3.6 Lab: Linear Regression

3.6.1 Importing packages

In this lab we will be utilizing the following libraries or packages, their corresponding versions are:

CSV → v0.10.13
DataFrames → v.0.22.7
DelimitedFiles → v.1.9.1
GLM → v1.9.0
Plots → v1.40.3

3.6.2 Simple Linear Regression

In order to understand how Linear Regression works, we will focus on the Boston housing dataset. This is a very popular dataset, which contains data from 506 neighborhoods around Boston. Our goal would be to predict median house value with 13 features or independent variables such as average number of rooms per dwelling (RM), crime rate (CRIM), distances to employment centres, among others. Loadint the dataset:

```
julia> Boston = DataFrame(readdlm("C:\\ISL 2024\\Datasets\\Boston Housing.csv"), :auto)
       first(Boston, 3)
```

3×14 DataFrame

Row	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	0.01	18	2.31	0	0.54	6.58	65.2	4.09	1	296	15.3	396.9	4.98	24
2	0.03	0	7.07	0	0.47	6.42	78.9	4.97	2	242	17.8	396.9	9.14	21.6
3	0.03	0	7.07	0	0.47	7.19	61.1	4.97	2	242	17.8	392.83	4.03	34.7

But the column names appeared as x-features, we can modify them by passing an array of names and using the function rename!. With the last line we display the first 5 rows and all the columns.

```
julia> col_names = ["crim", "zn", "indus", "chas", "nox", "rm", "age", "dis", "rad",
      "tax", "ptratio", "black", "lstat", "medv"]

       rename!(Boston, col_names)
       show(first(Boston, 5), allcols=true)
```

The first model that we'll be fitting it is Ordinary Least Squares – OLS. To fit the model, the input must be a DataFrame containing the feature(s) and the target variable.

```
julia> y = Boston[!, :medv]
       data = DataFrame()
       data.X = X[!, :lstat]
       data.y = y
       show(first(data, 3), allcols=true)
```

```
3×2 DataFrame Row
                  │ X            y
------------------Float64------Float64
      1           │ 4.98         24.0
      2           │ 9.14         21.6
      3           │ 4.03         34.7
```

The lm() function returns a summary table containing the coefficients, standard errors, t-statistic, p-values and confidence intervals.

```
julia> model = lm(@formula(y ~ X), data)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64, Matrix{Float64},
Vector{Int64}}}}, Matrix{Float64}}
y ~ 1 + X
Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	34.5538	0.562627	61.42	<1e-99	33.4485	35.6592
X	-0.950049	0.0387334	-24.53	<1e-87	-1.02615	-0.873951

A second alternatiive to obtain the coefficients, it is solving the equation system, X\y. We first convert the X DataFrame to a matrix and then solve the system.

```
julia> Array(X)\y
```

```
2-element Vector{Float64}:
34.55384087938309
-0.9500493537579905
```

Once the model is fitted, we can make some predictions. For example, let's define a DataFrame with 5, 10 and 15 as values.

```
julia> X_val = DataFrame(X = [5, 10, 15])
```

3×1 DataFrame

Row	X
	Int64
1	5
2	10
3	15

predict() function help us to compute the output values with X_val as input variable.

```
julia> pred = predict(model, X_val)
```

```
3-element Vector{Union{Missing, Float64}}:
29.803594110593103
25.053347341803175
20.303100573013246
```

Also to include the confidence intervals in our predictions.

```
julia> pred = predict(model, X_val, interval=:confidence, level=0.95)
```

3×3 DataFrame

Row	prediction	lower	upper
	Float64	Float64	Float64
1	29.8	29.01	30.6
2	25.05	24.47	25.63
3	20.3	19.73	20.87

Or to include the prediction intervals we type.

```
julia> pred = predict(model, X_val, interval=:prediction, level=0.95)
```

3×3 DataFrame

Row	prediction	lower	upper
	Float64	Float64	Float64
1	29.8	17.57	42.04
2	25.05	12.83	37.28
3	20.3	8.08	32.53

Defining functions

Julia recipes can simplify the way we plot the data. Within a recipe, it is possible to automate the definition of plot properties. To define a recipe we put the @ macro syntax.

```
julia> @userplot Myplot
       @recipe function h(object::Myplot)
          seriestype := :scatter
          seriescolor := :white
          markersize := 5
          markerstrokecolor --> :blue
          markerstrokewidth --> :2
          alpha --> 0.7
          xtickfont --> font(10)
```

```
        ytickfont --> font(10)
        size --> (800, 600)
        dpi --> 200
        object.args
    end
```

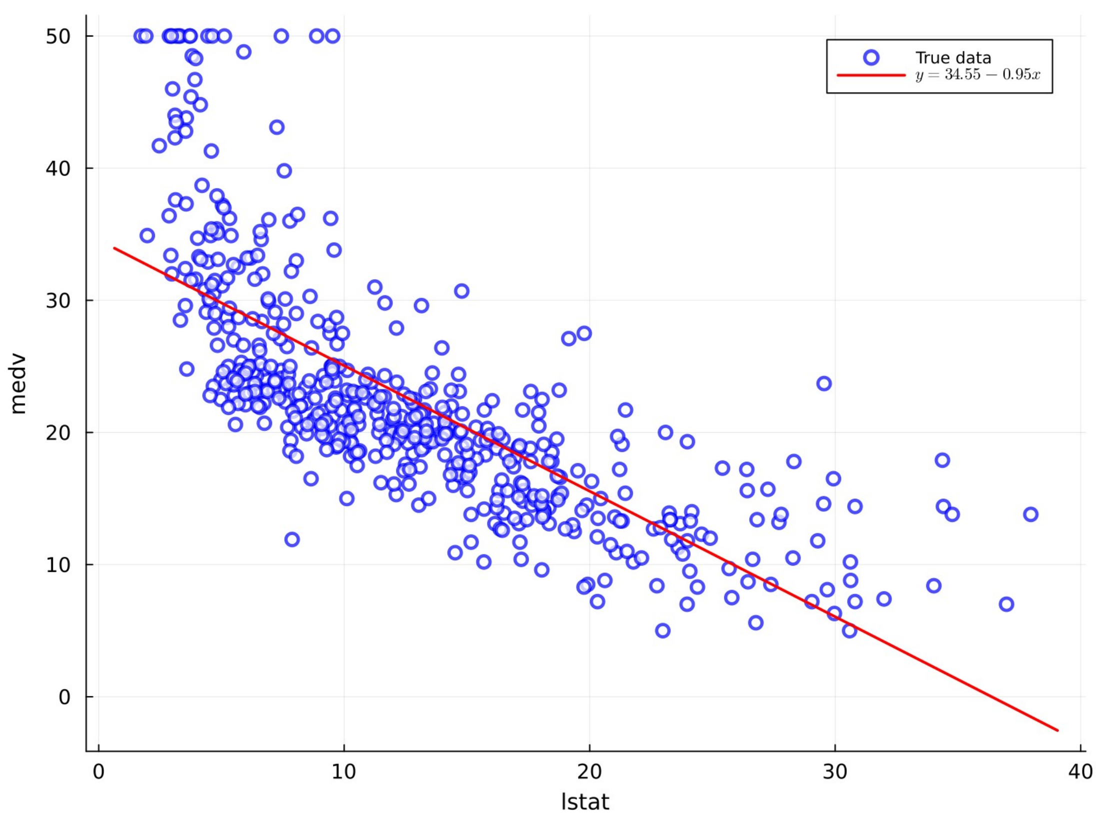
Then, we build a function to plot an abline between 2 points and a slope.

```
julia> using LateXStrings

        function abline(p, b, m)
            “Add a line with slope m and intercept b to p”
            xlim = collect(xlims(p))
            ylim = collect((m * xlim[1] + b, m * xlim[2] + b))
            b = round(b, digits=2)
            m = round(m, digits=2)
            plot!(xlim, ylim, c=:red, linewidth=2, label=L”y = %$b %$m x”)
        end
```

Now we are able to apply our recipe and abline function. Let’s define some additional parameters.

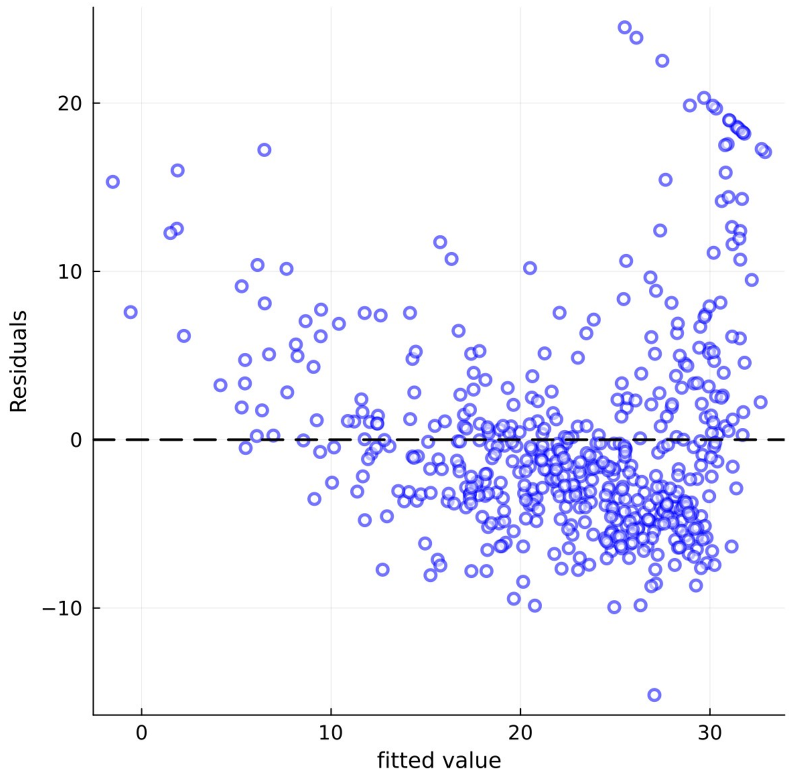
```
julia> p = myplot(Boston.:lstat,Boston.:medv,xlabel=“lstat”,ylabel=“medv”,label=“True
       data”)
       abline(p, coef(model)[1], coef(model)[2])
```



To retrieve the fitted and residual values of a model, fitted() and residuals() functions can help us.

```
julia> scatter(fitted(model), residuals(model), legend=false, c=:white,
        markersize=4.2, markerstrokecolor=:blue, markerstrokewidth=2, alpha=0.55,
        size=(600, 600), xlabel="fitted value", ylabel="Residuals")

        hline!([0], c=:black, linestyle=:dash, linewidth=2)
```

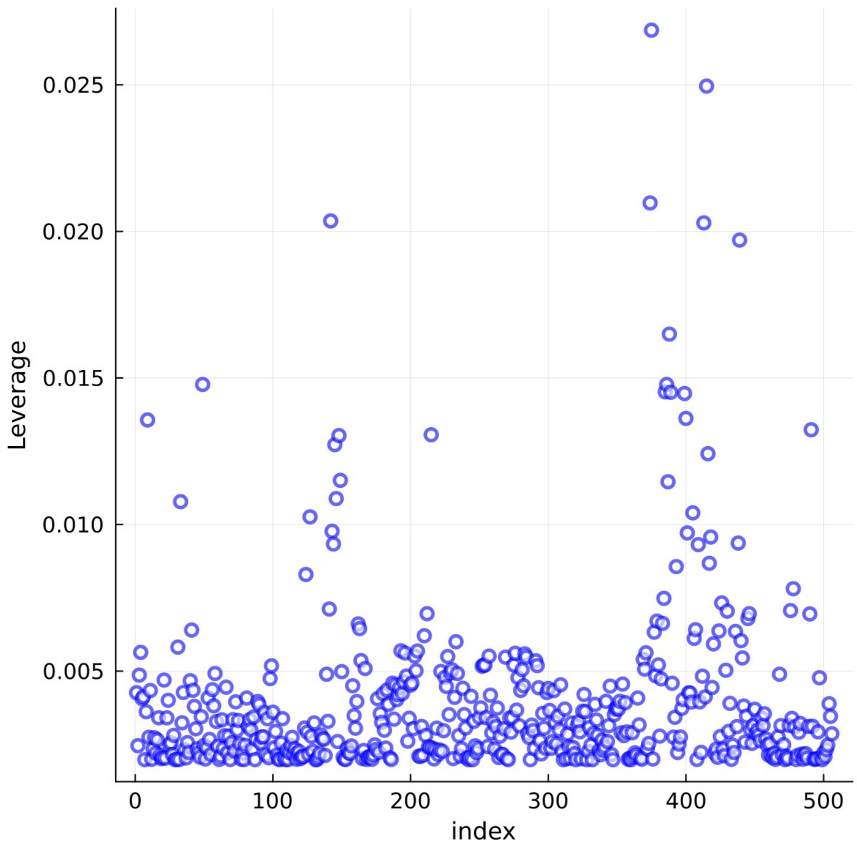


Leverage statistics is computed with the LinearAlgebra package and then, we can plot their behavior.

```
julia> using LinearAlgebra

       E = hcat(ones(506), Boston.:lstat)
       Lev = diag(E*inv((E'*E))*E')
```

```
julia> i = collect(1:506)
        scatter(i, Lev, legend=false, c=:white, markersize=4.5, markerstrokecolor=:blue,
       markerstrokewidth=2, alpha=0.6, size=(600, 600),xlabel="index",ylabel= "Leverage",
       dpi=200, xtickfont=10, ytickfont=10)
```



Findmax() function returns a tuple with the maximum value and it's index of the vector or array.

```
julia> findmax(Lev)
```

```
(0.026865166510283436, 375)
```

3.6.3 Multiple Linear Regression

If our interest is to include several independent variables in the model, the lm() function can receive some more terms. For example, adding the age as a regressor.

```
julia> X = Boston[:, [13, 7, 14]]
       model1 = lm(@formula(medv ~ lstat + age), X)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64, Matrix{Float64},
Vector{Int64}}}}, Matrix{Float64}}
medv ~ 1 + lstat + age
```

```
Coefficients:
────────────────────────────────────────────────────────────────────────
                 Coef.   Std. Error       t    Pr(>|t|)  Lower 95%   Upper 95%
────────────────────────────────────────────────────────────────────────
(Intercept) 33.2228     0.730847     45.46    <1e-99    31.7869     34.6586
lstat       -1.03207    0.0481907   -21.42    <1e-72    -1.12675    -0.937389
age          0.0345443  0.0122255     2.83    0.0049     0.0105251   0.0585636
────────────────────────────────────────────────────────────────────────
```

Also, we could have computed the coefficients solving the equation system.

```
julia> target = Boston.:medv
       feat = hcat(ones(size(X)[1]), X[:, [1, 2]])
       Array(feat)\target
```

```
3-element Vector{Float64}:
33.22276053179289
-1.0320685641826006
0.034544338571646l1
```

If we would like to include the 12 regressors of the Boston Dataset to predict the median value – medv, lm() function can receive all the features.

```
julia> model_all = lm(@formula(medv ~ crim + zn + indus + chas + nox + rm + age + dis
+ rad + tax + ptratio + lstat), Boston)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, LinearAlgebra.CholeskyPivoted{Float64, Matrix{Float64},
Vector{Int64}}}}, Matrix{Float64}}

medv ~ 1 + crim + zn + indus + chas + nox + rm + age + dis + rad + tax + ptratio + lstat

Coefficients:
────────────────────────────────────────────────────────────────────────
                 Coef.   Std. Error       t    Pr(>|t|)  Lower 95%   Upper 95%
────────────────────────────────────────────────────────────────────────
(Intercept) 41.6173     4.93604      8.43     <1e-15    31.919      51.3155
crim        -0.121389   0.0330004   -3.68     0.0003    -0.186227   -0.0565498
zn           0.0469635  0.0138791    3.38     0.0008     0.0196939   0.074233
indus        0.0134677  0.0621447    0.22     0.8285    -0.108633    0.135569
chas         2.83999    0.870007     3.26     0.0012     1.13061     4.54937
nox        -18.758      3.85135     -4.87     <1e-05    -26.3251    -11.1909
rm           3.65812    0.420246     8.70     <1e-16     2.83243     4.48381
age          0.00361071 0.0133294    0.27     0.7866    -0.0225788   0.0298002
dis         -1.49075    0.201623    -7.39     <1e-12    -1.8869     -1.09461
rad          0.289405   0.0669079    4.33     <1e-04     0.157945    0.420864
tax         -0.012682   0.00380098 -3.34      0.0009    -0.0201501 -0.00521387
ptratio     -0.937533   0.132206    -7.09     <1e-11    -1.19729    -0.677776
lstat       -0.552019   0.0506588  -10.90     <1e-24    -0.651553   -0.452485
────────────────────────────────────────────────────────────────────────
```

The “age” variable has a high p-value, showing not a good fit with the medv. As you can imagine, to remove it from the model, we just do not include it in the lm() function.

```
julia> model_all = lm(@formula(medv ~ crim + zn + indus + chas + nox + rm + dis + rad
+ tax + ptratio + lstat), Boston)
```

3.6.4 Multivariate Goodness of Fit

To compute the r^2 metric and the Residual Standard Error or RSE, we apply the r2 and dispersion functions.

```
julia> r2(model_age)
```

```
0.7342674984601645
```

```
julia> GLM.dispersion(model_age.model)
```

```
4.793532256301406
```

Another important metric it is the Variance Inflation Factor or VIF, using Linear Algebra equation we have.

```
julia>  X_all = select(Boston, Not([:medv]))
       vifm = diag(inv(cor(Matrix{Float64}(X_all))))
       round.(vifm, digits=3)
```

```
12-element Vector{Float64}:
1.767
2.298
3.987
1.071
4.369
1.913
3.088
3.954
7.445
9.002
1.797
2.871
```

3.6.5 Interaction Terms

To add the interaction terms in the model we simply put the "*" symbol in the lm() function.

```
julia> X_it = Boston[:, [13, 7, 14]]
       model2 = lm(@formula(medv ~ lstat * age), X_it)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}}},
Matrix{Float64}}

medv ~ 1 + lstat + age + lstat & age

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	36.0885	1.46984	24.55	<1e-87	33.2007	38.9763
lstat	-1.39212	0.167456	-8.31	<1e-15	-1.72112	-1.06312
age	-0.00072086	0.0198792	-0.04	0.9711	-0.0397775	0.0383358
lstat & age	0.00415595	0.0018518	2.24	0.0252	0.000517728	0.00779418

3.6.6 Non-linear Transformations of the Predictors

We can increase the degree of the polynomial by creating some polynomial features. Let's suppose we want a second degree polynomial.

```
julia> using CSV

       X_poly = CSV.read("X_poly.csv", DataFrame)
       X_poly = X_poly[:, 2:end]
       insertcols!(X_poly, 5, "medv" => Boston.:medv)
       rename!(X_poly, [:intercept, :lstat, :lstat2, :age, :medv])
       model3 = lm(@formula(medv ~ lstat + lstat2 + age), X_poly)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}}},
Matrix{Float64}}

medv ~ 1 + lstat + lstat2 + age

Coefficients:
```

	Coef.	Std. Error	t	Pr(>|t|)	Lower 95%	Upper 95%
(Intercept)	17.7151	0.781055	22.68	<1e-78	16.1806	19.2497
lstat	-179.228	6.73276	-26.62	<1e-97	-192.456	-166.0
lstat2	72.9908	5.48201	13.31	<1e-34	62.2203	83.7613
age	0.0702545	0.0108567	6.47	<1e-09	0.0489242	0.0915847

The function Anova_lm() compares 2 succesive models, in our case a second degree polynomial and a single model (models 3 and 1).

```
Julia> using DataFrames

  function Anova_lm(y::AbstractVector, y_hat_red::AbstractVector,
       yhat_full::AbstractVector, p_red::Union{Int64, Float64}, p_full::Union{Int64,
       Float64}, model, exp ; o=4)

       n = length(y)
       pval = coeftable(model).cols[o][exp+1]

       # Error Sum of Squares for the Reduced Model - SSER - (yhat_red - ymean)
       SSER = sum([y[i] - yhat_red[i])^2 for i in 1:length(y)])

       # Error Sum of Squares for the Full Model - SSEF - (yhat_red_i - ymean)
       SSEF = sum([y[i] - yhat_full[i])^2 for i in 1:length(y)])

       # Degrees of freedom for the Reduced Model - dfr
       dfr = n - p_red

       # Degrees of freedom for the Full Model - dff
       dff = n - p_full

       # Mean of Squares for Model - MSM
       MSR = SSER/dfr

       # Mean of Squares for the Error - MSE
       MSF = SSEF/dff

       # F-Statistics
       F = ((SSER - SSEF)/(dfr - dff))/(SSEF/dff)

       df_r = DataFrame(df_resid = dfr, ssr = SSER, df_diff = 0, ss_diff = NaN, F = Nan,
       P_val = NaN)

       df_f = DataFrame(df_resid = dff, ssr = SSEF, df_diff = dfr - dff, ss_diff = SSER -
       SSEF, F = F, P_val = pval)

       df = vcat(df_r, df_f)
       return df
  end
```

We proceed to create the vectors and the x_values and apply our created Anova_lm function.

```
julia> y_real = Boston.:medv
       y_hat1 = predict(model1, X)
       y_hat3 = predict(model3, X_poly)
       x_values = collect(1:length(y_real))
```

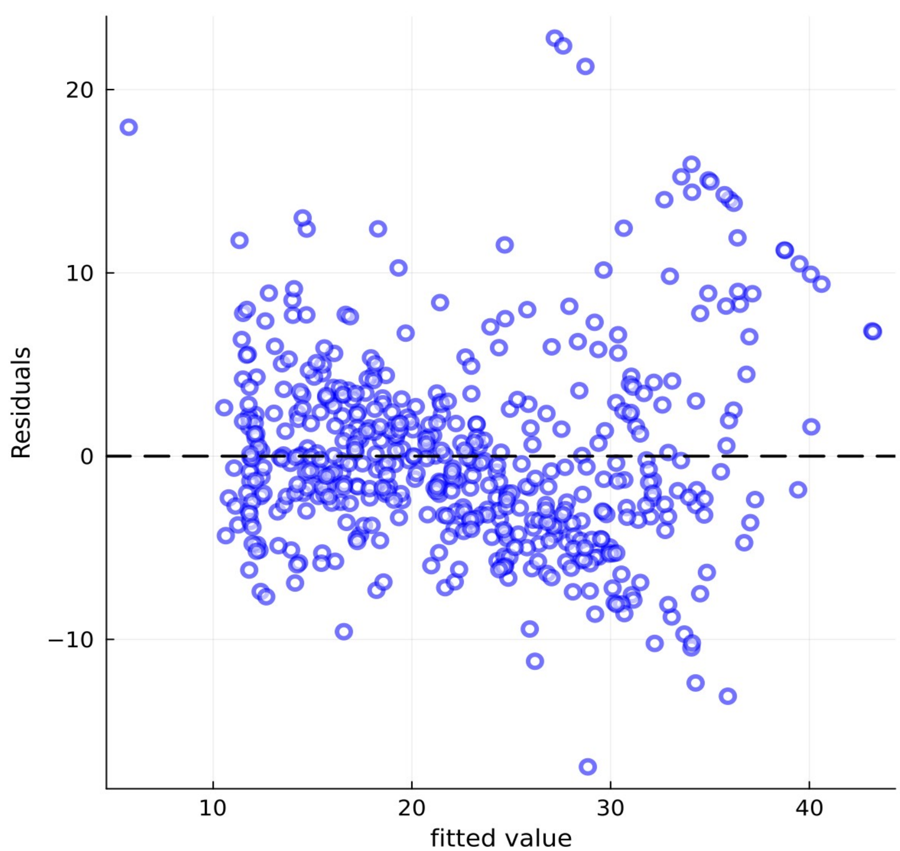
```
julia> Anova_lm(y_real, y_hat1, y_hat3, 3, 4, model3, 2)
```

2×6 DataFrame

Row	df_resid	ssr	df_diff	ss_diff	F	P_val
	Int64	Float64	Int64	Float64	Float64	Float64
1	503	19168.1	0	NaN	NaN	NaN
2	502	14165.6	1	5002.52	177.28	0

Finally, looking at the residuals plot for the model3 (second degree) we see some better results.

```
julia> scatter(fitted(model3), residuals(model3), legend=false, c=:white,
       markersize=5, markerstrokecolor=:blue, markerstrokewidth=2.5, alpha=0.55,
       size=(600, 600), xlabel="fitted value", ylabel="Residuals", dpi=200,
       xtickfont=10, ytickfont=10)
       hline!([0], c=:black, linestyle=:dash, linewidth=2)
```



3.6.7 Qualitative Predictors

For the Carseats dataset, it's easy to get a linear regression model, we just simply apply the lm() function, including the interaction terms Income*Advertising and Price*Age.

```
julia> Carseats = CSV.read("C:\\ISL_2024\\Datasets\\Carseats.csv", DataFrame)
       first(Carseats, 5)
```

5×11 DataFrame

Row	Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US
	Float64	Int64	Int64	Int64	Int64	Int64	String7	Int64	Int64	String3	String3
1	9.5	138	73	11	276	120	Bad	42	17	Yes	Yes
2	11.22	111	48	16	260	83	Good	65	10	Yes	Yes
3	10.06	113	35	10	269	80	Medium	59	12	Yes	Yes
4	7.4	117	100	4	466	97	Medium	55	14	Yes	Yes
5	4.15	141	64	3	340	128	Bad	38	13	Yes	No

```
julia> model_cars = lm(@formula(Sales ~ CompPrice + Income + Advertising + Population
       + Price + ShelveLoc + Age + Education + Urban + US + Income*Advertising +
       Price*Age), Carseats)
```

```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}}},
Matrix{Float64}}

Coefficients:
────────────────────────────────────────────────────────────────────────────────────
                          Coef.     Std. Error      t    Pr(>|t|)  Lower 95%     Upper 95%
────────────────────────────────────────────────────────────────────────────────────
(Intercept)           6.57557     1.00875        6.52   <1e-09    4.59224       8.55889
CompPrice             0.0929371   0.00411831    22.57   <1e-71    0.08484       0.101034
Income                0.010894    0.00260444     4.18   <1e-04    0.0057733     0.0160146
Advertising           0.0702462   0.0226091      3.11   0.0020    0.0257938     0.114699
Population            0.000159245 0.000367858    0.43   0.6653   -0.00056401    0.000882501
Price                -0.100806    0.00743989   -13.55   <1e-33   -0.115434     -0.0861786
ShelveLoc: Good       4.84868     0.152838      31.72   <1e-99    4.54818       5.14918
ShelveLoc: Medium     1.95326     0.125768      15.53   <1e-41    1.70599       2.20054
Age                  -0.0579466   0.0159506     -3.63   0.0003   -0.0893075    -0.0265857
Education            -0.0208525   0.0196131     -1.06   0.2884   -0.0594145     0.0177095
Urban: Yes            0.14016     0.112402       1.25   0.2132   -0.0808369     0.361156
US: Yes              -0.157557    0.148923      -1.06   0.2907   -0.45036       0.135245
Income & Advertising  0.000751039 0.000278409    2.70   0.0073    0.000203651   0.00129843
Price & Age           0.00010676  0.000133337    0.80   0.4238   -0.000155398   0.000368918
────────────────────────────────────────────────────────────────────────────────────
```