In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model as lm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
import warnings #this is to turn off DataConversionWarnings from normalizing in Sklearn
from sklearn.exceptions import DataConversionWarning
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
```

1. Look at the data

In [2]:
```
data = pd.read_csv("predictor_data_set.csv")
```

In [3]:
```
data.shape
```

Out[3]:
```
(300, 7)
```

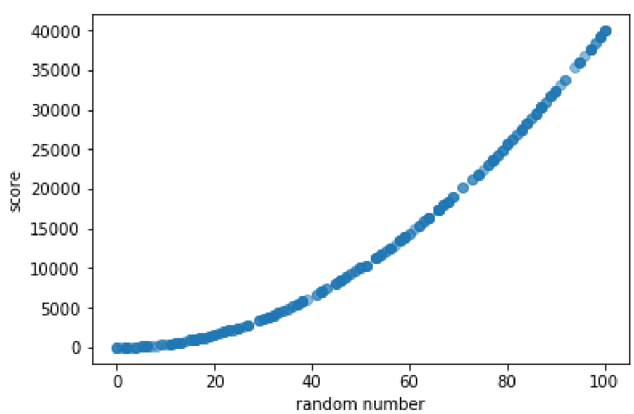
In [4]:
```
data.head()
```

Out[4]:

	index	random_num	times_two	square_that	subtract_4	score	above_sixty_sixth_percentile
0	1	11	22	484	480	480	0
1	2	31	62	3844	3840	3840	0
2	3	20	40	1600	1596	1596	0
3	4	45	90	8100	8096	8096	0
4	5	42	84	7056	7052	7052	0

In [5]:
```
#first step is to plot the data to see what it looks like
#the indepent variable is random_num
#the dependent variable is score
```

In [6]:
```
x = data.random_num
y = data.score
plt.scatter(x, y, alpha=0.5)
plt.xlabel("random number")
plt.ylabel("score")
plt.show()
```



2. The data is not linear, but I am going to start trying to fit a line to it just as practice.

In [7]:
```
#looks sort of exponential, but def not linear
```

In [8]:
```
#even though we know it's not linear...let's just see how a linear
#regression might look
```

In [9]:
```
x = x.values.reshape(-1,1) #to fit the shape needed for the regression #sometimes 'values' is n
ot needed
lin_mod = lm.LinearRegression()
x_train, x_test, y_train, y_test = train_test_split(x,y,train_size=0.8, test_size = 0.2, random
_state = 7)
lin_mod.fit(x_train, y_train)
print("Coef is {}".format(lin_mod.coef_))
print("Intercept is {}".format(lin_mod.intercept_))
```

```
Coef is [397.03727805]
Intercept is -6436.084918281134
```

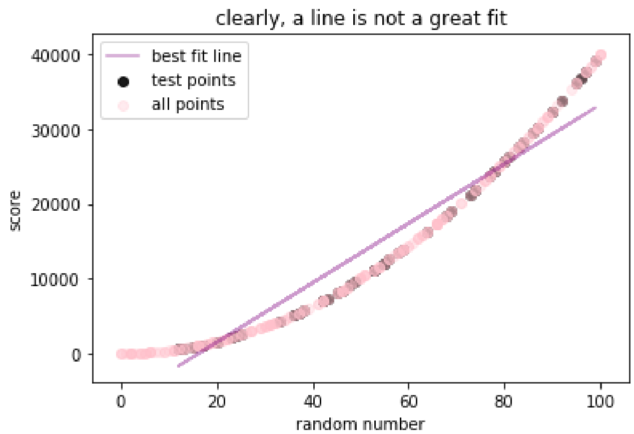
In [10]:
```
#credit to David Ziganto for these metrics (adjusted so it returns RMSE)
def calc_train_error(X_train, y_train, model):
    '''returns in-sample error for already fit model.'''
    predictions = model.predict(X_train)
    mse = mean_squared_error(y_train, predictions)
    rmse = np.sqrt(mse)
    return rmse

def calc_validation_error(X_test, y_test, model):
    '''returns out-of-sample error for already fit model.'''
    predictions = model.predict(X_test)
    mse = mean_squared_error(y_test, predictions)
    rmse = np.sqrt(mse)
    return rmse

def calc_metrics(X_train, y_train, X_test, y_test, model):
    '''fits model and returns the RMSE for in-sample error and out-of-sample error'''
    model.fit(X_train, y_train)
    train_error = calc_train_error(X_train, y_train, model)
    validation_error = calc_validation_error(X_test, y_test, model)
    return train_error, validation_error
```

2-b. Using the above RMSE code from David Ziganto ([https://dziganto.github.io/ (https://dziganto.github.io/)](https://dziganto.github.io/)), we can begin to look at how the linear regression is performing. The R^2 makes sense since score is a function of the random number, but we can see that the RMSE error is too big.

In [11]:
```
train_error, test_error = calc_metrics(x_train, y_train, x_test, y_test, lin_mod)
print('train RMSE error: {} | test RMSE error: {}'.format((round(train_error,1)), round(test_er
ror,1)))
#print('train error: {} | test error: {}'.format(train_error, test_error))
print('train/test: {}'.format(round(test_error/train_error, 2)))
#our model looks like it's underfitting or not having enough complexity if it performs bettter
 on the test then the train
print('R^2: {}'.format(round(lin_mod.score(x_test, y_test),3)))
#we do expect the R^2 to be high because it all depends on one feature
```

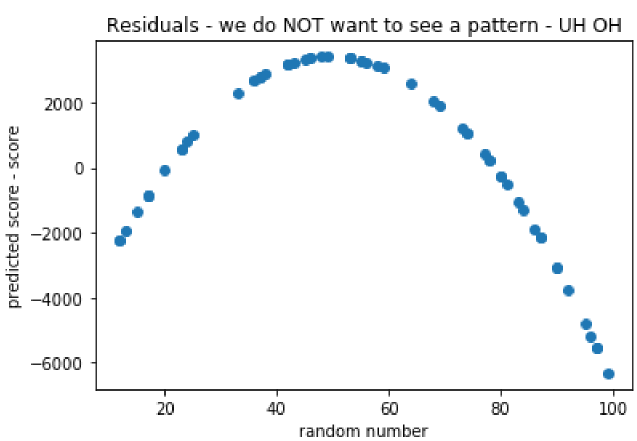
```
train RMSE error: 3072.2 | test RMSE error: 2791.5
train/test: 0.91
R^2: 0.945
```

In [12]:
```
predictions_1 = lin_mod.predict(x_test)
plt.plot(x_test, predictions_1, alpha=0.4, color="purple")
plt.scatter(x_test, y_test, alpha=0.9, color='black')
plt.scatter(x, y, alpha=0.3, color = "pink")
legend_labels = ["best fit line","test points","all points"]
plt.legend(legend_labels)
plt.title("clearly, a line is not a great fit")
plt.xlabel("random number")
plt.ylabel("score")
plt.show()
```



2-c. The residuals are not only showing a pattern, but they are also way too big. The model is not a good fit.

In [13]:
```
plt.scatter(x_test,(lin_mod.predict(x_test)-y_test))
plt.title("Residuals - we do NOT want to see a pattern - UH OH")
plt.xlabel("random number")
plt.ylabel("predicted score - score")
plt.show()
#this shoes the model is quite bad; there should not be a pattern seen in the residuals
#it systematically gets worse, as it gets bigger, and this makes sense b/c we are trying to map
a nonlinear function onto a linear regression
```



2-d. Now we look at the Cross-Validation score. It doesn't seem to have an RMSE capability, so I look to the Mean Absolute Error, which is reasonably close. Having run this model, effectively, 10x over here (cv=10), we still see that it has too much of an error. This makes sense because we are trying to fit a line to a curve.

In [14]:
```
#last let's look at the cross validation
print("Cross-Validation Scoring")
print('Mean Absolute Error: {}'.format(-1*round(cross_val_score(lm.LinearRegression(), x, y, cv
=10, scoring='neg_mean_absolute_error').mean(),2)))
#no option for RMSE in cross val score, so looking at mean absolute error
#multiply by -1 to get a positve value to take a look at it
print('R^2: {}'.format(round(cross_val_score(lm.LinearRegression(), x, y, cv=10, scoring='r2').
mean(),2)))
```

```
Cross-Validation Scoring
Mean Absolute Error: 2578.86
R^2: 0.93
```

In [15]:
```
#Now we know this did not work b/c we were trying to fit a linear model to curve!!!!!!
```

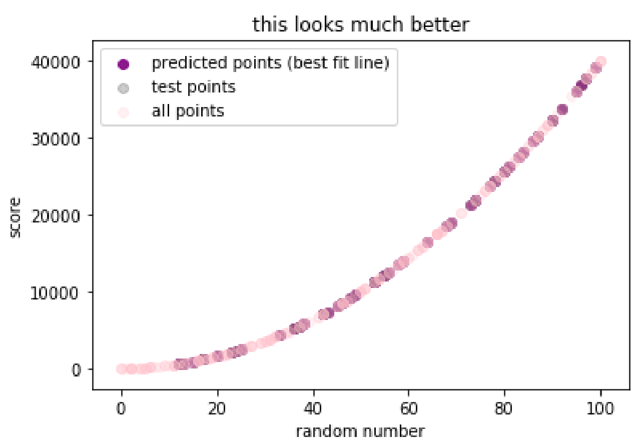
3. Let's try to fit a curve to a curve!!!!

In [16]:
```
#Let's now try to set a more sensible model.
#Let's use a support vector regression
```

In [17]:
```
svr_poly = SVR(kernel='poly', C=1e3, degree=2, gamma='auto')
y_poly = svr_poly.fit(x_train, y_train).predict(x_test)
```

3-a. It's already looking better!

In [18]:
```
predictions_1 = lin_mod.predict(x_test)
plt.scatter(x_test, y_poly, alpha=0.9, color="purple")
plt.scatter(x_test, y_test, alpha=0.2, color='black')
plt.scatter(x, y, alpha=0.2, color = "pink")
legend_labels = ["predicted points (best fit line)","test points","all points"]
plt.legend(legend_labels)
plt.title("this looks much better")
plt.xlabel("random number")
plt.ylabel("score")
plt.show()
```

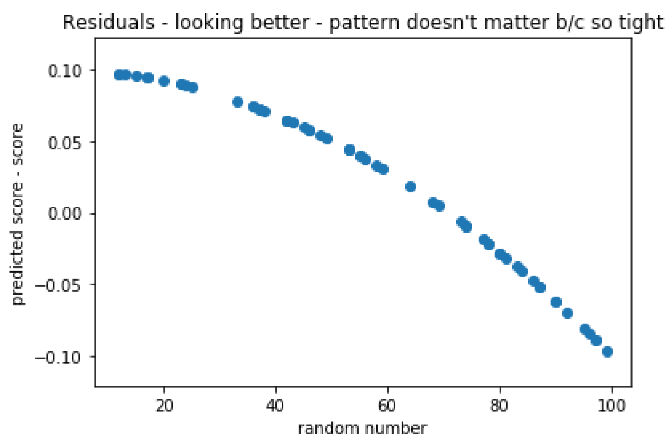


In [19]:
```
train_error, test_error = calc_metrics(x_train, y_train, x_test, y_test, svr_poly)
print('train RMSE error: {} | test RMSE error: {}'.format((round(train_error,1)), round(test_er
ror,1)))
#print('train error: {} | test error: {}'.format(train_error, test_error))
print('train/test: {}'.format(round(test_error/train_error, 2)))
#our model looks like it's underfitting or not having enough complexity if it performs bettter
 on the test then the train
print('R^2: {}'.format(round(lin_mod.score(x_test, y_test),3)))
#we do expect the R^2 to be high because it all depends on one feature
```

```
train RMSE error: 0.1 | test RMSE error: 0.1
train/test: 0.94
R^2: 0.945
```

3-b. The residuals still show a pattern, but now you can see that they really are only looking like that because they are all so accurate. On the scale of the data, are these residucals are bunched togethre in one small place. They are ranging from between 0.1 and -0.1. It looks like the model has figured out my scheme.

In [20]:
```
plt.scatter(x_test,(y_poly-y_test))
plt.title("Residuals - looking better - pattern doesn't matter b/c so tight")
plt.xlabel("random number")
plt.ylabel("predicted score - score")
plt.show()
```



In [21]:
```
#last let's look at the cross validation
print("Cross-Validation Scoring")
print('Mean Absolute Error: {}'.format(-1*round(cross_val_score(svr_poly, x, y, cv=10, scoring=
'neg_mean_absolute_error').mean(),2)))
#no option for RMSE in cross val score, so looking at mean absolute error
#multiply by -1 to get a positve value to take a look at it
print('R^2: {}'.format(round(cross_val_score(svr_poly, x, y, cv=10, scoring='r2').mean(),2)))
```

```
Cross-Validation Scoring
Mean Absolute Error: 0.06
R^2: 1.0
```

In [22]:
```
#Let's just take a look at how strong this model is
print(svr_poly.fit(x_train, y_train).predict([[83]]))
#answer is 27552
```

```
[27551.96222]
```

In [23]:
```
data.head()
```

Out[23]:

	index	random_num	times_two	square_that	subtract_4	score	above_sixty_sixth_percentile
0	1	11	22	484	480	480	0
1	2	31	62	3844	3840	3840	0
2	3	20	40	1600	1596	1596	0
3	4	45	90	8100	8096	8096	0
4	5	42	84	7056	7052	7052	0

In [24]:
```
#spot checking from the set
print(svr_poly.fit(x_train, y_train).predict([[11],[31],[20],[45],[42]]).reshape(-1,1))
```

```
[[ 480.09758]
 [3840.08078]
 [1596.092  ]
 [8096.0595 ]
 [7052.06472]]
```

In [25]:
```
#using totally new numbers
print(svr_poly.fit(x_train, y_train).predict([[55],[729]]).reshape(-1,1))
#looks great!!!
```

```
[[  12096.0395 ]
 [2125749.47118]]
```

In [26]:
```
#super exciting...that is sooo accurate!!
```

4. Now let's check a logistic regression to see if the model can figure out the >66th percentile binary condition

In [ ]:
```
#let's look into classification now
```

In [220]:
```
features = data[['random_num', 'score']]
outcome = data[['above_sixty_sixth_percentile']]
train_features, test_features, train_labels, test_labels = train_test_split(features, outcome,
test_size = 0.2)
#normalize this, since sklearn's logistic regression uses regularization
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)
test_features = scaler.transform(test_features) #we do NOT want to fit to the test
```

In [234]:
```
log_model = LogisticRegression(solver="liblinear") #to remove warning
log_model.fit(train_features, train_labels)
print('Accuracy Score: {}'.format(log_model.score(train_features, train_labels)))
```

```
Accuracy Score: 0.9833333333333333
```

4.a - Accuracy looks excellent

In [236]:
```
#model looking like it's figured out my percentile trick
print('Accuracy Score: {}'.format(log_model.score(test_features, test_labels)))
```

```
Accuracy Score: 1.0
```

In [259]:
```python
#let's look at the coefficients to see what is most important
print(features.columns)
print(log_model.coef_)
#excellent! it's figured out that the score is more important
#the score is a function of the random number, but the percentile is directly ranking the score
```

```
Index(['random_num', 'score'], dtype='object')
[[2.09394363 3.23301648]]
```

4-b Cross validation checks out!

In [256]:
```python
#last let's look at the cross-validation
print("Cross-Validation Scoring") #need to add solver="liblinear" #to remove warning
print('Accuracy Score: {}'.format(round(cross_val_score(LogisticRegression(solver="liblinear"),
features, outcome, cv=10, scoring='r2').mean(),2)))
```

```
Cross-Validation Scoring
Accuracy Score: 0.98
```

In [ ]:
```python
#let's watch this thing predict
```

In [257]:
```python
data.head(12)
```

Out[257]:

	index	random_num	times_two	square_that	subtract_4	score	above_sixty_sixth_percentile
0	1	11	22	484	480	480	0
1	2	31	62	3844	3840	3840	0
2	3	20	40	1600	1596	1596	0
3	4	45	90	8100	8096	8096	0
4	5	42	84	7056	7052	7052	0
5	6	27	54	2916	2912	2912	0
6	7	16	32	1024	1020	1020	0
7	8	73	146	21316	21312	21312	1
8	9	47	94	8836	8832	8832	0
9	10	82	164	26896	26892	26892	1
10	11	2	4	16	12	12	0
11	12	60	120	14400	14396	14396	0

In [268]:
```python
#sample random numbers to check from the set
Test1 = np.array([11, 480])
Test2 = np.array([16, 1020])
Test3 = np.array([73, 21312])
Test4 = np.array([47, 8832])
Test5 = np.array([82, 26892])
sample_test = np.array([Test1, Test2, Test3, Test4, Test5])
sample_test = scaler.transform(sample_test)
print(log_model.predict(sample_test).reshape(-1,1))
```

```
[[0]
 [0]
 [1]
 [0]
 [1]]
```

4.c - let's see if the predictions on new numbers work! They do!!

In [272]:

```
#and now for some totally new numbers I'm making up now, but that follwo the rules
Test6 = np.array([55, 12096])
Test7 = np.array([729, 2125760])
Test8 = np.array([2.2,15.36])
real_test = np.array([Test6, Test7, Test8])
real_test = scaler.transform(real_test)
print(log_model.predict(real_test).reshape(-1,1))
#bingo!!!
```

```
[[0]
 [1]
 [0]]
```