Diamond Price Analysis

Authors:

- Marcin Bereźnicki
- Jakub Burczyk

The problem:

The purpose of the project was to analyze diamond's pricing based on it's weight.

The goal:

We hope, that after creating sufficient model it will be possible to predict a price for gem given it's mass in carats.

It may be possible to estimate a price without any trade specific knowledge for example about type of cuts, which could prevent getting ripped off by sellers/buyers.

Table of contents

We highly recommend using provided hyperlinks to sections.

Dataset

Return to table of contents

The data was sourced from Kaggle.com which is an online community of data scientists. The dataset can be downloaded **here**.

Dataset contains 53 941 records containing description of 10 diamond properties.

The colums are as follows:

- **price** - in US dollars
- **carat** - weight of the gem
- **cut** - quality of the cut
- **color** - gem's color
- **clarity** - measurement how clear the gem is and it's defects
- **x** - length in milimiters
- **y** - width in milimiters
- **z** - depth in milimiters
- **table** width of top face of the diamond relative to widest point
- **depth** - depth percentage

$$depth = \frac{z}{mean(x, y)} \tag{1}$$

Imports

Return to table of contents

Necessary python modules for data analysis.

In [ ]:
```python
from cmdstanpy import CmdStanModel

import arviz as az
import numpy as np
import scipy.stats as stats

import matplotlib.pyplot as plt
import pandas as pd
import random as rd

import warnings
warnings.filterwarnings('ignore')
```

Data tidying

Return to table of contents

Before starting analysis it is necessary to clean up the dataset.

Dropping index

Return to table of contents

The first column contians record id without column name, but for our purpouses it is not necessary thus it gets dropped after loading the dataset file.

In [ ]:
```python
df = pd.read_csv("data/diamonds.csv")
df.drop(columns=["Unnamed: 0"], inplace=True)
df.head()
```

Out[ ]:

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

Mass and price extraction

Return to table of contents

For our analysis we will consider only one variable affecting gem pricing - it's mass. For each carat value, the average diamond price is calculated.

The data gets trimmed to contain only the necessary information and saved accordingly to `CSV` file located at `./output/caratPrice.csv` .

In [ ]:
```
caratPriceDict = dict()
caratAmountDict = dict()

for index, row in df.iterrows():
    if row['carat'] not in caratPriceDict.keys():
        caratPriceDict[row['carat']] = 0
    if row['carat'] not in caratAmountDict.keys():
        caratAmountDict[row['carat']] = 0

    caratPriceDict[row['carat']] += row['price']
    caratAmountDict[row['carat']] += 1

caratPriceDict = {k: v/caratAmountDict[k] for k, v in caratPriceDict.items()}
caratPriceDict = dict(sorted(caratPriceDict.items()))

carat = list(caratPriceDict.keys())
price = list(caratPriceDict.values())

caratPrice = {"carat":carat, "price":price}
dfCaratPrice = pd.DataFrame(caratPrice)
dfCaratPrice.to_csv(r'output/caratPrice.csv', index=False)
```

Plotting dataset and polynomial fitting

Return to table of contents

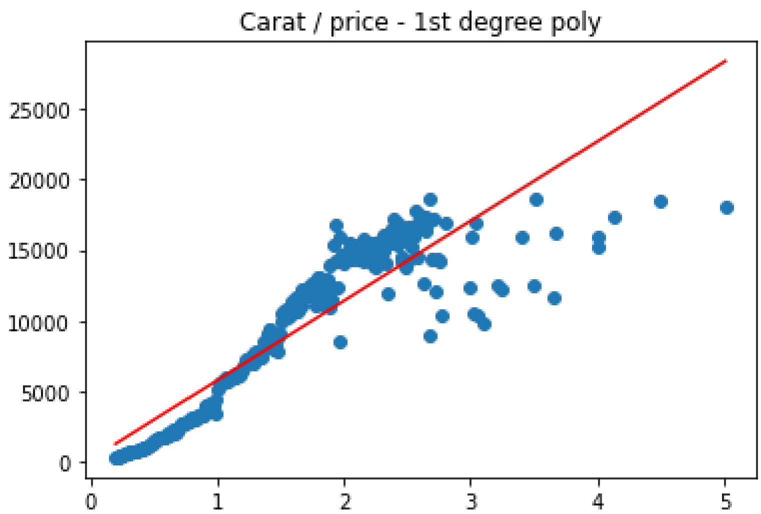
It is important to see the data before commencing analysis, afterall we should check in case it's utter nonsense as demonstrated **here**.

Before creating stan models, polynomial functions of 1st and 4th degree were fitted.

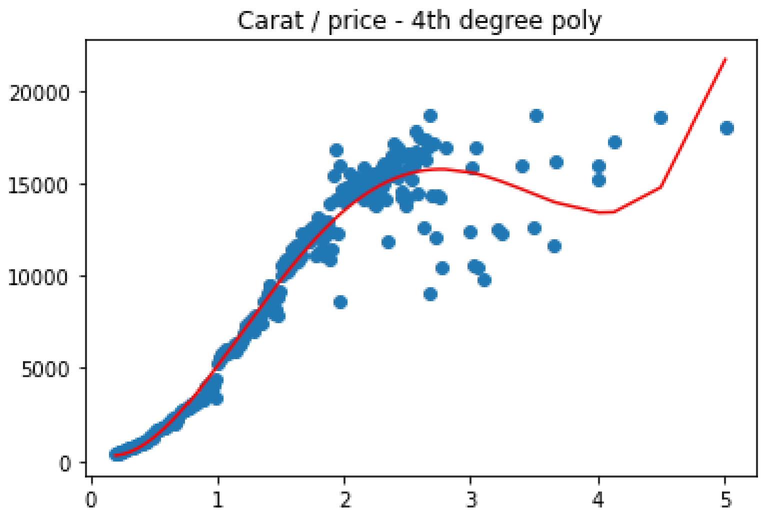
Calculated polynomial coefficients were basis for distribution of paramerters in `stan` models.

In [ ]:
```
plt.scatter(carat, price)
z = np.polyfit(carat, price, 1)
p = np.poly1d(z)

trend_h = p(carat)
plt.plot(carat,trend_h, "r-")
plt.title("Carat / price - 1st degree poly")
plt.show()

print("Linear function:")
print(p)
print("")

plt.scatter(carat, price)
```

```
z = np.polyfit(carat, price, 4)
p = np.poly1d(z)
trend_h = p(carat)
plt.plot(carat,trend_h, "r-")
plt.title("Carat / price - 4th degree poly")
plt.show()

print("Polynomial function:")
print(p)
print("")
```




```
Linear function:

5625 x + 192.5
```




```
Polynomial function:
       4          3              2
514.7 x - 4797 x + 1.274e+04 x - 3957 x + 616.6
```

Data Analysis

Return to table of contents

For analysis we have created 3 bayesian models.

- Model 1 - uses 1st degree polynomial (a linear function)
- Model 2 - uses 4th degree polynomial
- Model 3 - is based on a Gaussian Process.

Expanding the first model by increasing the order of the polynomial allows for a better fit of the model to the observations, in terms of the data, and for value prediction. The third model is the departure from polynomial regression to the simulating from a Gaussian Process conditional on non-Gaussian observations.

The equations, parameters and differences of individual models are presented in the corresponding chapters.

Loading dataset

Return to table of contents

In [ ]:
```
df = pd.read_csv("output/caratPrice.csv")
df.head()

print(df.describe())
```

```
            carat         price
count  273.000000    273.000000
mean     1.608791   9242.669570
std      0.894875   5606.869496
min      0.200000    365.166667
25%      0.880000   3342.322581
50%      1.560000  10424.000000
75%      2.240000  14481.333333
max      5.010000  18701.000000
```

Model 1 - Linear regression - 1st degree

Return to table of contents

Our first model was based on 1st degree polynomial function.

Model has form:

$$y \sim \mathrm{Normal}(\alpha + X\beta, \sigma)$$

With parameter distributions set as follows:

$$\alpha \sim \mathrm{Normal}(193, 5)$$

$$\beta \sim \mathrm{Normal}(5625, 5)$$

$$\sigma \sim \mathrm{Exponential}(5)$$

The required input data is the set of carats for which the user wants to make a prediction.

Model 1 - Prior predictive check

Return to table of contents

First step is prior predicive check whether parameter values and distributions "make sense".

Parameters simulated from priors are a result of the model definition. The first order polynomial requires two parameters to equate the line, and the third is the width of the fit.

On the basis of the obtained parameter values, it can be concluded that the prior selection was successful, the values are in line with the expectations.

On the basis of the obtained straight line fit to the measurements, it can be concluded that the prior predictive was successful. The obtained lines pass through the points as expected.

Priors were selected on the basis of the polynomial equation in the chapter Plotting dataset and polynomial fitting.

```
data {
  int N;
  real carat[N];
}

generated quantities {
    real alpha = normal_rng(193, 5);
    real beta = normal_rng(5625, 5);
    real sigma = exponential_rng(5);
    real price [N];
    for (i in 1:N) {
        price[i] = normal_rng(alpha + beta * carat[i], sigma);
    }
}
```

In [ ]:

```
model_1_ppc = CmdStanModel(stan_file='stanfiles/model_1_ppc.stan')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

In [ ]:

```
data_sim={'N':len(df), 'carat':df.carat}
model_1_sim = model_1_ppc.sample(data=data_sim, iter_sampling=1000, iter_warmup=0,
```

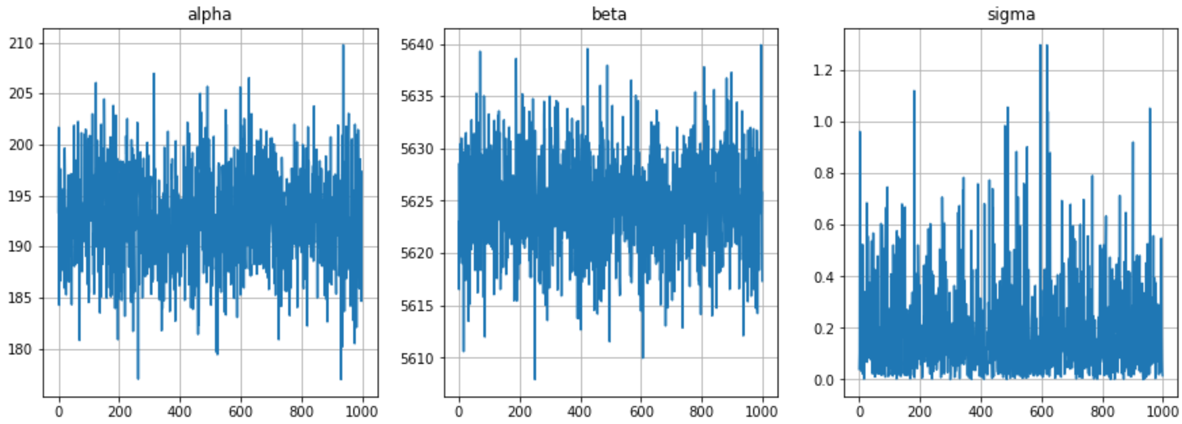
```
INFO:cmdstanpy:CmdStan start processing
chain 1 |          | 00:00 Sampling completed
```

```
INFO:cmdstanpy:CmdStan done processing.
```

In [ ]:

```
alpha_sim = model_1_sim.stan_variable('alpha')
beta_sim = model_1_sim.stan_variable('beta')
sigma_sim = model_1_sim.stan_variable('sigma')
price_sim = model_1_sim.stan_variable('price')

fig, axs = plt.subplots(1,3)
fig.set_size_inches(15, 5)
axs[0].plot(alpha_sim)
axs[0].grid()
axs[0].set_title('alpha')
```

```
axs[1].plot(beta_sim)
axs[1].grid()
axs[1].set_title('beta')
axs[2].plot(sigma_sim)
axs[2].grid()
axs[2].set_title('sigma')
plt.show()

az.summary(model_1_sim,var_names=['alpha','beta','sigma'],round_to=2,kind='stats')
```



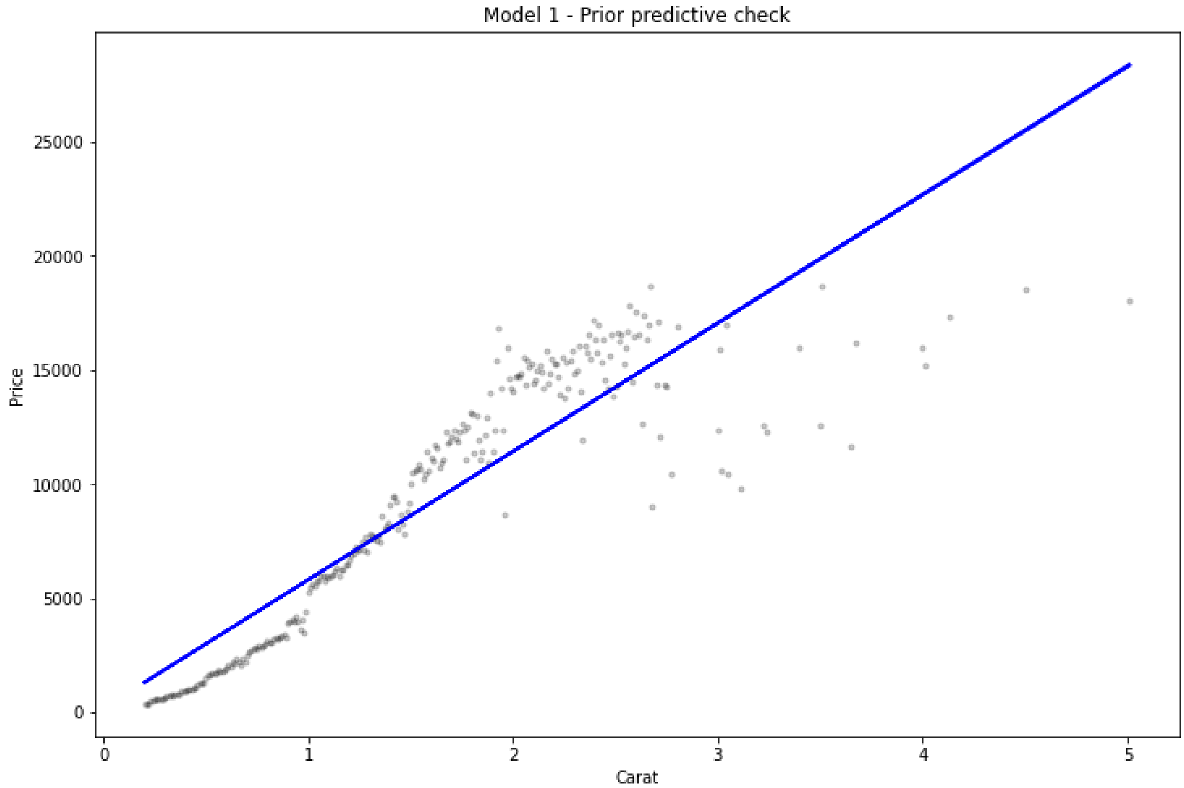
Out[ ]:

	mean	sd	hdi_3%	hdi_97%
alpha	192.69	4.84	184.52	201.98
beta	5624.82	4.84	5615.06	5633.17
sigma	0.19	0.19	0.00	0.54

On the basis of the obtained parameter values, it can be concluded that the prior selection was successful, the values are in line with the expectations.

In [ ]:

```
plt.figure(figsize=[12, 8])
for i in range(100):
    plt.plot(df.carat, alpha_sim[i] + beta_sim[i] * df.carat, alpha=0.1, color='blu
plt.scatter(df.carat, df.price, color='black', alpha=0.2, s=10)
plt.xlabel("Carat")
plt.ylabel("Price")
plt.title("Model 1 - Prior predictive check")
plt.show()
```



On the basis of the obtained straight line fit to the measurements, it can be concluded that the prior predictive was successful. The obtained lines pass through the points as expected.

Model 1 - Posterior analysis

Return to table of contents

After confirming that the priors values and trajectories are correct we can start a proper analysis.

No issues were detected during sampling.

A full model for 1st degree polynomial regression was created:

```stan
data {
    int <lower = 0> N;
    vector [N] carat;
    vector [N] price;
}

parameters {
    real alpha;
    real beta;
    real <lower = 0> sigma;
}

transformed parameters {
    vector[N] mu;
    for (i in 1:N) {
        mu[i] = alpha + beta * carat[i];
    }
}

model {
    alpha ~ normal(193, 5);
    beta ~ normal(5625, 5);
    sigma ~ exponential(5);
    price ~ normal(mu, sigma);
}

generated quantities {
    vector [N] price_sim;
    vector [N] log_lik;

    for(i in 1:N){
        log_lik[i] = normal_lpdf(price[i] | mu[i], sigma);
        price_sim[i] = normal_rng(mu[i], sigma);
    }

}
```

In [ ]:

```python
model_lr = CmdStanModel(stan_file='stanfiles/model_lr.stan')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

In [ ]:

```python
result_lr = model_lr.sample(data=dict(N=len(df), carat=df.carat, price=df.price))
#print(result_lr.draws_pd())
```

```
INFO:cmdstanpy:CmdStan start processing
chain 1 |          | 00:00 Status

chain 1 |█         | 00:00 Iteration:  200 / 2000 [ 10%]  (Warmup)

chain 1 |█████     | 00:00 Iteration: 1001 / 2000 [ 50%]  (Sampling)

chain 1 |███████   | 00:00 Iteration: 1400 / 2000 [ 70%]  (Sampling)

chain 1 |████████  | 00:01 Iteration: 1700 / 2000 [ 85%]  (Sampling)

chain 1 |██████████| 00:01 Iteration: 1900 / 2000 [ 95%]  (Sampling)
chain 1 |██████████| 00:01 Sampling completed
chain 2 |██████████| 00:01 Sampling completed
chain 3 |██████████| 00:01 Sampling completed
chain 4 |██████████| 00:01 Sampling completed
```

```
INFO:cmdstanpy:CmdStan done processing.
```

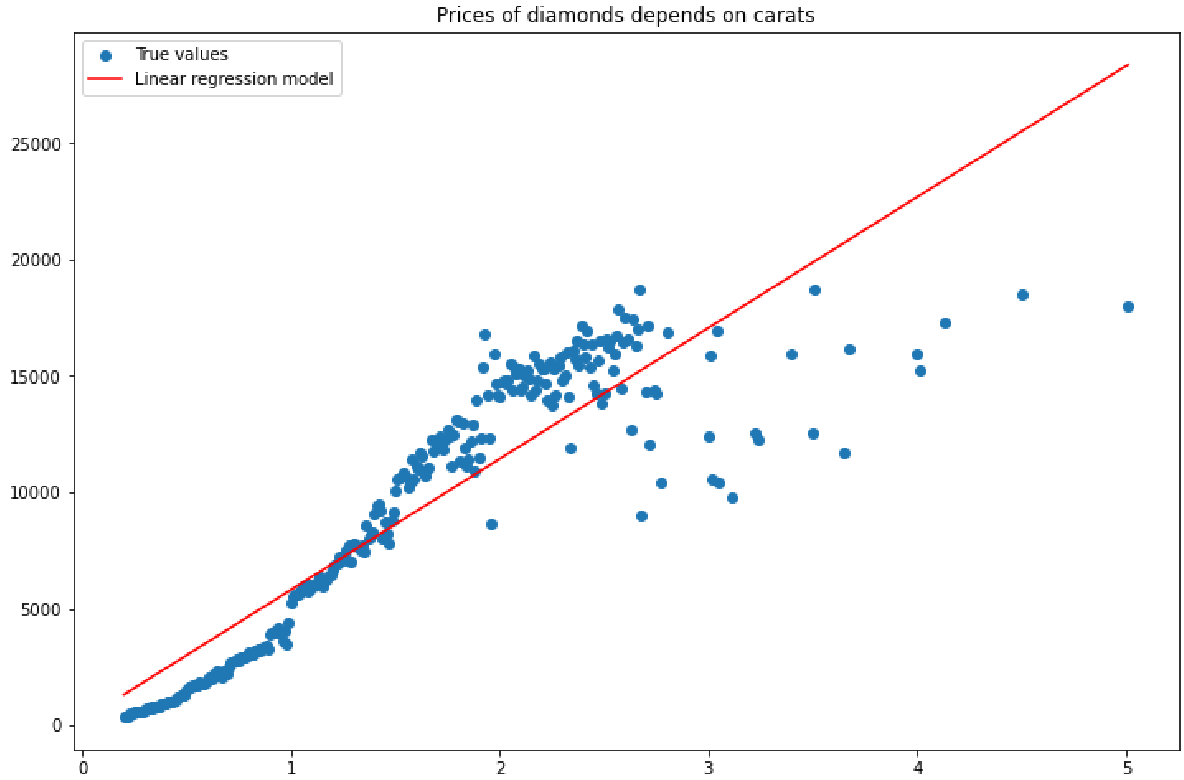
Model 1 - Stan linear regression

Return to table of contents

After creating, compiling and sampling the model we can observe the results. First of them is resulatant linear regression. The coefficients are read from results and function values are calculated for the plot.

In [ ]:

```
df_alpha = pd.DataFrame(result_lr.stan_variables()["alpha"])
df_beta = pd.DataFrame(result_lr.stan_variables()["beta"])
df_sigma = pd.DataFrame(result_lr.stan_variables()["sigma"])

alpha = df_alpha.mean().to_numpy()
beta = df_beta.mean().to_numpy()
sigma= df_sigma.mean().to_numpy()

x = df.carat.values
y = alpha + beta*x

plt.figure(figsize=[12, 8])
plt.scatter(df.carat.values, df.price.values)
plt.plot(x, y, "-r")
plt.title("Prices of diamonds depends on carats")
plt.legend(["True values", "Linear regression model"])
plt.show()
```



Model 1 - model parameters

Return to table of contents

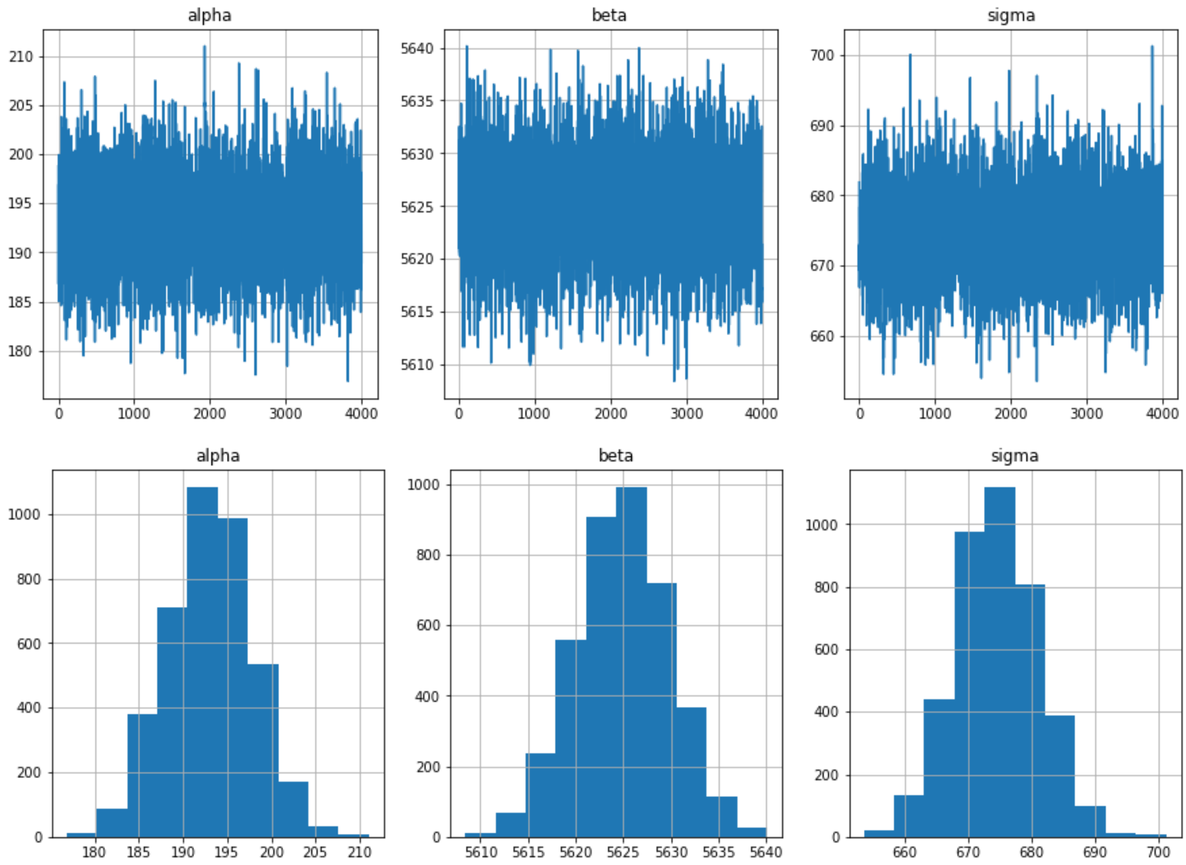
We can also extract stan variables that are used in the final price prediction equation.

Based on the presented graphs and histograms of parameters, it can be concluded that parameter values are relatively concentrated.

Their slight dispersion is good due to the fact that it is not possible to perfectly match the lines to observations.

In [ ]:
```
fig, axs = plt.subplots(1,3)
fig.set_size_inches(15, 5)
axs[0].plot(df_alpha)
axs[0].grid()
axs[0].set_title('alpha')
axs[1].plot(df_beta)
axs[1].grid()
axs[1].set_title('beta')
axs[2].plot(df_sigma)
axs[2].grid()
axs[2].set_title('sigma')
plt.show()

fig, axs = plt.subplots(1,3)
fig.set_size_inches(15, 5)
axs[0].hist(df_alpha)
axs[0].grid()
axs[0].set_title('alpha')
axs[1].hist(df_beta)
axs[1].grid()
axs[1].set_title('beta')
```

```
axs[2].hist(df_sigma)
axs[2].grid()
axs[2].set_title('sigma')
plt.show()

az.summary(result_lr,var_names=['alpha','beta','sigma'],round_to=2,kind='stats')
```



Out[ ]:

	mean	sd	hdi_3%	hdi_97%
alpha	193.03	4.83	183.83	201.81
beta	5624.93	4.90	5616.13	5634.50
sigma	674.56	6.51	662.45	686.63

Model 1 - evaluation

Return to table of contents

We can now observe the results.

Model 1 - quantiles

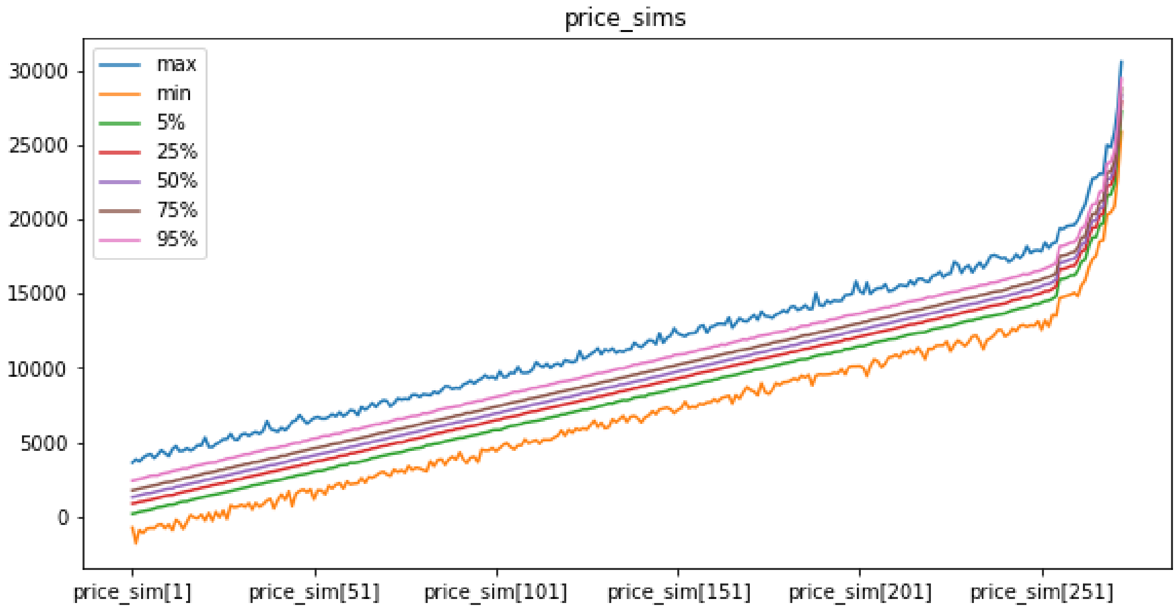
Return to table of contents

After simulating we can analyze predictions. The quantiles follow a linear function. The right side of the plot gets squished as there are fewer diamonds of higher weights.

In [ ]:

```
data = result_lr.draws_pd()
price_sims = data[data.columns[283:556]]
```

```
#print(price_sims)

quans = pd.DataFrame({'max': price_sims.max(), 'min': price_sims.min(), '5%': pric
quans.plot(figsize=(10,5))
plt.title("price_sims")
plt.show()
```



Model 1 - predictions and density plot

Return to table of contents

As we can see the model is not sufficient to describe the phenomenon.

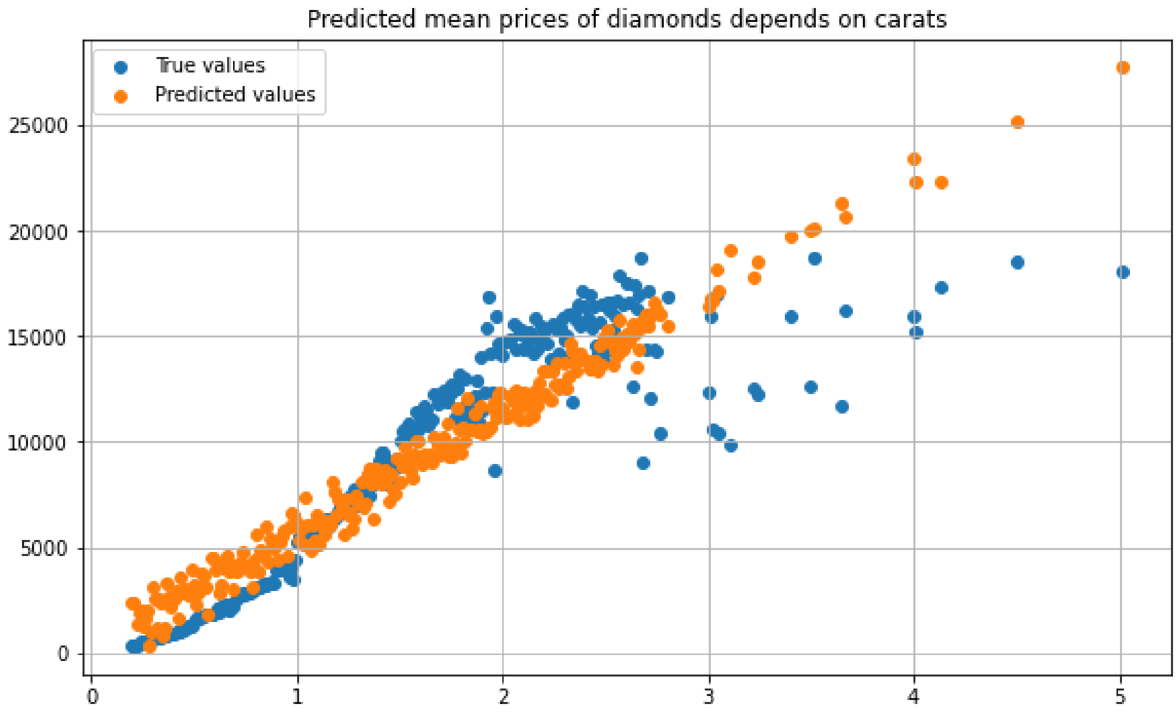
While it's somewhat true for a narrow band of weights (about 1-1.5 carats), it is not accurate for low and med-high gems. It also gets progressively worse as true diamond prices plateau above 3 carat mark, most likely due to higher chances of defects.

The model predicted that most diamonds oscillate around value of 10 000 while the ground truth is exacly opposite, there are more low and high costs diamonds rather than the mid ones.

In [ ]:

```
price_sim = result_lr.stan_variable('price_sim')
plt.figure(figsize=[10,6])
plt.scatter(df.carat.values, df.price.values)
plt.scatter(df.carat.values, price_sim[1])
plt.title("Predicted mean prices of diamonds depends on carats")
plt.legend(["True values", "Predicted values"])
plt.grid()
plt.show()


df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
for i in range(0,10):
    price_sims.iloc[i].plot.density(linewidth=0.25, color='blue')
df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
plt.title('Density plot for Price')
plt.legend(["True values", "Predicted values"])
```

```
plt.grid()
plt.show()
```



Model 2 - Linear regression - 4th degree

Return to table of contents

The 1st degree, quadratic and 3rd degree models proved to be insufficient, thus we expanded the first model to 4th degree polynomial.

Model has form:

$$y \sim \mathrm{Normal}(\alpha + X\beta + X^2\beta_2 + X^3\beta_3 + X^4\beta_4, \sigma)$$

With parameter distributions set as follows:

$$\alpha \sim \text{Normal}(617, 5)$$

$$\beta_1 \sim \text{Normal}(-3957, 5)$$

$$\beta_2 \sim \text{Normal}(12740, 5)$$

$$\beta_3 \sim \text{Normal}(-4797, 5)$$

$$\beta_4 \sim \text{Normal}(515, 5)$$

$$\sigma \sim \text{Exponential}(5)$$

The required input data is the set of carats for which the user wants to make a prediction.

Model 2 - Prior predictive check

Return to table of contents

First step is prior predicive check whether parameter values and distributions "make sense".

Parameters simulated from priors are a result of the model definition. The 4th order polynomial requires 5 parameters to equate the curve, and the 6th is the width of the fit.

On the basis of the obtained parameter values, it can be concluded that the prior selection was successful, the values are in line with the expectations.

On the basis of the obtained curve fit to the measurements, it can be concluded that the prior predictive was successful. The obtained lines pass through the points as expected.

Priors were selected on the basis of the polynomial equation in the chapter Plotting dataset and polynomial fitting.

```
data {
  int N;
  real carat[N];
}

generated quantities {
    real alpha = normal_rng(617, 5);
    real beta_1 = normal_rng(-3957, 5);
    real beta_2 = normal_rng(12740, 5);
    real beta_3 = normal_rng(-4797, 5);
    real beta_4 = normal_rng(515, 5);
    real sigma = exponential_rng(5);
    real price [N];
    for (i in 1:N) {
        price[i] = normal_rng(alpha + beta_1 * carat[i] + beta_2 * carat[i]^2 + beta_3 * carat[i]^3 + beta_4 * carat[i]^4, sigma);
    }
}
```

In [ ]:

```
model_2_ppc = CmdStanModel(stan_file='stanfiles/model_2_ppc.stan')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

In [ ]:

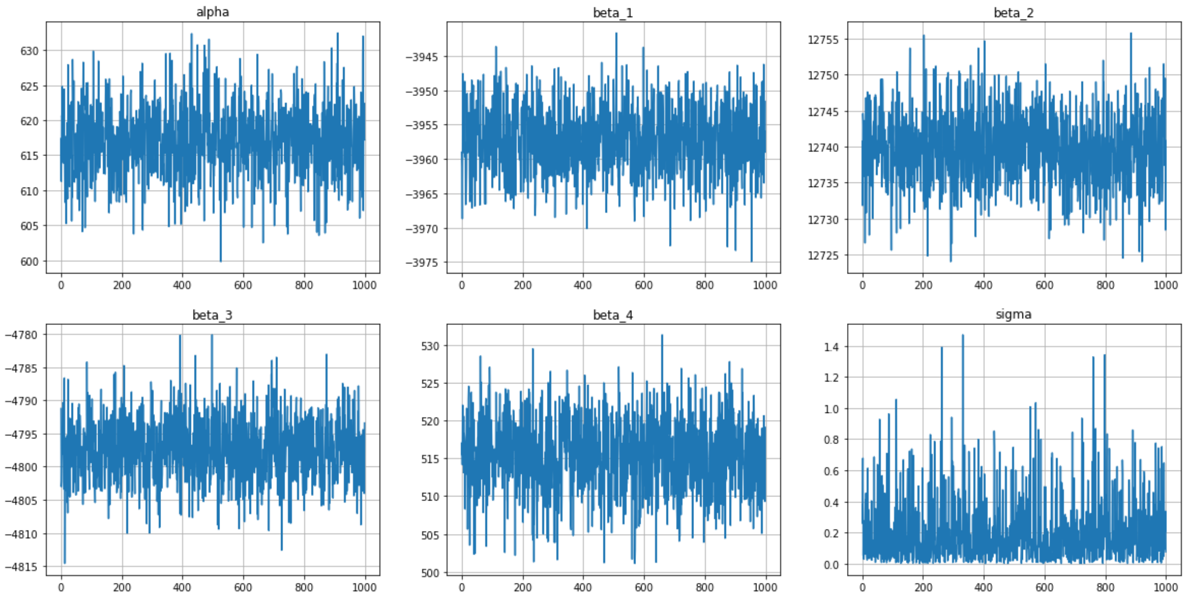
```
data_sim={'N':len(df), 'carat':df.carat}
model_2_sim = model_2_ppc.sample(data=data_sim, iter_sampling=1000, iter_warmup=0,
```

```
INFO:cmdstanpy:CmdStan start processing
chain 1 |          | 00:00 Sampling completed
```

```
INFO:cmdstanpy:CmdStan done processing.
```

In [ ]:

```
alpha_sim = model_2_sim.stan_variable('alpha')
beta_1_sim = model_2_sim.stan_variable('beta_1')
beta_2_sim = model_2_sim.stan_variable('beta_2')
beta_3_sim = model_2_sim.stan_variable('beta_3')
beta_4_sim = model_2_sim.stan_variable('beta_4')
sigma_sim = model_2_sim.stan_variable('sigma')
price_sim = model_1_sim.stan_variable('price')

fig, axs = plt.subplots(2,3)
fig.set_size_inches(20, 10)
axs[0][0].plot(alpha_sim)
axs[0][0].grid()
axs[0][0].set_title('alpha')
axs[0][1].plot(beta_1_sim)
axs[0][1].grid()
axs[0][1].set_title('beta_1')
axs[0][2].plot(beta_2_sim)
axs[0][2].grid()
axs[0][2].set_title('beta_2')
axs[1][0].plot(beta_3_sim)
axs[1][0].grid()
axs[1][0].set_title('beta_3')
axs[1][1].plot(beta_4_sim)
axs[1][1].grid()
axs[1][1].set_title('beta_4')
axs[1][2].plot(sigma_sim)
axs[1][2].grid()
axs[1][2].set_title('sigma')
plt.show()

az.summary(model_2_sim,var_names=['alpha','beta_1','beta_2','beta_3','beta_4','sig
```



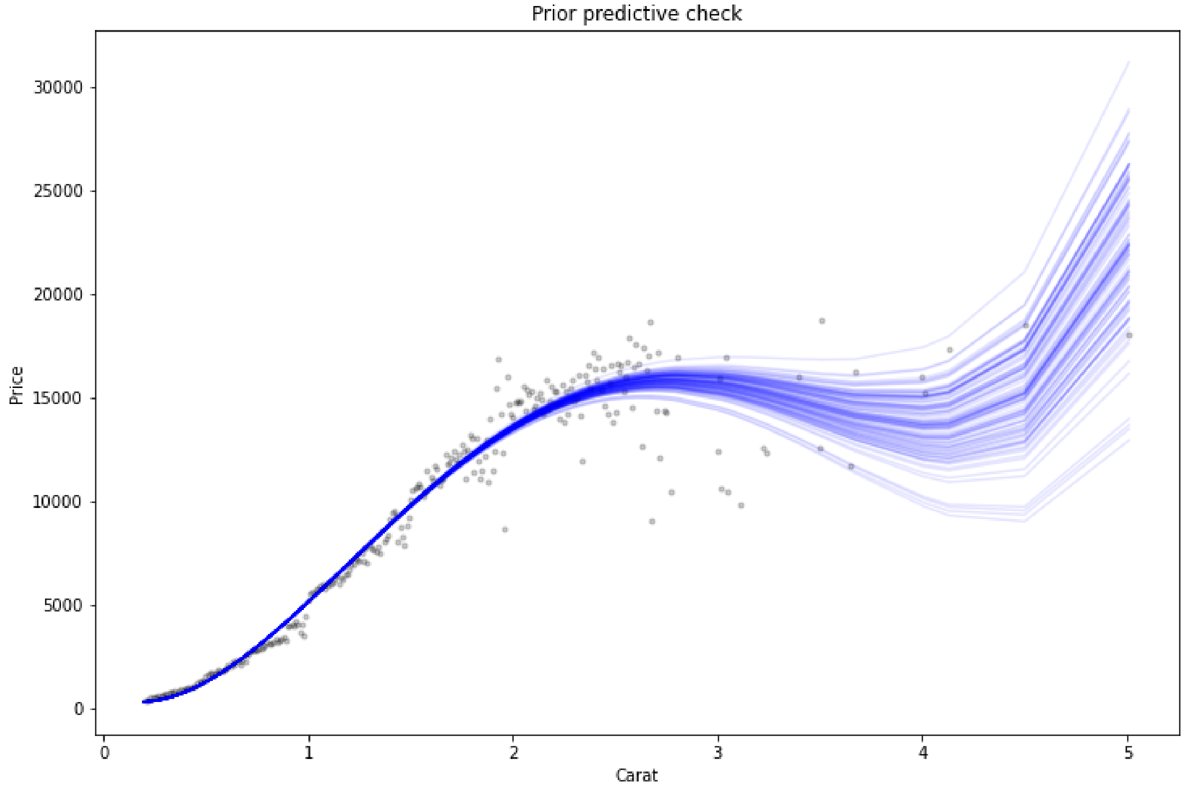
Out[ ]:

	mean	sd	hdi_3%	hdi_97%
alpha	617.08	5.06	607.90	627.35
beta_1	-3957.22	4.94	-3966.65	-3948.51
beta_2	12739.88	5.08	12730.90	12749.50
beta_3	-4797.26	4.86	-4805.79	-4788.22
beta_4	515.38	5.03	506.00	524.67
sigma	0.20	0.21	0.00	0.62

On the basis of the obtained parameter values, it can be concluded that the prior selection was successful, the values are in line with the expectations.

In [ ]:

```
plt.figure(figsize=[12, 8])
for i in range(100):
    plt.plot(df.carat, alpha_sim[i] + beta_1_sim[i] * df.carat + beta_2_sim[i] * d
plt.scatter(df.carat, df.price, color='black', alpha=0.2, s=10)
plt.xlabel("Carat")
plt.ylabel("Price")
plt.title("Prior predictive check")
plt.show()
```



On the basis of the obtained straight line fit to the measurements, it can be concluded that the prior predictive was successful. The obtained lines pass through the points as expected.

Model 2 - Posterior analysis

Return to table of contents

After confirming that the priors values and trajectories are correct we can start a proper analysis.

No issues were detected during sampling.

A full model for 4th degree polynomial regression was created:

```
data {
    int <lower = 0> N;
    vector [N] carat;
    vector [N] price;
}

parameters {
    real alpha;
    real beta_1;
    real beta_2;
    real beta_3;
    real beta_4;
    real <lower = 0> sigma;
}

transformed parameters {
    vector[N] mu;
    for (i in 1:N) {
        mu[i] = alpha + beta_1 * carat[i] + beta_2 * carat[i]^2 + beta_3 * carat[i]^3 + beta_4 * carat[i]^4;
    }
}

model {
    alpha ~ normal(617, 5);
    beta_1 ~ normal(-3957, 5);
    beta_2 ~ normal(12740, 5);
    beta_3 ~ normal(-4797, 5);
    beta_4 ~ normal(515, 5);
    sigma ~ exponential(5);

    price ~ normal(mu, sigma);
}

generated quantities {
    vector [N] price_sim;
    vector [N] log_lik;

    for(i in 1:N){
        log_lik[i] = normal_lpdf(price[i] | mu[i], sigma);
        price_sim[i] = normal_rng(mu[i], sigma);
    }

}
```

In [ ]:
```
model_pr = CmdStanModel(stan_file='stanfiles/model_pr.stan')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

In [ ]:
```
result_pr = model_pr.sample(data=dict(N=len(df), carat=df.carat, price=df.price))
#print(result_pr.draws_pd())
```

```
INFO:cmdstanpy:CmdStan start processing
chain 1 |          | 00:00 Status



chain 1 |█          | 00:00 Iteration:    1 / 2000 [  0%]  (Warmup)


chain 1 |█▌         | 00:00 Iteration:  200 / 2000 [ 10%]  (Warmup)

chain 1 |██         | 00:01 Iteration:  500 / 2000 [ 25%]  (Warmup)

chain 1 |██▋        | 00:01 Iteration:  600 / 2000 [ 30%]  (Warmup)

chain 1 |██▌       | 00:02 Iteration:  700 / 2000 [ 35%]  (Warmup)


chain 1 |██▉       | 00:02 Iteration:  800 / 2000 [ 40%]  (Warmup)
chain 1 |███▎      | 00:02 Iteration:  900 / 2000 [ 45%]  (Warmup)
chain 1 |████▎      | 00:03 Iteration: 1001 / 2000 [ 50%]  (Sampling)
chain 1 |████▏      | 00:03 Iteration: 1100 / 2000 [ 55%]  (Sampling)
chain 1 |████▊      | 00:03 Iteration: 1200 / 2000 [ 60%]  (Sampling)


chain 1 |████▊      | 00:03 Iteration: 1300 / 2000 [ 65%]  (Sampling)
chain 1 |█████▎     | 00:03 Iteration: 1400 / 2000 [ 70%]  (Sampling)
chain 1 |█████▎     | 00:04 Iteration: 1500 / 2000 [ 75%]  (Sampling)
chain 1 |█████▉     | 00:04 Iteration: 1600 / 2000 [ 80%]  (Sampling)


chain 1 |█████▉    | 00:04 Iteration: 1700 / 2000 [ 85%]  (Sampling)
chain 1 |██████▏   | 00:04 Iteration: 1800 / 2000 [ 90%]  (Sampling)
chain 1 |██████▌   | 00:04 Iteration: 1900 / 2000 [ 95%]  (Sampling)

























chain 1 |██████████| 00:07 Sampling completed
chain 2 |██████████| 00:07 Sampling completed
chain 3 |██████████| 00:07 Sampling completed
chain 4 |██████████| 00:07 Sampling completed
```

```
INFO:cmdstanpy:CmdStan done processing.
```

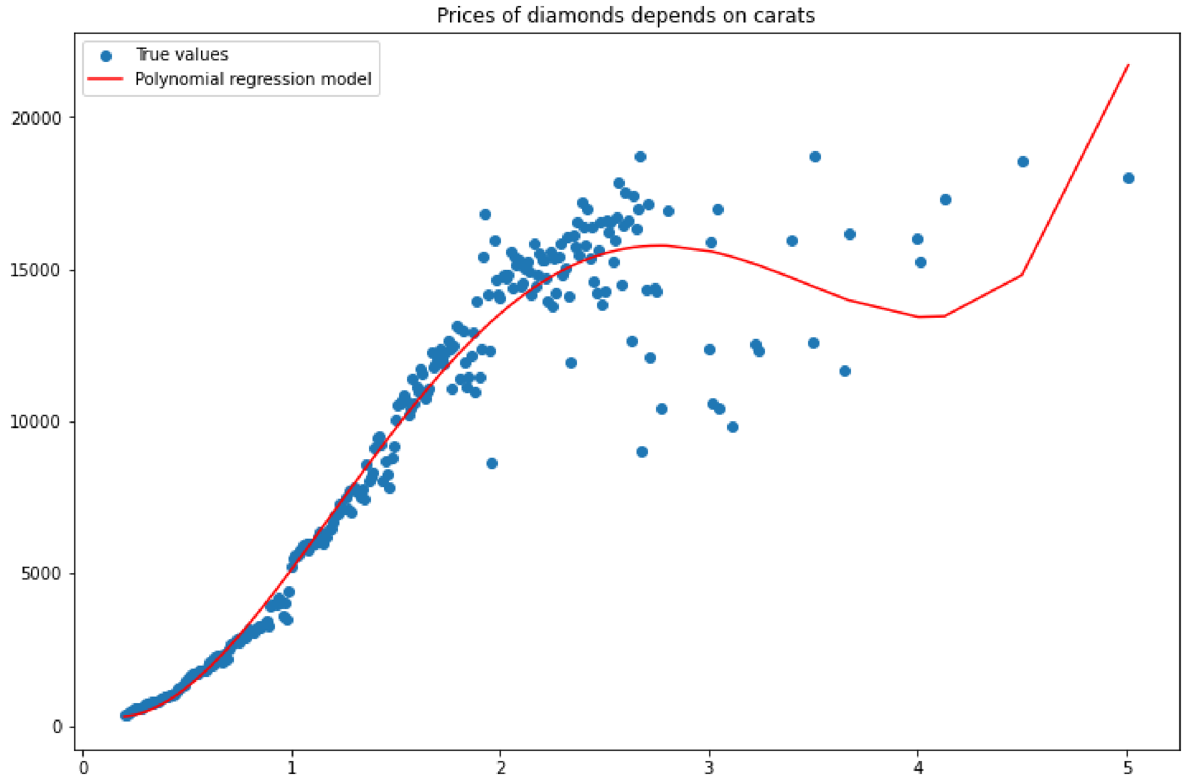
Model 2 - Stan linear regression

Return to table of contents

After creating, compiling and sampling the model we can observe the results. First of them is resulatant regression. The coefficients are read from results and function values are calculated for the plot.

In [ ]:

```
df_alpha = pd.DataFrame(result_pr.stan_variables()["alpha"])
df_beta_1 = pd.DataFrame(result_pr.stan_variables()["beta_1"])
df_beta_2 = pd.DataFrame(result_pr.stan_variables()["beta_2"])
df_beta_3 = pd.DataFrame(result_pr.stan_variables()["beta_3"])
df_beta_4 = pd.DataFrame(result_pr.stan_variables()["beta_4"])
df_sigma = pd.DataFrame(result_pr.stan_variables()["sigma"])

alpha = df_alpha.mean().to_numpy()
beta_1 = df_beta_1.mean().to_numpy()
beta_2 = df_beta_2.mean().to_numpy()
beta_3 = df_beta_3.mean().to_numpy()
beta_4 = df_beta_4.mean().to_numpy()

sigma= df_sigma.mean().to_numpy()

x = df.carat.values
y = alpha + beta_1*x + beta_2*(x**2) + beta_3*(x**3) + beta_4*(x**4)

plt.figure(figsize=[12, 8])
plt.scatter(df.carat.values, df.price.values)
plt.plot(x, y, "-r")
plt.title("Prices of diamonds depends on carats")
plt.legend(["True values", "Polynomial regression model"])
plt.show()
```



Model 2 - model parameters

[***Return to table of contents***]()

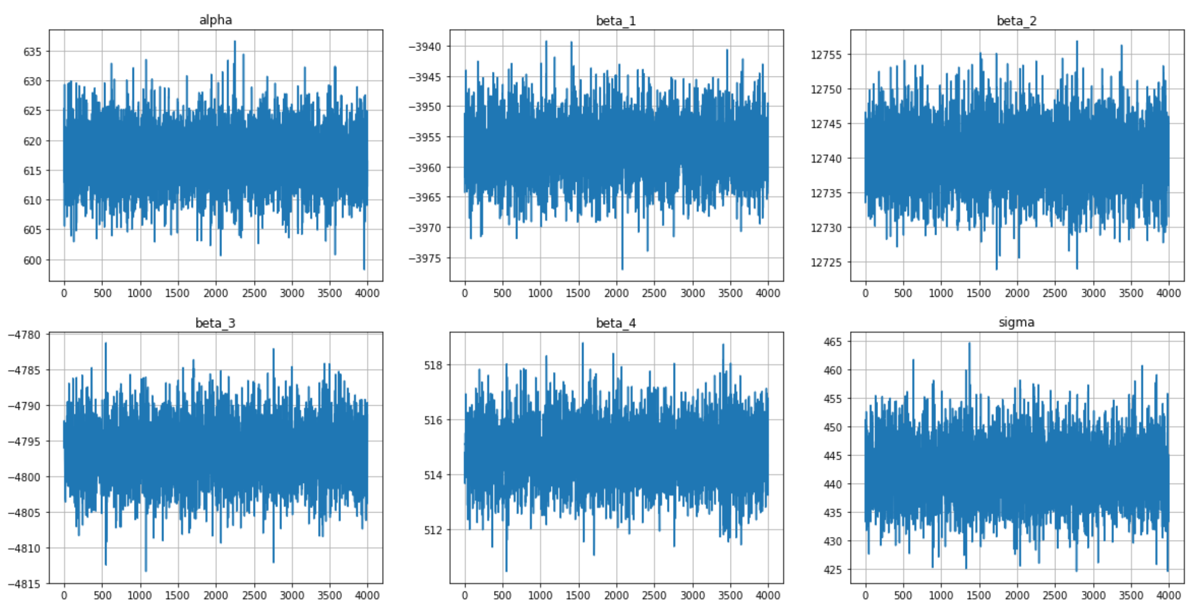
We can also extract stan variables that are used in the final price prediction equation.

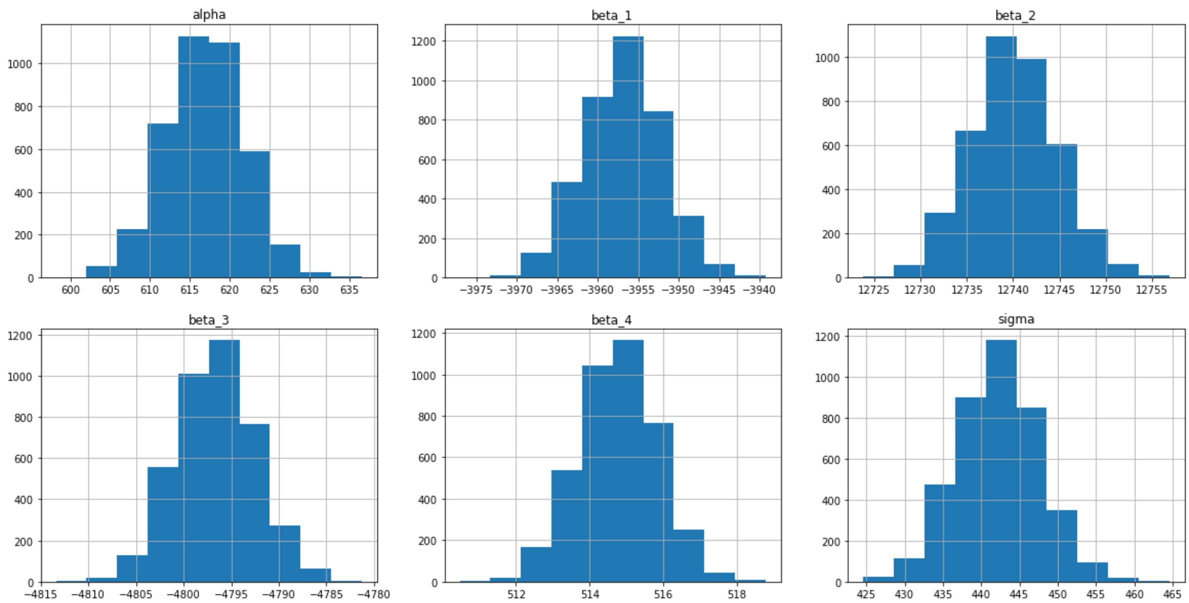
Based on the presented graphs and histograms of parameters, it can be concluded that parameter values are relatively concentrated.

Their slight dispersion is good due to the fact that it is not possible to perfectly match the lines to observations.

In [ ]:
```
fig, axs = plt.subplots(2,3)
fig.set_size_inches(20, 10)
axs[0][0].plot(df_alpha)
axs[0][0].grid()
axs[0][0].set_title('alpha')
axs[0][1].plot(df_beta_1)
axs[0][1].grid()
axs[0][1].set_title('beta_1')
axs[0][2].plot(df_beta_2)
axs[0][2].grid()
axs[0][2].set_title('beta_2')
axs[1][0].plot(df_beta_3)
axs[1][0].grid()
axs[1][0].set_title('beta_3')
axs[1][1].plot(df_beta_4)
axs[1][1].grid()
axs[1][1].set_title('beta_4')
axs[1][2].plot(df_sigma)
axs[1][2].grid()
axs[1][2].set_title('sigma')
plt.show()
```

```python
fig, axs = plt.subplots(2,3)
fig.set_size_inches(20, 10)
axs[0][0].hist(df_alpha)
axs[0][0].grid()
axs[0][0].set_title('alpha')
axs[0][1].hist(df_beta_1)
axs[0][1].grid()
axs[0][1].set_title('beta_1')
axs[0][2].hist(df_beta_2)
axs[0][2].grid()
axs[0][2].set_title('beta_2')
axs[1][0].hist(df_beta_3)
axs[1][0].grid()
axs[1][0].set_title('beta_3')
axs[1][1].hist(df_beta_4)
axs[1][1].grid()
axs[1][1].set_title('beta_4')
axs[1][2].hist(df_sigma)
axs[1][2].grid()
axs[1][2].set_title('sigma')
plt.show()

az.summary(model_2_sim,var_names=['alpha','beta_1','beta_2','beta_3','beta_4','sig
```





Out[ ]:

	mean	sd	hdi_3%	hdi_97%
alpha	617.08	5.06	607.90	627.35
beta_1	-3957.22	4.94	-3966.65	-3948.51
beta_2	12739.88	5.08	12730.90	12749.50
beta_3	-4797.26	4.86	-4805.79	-4788.22
beta_4	515.38	5.03	506.00	524.67
sigma	0.20	0.21	0.00	0.62

Model 2 - evaluation

Return to table of contents

We can now observe the results.

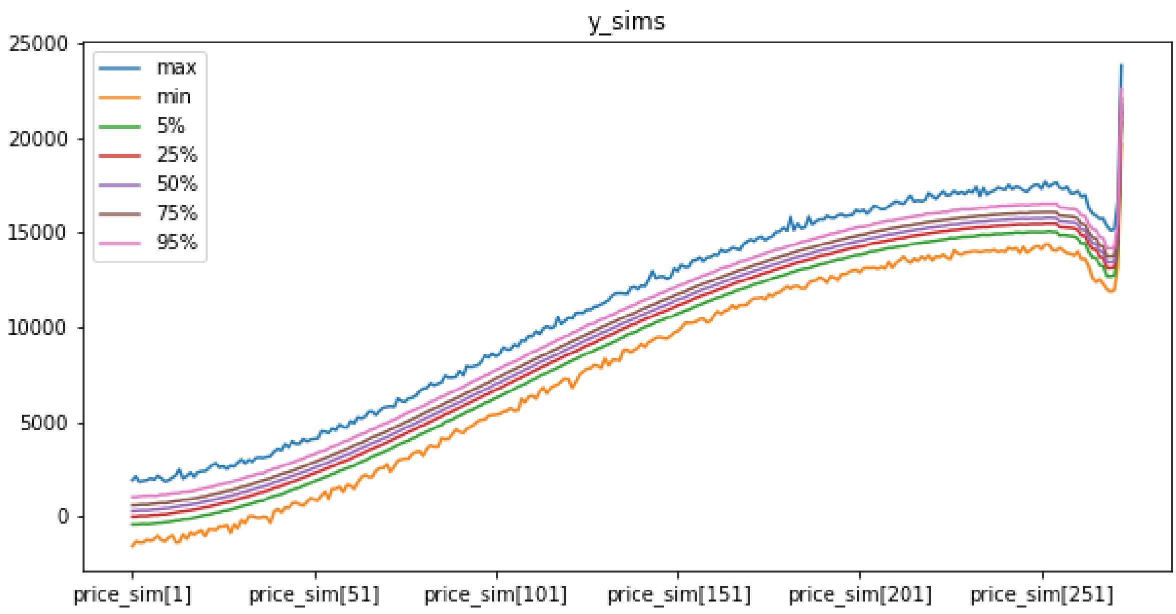
Model 2 - quantiles

Return to table of contents

After simulating we can analyze predictions. The quantiles follow a 4th degree polynomial function. The right side of the plot gets squished as there are fewer diamonds of higher weights.

In [ ]:

```
data = result_pr.draws_pd()
price_sims = data[data.columns[286:559]]
#print(price_sims)

quans = pd.DataFrame({'max': price_sims.max(), 'min': price_sims.min(), '5%': pric
quans.plot(figsize=(10,5))
plt.title("y_sims")
plt.show()
```



Model 2 - predictions and density plot

Return to table of contents

As we can see the model is somwehat sufficient to describe the phenomenon.

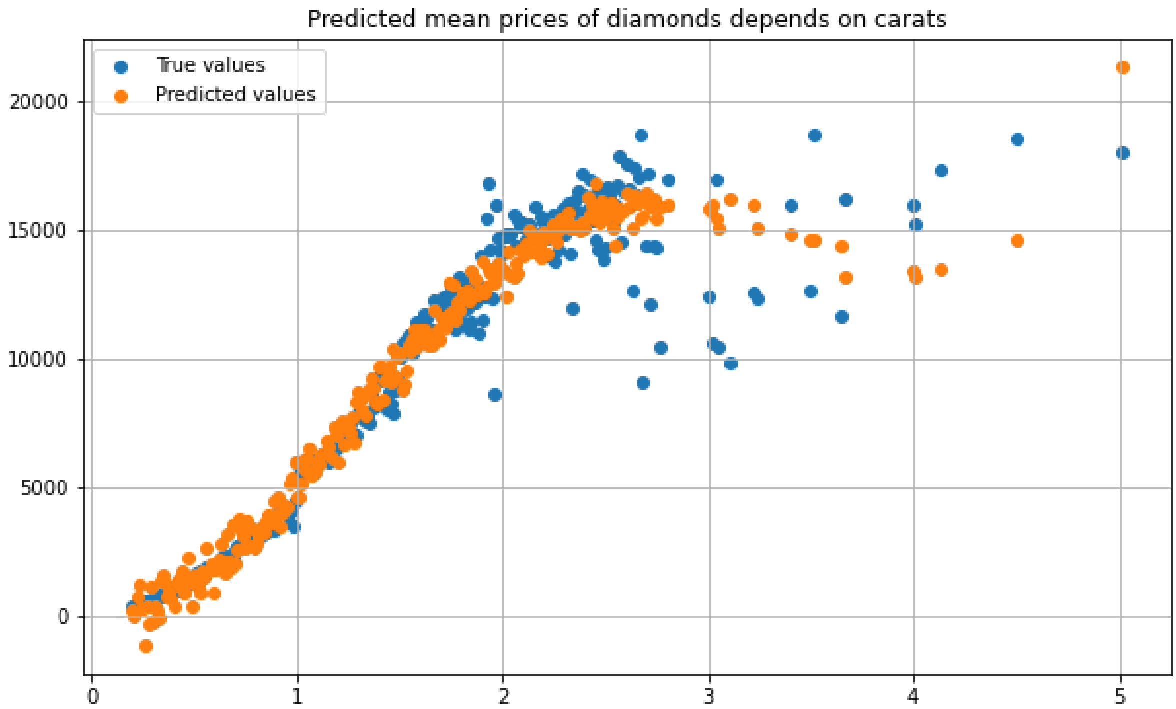
It stays fairly accurate up untill 4 carat mark when the behaviour of polynomial function can be observed. The values at the edges tend to diverge from fitted measurements towards infinity.

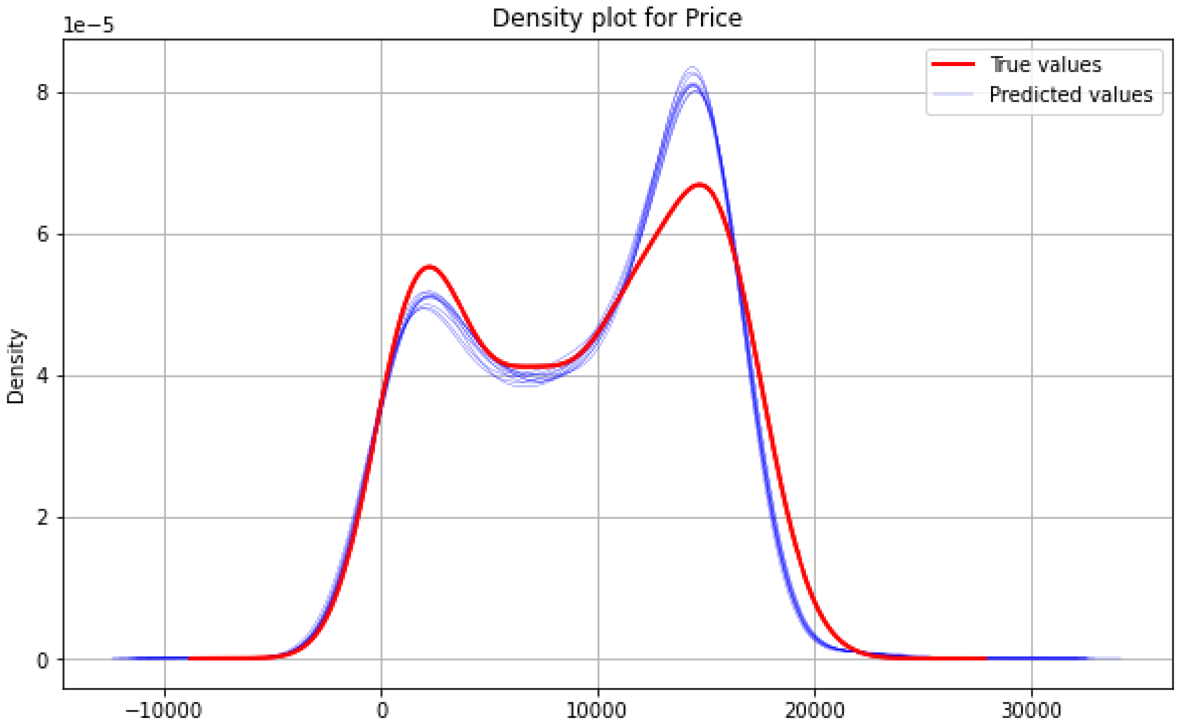
The model overestimates a little bit the amount of diamonds in each peak but the peak price range locations stay true to the measurements.

The model lacks a proper distribution of standard deviation, which for the real measurements increases as the weights go up - low weight diamonds tend to be of simmilar prices.

In [ ]:

```
price_sim = result_pr.stan_variable('price_sim')

plt.figure(figsize=[10,6])
plt.scatter(df.carat.values, df.price.values)
plt.scatter(df.carat.values, price_sim[1])
plt.title("Predicted mean prices of diamonds depends on carats")
plt.legend(["True values", "Predicted values"])
plt.grid()
plt.show()


df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
for i in range(0,10):
    price_sims.iloc[i].plot.density(linewidth=0.25, color='blue')
df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
plt.title('Density plot for Price')
plt.legend(["True values", "Predicted values"])
plt.grid()
plt.show()
```





Model 3 - Gaussian process

Return to table of contents

The data held on carats and diamond prices are non-Gaussian observations. When the observation model is non-Gaussian the posterior Gaussian process has not a closed form kernel. In this case we have to construct the multivariate Gaussian distribution joint over all of the covariates within the model itself, and allow the fit to explore the conditional realizations. A possible way to implement the model is to pull the covariates with the variate observations out of the vector of all the covariates in order to specify the observation model. Because the model is no longer conjugate we have to fit the latent Gaussian process with Markov chain Monte Carlo.

For this purpose, the non-centered parameterization of the latent multivariate Gaussian which takes advantage of the fact that:

$$\mathbf{f} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

is equivalent to:

$$\tilde{\mathbf{f}} \sim \mathcal{N}(0, 1)$$

$$\mathbf{f} = \boldsymbol{\mu} + \mathbf{L}\tilde{\mathbf{f}}$$

where:

$$\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^T$$

Source can be found **here**.

The required input data is the set of carats for which the user wants to make a prediction.

Model 3 - Prior optimization

Return to table of contents

First step is prior optimalization so that the parameter values "make sense".

Parameters simulated from priors are a result of the model definition. It requires 3 parameters.

Based on the obtained parameters, it can be concluded that they were successfully optimized.

A model for prior values optimization was created:

```
data {
    int<lower = 1> N;
    real carat[N];
    vector[N] price;
}

parameters {
    real<lower=0, upper=130> alpha;
    real<lower=0, upper=1> rho;
    real<lower=0, upper=100> sigma;
}

model {
    matrix[N, N] cov = cov_exp_quad(carat, alpha, rho) + diag_matrix(rep_vector(square(sigma), N));
    matrix[N, N] L_cov = cholesky_decompose(cov);

    price ~ multi_normal_cholesky(rep_vector(0, N), L_cov);
}
```

In [ ]:
```
model_gp_opt = CmdStanModel(stan_file='stanfiles/model_gp_opt.stan')
data = dict(N = len(df), carat = df.carat, price = df.price)
result_gp_opt = model_gp_opt.optimize(data=data, algorithm='Newton')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
INFO:cmdstanpy:Chain [1] start processing
INFO:cmdstanpy:Chain [1] done processing
```

In [ ]:
```python
result = result_gp_opt.optimized_params_pd

alpha = float(result['alpha'])
rho = float(result['rho'])
sigma = float(result['sigma'])

print("Optimization result:")
print(f"alpha = {alpha}")
print(f"rho = {rho}")
print(f"sigma = {sigma}")
```

```
Optimization result:
alpha = 130.0
rho = 1.0
sigma = 100.0
```

Model 3 - Posterior analysis

Return to table of contents

After receiving the optimized priors values, we can proceed to the proper analysis.

Sampling errors occurred when the optimized priors values were too big. For this reason, they have an upper limit in the optimization process, determined experimentally.

A full model for GP was created:

```stan
data {
    int<lower=1> N;
    real carat[N];
    real price[N];
    int<lower=1, upper=N> idx[N];

    real<lower=0> alpha;
    real<lower=0> rho;
    real<lower=0> sigma;
}

transformed data {
    matrix[N, N] cov = cov_exp_quad(carat, alpha, rho) + diag_matrix(rep_vector(1e-10, N));
    matrix[N, N] L_cov = cholesky_decompose(cov);
}

parameters {
    vector[N] f_t;
}

transformed parameters {
    vector[N] f = L_cov * f_t;
}

model {
    f_t ~ normal(0,1);
    price ~ normal(f[idx], sigma);
}

generated quantities {
    vector[N] log_lik;
    vector[N] price_sim;
    for (i in 1:N){
        log_lik[i] = normal_lpdf(price[i] | f[i], sigma);
        price_sim[i] = normal_rng(f[i], sigma);
    }
}
```

In [ ]:

```python
model_gp = CmdStanModel(stan_file='stanfiles/model_gp.stan')
```

```
INFO:cmdstanpy:found newer exe file, not recompiling
```

In [ ]:

```python
idx = range(1, len(df.price)+1)
data = dict(N = len(df), carat = df.carat, price = df.price, idx = idx, rho=rho, a
result_gp = model_gp.sample(data=data)
```

```
INFO:cmdstanpy:CmdStan start processing
chain 1 |          | 00:00 Status

chain 1 |          | 00:00 Iteration:    1 / 2000 [  0%]  (Warmup)

chain 1 |          | 00:01 Iteration:  100 / 2000 [  5%]  (Warmup)

chain 1 |          | 00:01 Iteration:  200 / 2000 [ 10%]  (Warmup)

chain 1 |          | 00:01 Iteration:  300 / 2000 [ 15%]  (Warmup)

chain 1 |          | 00:02 Iteration:  400 / 2000 [ 20%]  (Warmup)

chain 1 |          | 00:02 Iteration:  500 / 2000 [ 25%]  (Warmup)
chain 1 |          | 00:03 Iteration:  600 / 2000 [ 30%]  (Warmup)

chain 1 |         | 00:03 Iteration:  700 / 2000 [ 35%]  (Warmup)

chain 1 |         | 00:03 Iteration:  800 / 2000 [ 40%]  (Warmup)

chain 1 |         | 00:04 Iteration:  900 / 2000 [ 45%]  (Warmup)


chain 1 |          | 00:05 Iteration: 1001 / 2000 [ 50%]  (Sampling)

chain 1 |          | 00:05 Iteration: 1100 / 2000 [ 55%]  (Sampling)

chain 1 |          | 00:06 Iteration: 1200 / 2000 [ 60%]  (Sampling)

chain 1 |          | 00:07 Iteration: 1300 / 2000 [ 65%]  (Sampling)

chain 1 |          | 00:07 Iteration: 1400 / 2000 [ 70%]  (Sampling)

chain 1 |          | 00:08 Iteration: 1500 / 2000 [ 75%]  (Sampling)

chain 1 |          | 00:09 Iteration: 1600 / 2000 [ 80%]  (Sampling)

chain 1 |         | 00:09 Iteration: 1700 / 2000 [ 85%]  (Sampling)

chain 1 |         | 00:10 Iteration: 1800 / 2000 [ 90%]  (Sampling)

chain 1 |         | 00:11 Iteration: 1900 / 2000 [ 95%]  (Sampling)
```

```
chain 1 |██████████| 00:11 Sampling completed
chain 2 |██████████| 00:11 Sampling completed
chain 3 |██████████| 00:11 Sampling completed
chain 4 |██████████| 00:11 Sampling completed
```

```
INFO:cmdstanpy:CmdStan done processing.
```

Model 3 - evaluation

Return to table of contents

We can now observe the results.

Model 3 - predictions and density plot

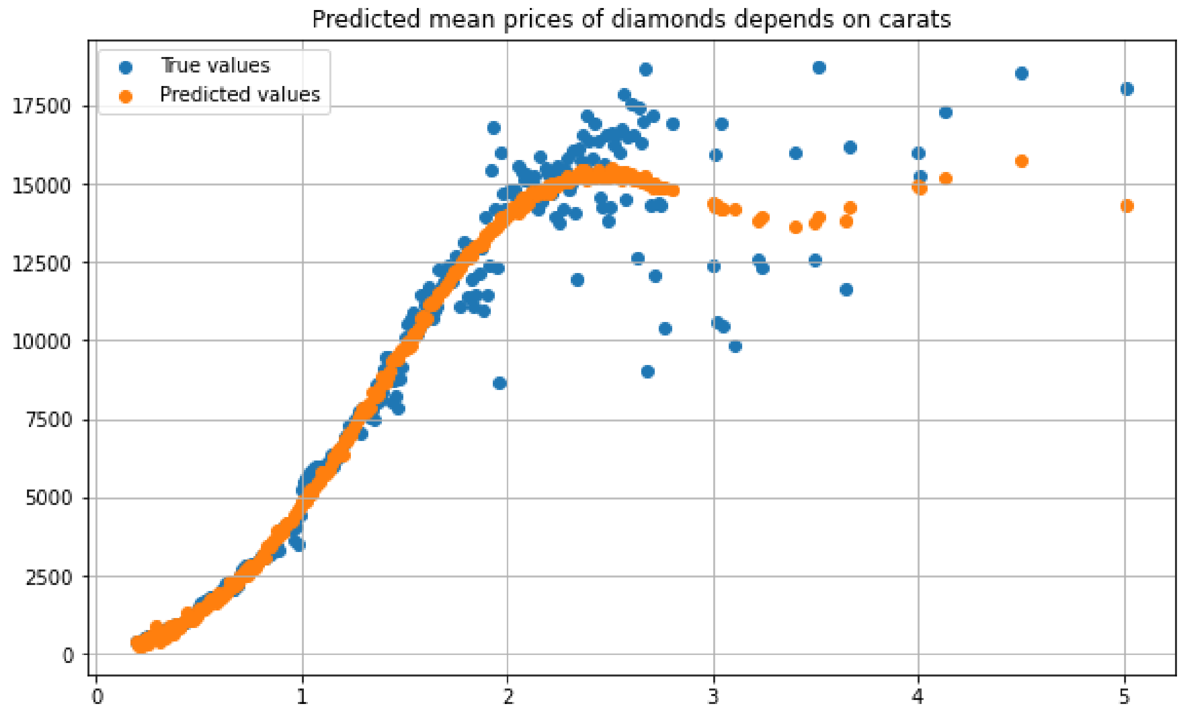
Return to table of contents

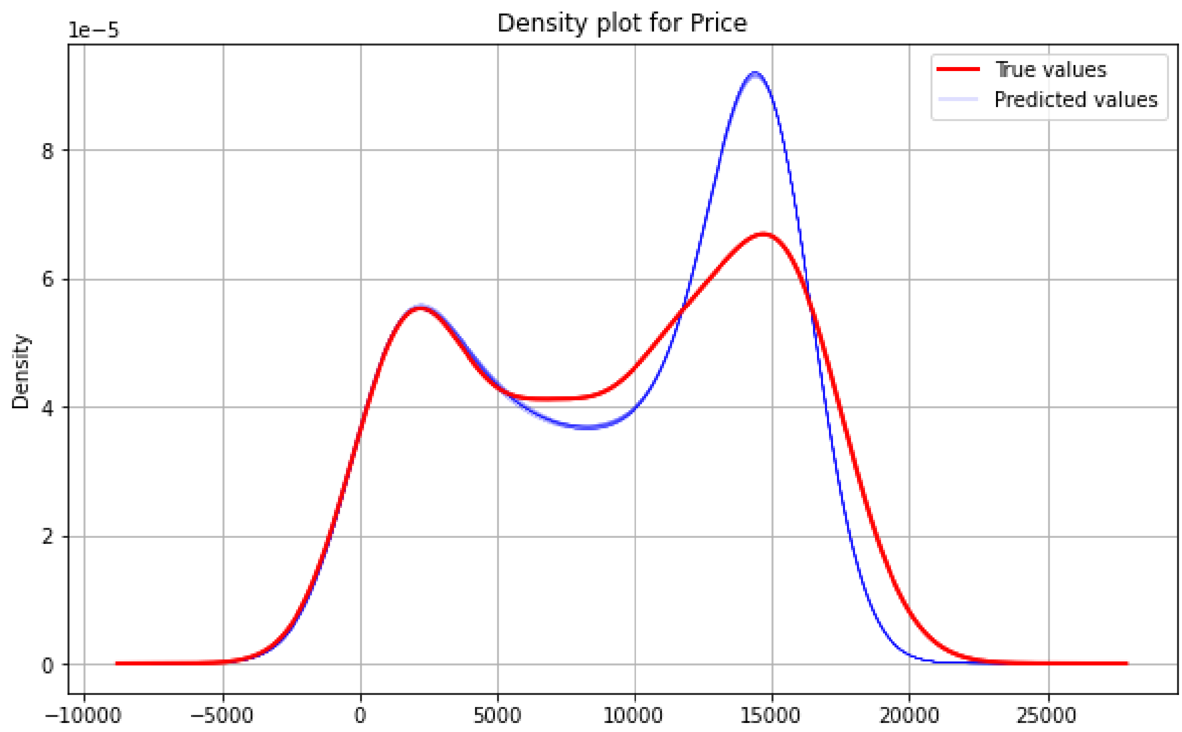
As we can see the model is somwehat sufficient to describe the phenomenon.

It stays fairly accurate up untill 2 carat mark. While it doesn't diverge from the mean for higher weights, standard deviation value is far smaller than in real data.

In [ ]:

```
data = result_gp.draws_pd()
price_sims = data[data.columns[826:]]
price_sim = result_gp.stan_variable('price_sim')

plt.figure(figsize=[10,6])
plt.scatter(df.carat.values, df.price.values)
plt.scatter(df.carat.values, price_sim[0])
plt.title("Predicted mean prices of diamonds depends on carats")
plt.legend(["True values", "Predicted values"])
plt.grid()
plt.show()

df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
for i in range(0,10):
    price_sims.iloc[i].plot.density(linewidth=0.25, color='blue')
df.price.plot.density(figsize=(10,6), linewidth=2, color='red')
plt.title('Density plot for Price')
plt.legend(["True values", "Predicted values"])
plt.grid()
plt.show()
```





Model Comparison

[Return to table of contents](#)

Leave-one-out cross-validation (LOO) and the widely applicable information criterion (WAIC) are methods for estimating pointwise out-of-sample prediction accuracy from a fitted Bayesian model using the log-likelihood evaluated at the posterior simulations of the parameter values. The comparison function used allows the models to be assessed against each of these criteria, ordering them from best to worst.

PSIS-LOO Criterion

Return to table of contents

Based on the comparison of the models using the PSIS-LOO criterion, it can be concluded that:

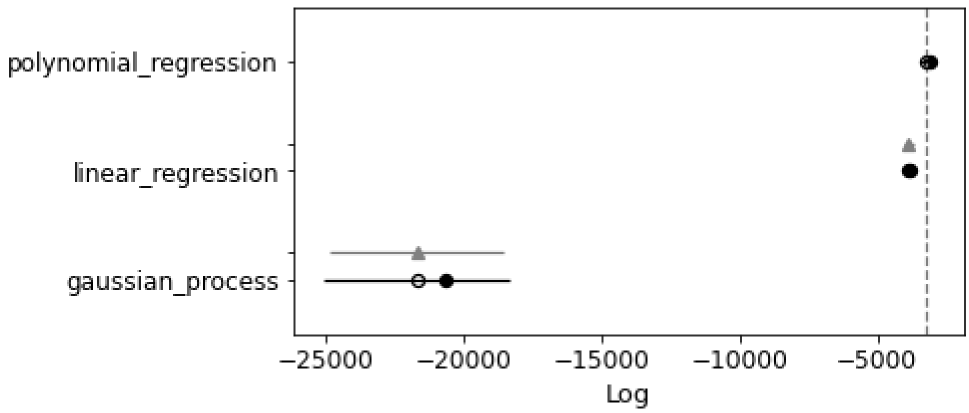
- 4th degree polynomial regression has the lowest rank (which means the best model of all)
- 4th degree polynomial regression has the highest out-of-sample predictive fit ('loo' column), while the gaussian process has the lowest,
- 4th degree polynomial regression has the highest probability of the correctness of the model ('weight' column), while the gaussian process has the lowest,
- standard error of the difference information criteria between each model and the top ranked model ('dse' column) show that the Gaussian process model deviates from the polynomial much more than linear,
- for all models there is a warning that indicates that the computation of the information criteria may not be reliable.

In [ ]:

```
data = dict(linear_regression = result_lr, polynomial_regression = result_pr, gaus
comp_loo = az.compare(data, ic = "loo")
print('\n')
print(comp_loo)
az.plot_compare(comp_loo)
```

```
                       rank           loo        p_loo         d_loo  \
polynomial_regression     0  -3204.942038    92.933615      0.000000
linear_regression         1  -3863.472001    24.108392    658.529963
gaussian_process          2 -21673.934583  1011.936128  18468.992545

                         weight           se          dse  warning loo_scale
polynomial_regression  0.692984   223.471071     0.000000     True       log
linear_regression      0.132521   225.242150   187.238572     True       log
gaussian_process       0.174495  3366.506530  3161.518218     True       log
```

Out[ ]: <AxesSubplot:xlabel='Log'>



WAIC Criterion

Return to table of contents

Based on the comparison of the models using the WAIC criterion, it can be concluded that the conclusions are identical to the previous criterion, i.e.:

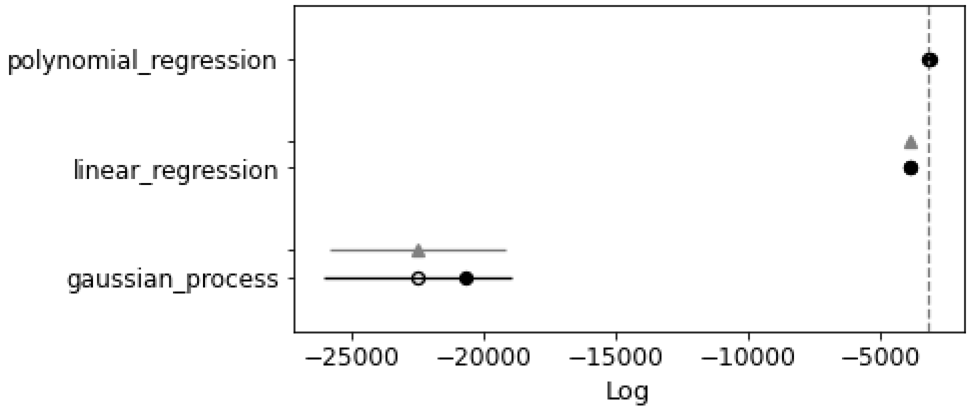
- 4th degree polynomial regression has the lowest rank (which means the best model of all)
- 4th degree polynomial regression has the highest out-of-sample predictive fit ('waic' column), while the gaussian process has the lowest,
- 4th degree polynomial regression has the highest probability of the correctness of the model ('weight' column), while the gaussian process has the lowest,
- standard error of the difference information criteria between each model and the top ranked model ('dse' column) show that the Gaussian process model deviates from the polynomial much more than linear,
- for all models there is a warning that indicates that the computation of the information criteria may not be reliable.

In [ ]:

```
comp_waic = az.compare(data, ic = "waic")
print('\n')
print(comp_waic)
az.plot_compare(comp_waic)
```

```
                       rank          waic       p_waic        d_waic    weight  \
polynomial_regression     0   -3210.76066    98.752237       0.00000  0.692617
linear_regression         1   -3863.52180    24.158191     652.76114  0.132801
gaussian_process          2  -22497.14111  1835.142655   19286.38045  0.174582

                                se          dse  warning waic_scale
polynomial_regression   224.625619     0.000000     True        log
linear_regression       225.264981   185.350005     True        log
gaussian_process       3564.983481  3358.838777     True        log
<AxesSubplot:xlabel='Log'>
```

Out[ ]:



Model Comparison - conclusions

Return to table of contents

Comparing the models, can agree with the results that the polynomial regression model produces the best results. If we are talking about the worst model, on the basis of the obtained price value prediction charts it can be concluded that the Gaussian process coped better with the fitting than the linear regression. This opinion differs from the result of the information criterion. As mentioned earlier, polynomial regression gave a satisfactory result for the given range. For higher carat values, the results could be different, but due to the fact that they are very rare, they were not taken into account in the final evaluation of the correctness of the model.