Learning Robots Exercise 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Dominik Marino - 2468378, Victor Jimenez - 2491031, Daniel Piendl - 2586991
January 4, 2021

Contents

1 Machine Learning in a Nutshell

1.1 Supervised vs Unsupervised Learning

- Supervised Learning
  - A supervised learning problem have besides of the observation $\vec{x}$ and a given label/class **y**. This means the observation is $(\vec{x}, y) \in (\mathrm{X} \times \mathrm{Y})$
- Unsupervised Learning
  - A unsupervised learning problem has only the observation $\vec{x}$
- The given problem is a supervised learning problem, because we have the input data **x** and it is also given the output value **y**. With the function $\mathbf{y} = \theta^T \phi(\mathbf{x})$ we also see that we have a regression problem with ϕ is vector of nonlinear functions, also called features or basis-functions and θ is the parameter that needs to be learned and called weight vector. ϕ projects x into some high-dimensional space, which the learning problem assume to become linear in the parameters.

1.2 Regression vs Classification

- **y** can be described as a qualitative or as a quantitative variable of the observation $\vec{x}$. If $\mathbf{y} = \mathbb{R}$ is a quantity variable, then its a regression problem and if **y** a quality variable then its a classification problem.
- Classification tasks
  - in a classification problem we try to separate our data as best we can and find the best decision boundary's to separate the classes.
  - Typical classification task are i.e. k-nearest-neighbour, support vector machines (SVM's), Neural Networks, etc.

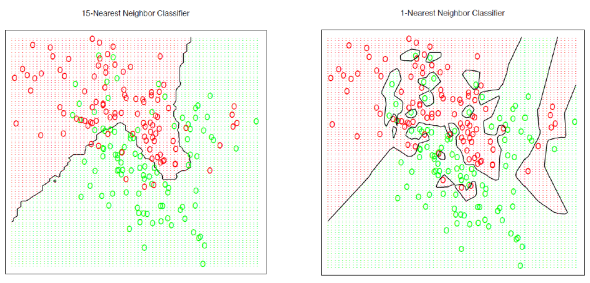

Figure 1: k-Nearest-Neighbor[1]

- Regression tasks
  - In a regression task we have a continuous output value $\mathbf{y} \in \mathbb{R}$.
  - Typical regression task is linear regression, polynomial regression, maximum likelihood regression, etc.

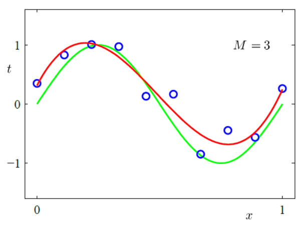

Figure 2: Polynomial regression with M = 3 [2]

[1]Source: Data Mining and Machine Learning Lecture
[2]Source: Statistical Machine Learning Lecture

1.3 Linear Least Squares

- Model:

$$\begin{aligned} y_i &= x^T w + b \\ 0 &= x^T w + b - y_i \\ &= \tilde{x}^T w - y_i \end{aligned}$$

- Lossfunction with our model:

$$\begin{aligned} \hat{w} = argmin_\theta ||\tilde{x}^T w - y_i||^2 \\ \nabla_{\hat{w}} ||\tilde{x}^T w - y_i||^2 \end{aligned}$$

- Rewriting:

$$\begin{aligned} L(w) &= (X\hat{w} - y)^T (X\hat{w} - y) \\ &= (\hat{w}^T X^T - y^T)(X\hat{w} - y) \\ &= \hat{w}^T X^T X\hat{w} - 2\hat{w}^T X^T y + y^T y \end{aligned}$$

- Derive over $\hat{w}$:

$$\begin{aligned} \frac{\partial L}{\partial \hat{w}} &= 2X^T X\hat{w} - 2X^T y \\ &= 2(X^T X)\hat{w} - 2X^T y \\ 2(X^T X)\hat{w} &= 2X^T y \\ \hat{w} &= (X^T X)^{-1} X^T y \end{aligned}$$

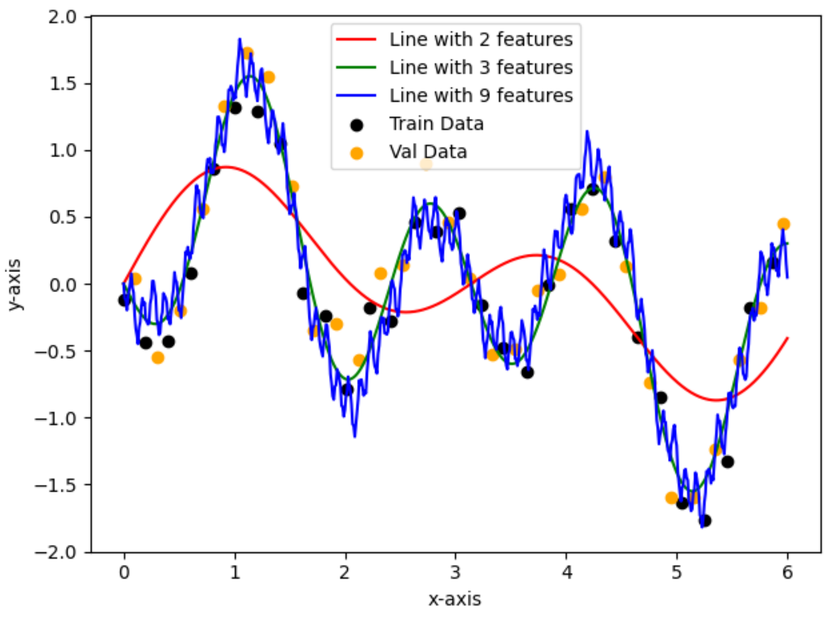

Figure 3: Linear Lest Squares with $\phi(x) = [sin(2^i * x)]_{i=0,...,n-1}$

```python
def task1c():
    x = np.arange(0, 6.01, 0.01)

    degree = [2, 3, 9]
    colors = ['r', 'g', 'b']
    plt.figure()
    plot_points(X_train, y_train, "Train Data", 'black')
    plot_points(X_val, y_val, "Val Data", "orange")
    for d, c in zip(degree, colors):
        w_hat = linear_regression(X_train, y_train, d)
        y_pred = predict_y(x, w_hat, d)
        plot_line(y_pred, d, c)

    plt.legend()
    plt.show()
```

Listing 1: Linear Least Squares

```
def linear_regression(x, y, d):
    X = phi_function(x, d)
    return np.linalg.inv(X.T @ X) @ X.T @ y
```

Listing 2: linear regression

```
def predict_y(X, y, d=0):
    return phi_function(X, d) @ y
```

Listing 3: predict y

```
def phi_function(x, degree):
    x_poly = []
    for i in range(degree):
        x_poly.append(np.sin(2**i * x))
    return np.asarray(x_poly).T
```

Listing 4: phi-function

1.4 Training a Model

- is in section "Model selection"

1.5 Model Selection

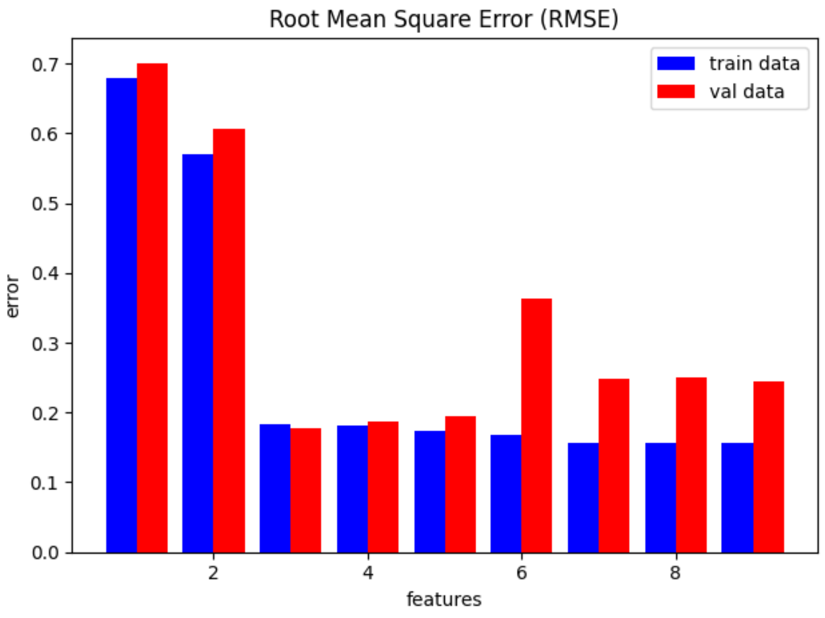

Figure 4: Root Mean Squared Error of different features

- What is the number of features that you should use to achieve a proper modeling?
  - For this dataset the number of features should be choosen of 3. The mean squared error of the validation set is the smallest, for n=3, but for n greater then 3 the dataset starts to overfitting. The plot shows also that the error of the train dataset gets smaller with every feature. Meanwhile the error of the validation set rises. This is also an indicatior for overfitting.
- How do they differ?
  - The difference comes because of we only concentrating on the training set. In general we dont care about the error of the training set, because we already discovered the data. We only want to prove our learning algorithm on unkown data and want to see how good our prediction is on unkown data. We can see this effect in this example. The trainingsdata error gets in every step smaller but at the same time the error for the validation set raises again.
- Can you explain what is the reason for these differences?
  - A validation only based on the trainingset is not a good idea. This effect is also called overfitting. After a the "learning process" we have to validate our machine learing algorithm on unkown data. The image 3 from section 1.3 shows this effect how i.e the plot look like with 9 features. The line oszilate very strong around the trainingsdata.

```python
def task1d_e():
    degree = np.arange(1, 9 + 1)
    errors = np.zeros((len(degree), 2))

    for i, d in enumerate(degree):
        """ <<<----- Learning process ---->>> """
        w_hat = linear_regression(X_train, y_train, d)
        y_train_pred = predict_y(X_train, w_hat, d)
        errors[i, 0] = mean_square_error(y_train, y_train_pred)

        """ <<<----- validation process ---->>> """
        y_val_pred = predict_y(X_val, w_hat, d)
        errors[i, 1] = mean_square_error(y_val, y_val_pred)

    plt.figure()
    plt.title("Root Mean Square Error (RMSE)")
    plt.bar(degree - 0.2, errors[:, 0], color='b', width=0.4, label='train data')
    plt.bar(degree + 0.2, errors[:, 1], color='r', width=0.4, label='val data')
    plt.xlabel("degrees")
    plt.ylabel("error")
    plt.legend()
    plt.show()
```

Listing 5: RMSE of Train- and Val-Set

1.6 Cross Validation

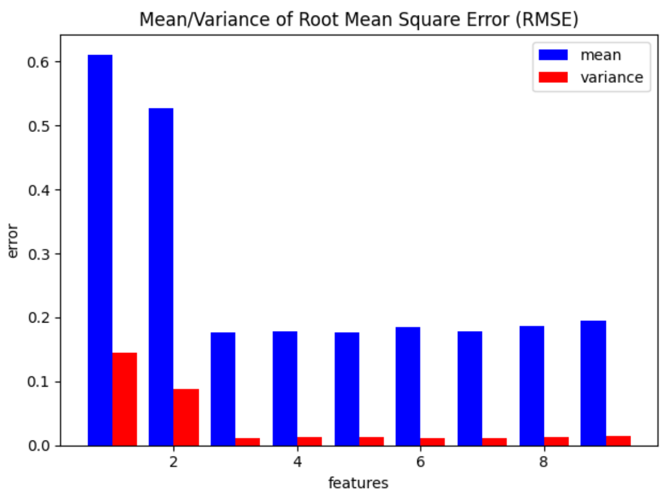

Figure 5: Mean and Variance bar plot

features	Mean	variance
1	0.6112497	0.14528838
2	0.52705745	0.08822847
3	0.17626565	0.0114376
4	0.17772128	0.0129961
5	0.17694891	0.0122322
6	0.18485587	0.01152085
7	0.17806914	0.01132834
8	0.18704623	0.01340935
9	0.19411939	0.01509499

	minimum mean	minimum variance
value	0.17626565	0.01132834
feature	3	7

- Model selection via bias-variance tradeoff:
  - In general, if the complexity of the model increases then the variance also tends to increase and the squared bias should decrease. If the model complexity decrease then the bias tends to increase and the variance decreases. We always want to minimize the test error for our chosen model complexity[3]. The figure 6 show that.

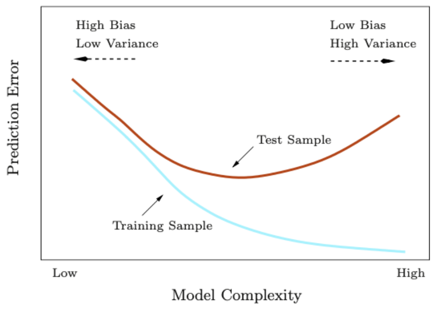

Figure 6: Test and training error as a function of model complexity

[3]The Elements of Statistical Learning - Data Mining, Inference, and Prediction Page 37

- Which is the optimal number of features now?
  - The best model is still the one with 3 features. It achieves the lowest RMSE-mean and even though the model with 7 features achieves a lower variance, the difference is negligible. Because of the principle of Occam's Razor, the simplest model with the best results should be chosen, which is the one with 3 features in this case.
- Discuss the results obtained and compare against model selection using train/validation set.
  - The difference in the trends of the RMSE in 1.5 and of the RMSE-mean in 1.6 is only very small. Both decrease quickly when increasing the number of features from 1 to 3 and then vary only slightly. When utilizing the cross validation and using the model selection via bias-variance trade-off it is not obvious which number of features between 3 and 9 to choose by only looking at the graph, since the variance and bias stay approximately the same from three features onward. However, when utilizing different training and validation data sets as in 1.5, the choice is narrowed down to 3, 4 or 5 features. However in the previous task we saw that our model starts to overfit when utilizing more than three features. With this knowledge, 3 features would be the best choice. Cross validation (in this case Leaf-One-Out) has the advantage of being able to properly train a model when there is only little available data. In this case however, the usual train-test approach yields better results.

```python
def task1f():
    degree = np.arange(1, 9 + 1)
    errors = np.zeros((len(degree), len(y_train) - 1))
    for i, d in enumerate(degree):
        for j in range(len(y_train) - 1):
            xtrain, ytrain, xtest, ytest = leave_one_out(X_train, y_train, j)

            """ <<<----- Learning process ---->>> """
            w_hat = linear_regression(xtrain, ytrain, d)
            y_train_pred = predict_y(xtest, w_hat, d)
            errors[i, j] = mean_square_error(ytest, y_train_pred)

    mean = compute_mean(errors)
    variance = compute_variance(errors)

    print("minimum mean-error: {}, idx: {}; mimumum variance-variance: {}, idx: {};".format(min(mean), np.argmin(mean),
                                                                                            min(variance),
                                                                                            np.argmin(variance)))

    plt.figure()
    plt.title("Mean/Variance of Root Mean Square Error (RMSE)")
    plt.bar(degree - 0.2, mean, color='b', width=0.4, label='mean')
    plt.bar(degree + 0.2, variance, color='r', width=0.4, label='variance')
    plt.xlabel("mean/variance")
    plt.ylabel("error")
    plt.legend()
    plt.show()
```

Listing 6: Validation with LOO

```python
def leave_one_out(X, y, index):
    return np.delete(X, index), np.delete(y, index), X[index], y[index]
```

Listing 7: Leave-One-Out Validation

```python
def compute_mean(errors):
    return errors.mean(axis=1)

def compute_variance(errors):
    return errors.std(axis=1) ** 2
```

Listing 8: Calculate mean and variance

1.7 Kernel Functions

For $n = 3$:

$$\begin{aligned}
\mathbf{\Phi}(\mathbf{x}) &= [\sin(\mathbf{x}) \quad \sin(2\mathbf{x}) \quad \sin(4\mathbf{x})] \\
k(\mathbf{x_i}, \mathbf{x_j}) &= \mathbf{\Phi}(\mathbf{x_i})^{\mathrm{T}} \cdot \mathbf{\Phi}(\mathbf{x_j}) \\
&= \sin(\mathbf{x_i}) \cdot \sin(\mathbf{x_j}) + \sin(2\mathbf{x_i}) \cdot \sin(2\mathbf{x_j}) + \sin(4\mathbf{x_i}) \cdot \sin(4\mathbf{x_j})
\end{aligned}$$

1.8 Kernel Regression

RMSE Train	RMSE Test
1.56807552e-16	2.42247105e-01

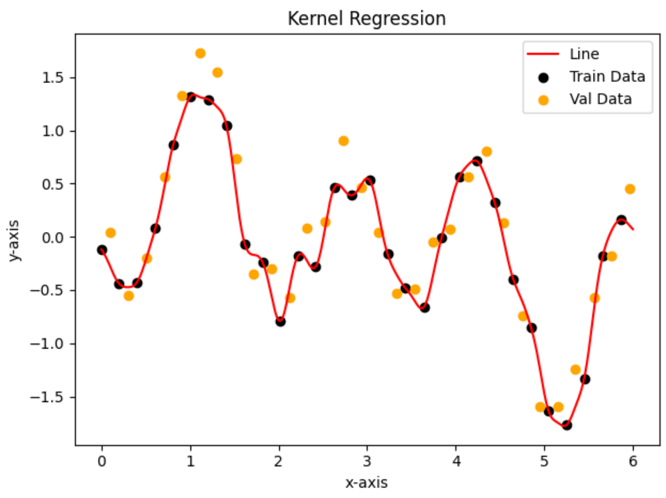

Figure 7: Kernel Regression

- We know from the lecture that "kernels can measure the similarity between data points in feature space without evaluating or explicitly knowing all features and the feature space can be possibly infinite dimensional and kernels are easier to design than features". Because of the relatively low kernel variance (0.15), the regression function approximates the training data very well. Since the approximation is only very good close to the training data it doesn't achieve a better RMSE in regard to the validation data in comparison with the models trained in 1.5 and 1.6. A better approach would be to train different kernel models with the same kernel function but utilizing different variances. Then the variances which performs best can be chosen.

```python
def task1h():
    sigma = 0.15
    n = 3
    K = exponential_squared_kernel(X_train, sigma)
    errors = np.zeros((1, 2))

    f_x_train = kernel_regression(X_train, X_train, y_train, K, sigma)
    errors[0, 0] = mean_square_error(y_train, f_x_train)

    """ <<<----- validation process ---->>>> """
    f_x_val = kernel_regression(X_val, X_train, y_train, K, sigma)
    errors[0, 1] = mean_square_error(y_val, f_x_val)

    print(errors)

    x = np.arange(0, 6.01, 0.01)
    f_x = kernel_regression(x, X_train, y_train, K, sigma)
    plt.figure()
    plt.title("Kernel Regression")
    plot_points(X_train, y_train, "Train Data", 'black')
    plot_points(X_val, y_val, "Val Data", "orange")
    plt.plot(x, f_x, c='r', label="Line")
    plt.xlabel("x-axis")
    plt.ylabel("y-axis")
    plt.legend()
    plt.show()
```

Listing 9: Plot Kernel Regression

```python
def kernel(xi, xj, sigma=0.15):
    return np.exp(- np.linalg.norm(xi - xj) ** 2 / sigma ** 2)

def exponential_squared_kernel(X, sigma):
    K = np.zeros((len(X), len(X)))
    for i in range(len(X)):
        for j in range(len(X)):
            K[i, j] = kernel(X[i], X[j], sigma)
    return K
```

Listing 10: calculation for k and K

```python
def kernel_regression(x, X, y, K, sigma):
    f_x = np.zeros((len(x)))
    for i in range(len(x)):
        k = kernel(x[i], X, sigma)
        f_x[i] = k.T @ np.linalg.inv(K) @ y
    return f_x
```

Listing 11: Kernel Regression

1.9 Derivation

- Explain the concept of ridge regression and why/when it is used
  - Ridge regression is a regularization method used to control the over-fitting phenomenon. In this type of regression an additional penalization term on the size of the weights is added in order to discourage the coefficients from reaching large values. The penalization term is the sum of squares of all of the coefficients. The hyperparameter λ controls the amount of shrinkage. The larger the value of λ, the greater the amount of shrinkage. The ridge regression allows the user to use complex and flexible models while also reducing the amount of over-fitting. The idea of penalizing by the sum of squares of the parameters is also used in neural networks, where it is known as weight decay. [4] [5]

Sum of squared Errors:

$$L(y, \hat{y}) = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Model:

$$\hat{y}_i = x^T w$$

Loss function for the Ridge Regression:

$$\begin{aligned}\hat{w} = \arg\min{}_w \frac{1}{2}||x^T w - y_i||^2 + \frac{\lambda}{2}||w||^2 \\ \nabla_w \frac{1}{2}||x^T w - y_i||^2 + \frac{\lambda}{2}||w||^2\end{aligned}$$

$$\begin{aligned}L(w, \lambda) &= \frac{1}{2}(Xw - y)^T(Xw - y) + \frac{\lambda}{2}w^T w \\ &= \frac{1}{2}(w^T X^T - y^T)(Xw - y)) + \frac{1}{2}w^T w \\ &= \frac{1}{2}(w^T X^T X w - 2w^T X^T y + y^T y) + \frac{1}{2}w^T w\end{aligned}$$

Now derive:

$$\begin{aligned}\frac{\partial L}{\partial w} &= X^T X w - X^T y + \lambda w \\ &= (X^T X + \lambda I)w - X^T y \\ 0 &= (X^T X + \lambda I)w - X^T y \\ X^T y &= (X^T X + \lambda I)w \\ w &= (X^T X + \lambda I)^{-1} X^T y\end{aligned}$$

[4]The Elements of Statistical Learning - Data Mining, Inference, and Prediction Page 63
[5]Bishop - Pattern Recognition And Machine Learning page 10