IFT 6390 Homework 2

Arlie Coles (20121051) and Yue (Violet) Guo (20120727)

October 16, 2018

1 Linear and non-linear regularized regression

Problem 1. *Linear regression.*

Solution.

1. The parameters θ consist of $\{w, b\}$, where w is a d-dimensional vector (the weight matrix) and b is a 1-dimensional scalar (the bias term).

2. The empirical risk:
$$\begin{aligned}\hat{R}(f, D) &= \sum_{(x,y)\in D} L(f(x) - t)^2 \\ &= \sum_{(x,t)\in D} \left(w^T x + b - t\right)^2\end{aligned}$$

3. The formulation of Empirical Risk Minimization on this problem:
$$\begin{aligned}\hat{f}(D_{\text{train}}) &= \text{argmin}_{f\in F}\hat{R}(f, D_{\text{train}}) \\ &= \text{argmin}_{\theta=\{w,b\}} \sum_{(x,t)\in D} \left(w^T x + b - t\right)^2\end{aligned}$$

4. To calculate the gradient, first we calculate the partial derivative of the empirical risk with respect to vector **w**:
$$\begin{aligned}\frac{\partial \hat{R}}{\partial w_j} &= \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})(x_j) \\ &= \sum_{i=1}^{n} 2x_j(w^T x_i + b - t^{(i)})\end{aligned}$$

Then we calculate the partial derivative of the empirical risk with respect to b:

$$\begin{aligned}\frac{\partial \hat{R}}{\partial b} &= \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})(1) \\ &= \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})\end{aligned}$$

Therefore, the gradient is the vector of these partial derivatives:

$$\nabla \hat{R} = \begin{pmatrix} \sum_{i=1}^{n} 2x_j(w^T x_i + b - t^{(i)}) \\ \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)}) \end{pmatrix}$$

5. If we let the error for a given point be $f(x) - t$, we can substitute the definition of $f(x)$ like so:

$$\begin{aligned}&f(x) - t \\ &= (w^T x + b) - t\end{aligned}$$

We remark that this is proportional to both elements of the gradient of the empirical risk. Therefore, on an intuitive level, the magnitude of the gradient reflects how much error can be "improved upon" if we take a step in the opposite direction, as is done in gradient descent.

Problem 2. *Ridge regression.*

Solution.

1. To express the gradient of the ridge-regression empirical risk, first we express the risk itself:

$$\tilde{R} = \sum_{(x,y)\in D} (w^T x + b - t)^2 + \lambda \|w\|_2^2$$

Then we take the derivative of the empirical risk with respect to w:

$$\frac{\partial \tilde{R}}{\partial w_j} = \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})(x_j) + 2\lambda w_j$$
$$= \sum_{i=1}^{n} 2(x_j)(w^T x_i + b - t^{(i)}) + 2\lambda w_j$$

Then we take the derivative of the empirical risk with respect to b:

$$\frac{\partial \hat{R}}{\partial b} = \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})(1)$$
$$= \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)})$$

Therefore, the gradient is the vector of these partial derivatives:

$$\nabla \hat{R} = \begin{pmatrix} \sum_{i=1}^{n} 2(x_j)(w^T x_i + b - t^{(i)}) + 2\lambda w_j \\ \sum_{i=1}^{n} 2(w^T x_i + b - t^{(i)}) \end{pmatrix}$$

The main difference between this and the un-regularized gradient is the continued presence of the λ term, i.e., the regularization term persists in the gradient and results in larger weights being penalized more heavily.

2. Pseudocode for gradient descent:

```
initialize theta = [weight, bias] randomly.
n = step size.
lambda = regularization hyperparemeter.
data = training inputs.
target = training targets.
j = number of steps.
for j steps:
    #  Full-batch gradient descent
    # Update weight
    pdv_w = avg(2*(weight * data + bias - target) * data)
    regularizer = avg(2 * lambda * weight)
    pdv_w = pdv_w + regularizer
    weight = weight - self.stepSize * pdv_w
    # Update bias
```

```
pdv_b = avg(2*(weight * data + bias - target))
bias = bias - self.stepSize * pdv_b
```

3. In matrix form, if we define $\theta = \begin{pmatrix} w \\ b \end{pmatrix}$ where $b = 0$ (or b as empty), the empirical risk can be written as:

$$\begin{aligned} &(X\theta - t)^T(X\theta - t) + ||\theta||^2 \\ =&(\theta^T X^T - t^T)(X\theta - t) + ||\theta||^2 \\ =&\theta^T X^T X\theta - t\theta^T X^T - t^T X\theta + t^T t + ||\theta||^2 \end{aligned}$$

Then, its gradient can be expressed by taking the derivative with respect to θ:

$$\begin{aligned} \frac{\partial}{\partial\theta} &= (2X^T X)\theta - X^T t - X^T t + 2\lambda\theta \\ &= (2X^T X)\theta - 2X^T t + 2\lambda\theta \end{aligned}$$

4. Matrix form analytical solution. Also, what happens when $N < d$ and $\lambda = 0$?

$$\begin{aligned} (2X^T X)\theta - 2X^T t + 2\lambda\theta &= 0 \\ (X^T X)\theta - X^T t + \lambda\theta &= 0 \\ (X^T X + \lambda I)\theta &= X^T t \\ \theta &= (X^T X + \lambda I)^{-1} X^T t \end{aligned}$$

If $\lambda = 0$, our expression simplifies to

$$\theta = (X^T X)^{-1} X^T t$$

When $N < d$, columns of X are linearly dependent. rank($X^T X$) = N (see proof below), where $N < d$, and X is $d \times d$. Therefore, it's not a matrix of full rank and it is not invertible.

In this case, we can apply other numerical methods such as a pseudo inverse

Proof. From this theorem, $rank(A^T A) \leq min(rank(A^T), rank(A))$, we have that $X^T X$ has rank N since $min(rank(X^T), rank(X)) = N$ □

Problem 3. *Regression with a fixed non-linear preprocessing.*

Solution.

1. When x is one-dimensional:

$$\tilde{f}(x) = f(\phi_{polyk}(x)) = w^T \begin{pmatrix} x \\ x^2 \\ \vdots \\ x^k \end{pmatrix} + b$$

2. The parameters are $k \in \mathbb{R}$, 1 dimensional, and w, which is k dimensional

3. For $d \geq 2$,

$$\phi_{poly1}(x) = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

Combination tile: Note: this is more of a diagram to visualize all the terms in ϕ, not a mathematically defined matrix, and hence blank spots.

$$\phi_{poly2}(x) = \begin{bmatrix} & x_1 & x_2 \\ x_1 & x_1^2 & x_1x_2 \\ x_2 & x_2x_1 & x_2^2 \end{bmatrix}$$

After removing redundancy, this collapses into: $\phi_{poly2}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix}$

Note: this is more of a diagram to visualize all the terms in ϕ, not a mathematically defined matrix, and hence blank spots. Combination tile:

$$\phi_{poly3}(x) = \begin{bmatrix} & x_1 & x_1^2 & x_2 & x_2^2 \\ x_1 & x_1^2 & x_1^3 & x_1x_2 & x_1x_2^2 \\ x_1^2 & x_1^3 & & x_1^2x_2 & \\ x_2 & x_2x_1 & x_2x_1^2 & x_2^2 & x_2^3 \\ x_2^2 & x_2^2x_1 & & x_2^3 & \end{bmatrix}$$

After removing redundancy, this collapses into

$$\phi_{poly3}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \\ x_1^2x_2 \\ x_1x_2^2 \\ x_1^3 \\ x_2^3 \end{bmatrix}$$

4. Dimension of $\phi_{polyk}(x)$:
   We can start by calculating the number of ways of placing d barriers among $k+d+1$ balls (using a typical combinatoric balls-and-bins image), where the bins define variables and interaction terms whose exponents must sum to k. This corresponds to $k+d$ places where a barrier might be placed, so we can represent the number of ways to place d barriers in $k+d$ places as $\binom{k+d}{d}$. Then, since we want to exclude the case of no variable (i.e. only a constant term with the exponent on all variables as 0), we subtract one from this count, giving:

$$\binom{k+d}{d} - 1$$

2 Practical part

Please see the attached iPython notebook for the code and writup for the practical part. *Note to TAs: our notebook runs on Python 3.*

We wrote a class of hyperparameter initializer and a stochastic gradient descent function. For every question, we simply declare a new object with varying parameters.

Instructions on how to run the code:

1. Open the 'Practical.ipynb' file

2. Click ‘Run All‘

3. Note: if you wish to re-run any part of the code, please click ‘Restart‘ kernel or ‘Restart and Run all‘ in order not to interfere the order in which we set our random seeds.

2. Practical Part

Arlie Coles (20121051) and Yue (Violet) Guo (20120727)

Instructions:

1. Open the `Practical.ipynb` file
2. Click `Run All`
3. Note: if you wish to re-run any part of the code, please click `Restart` kernel or `Restart and Run all` in order not to interfere the order in which we set our random seeds.

1. Ridge Regression

We implement ridge regression as `regression_gradient`, a function of a `gradDescent` class:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
np.random.seed(5)


class gradDescent():
    def __init__(self, weightedDecay = 0.01, stepSize = 0.01, numberSteps = 4000):
        self.weightedDecay = weightedDecay
        self.stepSize = stepSize
        self.numberSteps = numberSteps

    def regression_gradient(self, weight, bias, data, target):
        '''
        bias: 1d scalar
        weightedDecay: λ
        stepSize: η
        x: data
        t: target

        '''
        if(data.ndim == 1):
            data = data.reshape(data.size, 1)
        # Initialize weight randomly
        weight = np.random.randn(data.shape[1])

        # Full-batch gradient descent
        for i in range(0, self.numberSteps):
            t1 = (np.dot(data, weight) + bias - target)
            if(t1.ndim == 1):
                t1 = t1.reshape(t1.size, 1)
            grad = 2 * np.multiply(data, t1)

            # Calculate the pdv of the bias
            biasGrad =2*(np.dot(data, weight) + bias - target)
            biasGrad = np.mean(biasGrad, axis = 0)

            # Calculate the pdv of the weights and do regularization
            weightRegGrad = 2 * (self.weightedDecay* weight)
            regGrad = np.mean(weightRegGrad + grad, axis = 0)

            # Update parameters
            bias = bias - self.stepSize*biasGrad

            weight = weight - self.stepSize*regGrad

        return weight, bias
```

2. Draw Dn from h(x)

We define our target function $h(x)$, and then draw a dataset D_n from it:

In [2]:
```
def hX(x):
    '''
    defines the target function
    '''
    return np.sin(x) + 0.3*x -1

data = np.random.uniform(-5,5,15)
target = hX(data)
dataDn = [data, target]
```

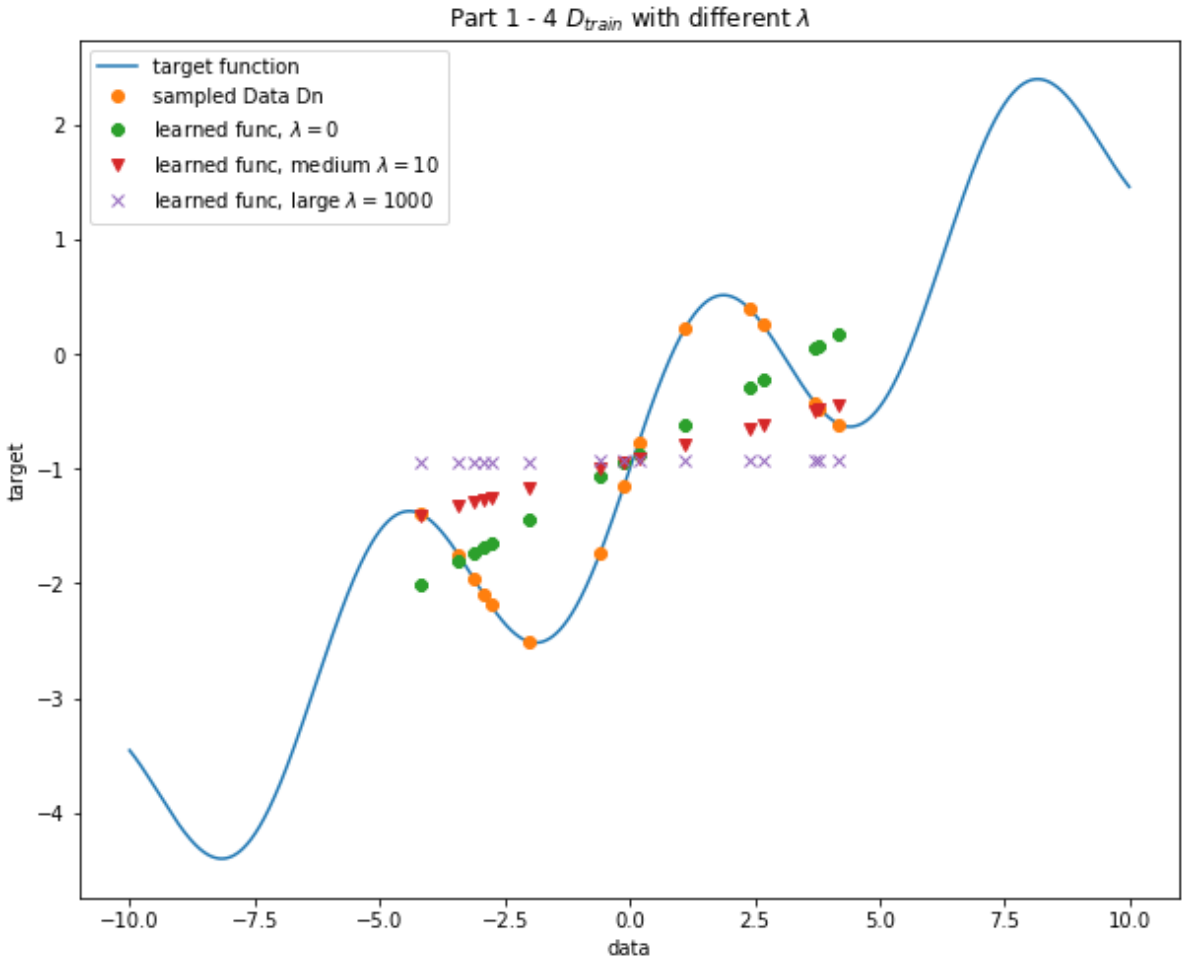
3. Train without regularization, λ = 0

We plot h(x), Dn, and our prediction function $f(X) = w^T X + b$ in the following cell.

4. Part 3 revisited, with different λ values

We extend the original plot to also plot predictions with W regularized by $\lambda \ ||w||^2$, choosing an intermediate and large value for λ.

In [3]:
```
#initialize weights
weight = np.random.rand(1)
bias = 0

#param holders
regGDsmallParam = []
regGDMedParam = []
regGDLargeParam = []


# No Lambda
regGDnoRegParam = []
regGDnoReg = gradDescent(weightedDecay = 0, stepSize = 0.00025)
regGDnoRegParam = regGDnoReg.regression_gradient(weight, bias, dataDn[0
], dataDn[1])


# Medium lambda
weight = np.random.rand(1)
bias = 0
regGDMed = gradDescent(weightedDecay = 10, stepSize = 0.00025)
regGDMedParam = regGDMed.regression_gradient(weight, bias, dataDn[0], da
taDn[1])

# Large lambda
weight = np.random.rand(1)
bias = 0
regGDLarge = gradDescent(weightedDecay = 1000, stepSize = 0.00025)
regGDLargeParam = regGDLarge.regression_gradient(weight, bias, dataDn[0
], dataDn[1])

# Use the learned parameters to define a linear function
learnedFuncZero = regGDnoRegParam[0] * dataDn[0] + regGDnoRegParam[1]
learnedFuncMed = (regGDMedParam[0] * dataDn[0]) + regGDMedParam[1]
learnedFuncLarge = (regGDLargeParam[0] * dataDn[0]) + regGDLargeParam[1]

# Then plot the functions
xvals = np.arange(-10, 10, 0.01)
plt.rcParams['figure.figsize'] = [10, 8]

plt.plot(xvals, hX(xvals), label = "target function")
plt.plot(dataDn[0], dataDn[1], 'o', label="sampled Data Dn")
plt.plot(dataDn[0], learnedFuncZero, '8', label = "learned func, $\lambd
a = 0$")
plt.plot(dataDn[0], learnedFuncMed, 'v', label = "learned func, medium
 $\lambda = 10$")
plt.plot(dataDn[0], learnedFuncLarge, 'x', label = "learned func, large
 $\lambda = 1000$")
plt.xlabel("data")
plt.ylabel("target")
plt.title("Part 1 - 4 $D_{train}$ with different $\lambda$")
plt.legend(loc='best')
plt.show()
```

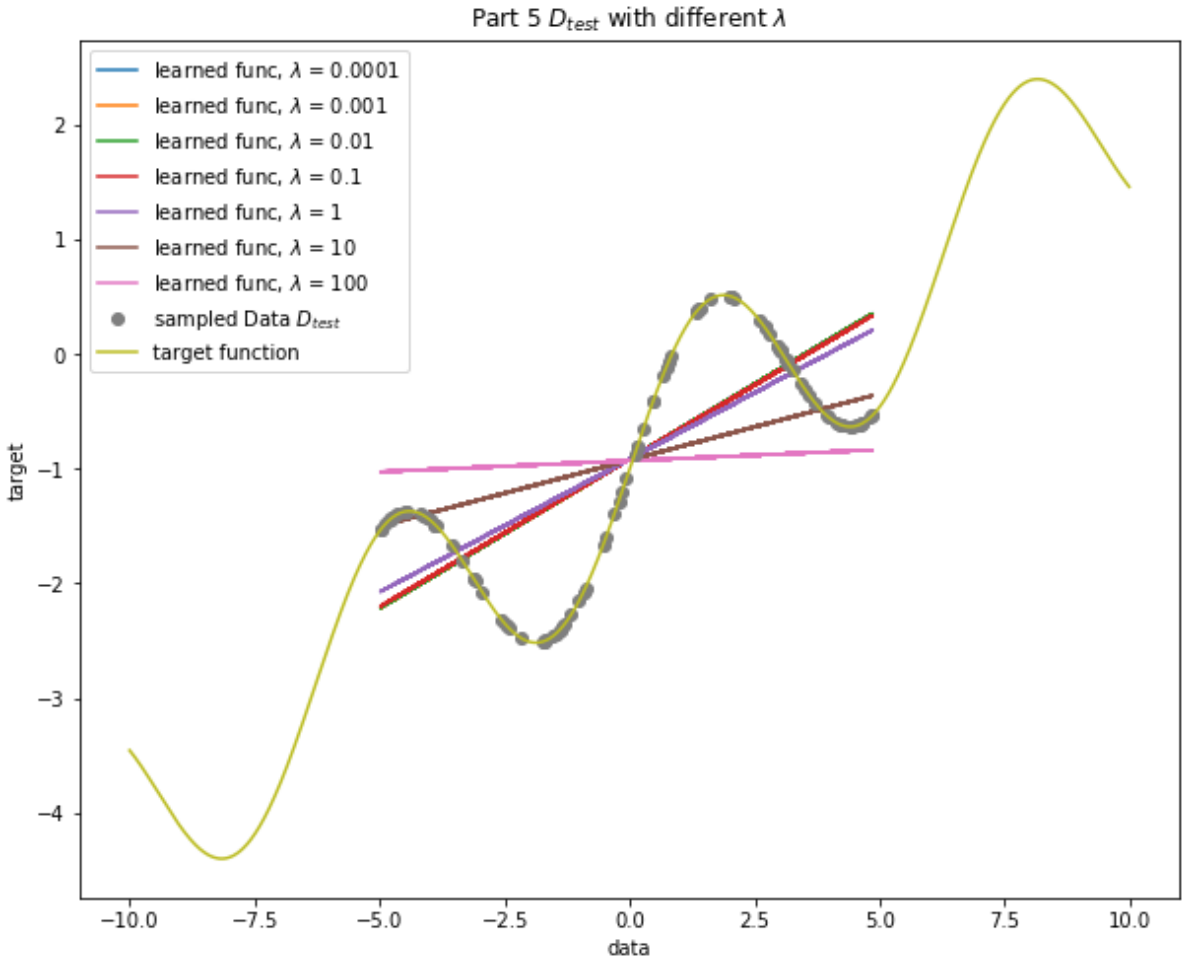


5. Sample D_{test} from $h(X)$

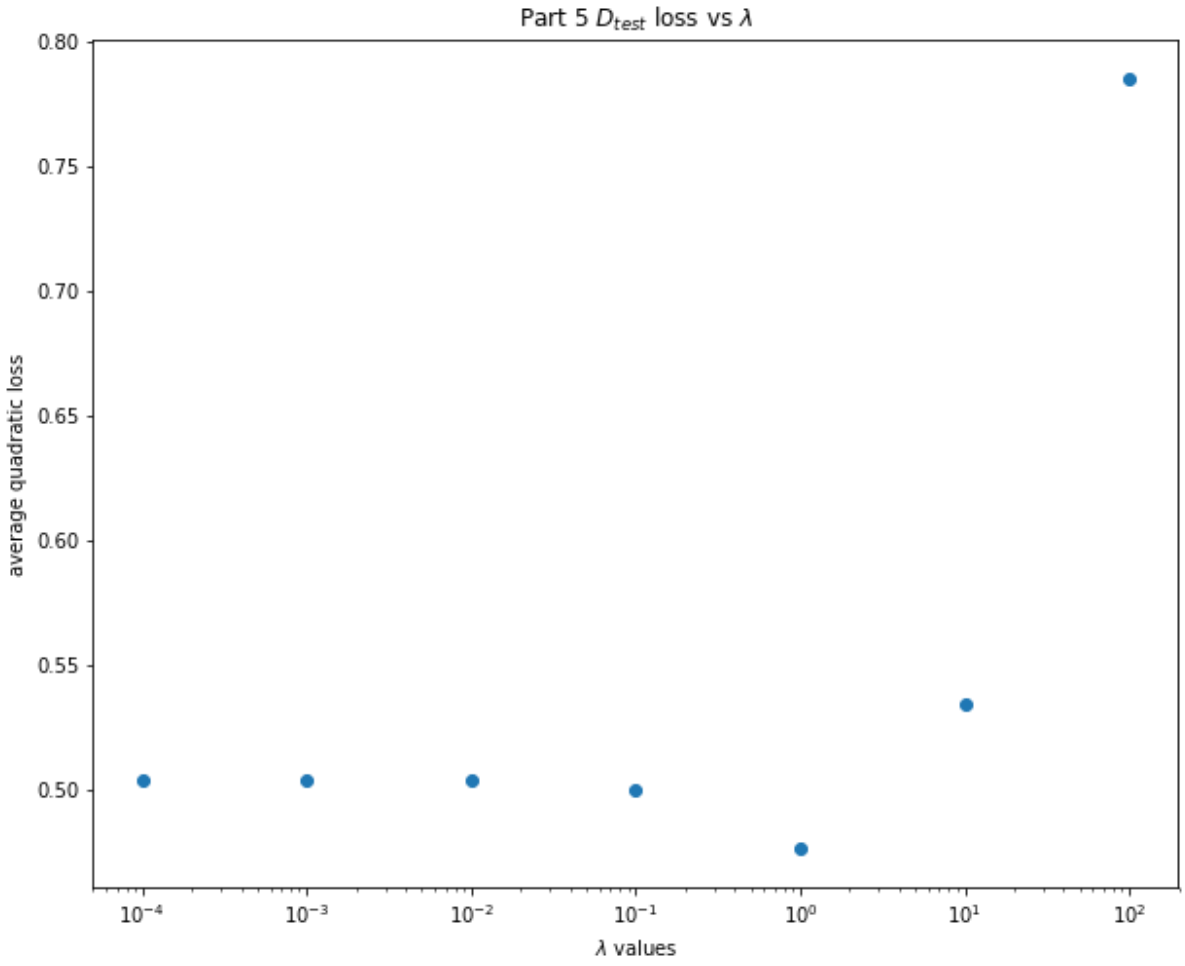
We sample our test set D_{test} and train models on D_n , using λ = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100].

Then we plot λ on the x-axis, and loss on the y-axis (N.B. to TAs: we plotted λ on a log scale):

In [4]:
```
# Sample D_test
data = np.random.uniform(-5, 5, 100)
target = hX(data)
dataDtest = [data, target]

lambdaVals = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]
weight = np.random.rand(1)
bias = 0

lossArr = []

# Do GD for each lambda value and plot
for i in lambdaVals:
    regGD = gradDescent(weightedDecay = i, stepSize = 0.00025)
    regGDParam = regGD.regression_gradient(weight, bias, dataDn[0], data
Dn[1])

    xvals = np.arange(-10, 10, 0.01)
    learnedFunc = (regGDParam[0] * dataDtest[0]) + regGDParam[1]

    # Calculate loss
    loss = np.mean((learnedFunc - dataDtest[1])**2)
    lossArr.append(loss)

    # Uncomment to plot
    plt.rcParams['figure.figsize'] = [10, 8]
    plt.plot(dataDtest[0], learnedFunc, label = "learned func, $\lambda$
 = {}".format(i) )

plt.plot(dataDtest[0], dataDtest[1], 'o', label="sampled Data $D_{test}
$")
plt.plot(xvals, hX(xvals), label = "target function")
plt.xlabel("data")
plt.ylabel("target")
plt.title("Part 5 $D_{test}$ with different $\lambda$")
plt.legend(loc='best')
plt.show()
```


Part 5 D_{test} with different λ
learned func, $\lambda = 0.0001$
learned func, $\lambda = 0.001$
learned func, $\lambda = 0.01$
learned func, $\lambda = 0.1$
learned func, $\lambda = 1$
learned func, $\lambda = 10$
learned func, $\lambda = 100$
sampled Data D_{test}
target function
target
data
2
1
0
−1
−2
−3
−4
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

In [5]:
```
# Plot the error
plt.plot(np.array(lambdaVals),np.array(lossArr), 'o')
plt.xlabel("$\lambda$ values")
plt.xticks(range(len(lambdaVals)), np.array(lambdaVals))
plt.xscale('log')
plt.ylabel("average quadratic loss")
plt.title("Part 5 $D_{test}$ loss vs $\lambda$")
plt.show()
```

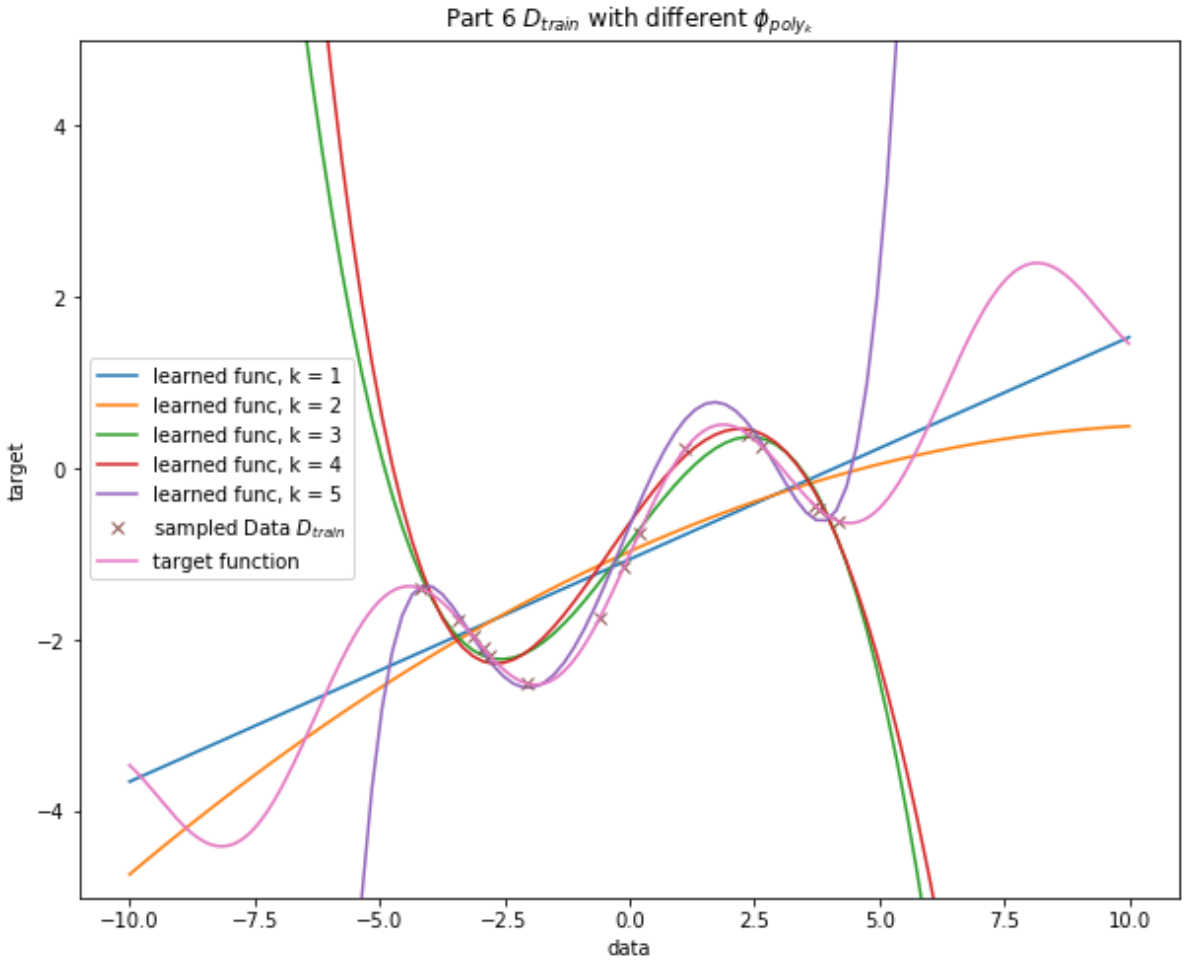


6. Nonlinear preprocessing

We set $\lambda = 0.01$ and try fitting different degrees of polynomials.

In [9]:
```
# Define step size and number of steps for each degree
kDegreeArr = [1,2, 3, 4, 5]


stepsizeArr = [0.01, 0.001, 1.0e-6, 1.0e-7, 1.0e-7]

numStepArr = [400, 40000, int(3e6),int(2e7), int(2e7)]
#numStepArr = [1,1,1,1,1]

plt.rcParams['figure.figsize'] = [10, 8]
plt.ylim(top = 5, bottom=-5)

lossArr_train = []
lossArr_test = []
learned_param = []
regLossArr_train = []
np.random.seed(5)

# Function to do a mapping from a 1d X.
def to_map(input_data, k):
    if k == 1: # Do nothing
        return input_data
    to_map = [input_data]
    for i in range(2, k+1):
        raised = np.power(input_data, i)
        to_map.append(raised)
    dataK = np.column_stack(to_map)[0]
    return dataK

for k in kDegreeArr:
    np.random.seed(5)

    # Initialize weights and bias randomly according to degree
    weight = np.random.rand(k)
    bias = 0

    # Get a mapping of the training data
    dataDM = []
    for data in dataDn[0]:
        dataDM.append(to_map(data, k))
    dataDM = np.array(dataDM)

    # Get a mapping of the dummy data (to make plot smooth)
    dataK = np.array(np.linspace(-10, 10, 100))
    dataKn = [dataK , hX(dataK)]
    dataKM = []
    for data in dataK:
        dataKM.append(to_map(data, k))
    dataKM = np.array(dataKM)

    # Do the descent
    regGD = 0
    regGD = gradDescent(weightedDecay = 0.01,
                        stepSize = stepsizeArr[kDegreeArr.index(k)],
                        numberSteps= numStepArr[kDegreeArr.index(k)])
    regGDParam = regGD.regression_gradient(weight, bias, dataDM, dataDn[
```

```
1])
    learned_param.append([regGDParam[0], regGDParam[1]])

    # Define the function using the learned parameters
    learnedFunc, learnedFunc_plot = [], []
    for i, point in enumerate(dataDM):
        learnedFunc.append(np.dot(regGDParam[0], point) + regGDParam[1])
    learnedFunc = np.array(learnedFunc)

    for point in dataKM:
        learnedFunc_plot.append(np.dot(regGDParam[0], point) + regGDPara
m[1])
    learnedFunc_plot = np.array(learnedFunc_plot)

    plt.plot(dataK, learnedFunc_plot, label = "learned func, k = {}".for
mat(k))

    # Calculate training error (for next part)
    loss = np.mean((learnedFunc - dataDn[1])**2)
    lossArr_train.append(loss)

    regLoss = loss + 0.01 * np.sum(np.power(regGDParam[0], 2))
    regLossArr_train.append(regLoss)


# Do the rest of the plot
xvals = np.arange(-10, 10, 0.01)
plt.plot(dataDn[0], dataDn[1], 'x', label="sampled Data $D_{train}$")

plt.plot(xvals, hX(xvals), label = "target function")
plt.xlabel("data")
plt.ylabel("target")
plt.title("Part 6 $D_{train}$ with different $\phi_{{poly}_k}$")
plt.legend(loc='best')
plt.show()
```



7. Comment on losses

We plot the the empirical risk (loss on D_n) and the true risk (loss on D_{test}) with increasing l below.

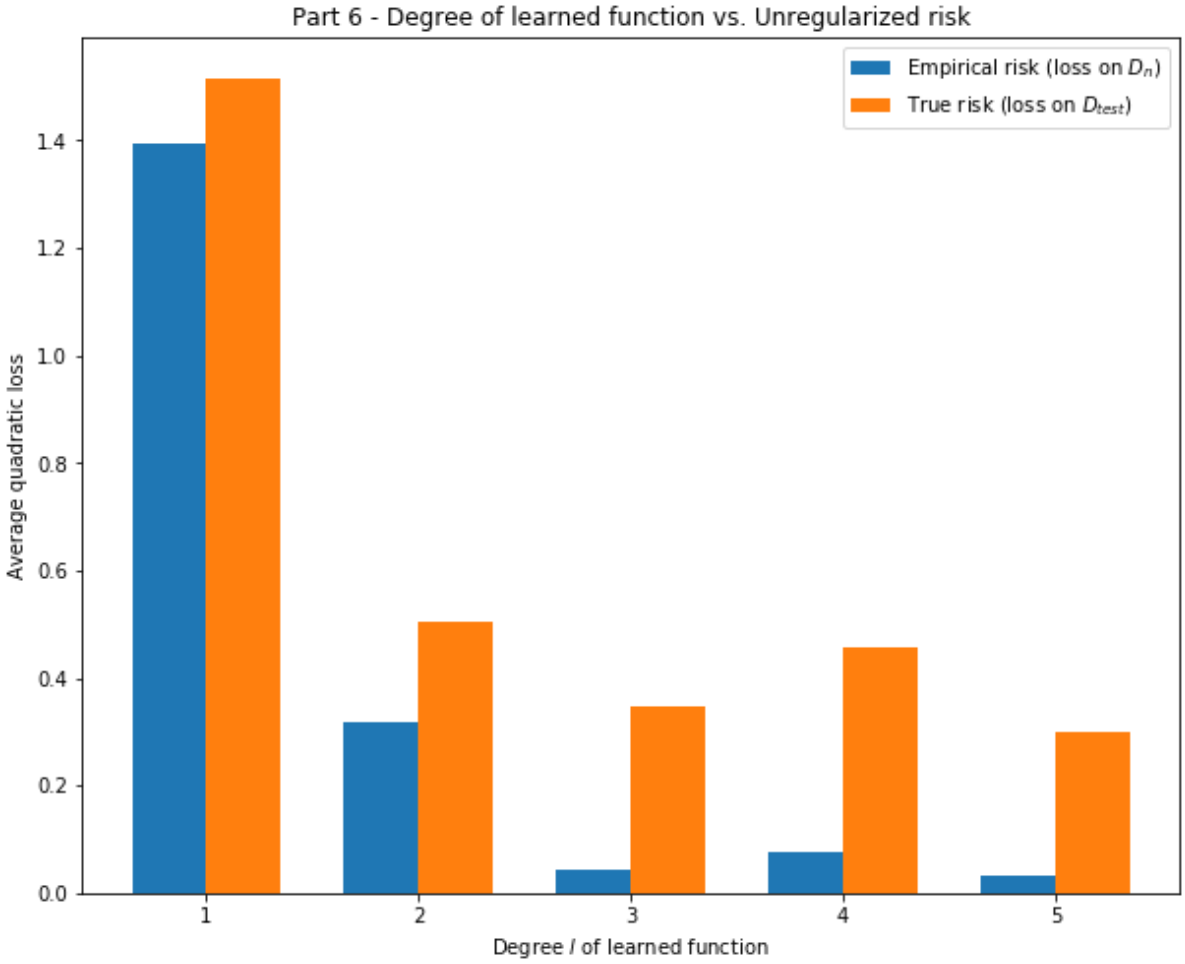
Note:

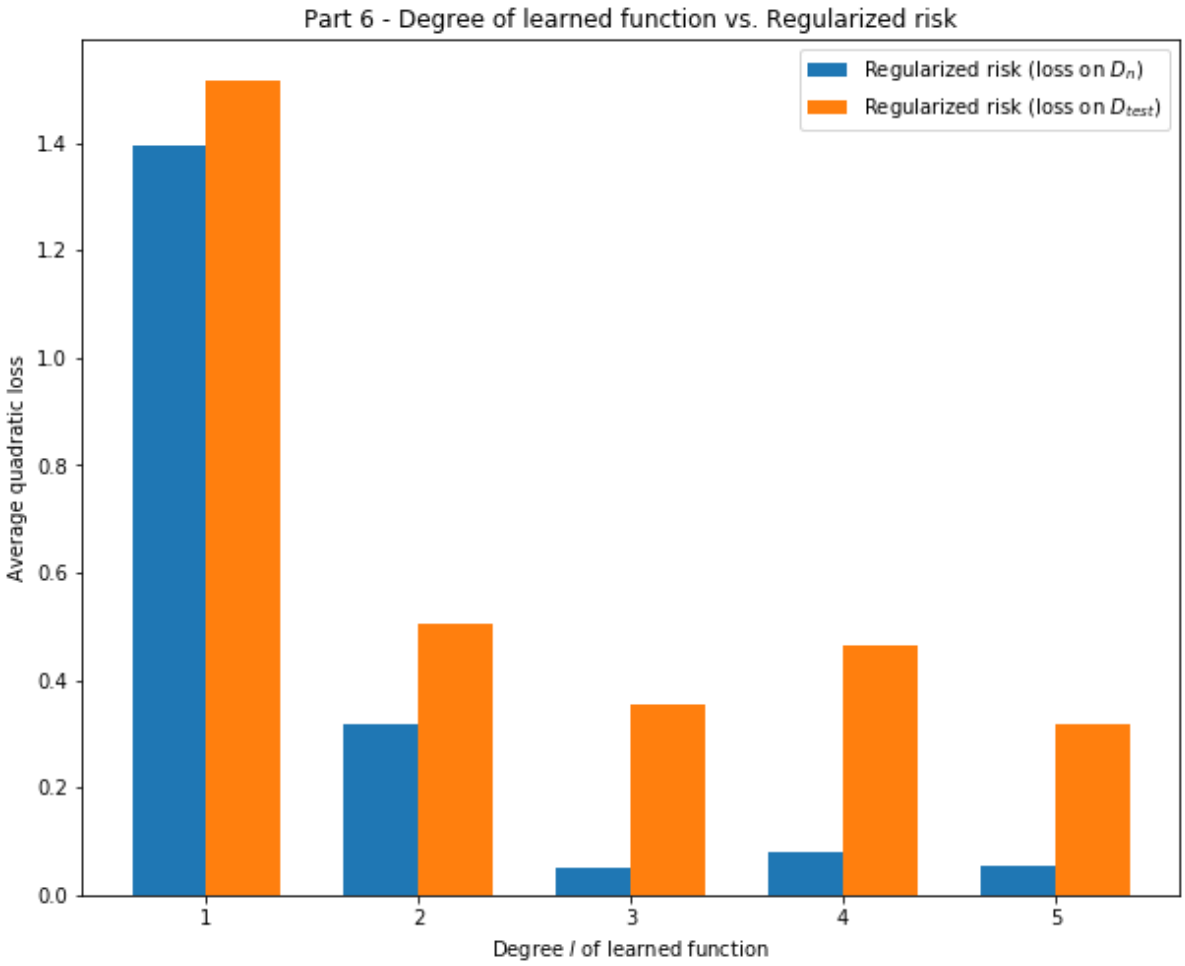
- We report loss with and without unregularization due to this post (https://studium.umontreal.ca/mod/forum/discuss.php?d=581157) on Studium by the professor.

In [10]:
```
# Already got training loss, so now get testing loss:
lossArr_test = []
regLossArr_test = []

for k in kDegreeArr:
    # Get the function
    dataDtestM = []
    for data in dataDtest[0]:
        dataDtestM.append(to_map(data, k))
    dataDtestM = np.array(dataDtestM)

    learnedFunc = []
    for point in dataDtestM:
        learnedFunc.append(np.dot(learned_param[k-1][0], point) + learne
d_param[k-1][1])
    learnedFunc = np.array(learnedFunc)

    # Calculate test error
    loss = np.mean((learnedFunc - dataDtest[1])**2)
    lossArr_test.append(loss)

    regLoss = loss + 0.01 * np.sum(np.power(learned_param[k-1][0], 2))
    regLossArr_test.append(regLoss)

print("loss on  D_{train}, " ,lossArr_train)
print("loss on test , ", lossArr_test)

print("regularized loss on  D_{train}, " ,regLossArr_train)
print("regularized loss on test , ", regLossArr_test)

# Plot training and test error
ind = np.arange(len(kDegreeArr))
width = 0.35

plt.bar(ind, np.array(lossArr_train), width, label="Empirical risk (loss
 on $D_{n}$)")
plt.bar(ind + width, np.array(lossArr_test), width, label="True risk (lo
ss on $D_{test}$)")
plt.xlabel("Degree $l$ of learned function")
plt.ylabel("Average quadratic loss")
plt.title("Part 6 - Degree of learned function vs. Unregularized risk")

plt.xticks(ind + width / 2, ('1', '2', '3', '4', '5'))
#plt.yscale('log')
plt.legend(loc='best')
plt.show()

plt.bar(ind, np.array(regLossArr_train), width, label="Regularized risk
 (loss on $D_{n}$)")
plt.bar(ind + width, np.array(regLossArr_test), width, label="Regularize
d risk (loss on $D_{test}$)")
plt.xlabel("Degree $l$ of learned function")
plt.ylabel("Average quadratic loss")
plt.title("Part 6 - Degree of learned function vs. Regularized risk")

plt.xticks(ind + width / 2, ('1', '2', '3', '4', '5'))
```

```
#plt.yscale('log')
plt.legend(loc='best')
plt.show()
```

```
loss on  D_{train},  [1.3926080150133606, 0.3165847457118256, 0.0444179
6896249361, 0.07464626566842804, 0.033555477961020035]
loss on test ,  [1.5162054646720906, 0.5044919305373453, 0.348626841170
753, 0.4569294237530785, 0.29840677524136]
regularized loss on  D_{train},  [1.393279735186666, 0.317269132052437
9, 0.05052650511625111, 0.08144113030460368, 0.05320518672911445]
regularized loss on test ,  [1.516877184845396, 0.5051763168779576, 0.3
547353773245105, 0.4637242883892541, 0.3180564840094544]
```





Generally, as the degree l of the learned function goes up, the empirical risk (loss on D_n) **should** decrease, while the true risk (loss on D_{test}) **should** increase. This is because with increasing degree, the learned function should be able to fit the training data better and better, using its higher capacity to effectively memorize the data.

We do not see this exact pattern on our set of learned functions above, **for which the test data are sampled from range** $[-5, 5]$ **as specified according to the handout**. Over this interval, the test data and label will be close to the training data, which was sampled over the same interval. Our risk plot over this interval does not reflect the fact that higher degree polynomials should overfit on the range $[-5, 5]$, and then overshoot outside this range.

We back this up by plotting true risk using the same learned weights and bias, **but on test data sampled from** $[-10, 10]$ below:

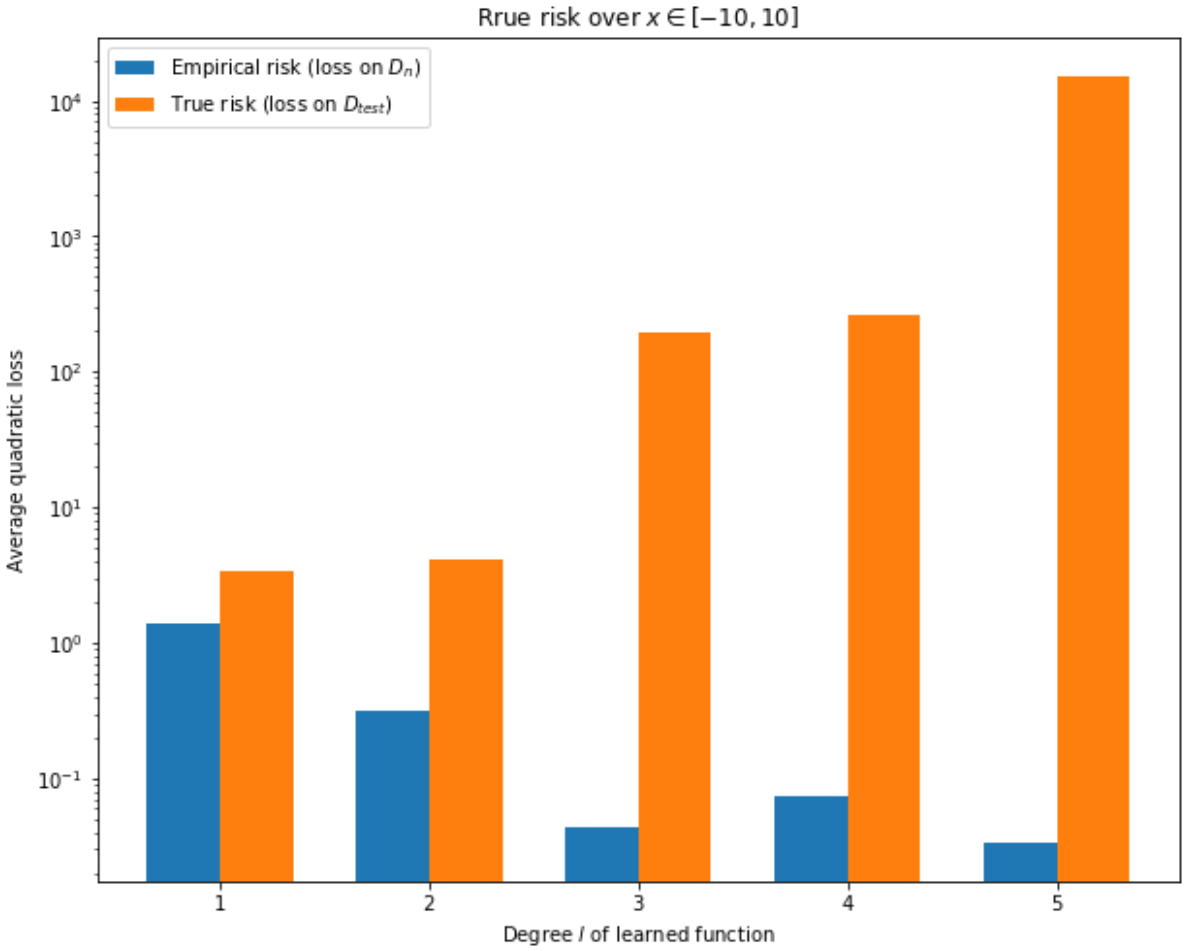
Plotting risk from test data sampled $\in [-10, 10]$

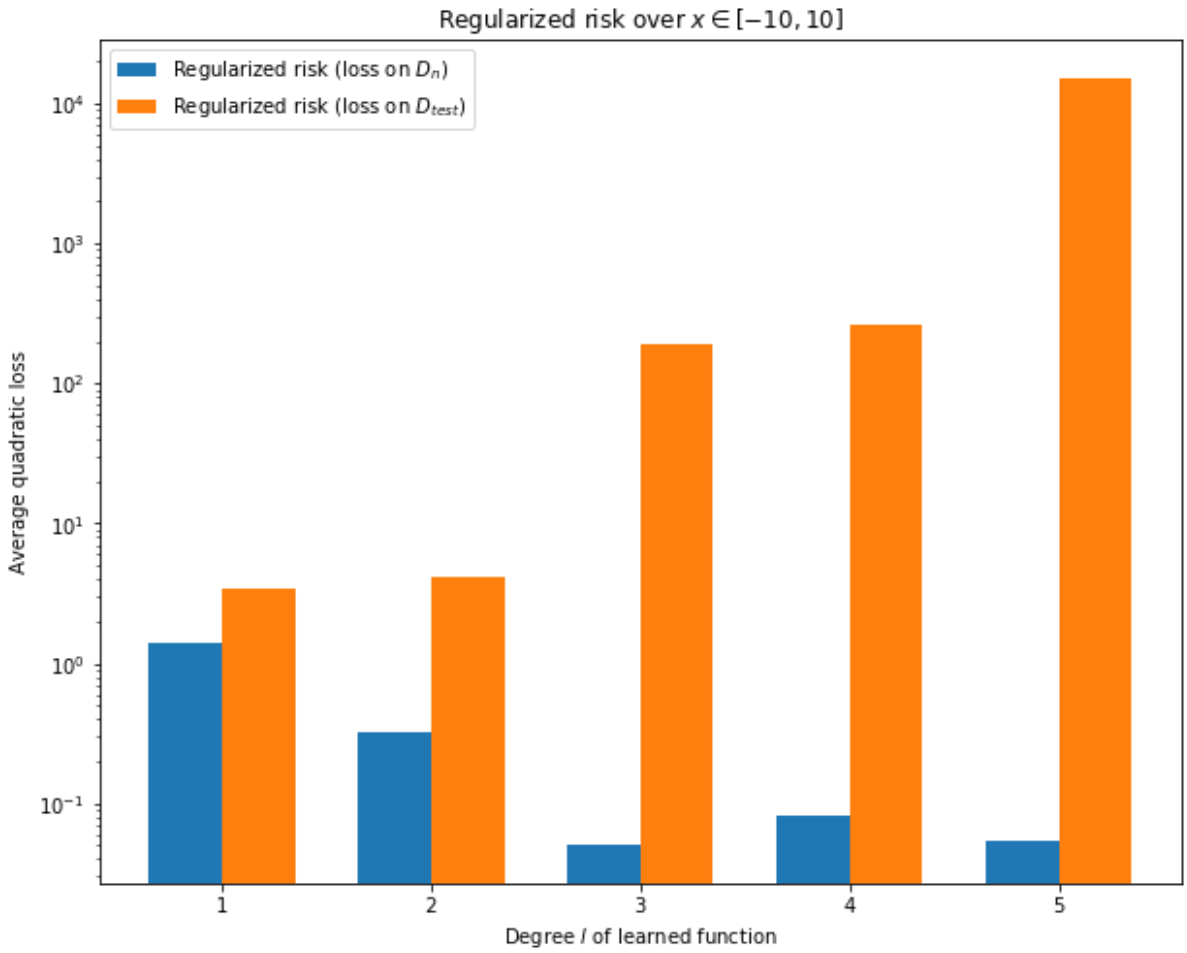
Note: these plots' y-axes are on a log scale for easier viewing.

In [11]:
```
newData = np.random.uniform(-10, 10, 100)
newTarget = hX(newData)
dataDtestNew = [newData, newTarget]


lossArr_test = []
regLossArr_test = []

for k in kDegreeArr:
    # Get the function
    dataDtestNewM = []
    for data in dataDtestNew[0]:
        dataDtestNewM.append(to_map(data, k))
    dataDtestNewM = np.array(dataDtestNewM)

    learnedFunc = []
    for point in dataDtestNewM:
        #print(learned_param[k-1])
        learnedFunc.append(np.dot(learned_param[k-1][0], point) + learne
d_param[k-1][1])
        learnedFunc
    learnedFunc = np.array(learnedFunc)

    # Calculate test error
    loss = np.mean((learnedFunc - dataDtest[1])**2)
    lossArr_test.append(loss)

    regLoss = loss + 0.01 * np.sum(np.power(learned_param[k-1][0], 2))
    regLossArr_test.append(regLoss)

print("loss train ", lossArr_train)
print("loss test , ", lossArr_test)
# Plot training and test error
ind = np.arange(len(kDegreeArr))
width = 0.35

plt.bar(ind, np.array(lossArr_train), width, label="Empirical risk (loss
 on $D_{n}$)")
plt.bar(ind + width, np.array(lossArr_test), width, label="True risk (lo
ss on $D_{test}$)")
plt.xlabel("Degree $l$ of learned function")
plt.ylabel("Average quadratic loss")
plt.title("Rrue risk over $ x \in [-10, 10]$")

plt.xticks(ind + width / 2, ('1', '2', '3', '4', '5'))
plt.yscale('log')
plt.legend(loc='best')
plt.show()



plt.bar(ind, np.array(regLossArr_train), width, label="Regularized risk
 (loss on $D_{n}$)")
plt.bar(ind + width, np.array(regLossArr_test), width, label="Regularize
d risk (loss on $D_{test}$)")
plt.xlabel("Degree $l$ of learned function")
```

```
plt.ylabel("Average quadratic loss")
plt.title("Regularized risk over $ x \in [-10, 10]$")

plt.xticks(ind + width / 2, ('1', '2', '3', '4', '5'))
plt.yscale('log')
plt.legend(loc='best')
plt.show()
```

```
loss train  [1.3926080150133606, 0.3165847457118256, 0.0444179689624936
1, 0.07464626566842804, 0.033555477961020035]
loss test ,  [3.419718153232333, 4.170288953451631, 192.14866483424876,
263.0606452335497, 15283.984700133065]
```





Having plotted over the interval $[-10, 10]$, here we remark:

- Overall, the empirical risk decreases as we increase polynomial degree. This is expected because a linear function has low capacity/expressitivity, whereas degree 2 to 5 can fit the curvature of $h(x)$.
- Overall, the true risk increases as we increase polynomial degree. This is expected because high degree polynomials overfit to the points sampled on $[-5, 5]$. The curvatures follow the curvature of $h(x)$ in $x \in [-5, 5]$ but overshoot/undershoot the target values for $x \notin [-5, 5]$ because of the nature of higher degree polynominal functions. Their y values do not oscillate like a sinusoid.
- The test of data generated from the true distribution matches our theory regarding true risk vs. empirical risk, outlined above.
- We can observe the same pattern in both regularized and unregularized cases, due to the regularized risk being $\text{empirical risk} + \lambda w^2$. The question asks us to fix $\lambda = 0.01$, which is relatively small and λw^2 does not dominate the empirical risk term in training or testing.