HW1

Database description

This dataset contains the prices and other attributes of 53940 diamonds which characterizes jewels by their cut, color, clarity, price, and other attributes(10 attributes total). Dataset doesn't contain any missing values.

Content

price: in US dollars
carat: weight of the diamond
cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
color: diamond colour, from J (worst) to D (best)
clarity: a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))
x: length in mm
y: width in mm
z: depth in mm
depth: total depth percentage
table: width of top of diamond relative to widest point

Example of problems:

The dataset can be used to analyze how different attributes affect the total cost of the diamond. Another possible usage of this dataset is diamond clusterization according to their physical features and analysis of how these features correlate with each other.

Dataset source This dataset source is https://www.kaggle.com/shivam2503/diamonds (https://www.kaggle.com/shivam2503/diamonds) .

In [1]:
```
%pylab inline
import pandas as pd
import sklearn
from sklearn.utils import resample
import numpy as np
import copy
from scipy.stats import chi2
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from scipy.stats import pearsonr
from sklearn.metrics import r2_score
from sklearn.decomposition import PCA
from sklearn.preprocessing import minmax_scale, scale
```

```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
full_dataset = pd.read_csv("diamonds.csv", index_col=0)
full_dataset.head(3)
```

Out[2]:

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31

In [3]:
```
dataset = sklearn.utils.resample(full_dataset, n_samples=400, random_state=123
, replace=False)
```

HW-2-I K-means

In [4]:
```
features = ["price", "carat", "depth", "table"]
selected_features = dataset[features]
```

We decided to use price, carat, depth and table because they represent different aspects of each diamond.

In [5]:
```
scaler = StandardScaler().fit(selected_features)
selected_features_normalized = scaler.transform(selected_features)
```

In [6]:
```
kmeans5 = KMeans(n_clusters=5, init="random")
kmeans9 = KMeans(n_clusters=9, init="random")
```

In [7]:
```
best_5_score = 1e9
best_9_score = 1e9

for i in range(10):
    classes_5 = kmeans5.fit_predict(selected_features_normalized)
    classes_9 = kmeans9.fit_predict(selected_features_normalized)
    if kmeans5.inertia_ < best_5_score:
        best_5_score = kmeans5.inertia_
        best_5_classes = classes_5
    if kmeans5.inertia_ < best_9_score:
        best_9_score = kmeans9.inertia_
        best_9_classes = classes_9
```

In [8]:
```
print("Grand Mean: ", ' '.join(["%.2f"% j for j in np.mean(selected_features
)]))
for i in range(0, 5):
    print("CENTROID %d: " % (i + 1), ' '.join(["%.2f" % j for j in scaler.inve
rse_transform(kmeans5.cluster_centers_[i])]))
```

```
Grand Mean:  4217.14 0.84 61.81 57.29
CENTROID 1:  5445.93 1.11 62.54 57.76
CENTROID 2:  13721.54 1.77 61.75 57.28
CENTROID 3:  4164.93 0.96 59.51 60.93
CENTROID 4:  1661.45 0.52 62.51 55.37
CENTROID 5:  1497.39 0.48 61.28 57.72
```

In [9]:
```
print("Grand Mean: ", ' '.join(["%.2f"% j for j in np.mean(selected_features
)]))
for i in range(0, 9):
    print("CENTROID %d: " % (i + 1), ' '.join(["%.2f"% j for j in scaler.inver
se_transform(kmeans9.cluster_centers_[i])]))
```

```
Grand Mean:  4217.14 0.84 61.81 57.29
CENTROID 1:  5006.94 1.08 61.64 60.25
CENTROID 2:  15452.70 2.08 61.88 58.74
CENTROID 3:  11296.59 1.48 61.74 56.31
CENTROID 4:  1437.16 0.48 62.08 55.15
CENTROID 5:  4177.62 1.02 58.04 62.85
CENTROID 6:  3538.88 1.12 66.85 56.75
CENTROID 7:  1083.97 0.40 62.23 57.78
CENTROID 8:  2287.24 0.63 60.37 58.04
CENTROID 9:  4733.15 1.01 62.51 56.59
```

HW-2-II Bootstrap

Lets choose cluster 7 for futher analysis. It contains most cheap diamonds. As an interesing feature lets select depth.

In [10]:
```
def bootstrap(data, iterations, confidence, func, pivotal=False):
    func_vals = []
    confidence_margin = (100-confidence)/2
    data_func = func(data)
    for i in range(iterations):
        resampled = sklearn.utils.resample(data)
        func_vals.append(func(resampled))
    if not pivotal:
        return np.percentile(func_vals, confidence_margin), np.percentile(func
_vals, 100-confidence_margin)
    else:
        return 2 * data_func - np.percentile(func_vals, 100 - confidence_margi
n), 2 * data_func - np.percentile(func_vals, confidence_margin)
```

In [14]:
```
cluster_samples = selected_features[classes_9==6]
cluster_samples.shape
```

Out[14]:
```
(65, 4)
```

In [15]:
```
print(
    "95% non-pivotal confidence interval for selected_cluster:",
    bootstrap(cluster_samples["depth"], 10000, 95, np.mean)
)
print(
    "95% pivotal confidence interval for selected_cluster:",
    bootstrap(cluster_samples["depth"], 10000, 95, np.mean, pivotal=True)
)
```

```
95% non-pivotal confidence interval for selected_cluster: (62.06153846153846, 62.410769230769226)
95% pivotal confidence interval for selected_cluster: (62.058461538461536, 62.413884615384625)
```

In [16]:
```
grand_mean = 61.81
print(
    "95% non-pivotal confidence interval for selected_cluster:",
    bootstrap(cluster_samples["depth"] - grand_mean, 10000, 95, np.mean)
)
print(
    "95% pivotal confidence interval for selected_cluster:",
    bootstrap(cluster_samples["depth"] - grand_mean, 10000, 95, np.mean, pivotal=True)
)
```

```
95% non-pivotal confidence interval for selected_cluster: (0.24538461538461334, 0.6023076923076901)
95% pivotal confidence interval for selected_cluster: (0.24384615384615205, 0.6023076923076897)
```

Confidence interval of difference of cluster mean and grand mean greater then 0.
That means that average depth in this cluster is greater than grand_mean with 95% confidence.

HW3 Contingency Table

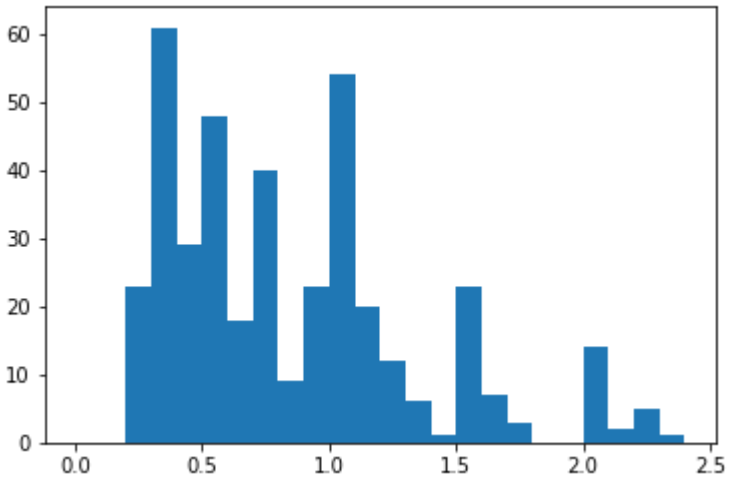
Features: cut, carat, price

In [17]:
```
def new_feature(f, points):
    x = np.zeros_like(f)
    for i, p in enumerate(points):
        x[f >= p] = i + 1
    points = [np.min(f)] + points + [np.max(f)]
    print("border points: ", points)
    return x

def cut_feature(f):
    x = np.zeros_like(f) # Fair
    x[f == 'Good'] = 1
    x[f == 'Very Good'] = 2
    x[f == 'Premium'] = 3
    x[f == 'Ideal'] = 4
    return x
```

In [18]:
```
plt.hist(dataset['carat'], bins=list(np.arange(0.0, 2.5, 0.1)))
```

Out[18]:
```
(array([ 0.,  0., 23., 61., 29., 48., 18., 40.,  9., 23., 54., 20., 12.,
         6.,  1., 23.,  7.,  3.,  0.,  0., 14.,  2.,  5.,  1.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
        1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4]),
 <a list of 24 Patch objects>)
```

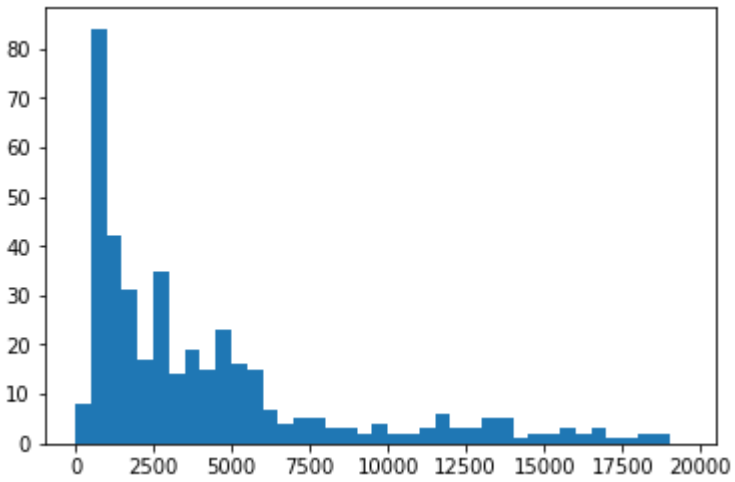


In [19]:
```
nominal_carat = new_feature(dataset['carat'], [0.5, 0.9, 1.4])
```

```
border points:  [0.24, 0.5, 0.9, 1.4, 2.48]
```

In [20]:
```
plt.hist(dataset['price'], bins=list(np.arange(0.0, 20000, 500)))
```

Out[20]:
```
(array([ 8., 84., 42., 31., 17., 35., 14., 19., 15., 23., 16., 15.,  7.,
         4.,  5.,  5.,  3.,  3.,  2.,  4.,  2.,  2.,  3.,  6.,  3.,  3.,
         5.,  5.,  1.,  2.,  2.,  3.,  2.,  3.,  1.,  1.,  2.,  2.,  0.]),
 array([    0.,   500.,  1000.,  1500.,  2000.,  2500.,  3000.,  3500.,
         4000.,  4500.,  5000.,  5500.,  6000.,  6500.,  7000.,  7500.,
         8000.,  8500.,  9000.,  9500., 10000., 10500., 11000., 11500.,
        12000., 12500., 13000., 13500., 14000., 14500., 15000., 15500.,
        16000., 16500., 17000., 17500., 18000., 18500., 19000., 19500.]),
 <a list of 39 Patch objects>)
```



In [21]:
```
nominal_price = new_feature(dataset['price'], [2400, 6000, 11000])
```

```
border points:  [418, 2400, 6000, 11000, 18781]
```

In [22]:
```
nominal_cut = cut_feature(dataset['cut'])
```

In [23]:
```
def table_(f1, f2):
    bin1 = len(np.unique(f1))
    bin2 = len(np.unique(f2))
    res = np.zeros((bin1, bin2))
    for i in range(len(f1)):
        res[int(f1[i]), int(f2[i])] += 1
    return res

def table(f1, f2, name1, name2):
    res = table_(f1, f2)
    array = list([0]*(res.shape[1] + 2) for i in range(res.shape[0] + 2))
    for i in range(res.shape[0]):
        for j in range(res.shape[1]):
            array[i + 1][j + 1] = res[i][j]
    array[0][0] = name1 + ' \ ' + name2
    for i in range(res.shape[0]):
        array[i + 1][0] = name1 + str(i)
        array[i + 1][-1] = np.sum(res[i, :])
    for i in range(res.shape[1]):
        array[0][i + 1] = name2 + str(i)
        array[-1][i + 1] = np.sum(res[:, i])
    array[0][-1] = "Total"
    array[-1][0] = "Total"
    array[-1][-1] = np.sum(res)
    return pd.DataFrame(data=array)

def probability(f1, f2, name1, name2):
    data = table(f1, f2, name1, name2)
    for i in range(1, data.shape[1]):
        data.iloc[1:-1, i] /= np.sum(data.iloc[1:-1, i])
    data.iloc[0, -1] = "Freq"
    return data

def quetelet(f1, f2, name1, name2):
    data = probability(f1, f2, name1, name2)
    for i in range(1, data.shape[1] - 1):
        x = np.sum(data.iloc[1:-1, i])
        data.iloc[1:-1, i] =  data.iloc[1:-1, i] / data.iloc[1:-1, -1] - 1
    data.iloc[0, 0] = "Quetelet"
    return data

def chi_squared(f1, f2, name1, name2):
    data = table(f1, f2, name1, name2)
    data.iloc[1:, 1:] /= data.iloc[-1, -1]#np.sum(data.iloc[1:-1, 1:-1].value
s)
    freq = copy.deepcopy(data)
    for i in range(1, freq.shape[0]):
        for j in range(1, freq.shape[1]):
            freq.iloc[i,j] = data.iloc[i,-1] * data.iloc[-1,j]
    res = np.sum((((data.iloc[1:-1, 1:-1] - freq.iloc[1:-1, 1:-1])**2) / freq.
iloc[1:-1,1:-1]).values)
    return res
```

In [24]:
```
table(nominal_price, nominal_cut, "price", "cut")
```

Out[24]:

	0	1	2	3	4	5	6
0	price \ cut	cut0	cut1	cut2	cut3	cut4	Total
1	price0	4	12	31	40	91	178
2	price1	7	18	28	39	49	141
3	price2	3	2	7	9	16	37
4	price3	0	3	8	17	16	44
5	Total	14	35	74	105	172	400

In [25]:
```
table(nominal_price, nominal_carat, "price", "carat")
```

Out[25]:

	0	1	2	3	4	5
0	price \ carat	carat0	carat1	carat2	carat3	Total
1	price0	113	64	1	0	178
2	price1	0	51	88	2	141
3	price2	0	0	20	17	37
4	price3	0	0	6	38	44
5	Total	113	115	115	57	400

In [26]:
```
probability(nominal_price, nominal_cut, "price", "cut")
```

Out[26]:

	0	1	2	3	4	5	6
0	price \ cut	cut0	cut1	cut2	cut3	cut4	Freq
1	price0	0.285714	0.342857	0.418919	0.380952	0.52907	0.445
2	price1	0.5	0.514286	0.378378	0.371429	0.284884	0.3525
3	price2	0.214286	0.0571429	0.0945946	0.0857143	0.0930233	0.0925
4	price3	0	0.0857143	0.108108	0.161905	0.0930233	0.11
5	Total	14	35	74	105	172	400

In [27]:
```
probability(nominal_price, nominal_carat, "price", "carat")
```

Out[27]:

	0	1	2	3	4	5
0	price \ carat	carat0	carat1	carat2	carat3	Freq
1	price0	1	0.556522	0.00869565	0	0.445
2	price1	0	0.443478	0.765217	0.0350877	0.3525
3	price2	0	0	0.173913	0.298246	0.0925
4	price3	0	0	0.0521739	0.666667	0.11
5	Total	113	115	115	57	400

Conditional frequency tables show distribution of the second feature over the first feature. For instance, 50% of all cut0 are price1 and none of them are price3.

In [28]:
```
quetelet(nominal_price, nominal_cut, "price", "cut")
```

Out[28]:

	0	1	2	3	4	5	6
0	Quetelet	cut0	cut1	cut2	cut3	cut4	Freq
1	price0	-0.357945	-0.229535	-0.0586092	-0.143927	0.188921	0.445
2	price1	0.41844	0.458967	0.0734138	0.0536981	-0.191819	0.3525
3	price2	1.3166	-0.382239	0.0226443	-0.0733591	0.00565682	0.0925
4	price3	-1	-0.220779	-0.017199	0.471861	-0.154334	0.11
5	Total	14	35	74	105	172	400

In [29]:
```
quetelet(nominal_price, nominal_carat, "price", "carat")
```

Out[29]:

	0	1	2	3	4	5
0	Quetelet	carat0	carat1	carat2	carat3	Freq
1	price0	1.24719	0.250611	-0.980459	-1	0.445
2	price1	-1	0.258094	1.17083	-0.90046	0.3525
3	price2	-1	-1	0.880141	2.22428	0.0925
4	price3	-1	-1	-0.525692	5.06061	0.11
5	Total	113	115	115	57	400

We can get some information from this Quetelet indices table. For example: price2, given carat2 is 88% more frequent, than on average, but price2, given cut1, is 38% less frequent, than on average. These relations we could not see using just conditional probabilities tables.

In [30]:
```
chi_squared(nominal_price, nominal_cut, "price", "cut")
```

Out[30]:
```
0.04724814512885877
```

In [31]:
```
print(chi2.ppf(0.95, 12))
print(400 * 0.04724814512885877)
chi2.ppf(0.95, 12) < 400 * 0.04724814512885877
```

```
21.02606981748307
18.89925805154351
```

Out[31]:
```
False
```

In [32]:
```
chi2.ppf(0.95, 12) < 445 * 0.04724814512885877
```

Out[32]:
```
False
```

In [33]:
```
print(chi2.ppf(0.99, 12))
print(400 * 0.04724814512885877)
chi2.ppf(0.99, 12) < 400 * 0.04724814512885877
```

```
26.216967305535853
18.89925805154351
```

Out[33]:
```
False
```

In [34]:
```
chi2.ppf(0.99, 9) < 458 * 0.04724814512885877
```

Out[34]:
```
False
```

We cannot reject hypothesis that cut and price are independent. If we had more then 445 and 458 samples, we could reject with 95% and 98% confidence respectively

In [35]:
```
chi_squared(nominal_price, nominal_carat, "price", "carat")
```

Out[35]:
```
1.2873762500391155
```

In [36]:
```
print(chi2.ppf(0.95, 9))
print(400 * 1.2873762500391155)
chi2.ppf(0.95, 9) < 400 * 1.2873762500391155
```

```
16.918977604620448
514.9505000156462
```

Out[36]:
```
True
```

In [37]:
```
chi2.ppf(0.95, 9) < 13 * 1.2873762500391155
```

Out[37]:
```
False
```

In [38]:
```
print(chi2.ppf(0.99, 9))
print(400 * 1.2873762500391155)
chi2.ppf(0.99, 9) < 400 * 1.2873762500391155
```

```
21.665994333461924
514.9505000156462
```

Out[38]: True

In [39]:
```
chi2.ppf(0.99, 9) < 16 * 1.2873762500391155
```

Out[39]: False

We reject eject hypothesis that cut and price are independent. However, we rejected it if we had less then 14 and 17 samples with 95% and 98% confidence respectively.

HW4 PCA/SVD

In [77]:
```
selected_features = dataset[['x', 'y', 'z', 'price']]
```

In [78]:
```
scaler = StandardScaler().fit(selected_features)
selected_features_normalized = scaler.transform(selected_features)
data_scatter = np.sum(selected_features_normalized*selected_features_normalized)
```

In [79]:
```
pca = PCA(n_components=4)
components = pca.fit_transform(selected_features_normalized)
for i in range(4):
    contribution = np.sum(components[:, i] * components[:, i])
    total = contribution/data_scatter
    print("COMPONENT %d contributon: value %.2f Percent %.4f" % (i+1,contribution, contribution/data_scatter), '%')
```

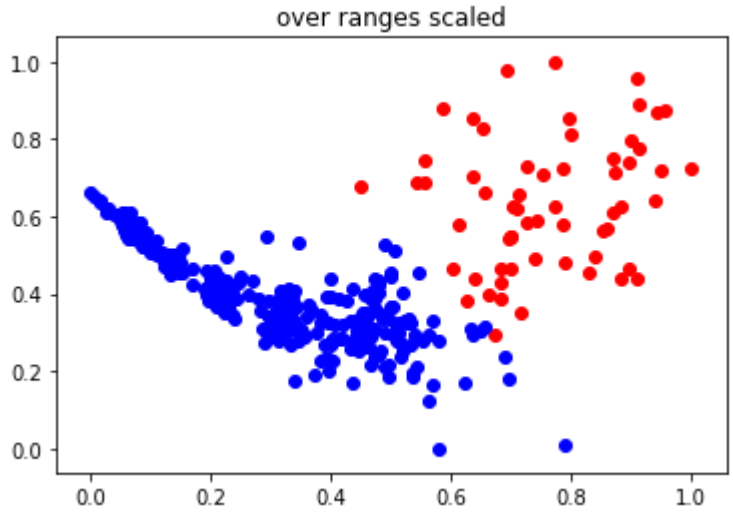
```
COMPONENT 1 contributon: value 1525.24 Percent 0.9533 %
COMPONENT 2 contributon: value 69.62 Percent 0.0435 %
COMPONENT 3 contributon: value 4.73 Percent 0.0030 %
COMPONENT 4 contributon: value 0.41 Percent 0.0003 %
```

Lets chouse diamonds with price > 8000 to visualize with red color.

In [80]:
```
mask = dataset['price'] > 8000
```

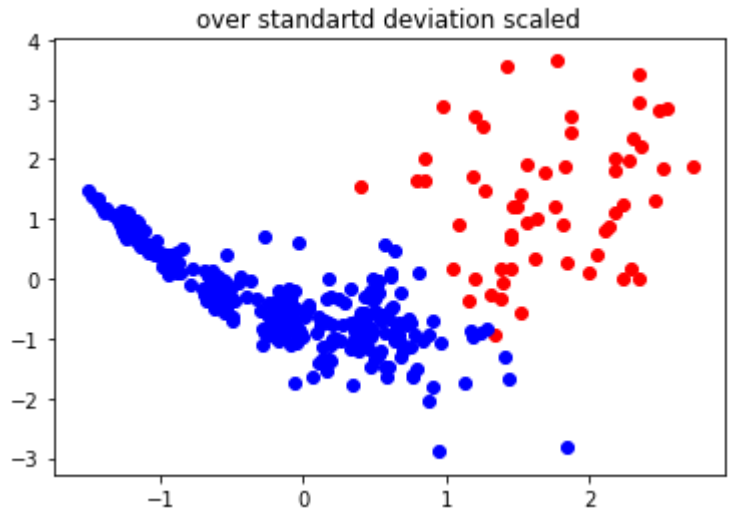
In [81]:
```
plt.title("over ranges scaled")
minmax_comp = minmax_scale(components)
plt.scatter(minmax_comp[:, 0][mask], minmax_comp[:, 1][mask], c='r')
plt.scatter(minmax_comp[:, 0][~mask], minmax_comp[:, 1][~mask], c='b')
```

Out[81]: <matplotlib.collections.PathCollection at 0x7f8d73cf2978>



In [82]:
```
plt.title("over standartd deviation scaled")
scaled_comp = scale(components)
plt.scatter(scaled_comp[:, 0][mask], scaled_comp[:, 1][mask], c='r')
plt.scatter(scaled_comp[:, 0][~mask], scaled_comp[:, 1][~mask], c='b')
```

Out[82]: <matplotlib.collections.PathCollection at 0x7f8d73c4a908>

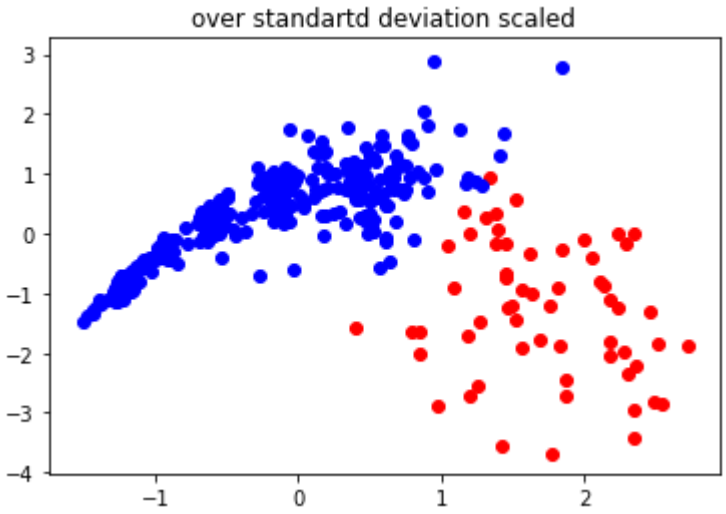


Standart deviation and over ranges scales of pca features looks similar. That means it is not important which scaler to choose.

In [83]:
```
def ConventionalPCA(X):
    N = X.shape[0]
    B = X.T.dot(X)/N
    components = []
    for i in range(0, X.shape[1]):
        vals, vecs = numpy.linalg.eig(B)
        val, vec = vals[np.argmax(vals)], vecs[:, np.argmax(vals)]
        component = X.dot(vec)
        components.append(component)
        B = B - val * np.outer(vec, vec)
    return np.array(components).T
res = ConventionalPCA(selected_features_normalized)
```

In [84]:
```
plt.title("over standartd deviation scaled")
scaled_comp = scale(res)
plt.scatter(scaled_comp[:, 0][mask], scaled_comp[:, 1][mask], c='r')
plt.scatter(scaled_comp[:, 0][~mask], scaled_comp[:, 1][~mask], c='b')
```

Out[84]: <matplotlib.collections.PathCollection at 0x7f8d73c156d8>



Result for convential PCA accurate to the inversion over y axis.

In [87]:
```
features = minmax_scale(selected_features, (0, 100))
_, _, v = np.linalg.svd(features)
factor = v[:,0]/np.sum(v[:,0])
```

In [88]:
```
features = minmax_scale(selected_features, (0, 100))
_, vals, v = np.linalg.svd(features)
factor_mul = v[0]/np.sum(v[0])
hidden_factor = features.dot(factor_mul)
for hidden_factor_val, feature in zip(hidden_factor[::66], range(0, len(hidden
_factor), 66)):
    print("Hidden factor value %d \n" % hidden_factor_val, selected_features.i
loc[feature])
    print("\n")
```

```
Hidden factor value 6
 x          4.32
y          4.28
z          2.71
price    802.00
Name: 32686, dtype: float64


Hidden factor value 2
 x          4.13
y          4.16
z          2.53
price    545.00
Name: 51970, dtype: float64


Hidden factor value 6
 x          4.33
y          4.38
z          2.70
price    489.00
Name: 38950, dtype: float64


Hidden factor value 44
 x           6.01
y           6.10
z           3.81
price    8239.00
Name: 19623, dtype: float64


Hidden factor value 52
 x           6.79
y           6.74
z           4.03
price    5544.00
Name: 13530, dtype: float64


Hidden factor value 34
 x           5.79
y           5.81
z           3.60
price    2952.00
Name: 1294, dtype: float64


Hidden factor value 6
 x          4.31
y          4.34
z          2.67
price    680.00
Name: 28760, dtype: float64
```

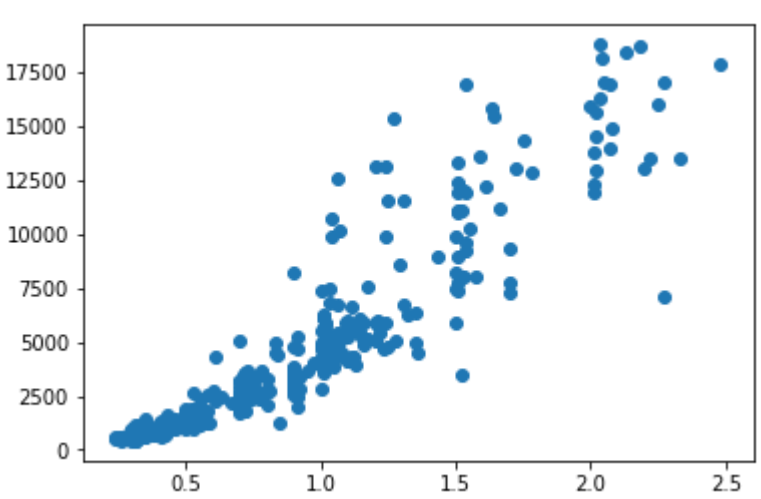
Hidden factor can be interpreted as size/price of diamonds. Both of them can be accepted as the interpretation because they are highly correlated with each other.

HW5 Linear regression

Lets take `carat` and `price` features

In [89]:
```
plt.scatter(dataset['carat'], dataset['price'])
```

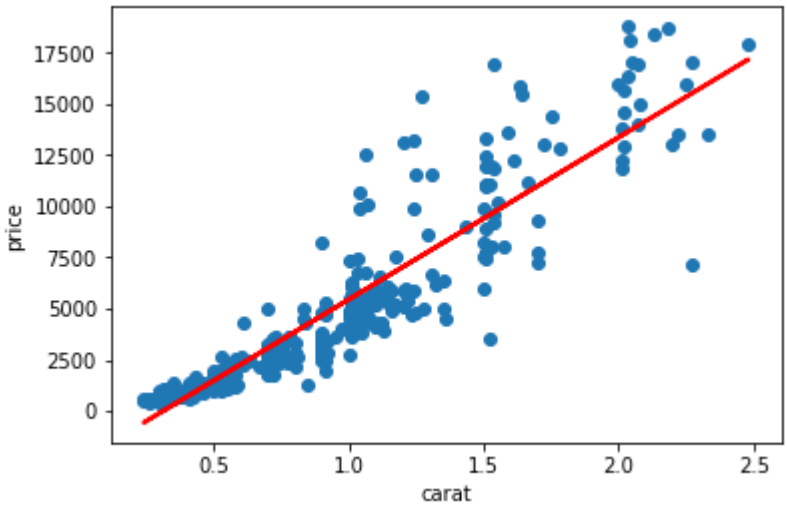
Out[89]:
```
<matplotlib.collections.PathCollection at 0x7f8d73bfcac8>
```



In [90]:
```
lr = LinearRegression()
lr = lr.fit(dataset[["carat"]].values, dataset["price"].values)
preds = lr.predict(dataset[["carat"]].values)
```

In [91]:
```
plt.scatter(dataset['carat'], dataset['price'])
plt.plot(dataset["carat"], preds, c='red', linewidth=2)
plt.xlabel("carat")
plt.ylabel("price")
```

Out[91]: Text(0,0.5,'price')



In [100]:
```
print("Slope: %.2f" % lr.coef_[0])
```

```
Slope: 7922.52
```

Slope is about 7900. It means that price of additional carat is about 7900$.

In [93]:
```
pearson_corr = pearsonr(dataset.carat.values, dataset.price.values)[0]
print("correlation score: %f" % pearson_corr)
```

```
correlation score: 0.920410
```

Correlation score about 0.92 means that carats and prices are highly correlated and in most cases higher carats leads to higher price.

In [94]:
```
r2_score_val = r2_score(dataset.price.values, preds)
print("determinacy score: %f" % r2_score_val)
```

```
determinacy score: 0.847155
```

Determinacy score about 0.85 means that model explains 85% of variability of the response.

In [95]:
```
print ("result for 3 carat:", lr.predict(3)[0])
print ("result for 0.5 carat:", lr.predict(0.5)[0])
print ("result for 0 carat:", lr.predict(0)[0])
```

```
result for 3 carat: 21296.71134924944
result for 0.5 carat: 1490.4030342778633
result for 0 carat: -2470.8586287164517
```

Model behaves badly on small(about 0) carats of diamonds. For small carats it can predict negative price that couldn't be possible. For other points predictions seams reasonable.

In [96]:
```
def compute_rae(y_true, y_pred):
    y_mean = np.mean(y_true)
    return np.sum(np.abs(y_true-y_pred))/np.sum(np.abs(y_true-y_mean))
rae_score_val = compute_rae(dataset.price.values, preds)
```

In [97]:
```
print("determinacy score: %f" % r2_score_val)
print("mean relative absolute error: %f" % rae_score_val)
```

```
determinacy score: 0.847155
mean relative absolute error: 0.352223
```

Optimal mean relative absolute error is 0 and optimal determinacy score is 1. They measures similar things, but determinacy coefficient uses l_2 norm and MRAE uses l_1 norm. If we change l_1 norm to l_2 in MRAE we will get 1 - determinacy coefficient. So usually higher result of determinacy coefficient will lead to lower of mean relative absolute error.