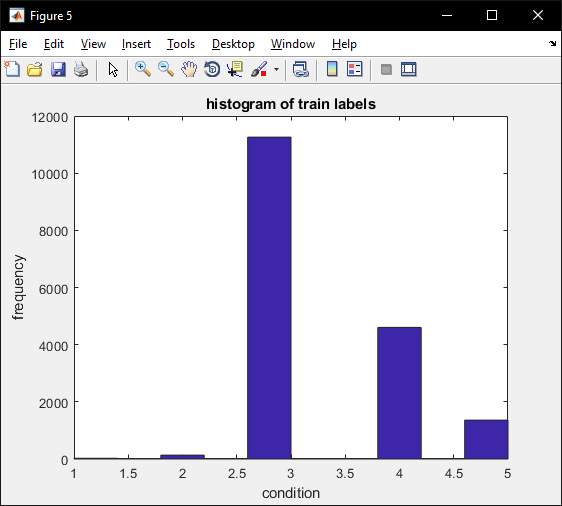
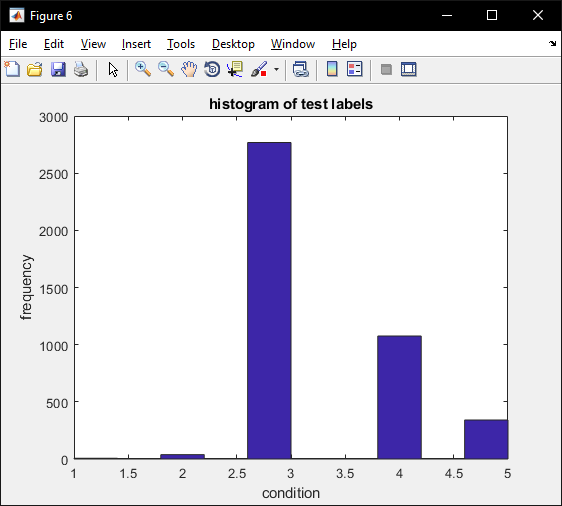
### Fazit

Alle Funktionalitäten laufen wie gewünscht und können jetzt für das «house\_data» Dataset verwendet werden.

## Anwendung des Entwickelten Algorithmus zur Vorhersage von «Condition» (Matlab script task2.m)

### Analyse des Datasets

Die Histogramme der Datasets zeigen eine schiefe Verteilung der «Condition». Es gibt praktisch keine Häuser mit condition = 1 oder condition = 2 und nur wenige condition = 5.

### Vorbereiten der Daten

Wie auch schon vorher verwende ich nach Auswahl der Features jeweils 10% der Trainingsdaten als Validierungs-Set zur Bestimmung optimaler Parameter.

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

### Auswertungsfunktion/Metriken

Um die Multiclass-Klassifizierung anständig auszuwerten berechne ich zuerst jeweils die Accuracy, Precision, Recall und F-Score der einzelnen Klassen und verwende dann einen gewichteten Durchschnitt der einzelnen Werte für die Gesamtauswertung.

precision = sum(precision .\* total\_per\_class) / total;

recall = sum(recall .\* total\_per\_class) / total;

accuracy = sum(accuracy .\* total\_per\_class) / total;

fscore = sum(fscore .\* total\_per\_class) / total;

Weitere Details in der Funktion evaluateMultiClass.

### Erstes Model

Ich verwende als erstes alle normalisierten Features, wobei ich den Zipcode noch one-hot codiere.

%% feature selection

X = [

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

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

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

];

**Netzwerk Architektur:** 1 hidden Layer mit gleich vielen Nodes wie der Inputlayer (mit bias unit) und sigmoid Aktivierungsfunktion.

%% setup neural network

inputLayer = Layer(size(X, 2), true, @linear);

outputLayer = Layer(size(Y, 2), false, @sigmoid);

nn = NeuralNetwork(inputLayer, outputLayer);

nn = nn.addLayer(Layer(size(X, 2), true, @sigmoid));

Zur ersten Wahl von lambda verwende ich hier nur 200 Gradient Descent Iterationen, da das Training sehr rechenintensiv ist und relativ lange dauert.

%% set parameters and train network using different lambdas

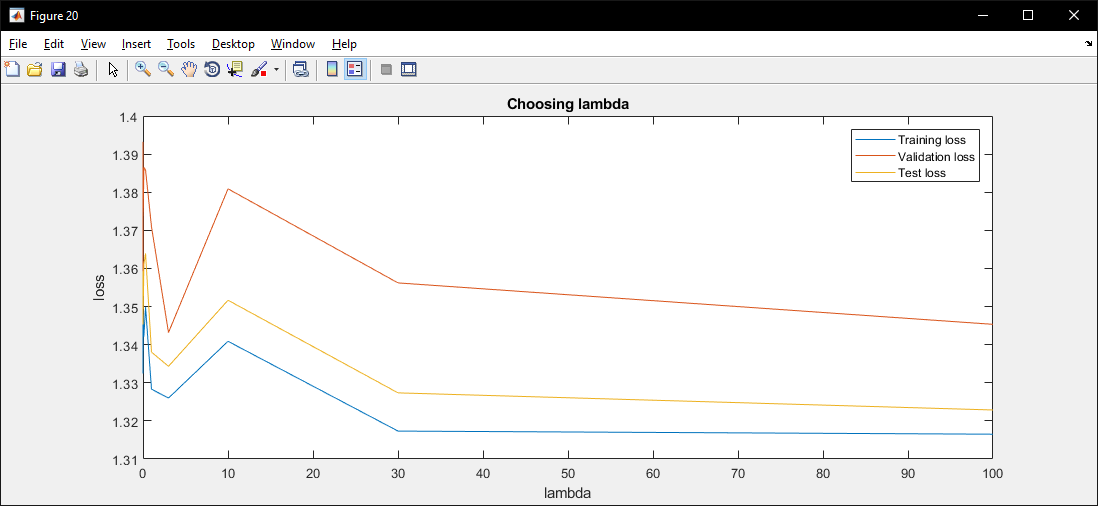
alpha = .2; %learning rate

n\_iters = 200;

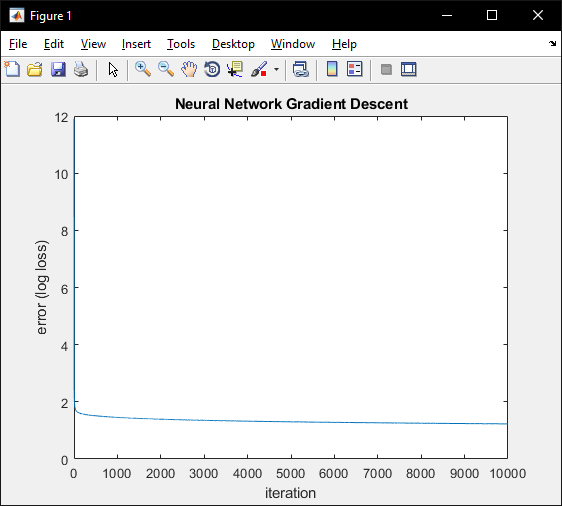
init\_weight\_min = -1; %lower bound for random weight initialization

init\_weight\_max = 1; % upper bound

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



**lambda = 3** scheint gemäss diesem Plot keine schlechte Wahl zu sein. Mit **alpha = 0.2** konvergiert Gradient Descent gut. Höhere Werte enden oft in Divergenz mit diesem Model. Nach dieser Auswertung liess ich das Training nochmal mit **10000 Iterationen** laufen, was ca. 15min dauerte.



Letzter Wert der Kostenfunktion = 1.2288

Letzte Gradientenlänge (norm) = 0.0015  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

Evaluation on training set:

Confusion Matrix:

0 2 13 5 0

0 2 80 30 2

0 4 9104 973 73

1 1 2145 1852 120

0 0 443 610 185

Accuracy: 0.773

Precision: 0.682

Recall: 0.712

F-score: 0.685

Evaluation on validation set:

Confusion Matrix:

0 0 3 1 0

0 0 15 4 1

0 0 984 113 12

0 0 249 220 14

0 0 50 60 12

Accuracy: 0.762

Precision: 0.658

Recall: 0.700

F-score: 0.670

Evaluation on test set:

Confusion Matrix:

0 1 4 1 0

0 0 30 7 1

0 5 2435 306 22

1 1 548 479 47

0 0 127 175 39

Accuracy: 0.766

Precision: 0.662

Recall: 0.698

F-score: 0.672

### Zweites Model

Intuitiv sind die sinnvollsten Features für die condition folgende:

X = [

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

normalizeFeatures(features\_train(:, 12:13)), ... % year built & renovated

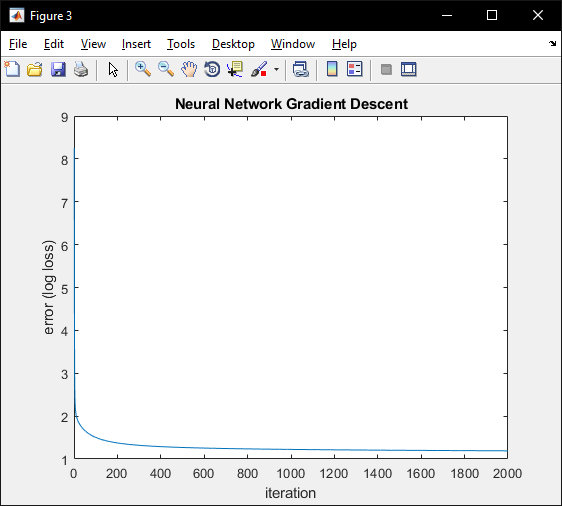
normalizeFeatures(features\_train(:, 15:16)), ... % long & lat

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

];

**Netzwerk Architektur:** wie vorhin, aber mit RELU anstatt sigmoid im hidden layer mit Hoffnung auf schnellere Konvergenz. **Num\_iters = 1000, alpha = 0.3, lambda = 0.1**

Das Modell macht sich minim schlechter, trainiert aber viel schneller (und ich habe 5x weniger Iterationen benutzt und trotzdem fast die gleiche F-Score erzielt).



Letzter Wert der Kostenfunktion = 1.1905

Letzte Gradientenlänge (norm) = 0.0042  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

Evaluation on training set:

Confusion Matrix:

0 0 15 7 0

0 0 80 40 1

0 0 9047 1083 15

0 0 2091 2019 23

0 0 551 648 25

Accuracy: 0.769

Precision: 0.669

Recall: 0.709

F-score: 0.673

Evaluation on validation set:

Confusion Matrix:

0 0 2 0 0

0 0 9 4 0

0 0 987 130 1

0 0 249 218 2

0 0 66 70 0

Accuracy: 0.754

Precision: 0.623

Recall: 0.693

F-score: 0.654

Evaluation on test set:

Confusion Matrix:

0 0 3 2 1

0 0 29 9 0

0 0 2431 326 11

0 0 534 533 9

0 0 157 178 6

Accuracy: 0.765

Precision: 0.652

Recall: 0.702

F-score: 0.668

### Drittes Model

In diesem letzten Versuch werden wieder alle Features verwendet, jedoch noch ein zusätzlicher hidden Layer mit 20 Neuronen.

X = [

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

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

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

];

**Netzwerk Architektur:**

inputLayer = Layer(size(X, 2), true, @linear);

outputLayer = Layer(size(Y, 2), false, @sigmoid);

nn = NeuralNetwork(inputLayer, outputLayer);

nn = nn.addLayer(Layer(size(X, 2), true, @relu));

nn = nn.addLayer(Layer(20, true, @sigmoid)); % additional sigmoid layer with 20 neurons + bias

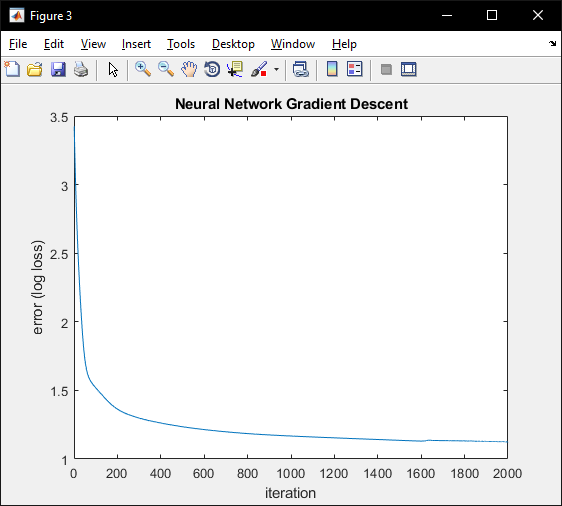
alpha = .1; %learning rate

init\_weight\_min = -.1; %lower bound for random weight initialization

init\_weight\_max = .1; % upper bound

lambda = .0005; %regularization parameter

n\_iters = 3000;



Letzter Wert der Kostenfunktion = 1.1557  
Letzte Gradientenlänge (norm) = 0.0049  
Abbruchgrund: Konvergiert, Wert ändert sich kaum mehr

Evaluation on training set:

Confusion Matrix:

0 0 21 2 0

0 0 87 29 0

0 0 9347 790 0

0 0 2233 1922 0

0 0 387 827 0

Accuracy: 0.782

Precision: 0.645

Recall: 0.720

F-score: 0.677

Evaluation on validation set:

Confusion Matrix:

0 0 0 1 0

0 0 15 3 0

0 0 1015 111 0

0 0 239 208 0

0 0 45 101 0

Accuracy: 0.773

Precision: 0.627

Recall: 0.704

F-score: 0.662

Evaluation on test set:

Confusion Matrix:

0 0 4 2 0

0 0 33 5 0

0 0 2532 236 0

0 0 561 515 0

0 0 121 220 0

Accuracy: 0.784

Precision: 0.644

Recall: 0.721

F-score: 0.678

### Fazit & Interpretation der Resultate

Auch wenn es intuitiv Features wie zum Beispiel die Anzahl Schlafzimmer nichts auf den Hauszustand ausmachen, kann es eben doch eine gewisse versteckte Korrelation geben, die vom ML Algorithmus ausgenutzt werden kann. Im letzten Versuch wurde insgesamt das beste Ergebnis erzielt mit einer F-Score von 0.678. In der Confusion Matrix sieht man, dass alle Häuser nur als condition = 3 oder condition = 4 vorausgesagt wurden. Dies ist nicht wahnsinnig überraschend, denn die meisten Häuser befinden sich in genau diesen Kategorien. Auch hier wäre deshalb eine ausgeglichene Verteilung der condition in den Trainingsdaten gut, um bessere Resultate erzielen zu können.