Simple Classifier / Logistic Regression

After having worked with the Dataloading part last week, we want to start this week to take a more detailed look into how the training process looks like. So far, our tools are limited and we must restrict ourselves to a simplified model. But nevertheless, this gives us the opportunity to look at the different parts of the training process in more detail and builds up a good base when we turn to more complicated model architectures in the next exercises.

This notebook will demonstrate a simple logistic regression model predicting whether a house is `low-priced` or `expensive`. The data that we will use here is the HousingPrice dataset. Feeding some features in our classifier, the output should then be a score that determines in which category the considered house is.

classifierTeaser

Before we start, let us first import some libraries and code that we will need along the way.

In [1]:
```
from exercise_code.data.csv_dataset import CSVDataset
from exercise_code.data.csv_dataset import FeatureSelectorAndNormalizationT
from exercise_code.data.dataloader import DataLoader

import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import seaborn as sns


pd.options.mode.chained_assignment = None  # default='warn'

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

0. Dataloading and Data Preprocessing

Let us load the data that we want to use for our training. The method `get_housing_data()` is providing you with a training, validation and test set that is ready to use.

For more information about how to prepare the data and what the final data look like, you can have a look at the notebook `housing_data_preprocessing(optional).ipynb`. We reduced our data and the remaining houses in our dataset are now either labeled with `1` and hence categorized as `expensive`, or they are labeled with `0` and hence categorized as `low-priced`.

In [2]:
```
from exercise_code.networks.utils import *

X_train, y_train, X_val, y_val, X_test, y_test, train_dataset = get_housing

print("train data shape:", X_train.shape)
print("train targets shape:", y_train.shape)
print("val data shape:", X_val.shape)
print("val targets shape:", y_val.shape)
print("test data shape:", X_test.shape)
print("test targets shape:", y_test.shape, '\n')

print('The original dataset looks as follows:')
train_dataset.df.head()
```

```
You successfully loaded your data!

train data shape: (533, 1)
train targets shape: (533, 1)
val data shape: (167, 1)
val targets shape: (167, 1)
test data shape: (177, 1)
test targets shape: (177, 1)

The original dataset looks as follows:
```

Out[2]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Ut
529	530	20	RL	NaN	32668	Pave	NaN	IR1	Lvl	/
491	492	50	RL	79.0	9490	Pave	NaN	Reg	Lvl	/
459	460	50	RL	NaN	7015	Pave	NaN	IR1	Bnk	/
279	280	60	RL	83.0	10005	Pave	NaN	Reg	Lvl	/
655	656	160	RM	21.0	1680	Pave	NaN	Reg	Lvl	/

5 rows × 81 columns

The data is now ready and can be used to train our classifier model.

1. Set up a Classifier Model

Let $\mathbf{X} \in \mathbb{R}^{N\times(D+1)}$ be our data with N samples and D feature dimensions. With our classifier model, we want to predict binary labels $\hat{\mathbf{y}} \in \mathbb{R}^{N\times 1}$. Our classifier model should be of the form

$$\hat{\mathbf{y}} = \sigma\left(\mathbf{X} \cdot \mathbf{w}\right),$$

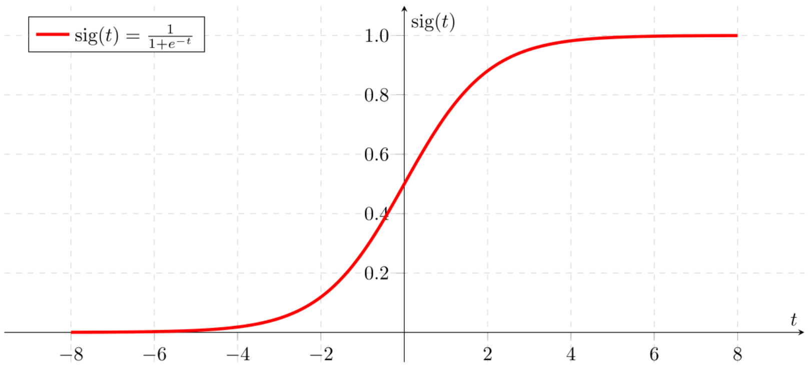
where $\mathbf{w} \in \mathbb{R}^{(D+1)\times 1}$ is the weight matrix of our model.

The **sigmoid function** $\sigma : \mathbb{R} \to [0, 1]$, defined by

$$\sigma(t) = \frac{1}{1+e^{-t}},$$

is used to squash the outputs of the linear layer into the interval $[0, 1]$. Remember that the sigmoid function is a real-valued function. When applying it on a vector, the sigmoid is operating component-wise.

The output of the sigmoid function can be seen as the probability that our sample is indicating a house that can be categorized as `expensive`. As the probability gets closer to 1, our model is more confident that the input sample is in the class `expensive`.



Task: Check Code

Take a look at the implementation of the `Classifier` class in `exercise_code/networks/classifier.py`. To create a `Classifier` object, you need to define the number of features that our classifier model takes as input.

2. Loss: Binary Cross Entropy

For a binary classification like our task, we use a loss function called Binary Cross-Entropy (BCE).

$$BCE(y, \hat{y}) = -y \cdot log(\hat{y}) - (1 - y) \cdot log(1 - \hat{y})$$

where $y \in \mathbb{R}$ is the ground truth and $\hat{y} \in \mathbb{R}$ is the predicted probability of the house being expensive.

Since the BCE function is a non-convex function, there is no closed-form solution for the optimal weights vector. In order to find the optimal parameters for our model, we need to use numeric methods such as Gradient Descent. But let us have a look at that later. First, you have to complete your first task:

Task: Implement

In `exercise_code/networks/loss.py` complete the implementation of the BCE loss function. You need to write the forward and backward pass of BCE as `forward()` and `backward()` function. The backward pass of the loss is needed to later optimize your weights of the model. You can test your implementation by the included testing code in the cell below.

In [3]:
```
from exercise_code.tests.loss_tests import *
from exercise_code.networks.loss import BCE

bce_loss = BCE()
print (BCETest(bce_loss)())
```

```
BCEForwardTest passed.
BCEBackwardTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 2/2
(0, 2)
```

3. Backpropagation

The backpropagation algorithm allows the information from the loss flowing backward through the network in order to compute the gradient of the loss function L w.r.t the weights w of the model.

The key idea of backpropagation is decomposing the derivatives by applying the chain rule to the loss function.

$$\frac{\partial L(w)}{\partial w} = \frac{\partial L(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$

You have already completed the `forward()` and `backward()` pass of the loss function, which can be used to compute the derivative $\frac{\partial L(w)}{\partial \hat{y}}$. In order to compute the second term $\frac{\partial \hat{y}}{\partial w}$, we need to implement a similar `forward()` and `backward()` method in our `Classifier` class.

Backward Pass

The backward pass consists of computing the derivative $\frac{\partial \hat{y}}{\partial w}$. Again, we can decompose this derivative by the chain rule: For $s = X \cdot w$ we obtain

$$\frac{\partial \hat{y}}{\partial w} = \frac{\partial \sigma(s)}{\partial w} = \frac{\partial \sigma(s)}{\partial s} \cdot \frac{\partial s}{\partial w}$$

Hint: Taking track of the dimensions in higher-dimensional settings can make the task a little bit complicated. Make sure you understand the operations here. If you have difficulties, first try to understand the forward and backward pass if the input is only one sample consisting of $D + 1$ features. Then our data matrix has dimension $X \in \mathbb{R}^{1\times(D+1)}$. After you understood this situation, you can go back to the setting where our data matrix has dimension $X \in \mathbb{R}^{N\times(D+1)}$ and consists of N samples each having $D + 1$ features.

Task: Implement

Implement the `forward()` and `backward()` pass as well as the `sigmoid()` function in the `Classifier` class in `exercise_code/networks/classifier.py` . Check your implementation using the following testing code.

In [10]:
```
from exercise_code.networks.classifier import Classifier
from exercise_code.tests.classifier_test import *
test_classifier(Classifier(num_features=2))
```

```
Sigmoid_Of_Zero passed.
Sigmoid_Of_Zero_Array passed.
Sigmoid_Of_100 passed.
Sigmoid_Of_Array_of_100 passed.
Method sigmoid() correctly implemented. Tests passed: 4/4
ClassifierForwardTest passed.
Method forward() correctly implemented. Tests passed: 1/1
ClassifierBackwardTest passed.
Method backward() correctly implemented. Tests passed: 1/1
Congratulations you have passed all the unit tests!!! Tests passed: 6/6
Score: 100/100
```

Out[10]: 100

4. Optimizer and Gradient Descent

Previously, we have successfully dealt with the loss function, which is a method of measuring how well our model fits the given data. The idea of the training process is to adjust iteratively the weights of our model in order to minimize the loss function.

And this is where the optimizer comes in. In each training step, the optimizer updates the weights of the model w.r.t. the output of the loss function, thereby linking the loss function and model parameters together. The goal is to obtain a model which is accurately predicting the class for a new sample.

Any discussion about optimizers needs to begin with the most popular one, and it's called Gradient Descent. This algorithm is used across all types of Machine Learning (and other math problems) to optimize. It's fast, robust, and flexible. Here's how it works:

0. Initialize the weights with random values.
1. Calculate loss with the current weights and the loss function.
2. Calculate the gradient of the loss function w.r.t. the weights.
3. Update weights with the corresponding gradient.
4. Iteratively perform Step 1 to 3 until converges.

The name of the optimizer already hints at the required concept: We use gradients which are very useful for minimizing a function. The gradient of the loss function w.r.t to the weights w of our model tells us how to change our weights w in order to minimize our loss function.

The weights are updated each step as follows:

$$w^{(n+1)} = w^{(n)} - \alpha \cdot \frac{dL}{dw},$$

where $\frac{dL}{dw}$ is the gradient of your loss function w.r.t. the weights w and α is the learning rate which is a predefined positive scalar determining the size of the step.

Task: Implement

In our model, we will use gradient descent to update the weights. Take a look at the `Optimizer` class in the file `networks/optimizer.py` . Your task is now to implement the gradient descent step in the `step()` method. You can test your implementation by the following testing code.

In [11]:
```
from exercise_code.networks.optimizer import Optimizer
from exercise_code.networks.classifier import Classifier
from exercise_code.tests.optimizer_test import *
TestClassifier=Classifier(num_features=2)
TestClassifier.initialize_weights()
test_optimizer(Optimizer(TestClassifier))
```

```
OptimizerStepTest passed.
Congratulations you have passed all the unit tests!!! Tests passed: 1/1
Score: 100/100
```

Out[11]: 100

5. Training

We have now implemented all the necessary parts of our training process, namely:

- **Classifier Model:** We set up a simple classifier model and you implemented the corresponding `forward()` and `backward()` methods.
- **Loss function:** We chose the Binary Cross Entropy Loss for our model to measure the distance between the prediction of our model and the ground-truth labels. You implemented a forward and backward pass for the loss function.
- **Optimizer**: We use the Gradient Descent method to update the weights of our model. Here, you implemented the `step()` function which performs the update of the weights.

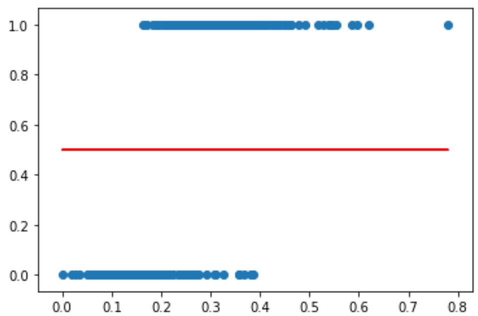
Task: Check Code

Before we start our training and put all the parts together, let us shortly talk about the weight initialization. In `networks/classifier.py` you can check the `Classifier` class. It contains a method called `initialize_weights()` that randomly initializes the weights of our classifier model. Later in the lecture, we will learn about more efficient methods to initialize the weights. But for now, a random initialization as it happens in the `initialize_weights()` method is sufficient.

Let's start with our classifier model and look at its performance before any training happened.

In [12]:
```python
from exercise_code.networks.classifier import Classifier

#initialization
model = Classifier(num_features=1)
model.initialize_weights()

y_out, _ = model(X_train)

# plot the prediction
plt.scatter(X_train, y_train)
plt.plot(X_train, y_out, color='r')
```

Out[12]:
```
[<matplotlib.lines.Line2D at 0x7f93011679d0>]
```



As you can see the predictions of our model without any training are very bad. Let's see how the performance improves when we start our training, which means that we update our weights by applying the gradient descent method. The following cell combines the forward- and backward passes with the gradient update step and performs a training step for our classifier:

Task: Check Code

Note that the `Classifier` class is derived from the more general `Network` class. It is worth having a look at the basis class `Network` in the file `exercise_code/networks/base_networks.py` . We will make use of the `__call__()` method, which computes the forward and backward pass of your classifier. In a similar manner, we use the `__call__()` function for our Loss function.

The following cell performs training with 400 training steps:

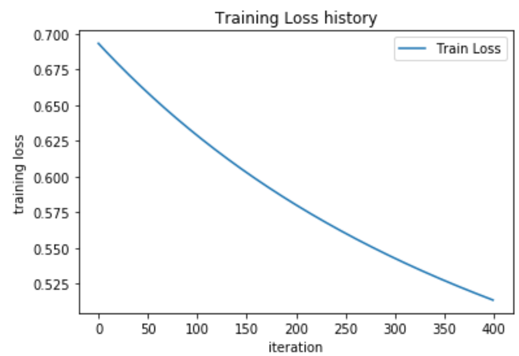
In [13]:
```python
from exercise_code.networks.optimizer import *
from exercise_code.networks.classifier import *
# Hyperparameter Setting, we will specify the loss function we use, and imp
num_features = 1

# initialization
model = Classifier(num_features=num_features)
model.initialize_weights()

loss_func = BCE()
learning_rate = 5e-1
loss_history = []
opt = Optimizer(model,learning_rate)

steps = 400
# Full batch Gradient Descent
for i in range(steps):

    # Enable your model to store the gradient.
    model.train()

    # Compute the output and gradients w.r.t weights of your model for the
    model_forward, model_backward = model(X_train)

    # Compute the loss and gradients w.r.t output of the model.
    loss, loss_grad = loss_func(model_forward, y_train)

    # Use back prop method to get the gradients of loss w.r.t the weights.
    grad = loss_grad * model_backward

    # Compute the average gradient over your batch
    grad = np.mean(grad, 0, keepdims = True)

    # After obtaining the gradients of loss with respect to the weights, we
    # do gradient descent step.
    # Take transpose to have the same shape ([D+1,1]) as weights.
    opt.step(grad.T)

    # Average over the loss of the entire dataset and store it.
    average_loss = np.mean(loss)
    loss_history.append(average_loss)
    if i%10 == 0:
        print("Epoch ",i,"--- Average Loss: ", average_loss)
```

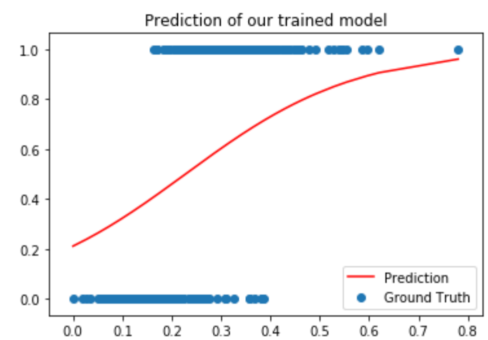
```
Epoch  0 --- Average Loss:  0.6931846493618942
Epoch  10 --- Average Loss:  0.6857938364646906
Epoch  20 --- Average Loss:  0.6786785421779972
Epoch  30 --- Average Loss:  0.6717723482755023
Epoch  40 --- Average Loss:  0.6650653525465111
Epoch  50 --- Average Loss:  0.658551067519396
Epoch  60 --- Average Loss:  0.6522232938017201
Epoch  70 --- Average Loss:  0.6460759503807414
Epoch  80 --- Average Loss:  0.6401030776912475
Epoch  90 --- Average Loss:  0.6342988487602588
Epoch  100 --- Average Loss:  0.6286575788224299
```

```
Epoch  110 --- Average Loss:  0.6231737329709143
Epoch  120 --- Average Loss:  0.6178419319596162
Epoch  130 --- Average Loss:  0.6126569563173097
Epoch  140 --- Average Loss:  0.6076137489448004
Epoch  150 --- Average Loss:  0.6027074163693864
Epoch  160 --- Average Loss:  0.5979332288283317
Epoch  170 --- Average Loss:  0.5932866193461281
Epoch  180 --- Average Loss:  0.5887631819601901
Epoch  190 --- Average Loss:  0.5843586692374213
Epoch  200 --- Average Loss:  0.580068989210767
Epoch  210 --- Average Loss:  0.5758902018511216
Epoch  220 --- Average Loss:  0.5718185151764105
Epoch  230 --- Average Loss:  0.56785028108666
Epoch  240 --- Average Loss:  0.5639819910017176
Epoch  250 --- Average Loss:  0.5602102713671161
Epoch  260 --- Average Loss:  0.55653187908348
Epoch  270 --- Average Loss:  0.5529436969058714
Epoch  280 --- Average Loss:  0.5494427288515231
Epoch  290 --- Average Loss:  0.5460260956474575
Epoch  300 --- Average Loss:  0.5426910302434803
Epoch  310 --- Average Loss:  0.5394348734108529
Epoch  320 --- Average Loss:  0.5362550694425366
Epoch  330 --- Average Loss:  0.53314916196715
Epoch  340 --- Average Loss:  0.5301147898856238
Epoch  350 --- Average Loss:  0.5271496834368935
Epoch  360 --- Average Loss:  0.5242516603967665
Epoch  370 --- Average Loss:  0.5214186224122841
Epoch  380 --- Average Loss:  0.5186485514724081
Epoch  390 --- Average Loss:  0.5159395065146465
```

We can see that our average loss is decreasing as expected. Let us visualize the average loss and the prediction after our short training:

In [14]:
```
# Plot the loss history to see how it goes after several steps of gradient
plt.plot(loss_history, label = 'Train Loss')
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.legend()
plt.show()


# forward pass
y_out, _ = model(X_train)


# plot the prediction
plt.scatter(X_train, y_train, label = 'Ground Truth')
inds = X_train.argsort(0).flatten()
plt.plot(X_train[inds], y_out[inds], color='r', label = 'Prediction')
plt.title('Prediction of our trained model')
plt.legend()
plt.show()
```





This looks pretty good already and our model gets better in explaining the underlying relationship of data.

6. Solver

Now we want to put everything we have learned so far together in an organized and concise way, that provides easy access to train a network/model in your own script/code. The purpose of a solver is mainly to provide an abstraction for all the gritty details behind training your parameters, such as logging your progress, optimizing your model, and handling your data.

This part of the exercise will require you to complete the missing code in the `Solver` class and to train your model end to end.

Task: Implement

Open the file `exercise_code/solver.py` and have a look at the `Solver` class. The `_step()` function is representing one single training step. So when using the Gradient Descent method, it represents one single update step using the Gradient Descent method. Your task is now to finalize this `_step()` function. You can test your implementation with the testing code included in the following cell.

Hint: The implementation of the `_step()` function is very similar to the implementation of a training step as we observed above. You may have a look at that part first.

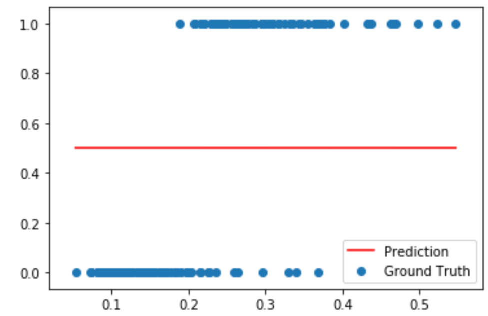
In [43]:
```
from exercise_code.solver import Solver
from exercise_code.networks.classifier import Classifier
from exercise_code.tests.solver_tests import *
weights = np.array([[0.1],[0.1]])
TestClassifier = Classifier(num_features=1)
TestClassifier.initialize_weights(weights)
learning_rate = 5e-1
data = {'X_train': X_train, 'y_train': y_train,
        'X_val': X_val, 'y_val': y_val}
loss = BCE()
solver = Solver(TestClassifier,data,loss,learning_rate,verbose=True)

test_solver(solver)
```

```
model forward:  [[0.53521384]
 [0.53081996]
 [0.52906009]
 [0.52853903]
 [0.5282104 ]
 [0.53028504]
 [0.5443911 ]
 [0.52975943]
 [0.53271975]
 [0.52763759]
 [0.52849208]
 [0.52806016]
 [0.52724315]
 [0.52636963]
 [0.53138296]
 [0.52912111]
 [0.53174886]
 [0.53174417]
 [0.5337279 ]
```

After having successfully implemented the `step()` function in the `Optimizer` class, let us now train our classifier. We train our model with a learning rate $\lambda = 0.1$ and with 25000 epochs. Your model should reach an accuracy which is higher than 85%.

In [36]:
```
from exercise_code.solver import Solver
from exercise_code.networks.utils import test_accuracy
from exercise_code.networks.classifier import Classifier
# Select the number of features, you want your task to train on.
# Feel free to play with the sizes.
num_features = 1

# initialize model and weights
model = Classifier(num_features=num_features)
model.initialize_weights()

y_out, _ = model(X_test)

accuracy = test_accuracy(y_out, y_test)
print("Accuracy BEFORE training {:.1f}%".format(accuracy*100))


if np.shape(X_test)[1]==1:
    plt.scatter(X_test, y_test, label = "Ground Truth")
    inds = X_test.flatten().argsort(0)
    plt.plot(X_test[inds], y_out[inds], color='r', label = "Prediction")
    plt.legend()
    plt.show()

data = {'X_train': X_train, 'y_train': y_train,
        'X_val': X_val, 'y_val': y_val}

#We use the BCE loss
loss = BCE()

# Please use these hyperparmeter as we also use them later in the evaluatio
learning_rate = 1e-1
epochs = 25000

# Setup for the actual solver that's going to do the job of training
# the model on the given data. set 'verbose=True' to see real time
# progress of the training.
solver = Solver(model,
                data,
                loss,
                learning_rate,
                verbose=True,
                print_every = 1000)
# Train the model, and look at the results.
solver.train(epochs)


# Test final performance
y_out, _ = model(X_test)

accuracy = test_accuracy(y_out, y_test)
print("Accuracy AFTER training {:.1f}%".format(accuracy*100))
```

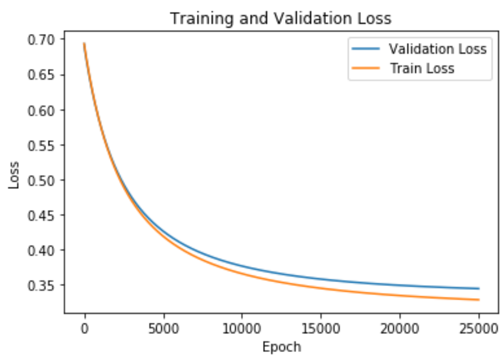
```
Accuracy BEFORE training 41.8%
```

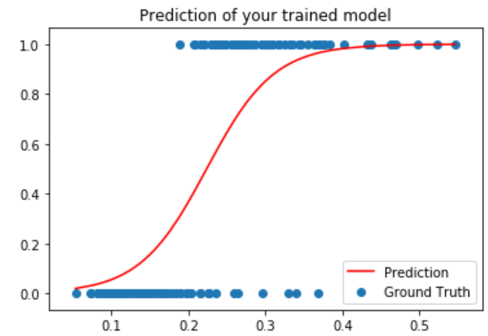




```
(Epoch 0 / 25000) train loss: 0.693044; val_loss: 0.692754
(Epoch 1000 / 25000) train loss: 0.580046; val_loss: 0.580283
(Epoch 2000 / 25000) train loss: 0.513301; val_loss: 0.516032
(Epoch 3000 / 25000) train loss: 0.470397; val_loss: 0.475013
(Epoch 4000 / 25000) train loss: 0.440860; val_loss: 0.446909
(Epoch 5000 / 25000) train loss: 0.419452; val_loss: 0.426636
(Epoch 6000 / 25000) train loss: 0.403316; val_loss: 0.411438
(Epoch 7000 / 25000) train loss: 0.390781; val_loss: 0.399701
(Epoch 8000 / 25000) train loss: 0.380806; val_loss: 0.390423
(Epoch 9000 / 25000) train loss: 0.372711; val_loss: 0.382949
(Epoch 10000 / 25000) train loss: 0.366036; val_loss: 0.376833
(Epoch 11000 / 25000) train loss: 0.360456; val_loss: 0.371764
(Epoch 12000 / 25000) train loss: 0.355739; val_loss: 0.367516
(Epoch 13000 / 25000) train loss: 0.351712; val_loss: 0.363923
(Epoch 14000 / 25000) train loss: 0.348244; val_loss: 0.360859
(Epoch 15000 / 25000) train loss: 0.345235; val_loss: 0.358229
(Epoch 16000 / 25000) train loss: 0.342607; val_loss: 0.355957
(Epoch 17000 / 25000) train loss: 0.340299; val_loss: 0.353983
(Epoch 18000 / 25000) train loss: 0.338261; val_loss: 0.352262
(Epoch 19000 / 25000) train loss: 0.336453; val_loss: 0.350753
(Epoch 20000 / 25000) train loss: 0.334842; val_loss: 0.349427
(Epoch 21000 / 25000) train loss: 0.333401; val_loss: 0.348256
(Epoch 22000 / 25000) train loss: 0.332108; val_loss: 0.347221
(Epoch 23000 / 25000) train loss: 0.330944; val_loss: 0.346302
(Epoch 24000 / 25000) train loss: 0.329893; val_loss: 0.345486
Accuracy AFTER training 91.5%
```

During the training process losses in each epoch are stored in the lists `solver.train_loss_history` and `solver.val_loss_history`. We can use them to plot the training result easily.

In [37]:
```
plt.plot(solver.val_loss_history, label = "Validation Loss")
plt.plot(solver.train_loss_history, label = "Train Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title('Training and Validation Loss')
plt.show()


if np.shape(X_test)[1]==1:

    plt.scatter(X_test, y_test, label = "Ground Truth")
    inds = X_test.argsort(0).flatten()
    plt.plot(X_test[inds], y_out[inds], color='r', label = "Prediction")
    plt.legend()
    plt.title('Prediction of your trained model')
    plt.show()
```





7. Save your BCE Loss, Classifier and Solver for Submission

Your model should be trained now and able to predict whether a house is expensive or not. Hooooooray, you trained your very first model! The model will be saved as a pickle file to `models/simple_classifier.p` .

In [38]:
```
from exercise_code.tests import save_pickle

save_pickle(
    data_dict={
        "BCE_class": BCE,
        "Classifier_class": Classifier,
        "Optimizer": Optimizer,
        "Solver_class": Solver
    },
    file_name="simple_classifier.p"
)
```

Submission Instructions

Now, that you have completed the necessary parts in the notebook, you can go on and submit your files.

1. Go on our submission page (https://i2dl.vc.in.tum.de/), register for an account and login. We use your matriculation number and send an email with the login details to the mail account associated. When in doubt, login into tum-online and check your mails there. You will get an id which we need in the next step.
2. Log into our submission page (https://i2dl.vc.in.tum.de/) with your account details and upload the zip file.
3. Your submission will be evaluated by our system and you will get feedback about the performance of it. You will get an email with your score as well as a message if you have surpassed the threshold.
4. Within the working period, you can submit as many solutions as you want to get the best possible score.

In [39]:
```
from exercise_code.submit import submit_exercise

submit_exercise('exercise04')
```

```
relevant folders: ['exercise_code', 'models']
notebooks files: ['housing_data_preprocessing(optional).ipynb', '1_simple
_classifier.ipynb']
Adding folder exercise_code
Adding folder models
Adding notebook housing_data_preprocessing(optional).ipynb
Adding notebook 1_simple_classifier.ipynb
Zipping successful! Zip is stored under: /Users/meiqiliu/Downloads/exerci
se_04/exercise04.zip
```

Submission Goals

For this exercise we only test your implementations which are tested throughout the notebook. In total we have 10 test cases where you are required to complete 8 of. Here is an overview split among the notebook:

- Goal:
  - To implement:
    1. `exercise_code/networks/loss.py` : `forward()` , `backward()`
    2. `exercise_code/networks/classifier.py` : `forward()` , `backward()` , `sigmoid()`
    3. `exercise_code/networks/optimizer.py` : `step()`
    4. `exercise_code/solver.py` : `_step()`
  - Test cases:
    1. Does `forward()` of `BCE` return the correct value?
    2. Does `backward()` of `BCE` return the correct value?
    3. Does `sigmoid()` of `Classifier` return the correct value when `x=0` ?
    4. Does `sigmoid()` of `Classifier` return the correct value when `x=np.array([0,0,0,0,0])` ?
    5. Does `sigmoid()` of `Classifier` return the correct value when `x=100` ?
    6. Does `sigmoid()` of `Classifier` return the correct value when `x=np.asarray([100, 100, 100, 100, 100])` ?
    7. Does `forward()` of `Classifier` return the correct value?
    8. Does `backward()` of `Classifier` return the correct value?
    9. Does `Optimizer` update the model parameter correctly?
    10. Does `Solver` update the model parameter correctly?
- Reachable points [0, 100]: 0 if not implemented, 100 if all tests passed, 10 per passed test
- Threshold to clear exercise: 80
- Submission start: **May 06, 2021 13.00**
- Submission deadline: **May 12, 2021 15.59**
- You can make multiple submission until the deadline. Your **best submission** will be considered for bonus.