Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

In [2]:
```
%matplotlib inline

# Numpy
import numpy as np
import numpy.random as rnd

#  Pandas
import pandas as pd

# Scikit learn imports for this NB (regression tools,linear and nonlinear)
from sklearn import preprocessing
from sklearn import pipeline
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score

#  Basic python modules
import os
import requests
import codecs
import io

# to make this notebook's output stable across runs
rnd.seed(42)

# To plot pretty figure

import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

def url_to_content (url,encoding='utf-8',bufferize=False):
    r = requests.get(url)
    content_raw = r.content
    # Content downloaded by request is a byte string.  We're pretty sure thi
    content = codecs.decode(content_raw,encoding=encoding)
    if bufferize:
        return io.StringIO(content)
    else:
        return content
```

Load and prepare Life satisfaction data

Before starting this exercise you should create a folder called `datasets` in the folder where this notebook is stored. We will be placing various data files in that folder, and the code below is written so as to look there. We will be making various subfolders in the `datasets` folder, and if you want, you can start by creating the first one right away. It's called `lifesat`.

The **Organization for Economic Cooperation and Development** (OECD) stats website contains all kinds if economic statistics on countries in downloadable form, in particular in a very popular stripped-down spreadsheet format call ".csv" (for comma-separated values). You will get a local copy. The particular dataset we want is the BLI data ("Better Life Index").

In [3]:
```
notebook_lifesat_url0 = 'https://github.com/gawron/python-for-social-science
lifesat_url = notebook_lifesat_url0.replace('github', 'raw.githubusercontent
lifesat_url = lifesat_url.replace('blob/','')

def load_lifesat_data (lifesat_url):
    oecd_file = 'oecd_bli_2015.csv'
    oecd_url= f'{lifesat_url}{oecd_file}'
    return pd.read_csv(oecd_url, thousands=',',encoding='utf-8')

oecd_bli = load_lifesat_data(lifesat_url)
print(len(oecd_bli))
oecd_bli = oecd_bli[oecd_bli["INEQUALITY"]=="TOT"]
print(len(oecd_bli))
oecd_bli = oecd_bli.pivot(index="Country", columns="Indicator", values="Valu
```

In the next cell we go through exactly the same processing steps we discussed for the data in the pandas module introduction part II notebook. To review the data, and the motivations for these steps, please visit that notebook.

In the exercise ahead, we're going to take particular interest in the `Life satisfaction` score, a kind of general "quality of life" or "happiness" score computed from a formula combining many of the indicators in this data.

> Since 2002, the World Happiness Report has used statistical analysis to determine the world's happiest countries. In its 2021 update, the report concluded that Finland is the happiest country in the world. To determine the world's happiest country, researchers analyzed comprehensive Gallup polling data from 149 countries for the past three years, specifically monitoring performance in six particular categories: gross domestic product per capita, social support, healthy life expectancy, freedom to make your own life choices, generosity of the general population, and perceptions of internal and external corruption levels.

Load, prepare, and merge the GDP per

capita data

Elsewhere, on the world wide web, with help from Google, we find data about GDP ("gross domestic product") here. Hit the download butten and place another csv file in the same directory as the last data.

In [4]:
```
# Downloaded data from http://goo.gl/j1MSKe (=> imf.org) to github
def load_gdp_data ():
    gdp_file = "gdp_per_capita.csv"
    gdp_url = f'{lifesat_url}{gdp_file}'
    return pd.read_csv(gdp_url, thousands=',', delimiter='\t',
                                 encoding='latin1', na_values="n/a")

gdp_per_capita = load_gdp_data()
```

In [5]:
```
gdp_per_capita.rename(columns={"2015": "GDP per capita"}, inplace=True)
# Make "Country" the index column.  We are going to merge data on this colum
gdp_per_capita.set_index("Country", inplace=True)
full_country_stats = pd.merge(left=oecd_bli, right=gdp_per_capita,
                              left_index=True, right_index=True)
full_country_stats.sort_values(by="GDP per capita", inplace=True)
full_country_stats
```

Out[5]:

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Brazil	18.0	7.9	4.0	6.7	45.0	10.41	
Mexico	30.0	12.8	9.0	4.2	37.0	28.83	
Russia	15.0	3.8	2.5	15.1	94.0	0.16	
Turkey	35.0	5.0	5.5	12.7	34.0	40.86	
Hungary	15.0	3.6	7.9	4.8	82.0	3.19	
Poland	33.0	1.4	10.8	3.2	90.0	7.41	
Chile	46.0	6.9	2.0	9.4	57.0	15.42	

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Slovak Republic	13.0	3.0	6.6	0.6	92.0	7.02	
Czech Republic	16.0	2.8	6.8	0.9	92.0	6.98	
Estonia	9.0	5.5	3.3	8.1	90.0	3.30	
Greece	27.0	3.7	6.5	0.7	68.0	6.16	
Portugal	18.0	5.7	6.5	0.9	38.0	9.62	
Slovenia	26.0	3.9	10.3	0.5	85.0	5.63	
Spain	24.0	4.2	7.3	0.1	55.0	5.89	

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Korea	30.0	2.1	10.4	4.2	82.0	18.72	
Italy	21.0	4.7	5.0	1.1	57.0	3.66	
Japan	24.0	1.4	7.3	6.4	94.0	22.26	
Israel	21.0	6.4	2.5	3.7	85.0	16.03	
New Zealand	11.0	2.2	10.3	0.2	74.0	13.87	
France	12.0	5.0	3.5	0.5	73.0	8.15	
Belgium	21.0	6.6	4.5	2.0	72.0	4.57	

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Germany	16.0	3.6	4.5	0.1	86.0	5.25	
Finland	15.0	2.4	9.0	0.6	85.0	3.58	
Canada	15.0	1.3	10.5	0.2	89.0	3.94	
Netherlands	30.0	4.9	6.1	0.0	73.0	0.45	
Austria	27.0	3.4	7.1	1.0	83.0	7.61	
United Kingdom	13.0	1.9	11.5	0.2	78.0	12.70	
Sweden	10.0	5.1	10.9	0.0	88.0	1.13	

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Iceland	18.0	2.7	5.1	0.4	71.0	12.25	
Australia	13.0	2.1	10.5	1.1	76.0	14.02	
Ireland	13.0	2.6	9.0	0.2	75.0	4.20	
Denmark	15.0	3.9	7.0	0.9	78.0	2.03	
United States	18.0	1.5	8.3	0.1	89.0	11.30	
Norway	16.0	3.3	8.1	0.3	82.0	2.82	
Switzerland	20.0	4.2	8.4	0.0	86.0	6.72	

Country	Air pollution	Assault rate	Consultation on rule-making	Dwellings without basic facilities	Educational attainment	Employees working very long hours	Em
Luxembourg	12.0	4.3	6.0	0.1	78.0	3.47	

36 rows × 30 columns

Notice that when we did this merge, we lost some rows, in particular, countries for which we have GDP information, but no happiness stats. Consider Cypress.

In [6]:
```
cyprus_gdp_per_capita  = gdp_per_capita.loc["Cyprus"][["GDP per capita"]]
cyprus_gdp_per_capita
```

Out[6]:
```
GDP per capita    22587.49
Name: Cyprus, dtype: object
```

But when we try to look up Cypress in the merged data, we get an error. So as things stand now, we can't find out the quality of life score for Cypress. We return to this problem below.

In [91]:
```
# This is a KeyError!
#full_country_stats.loc['Cyprus']
```

Make a linear regression model

We are going to try to build a model that predicts life satisfaction from GDP. The kind of model we're going to use is called a **LInear Regression model**.

To start with, we are going to separate our country data into two sets, one set of countries which we use to train our model, and the other set which will be held back during training. We call this second part of the data the **test set**. We are going to ask our model to try to **predict** the life satisfaction of the test countries. Let's choose 7 countries for the test, and since we have our countries sorted by GDP, let's pick 3 countries from the low end and 3 countries from the high end to test on.

In [7]:
```
remove_indices = [0, 1, 6, 8, 33, 34, 35]
keep_indices = list(set(range(36)) - set(remove_indices))
```

```
#Train on this!
sample_data = full_country_stats[["GDP per capita", 'Life satisfaction']].il
#Test on this!
missing_data = full_country_stats[["GDP per capita", 'Life satisfaction']].i
```

In [8]:
```
sample_data.head()
```

Out[8]:

Country	GDP per capita	Life satisfaction
Russia	9054.914	6.0
Turkey	9437.372	5.6
Hungary	12239.894	4.9
Poland	12495.334	5.8
Slovak Republic	15991.736	6.1

In [9]:
```
missing_data
```

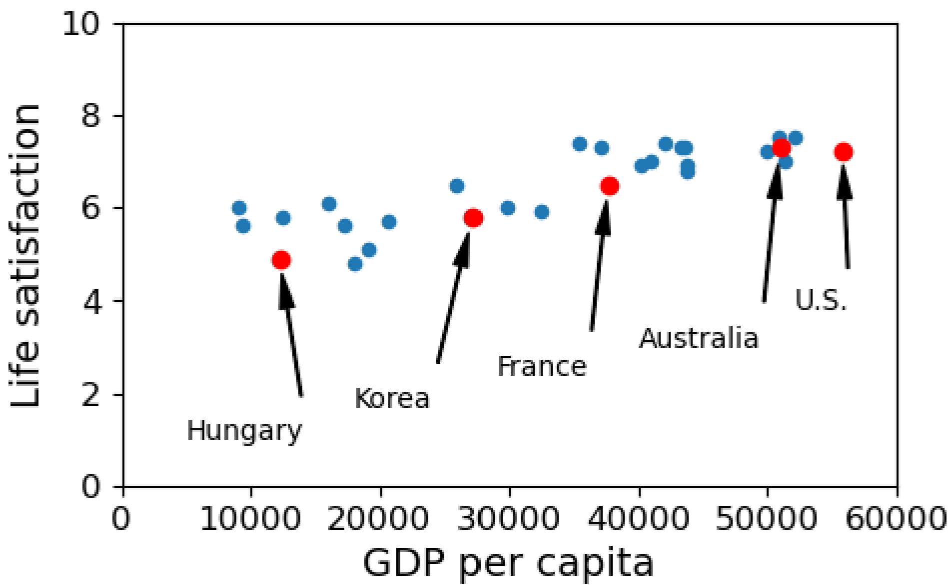
Out[9]:

Country	GDP per capita	Life satisfaction
Brazil	8669.998	7.0
Mexico	9009.280	6.7
Chile	13340.905	6.7
Czech Republic	17256.918	6.5
Norway	74822.106	7.4
Switzerland	80675.308	7.5
Luxembourg	101994.093	6.9

It's pretty easy to show that data like this has some potential for making accurate predictions.

In [10]:
```
# Scatter all the points, using the pandas DF plot method.
sample_data.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',
# GDP 0-60K X-axis. LifeSat 0-10 Y-axis
plt.axis([0, 60000, 0, 10])
# Text positions need some eyeballing, and so are entered by hand. x = left
position_text = {
    "Hungary": (5000, 1),
    "Korea": (18000, 1.7),
    "France": (29000, 2.4),
    "Australia": (40000, 3.0),
    "United States": (52000, 3.8),
}
for country, pos_text in position_text.items():
    pos_data_x, pos_data_y = sample_data.loc[country]
```

```
    country = "U.S." if country == "United States" else country
    plt.annotate(country, xy=(pos_data_x, pos_data_y), xytext=pos_text,
            arrowprops=dict(facecolor='black', width=0.5, shrink=0.1, headwi
    # Make these points red circles
    plt.plot(pos_data_x, pos_data_y, "ro")
plt.show()
```



As the GDP grows the Life satisfaction grows. Over on the right, the life satisfaction of the US isn't quite where we'd expect it to be. Although the US has a higher GDP than Australia, its life satisfaction trails behind. But all in all, it's a pretty strong trend. More money equals more happiness.

In [10]:
```
sample_data.loc[list(position_text.keys())]
```

Out[10]:

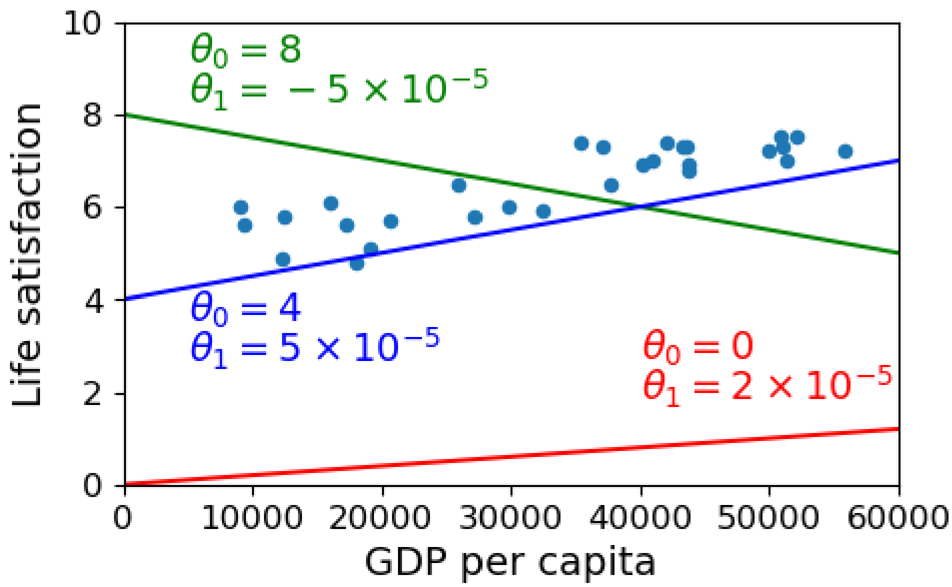
Country	GDP per capita	Life satisfaction
Hungary	12239.894	4.9
Korea	27195.197	5.8
France	37675.006	6.5
Australia	50961.865	7.3
United States	55805.204	7.2

Doing it by hand

In this notebook, we're looking at **linear regression models**. A regression model is a kind of model that tries to predict some dependent variable on the basis of one or more independent variables. Sometimes we say the model attempts to **explain** the dependent variable in terms of the independent variables. A **linear regression model** tries to predict with a linear model, a model that can actually be represented as a line or a plane or a hyperplane on a plot. In this notebook we'll study the simplest case, trying to predict one dependent variable with one independent variable. In that case the line that represents the model can be drawn on a 2D plot. For our dependent variable we'll use life satisfaction and for our independent variable we'll use GDP. So we're trying to explain life satisfaction in terms of GDP with a linear model.

Let's try to draw some lines that capture the trends, just using our eyeballs. Here are some hot-or-miss attempts. A line be defined by two numbers, a slope value θ_1 and an intercept value θ_0; θ_0 is y-value at which the line intersects the y-axis; θ_1 is the slope. Here are some guesses for those values, together with a sample of points to evaluate the guesses. The closer the lines are to the points, the better the model.

In [11]:
```
#import numpy as np

# The data
sample_data.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',
plt.axis([0, 60000, 0, 10])
X=np.linspace(0, 60000, 1000)

# red line
plt.plot(X, 2*X/100000, "r")
plt.text(40000, 2.7, r"$\theta_0 = 0$", fontsize=14, color="r")
plt.text(40000, 1.8, r"$\theta_1 = 2 \times 10^{-5}$", fontsize=14, color="r

# green line
plt.plot(X, 8 - 5*X/100000, "g")
plt.text(5000, 9.1, r"$\theta_0 = 8$", fontsize=14, color="g")
plt.text(5000, 8.2, r"$\theta_1 = -5 \times 10^{-5}$", fontsize=14, color="g

# blue line
plt.plot(X, 4 + 5*X/100000, "b")
plt.text(5000, 3.5, r"$\theta_0 = 4$", fontsize=14, color="b")
plt.text(5000, 2.6, r"$\theta_1 = 5 \times 10^{-5}$", fontsize=14, color="b"

# Let's have this cell return nothing.  Just a display cell.
None
```



The blue line isn't bad. Maybe we can do better if we use some math.

The scitkit_learn (aka sklearn) linear regression module

We load up the `sklearn` linear regression module and ask it to find the line that best fits our sample data.

In [86]:
```
from sklearn import linear_model

#Data
# Independent variable
Xsample = sample_data[["GDP per capita"]]
# Dependent variable
ysample = sample_data[["Life satisfaction"]]

#Create model
lin1 = linear_model.LinearRegression()
#Train model on data
lin1.fit(Xsample, ysample)

# Here's the result of the learning.
theta0, theta1 = lin1.intercept_[0], lin1.coef_[0][0]
theta0, theta1
```

Out[86]:
```
(4.853052800266436, 4.911544589158484e-05)
```

Notice creating a `linear_model` produced a python object called `lin1`. That's what we run the `fit` method on. This method "fits" the model to the data. After fitting, the `lin1` object has learned two numbers called the `intercept` and the `coefficient`. These are the slope and intercept of the regression line.

Note that sklearn is fully integrated with pandas, but the training data passed in must be a `DataFrame`. This is why the lines defining `Xsamples` and `Ysample` have two square brackets instead of just one:

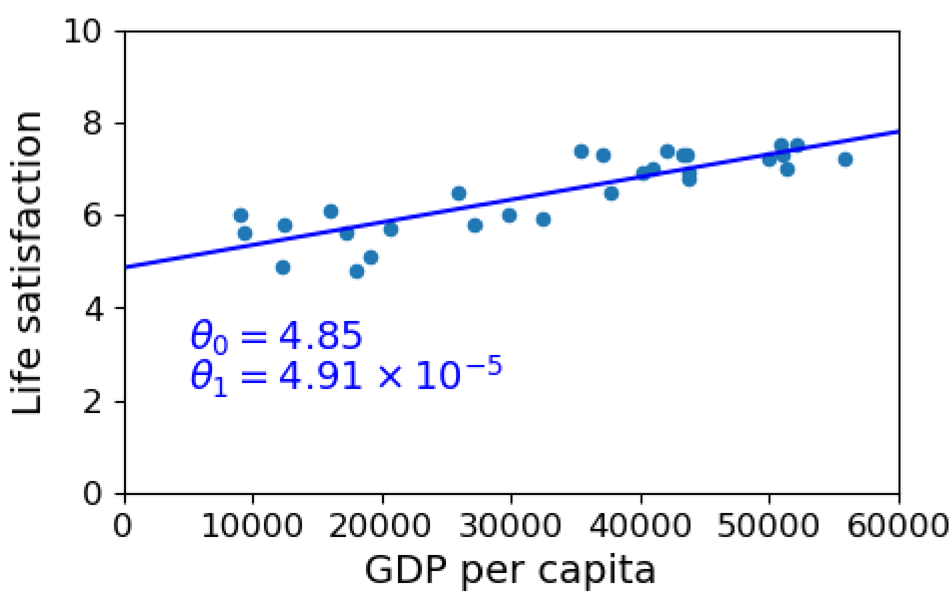
In [143…
```
type(sample_data[["GDP per capita"]]),type(sample_data["GDP per capita"])
```

Out[143]:
```
(pandas.core.frame.DataFrame, pandas.core.series.Series)
```

Essentially the data must be 2-dimensional. The same would hold true if we were passing in numpy arrays with one feature. In place of training data with shape (n,), we would pass in data with shape (n,1), reshaping the data if necessary.

We draw our data again with the line defined by the slope and intercept. Pretty good fit, eyeballing it.

In [14]:
```
sample_data.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',
plt.axis([0, 60000, 0, 10])
X=np.linspace(0, 60000, 1000)
plt.plot(X, theta0 + theta1*X, "b")
plt.text(5000, 3.1, r"$\theta_0 = 4.85$", fontsize=14, color="b")
plt.text(5000, 2.2, r"$\theta_1 = 4.91 \times 10^{-5}$", fontsize=14, color=
None
```



Predicting the Life Satisfaction Score for Cypress

First, let's try to predict the `Life Satisfaction` for a country we know the GDP of, which is missing from our quality of life data, `Cypress`.

What we do is plug the GDP for Cypress into the model using the `predict` method, and it returns the predicted `Life Satisfaction` score, which is 5.96 or so.

In [111…
```
cyprus_gdp_per_capita = gdp_per_capita.loc["Cyprus"][["GDP per capita"]]
print(f"{'Model input GDP val:':<30} {cyprus_gdp_per_capita['GDP per capita'
cyprus_predicted_life_satisfaction = lin1.predict(pd.DataFrame({"GDP per cap
```

```
                                                                  cyprus_gdp_p
# What we get out
print(f"{'Model output lifesat score:':<30} {cyprus_predicted_life_satisfact
```

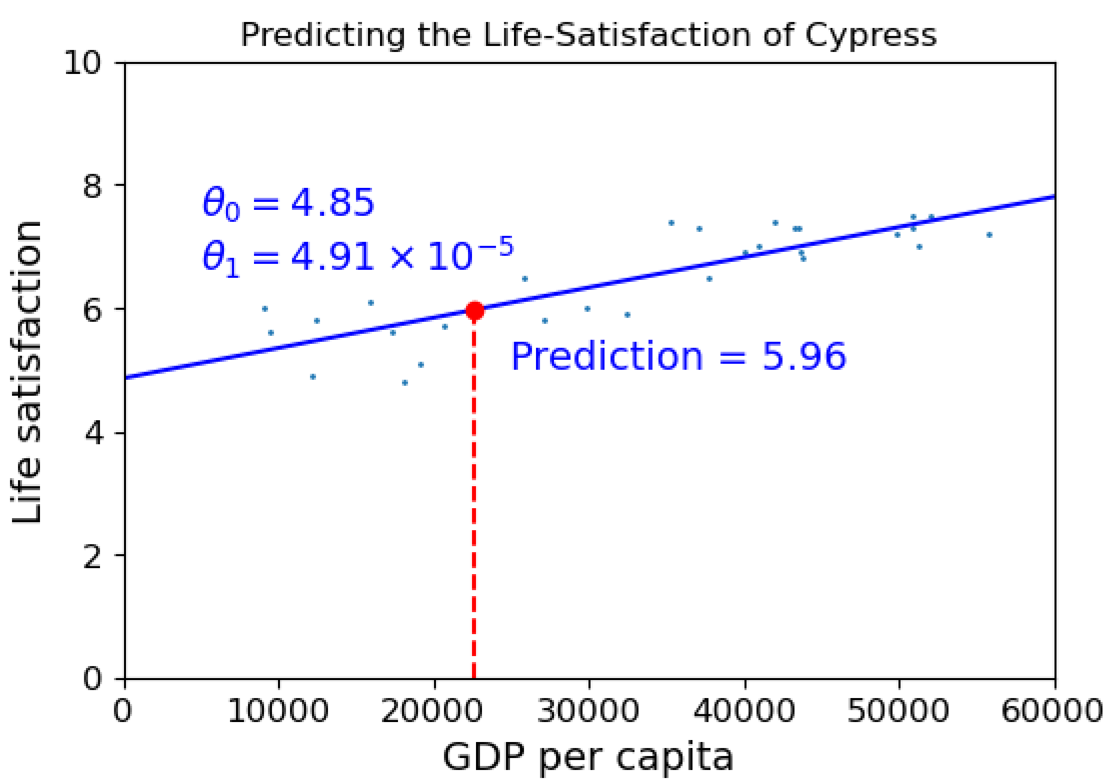
```
Model input GDP val:           22,587.49
Model output lifesat score:         5.96
```

In [103…

```
from matplotlib import pyplot as plt

fig, ax = plt.subplots(1,1,figsize=(6,4))
sample_data.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',
X=np.linspace(0, 60000, 1000)

# Plot the line our linear regression model learned.
ax.set_title("Predicting the Life-Satisfaction of Cypress")
ax.plot(X, theta0 + theta1*X, "b")
ax.axis([0, 60000, 0, 10])
ax.text(5000, 7.5, r"$\theta_0 = 4.85$", fontsize=14, color="b")
ax.text(5000, 6.6, r"$\theta_1 = 4.91 \times 10^{-5}$", fontsize=14, color="

# Plot a vertical red dashed line where the GDP of CYPRESS is.
# It goes from the x axis right up to where our predicted happiness is.
ax.plot([cyprus_gdp_per_capita, cyprus_gdp_per_capita], [0, cyprus_predicted

ax.text(25000, 5.0, r"Prediction = 5.96", fontsize=14, color="b")
# Plot a fat red dot right where our predicted happiness is
# Notice it lands right on the line, and it has to, because our model just i
ax.plot(cyprus_gdp_per_capita, cyprus_predicted_life_satisfaction, "ro")
#plt.show()
```

Out[103]:

```
[<matplotlib.lines.Line2D at 0x7fe5384bf250>]
```



Look at the sample data that has around the same GDP as Cypress.

In [29]:
```
sample_data[7:10]
```

Out[29]:

	GDP per capita	Life satisfaction
Country		
Portugal	19121.592	5.1
Slovenia	20732.482	5.7
Spain	25864.721	6.5

In [30]:
```
cyprus_gdp_per_capita
```

Out[30]:
```
GDP per capita    22587.49
Name: Cyprus, dtype: object
```

Suppose we try to predict our life satisfaction by taking the average of these 3 points.

In [31]:
```
(5.1+5.7+6.5)/3
```

Out[31]:
```
5.766666666666667
```

So you can see this estimate is lower than the 5.96 estimated by our model. This is because the linear model tries for the best line that fits **all** the points, so it responds to the fact that there is a steady upward trend with a particular slope, and because Spain falls below that line, it is treated as a point whose value has been reduced by **noise**.

So far, so good. But what we really need to do is TEST our model on some data it didn't see during training, but for which we know the answers. That was the point of setting aside some test data.

In [32]:
```
missing_data
```

Out[32]:

	GDP per capita	Life satisfaction
Country		
Brazil	8669.998	7.0
Mexico	9009.280	6.7
Chile	13340.905	6.7
Czech Republic	17256.918	6.5
Norway	74822.106	7.4
Switzerland	80675.308	7.5
Luxembourg	101994.093	6.9

In [17]:
```
position_text2 = {
    "Brazil": (1000, 9.0),
```

```
    "Mexico": (11000, 9.0),
    "Chile": (25000, 9.0),
    "Czech Republic": (35000, 9.0),
    "Norway": (60000, 3),
    "Switzerland": (72000, 3.0),
    "Luxembourg": (90000, 3.0),
}
```

The cell below compares two models, the one we used before (the dotted blue line), trained on our data sample, and another model trained on all the data (the solid black line). The significant difference between these models migfht be explained in three different ways, not necessarily mutually exclusive: (a) the points we trained on before are not all that representative of the entire data set or (b) our test points are outliers (they sort of are if you look back at how we chose them); or (c) the behavior of the data cannot be captured by a linear model. The fact that the model trained on all the data is still significantly off suggests that some of the error is because of (c), although it does not eliminate the possibility that noise plays a role.

Mark	Model trained on	Act/Pred
black line	train + test	Pred
dotted blue line	train	Pred
red squares		Act test

| big blue dots | | Act train

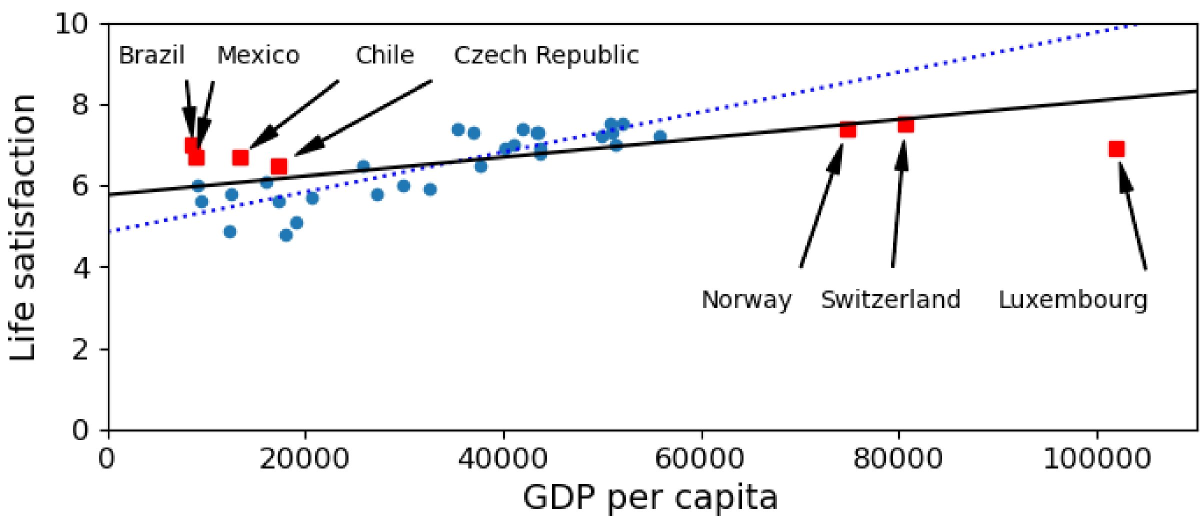
The red squares are the actual locations of our test points. The dot on the dotted blue line that lies directly above or below the red square is its predicted value according to the model we trained before. If the red square lands directly on the dotted blue line, the model got it it exactly right. If the red square is some distance above the line (like "Brazil"), the model's estimate is low; if the red square is some distance below the blue line (like "Luxembourg"), the model's estimate is high.

Which of the following statements about **the dotted blue line** do the red dots provide evidence for?

1. Does the blue model underestimate or overestimate the happiness of poor countries?
2. Does the blue model underestimate or overestimate the happiness of rich countries?

In [18]:
```
sample_data.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',
plt.axis([0, 110000, 0, 10])

for country, pos_text in position_text2.items():
    pos_data_x, pos_data_y = missing_data.loc[country]
    plt.annotate(country, xy=(pos_data_x, pos_data_y), xytext=pos_text,
            arrowprops=dict(facecolor='black', width=0.5, shrink=0.1, headwi
    plt.plot(pos_data_x, pos_data_y, "rs")
```

```
X=np.linspace(0, 110000, 1000)
# Plot the dotted blue line
plt.plot(X, theta0 + theta1*X, "b:")

lin_reg_full = linear_model.LinearRegression()
Xfull = full_country_stats[["GDP per capita"]]
yfull = full_country_stats[["Life satisfaction"]]
lin_reg_full.fit(Xfull, yfull)

t0full, t1full = lin_reg_full.intercept_[0], lin_reg_full.coef_[0][0]
X = np.linspace(0, 110000, 1000)
# Plot the black line
plt.plot(X, t0full + t1full * X, "k")

plt.show()
```



Which of the following statements about **the dotted blue line** do the red dots provide evidence for?

1. Does the blue model underestimate or overestimate the happiness of poor countries?
2. Does the blue model underestimate or overestimate the happiness of rich countries?

Q1. If we include both the training points (blue) and test points (red), it does a fair bit of both under estimate and overestimating. It underestimates the happiness of all the test points.

Q2. The three richest countries are all in the test set, and the dotted blue model overestimates the happiness of all three rich countries. Indeed, so does the model trained on all the data, but not by as much.

Other models

The next cell has some code computing a complicated model, one which does not have to be a line. It is a higher order polynomial model which can curve up and down to capture all the data points. We can measure the aggregate error of a model by computing the distance of each actual point from the curve/line of the model. By that definition, the polynomial model is reducing the "error" on the training set by a large amount. But is it a better model? Only the performance on a test set will tell us for sure.

Note: The machine learning toolkit demoed here (`scikit_learn`) has a variety of robust regression strategies implemented; see Scikit learn docs for discussion.

Pipeline

We choose a degree, the highest exponent in the polynomial function we will use to make predictions. Below we choose 20.

1. Polynomial features. Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree k. For example, if an input sample is two dimensional and of the form [a, b] (say using the two features GDP and employment), the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.
   What this notation means is that the model has to learn coefficients or **weights** to associate with each of these values such that when the following polynomial is summed, it gets as close as possible to the real life-satisfaction value:

   $$w_1 + w_2 a + w_3 b + w_4 a^2 + w_5 ab + w_6 b^2$$

   These polynomial models have a lot of numbers to learn. For a 1-dimensional input model (one feature, GDP, in our case) and k=60, the model will compute 20 new feature values [1, a^2, ..., a^20] and need to learn weights for all of them. If we used employment in addition to GDP, it would be a lot more.

2. Scaler. The standard scaler computes the **z-score**, that is, it maps each feature value x onto z where

   $$z = (x - \mu)/s$$

   where μ is the mean of the feature and s is the standard deviation. As we saw in the pandas module, this is known as centering and scaling. One motivation for scaling and centering is that it reduces the possibility of over- or under- valuing features with large/small value ranges.

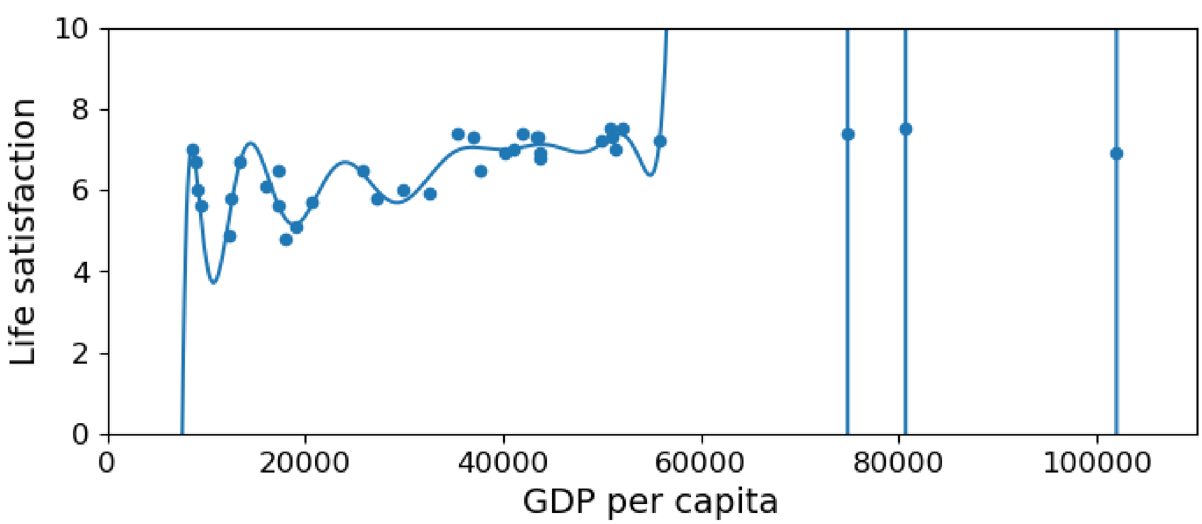
3. With the new features we still do a linear regression, just in a much higher dimensional space. In other words, the learning computation in the example depicted below is exactly the same as if we had 60 independent features instead of 1 value we had raised to 60o different powers. In both cases it's a linear model. In both cases, we seek to learn the weights that define a hyper plane that provides the least squared error solution. When we do a 2D plot of the predicted values our

dimension-60 "linear" model assigns to our data (using the original data GDP feature as the X coordinate), the predicted Life Satisfaction value is a sinuous curve.

Note: This is a pretty silly example but we use it to illustrate a simple point. With a model with enough parameters (each of those Polynomial features gets its own coefficient), you can fit anything.

In [21]:

```
full_country_stats.plot(kind='scatter', x="GDP per capita",
                        y='Life satisfaction', figsize=(8,3))
plt.axis([0, 110000, 0, 10])



poly = preprocessing.PolynomialFeatures(degree=20, include_bias=True)
scaler = preprocessing.StandardScaler()
lin_reg2 = linear_model.LinearRegression()

pipeline_reg = pipeline.Pipeline([('poly', poly), ('scal', scaler), ('lin',
# Train on the full GDP dataset for the sake of the picture
pipeline_reg.fit(Xfull, yfull)

# Pass in a large set of sample GDP values from poor to rich
Xvals = np.linspace(0, 110000, 1000)
# ... as a DF as per training
X = pd.DataFrame(Xvals, columns=["GDP per capita"])
curve = pipeline_reg.predict(X)
plt.plot(Xvals, curve)
plt.show()
```



So this model **really** minimizes mean squared error. It pretty much threads the curve through every point.

But that's every point of the **training** set. What about unseen data, the **test set**?

Let's evaluate by computing the mean squared error of the polynomial and linear model when both are trained on the same training data (with 6 outliers removed).

In [22]:
```
pipeline_reg = pipeline.Pipeline([('poly', poly), ('scal', scaler), ('lin',
# Now Train on the sample dataset
pipeline_reg.fit(Xsample, ysample)

# Polynomial Model
#vals_poly = pipeline_reg.predict(missing_data["GDP per capita"].values[:,np
vals_poly = pipeline_reg.predict(missing_data[["GDP per capita"]])
# Compare predicted values to actual values
poly_mse = mean_squared_error (vals_poly[:,0], missing_data["Life satisfacti

# Linear Model
#vals_lin = lin1.predict(missing_data["GDP per capita"].values[:,np.newaxis]
vals_lin = lin1.predict(missing_data[["GDP per capita"]])
lin_mse = mean_squared_error (vals_lin[:,0], missing_data["Life satisfaction
```

And now the results, with a fluorish:

In [23]:
```
print(f' Lin Model MSE: {lin_mse:.3e}')
print(f'Poly Model MSE: {poly_mse:.3e}')
```

```
 Lin Model MSE: 2.682e+00
Poly Model MSE: 5.040e+24
```

The mean squared error of the polynomial model is crazy bad!

By comparison, the mean squared error of the linear model is pretty reasonable!

To look into this, let's cook up a squared error function so as to look at the individual squared errors before the mean is taken.

In [24]:
```
def get_squared_error(predicted, actual):
    return (predicted - actual)**2
```

In [104…
```
##  The general argument signature for evaluation metrics is
##  evaluation_metric(predicted_values,actual_values)
se_poly = get_squared_error (vals_poly[:,0], missing_data["Life satisfaction
# Save this to look at later.
missing_data['sq_err_poly'] = se_poly
# Now take the mean of squared error (to compare with sklearn function)
mse_poly = se_poly.mean()
## pretty good egreement with sklearn's mean_squared_error function
mse_poly
```

Out[104]:
```
5.039974893489065e+24
```

In [106…
```
# Same for the linear model
se_lin = get_squared_error (vals_lin[:,0], missing_data["Life satisfaction"]
# Save this too look at later
missing_data['sq_err_lin'] = se_lin
# Get the mean of squared error
mse_lin = se_lin.mean()
# Also good agreement
mse_lin
```

Out[106]: 2.681893248747465

What happened?

Here is our data with squared errors from the models appended.

In [107…

```
#  Think about why the squared error columns are so nicely aligned with the
#  What Python type does get_squared_error error return?
missing_data
```

Out[107]:

Country	GDP per capita	Life satisfaction	sq_err_poly	sq_err_lin
Brazil	8669.998	7.0	9.936943e+00	2.962242
Mexico	9009.280	6.7	5.971153e-01	1.972487
Chile	13340.905	6.7	2.379451e-01	1.420155
Czech Republic	17256.918	6.5	1.558291e+00	0.638986
Norway	74822.106	7.4	5.020802e+17	1.272325
Switzerland	80675.308	7.5	6.602070e+19	1.730426
Luxembourg	101994.093	6.9	3.527976e+25	8.776632

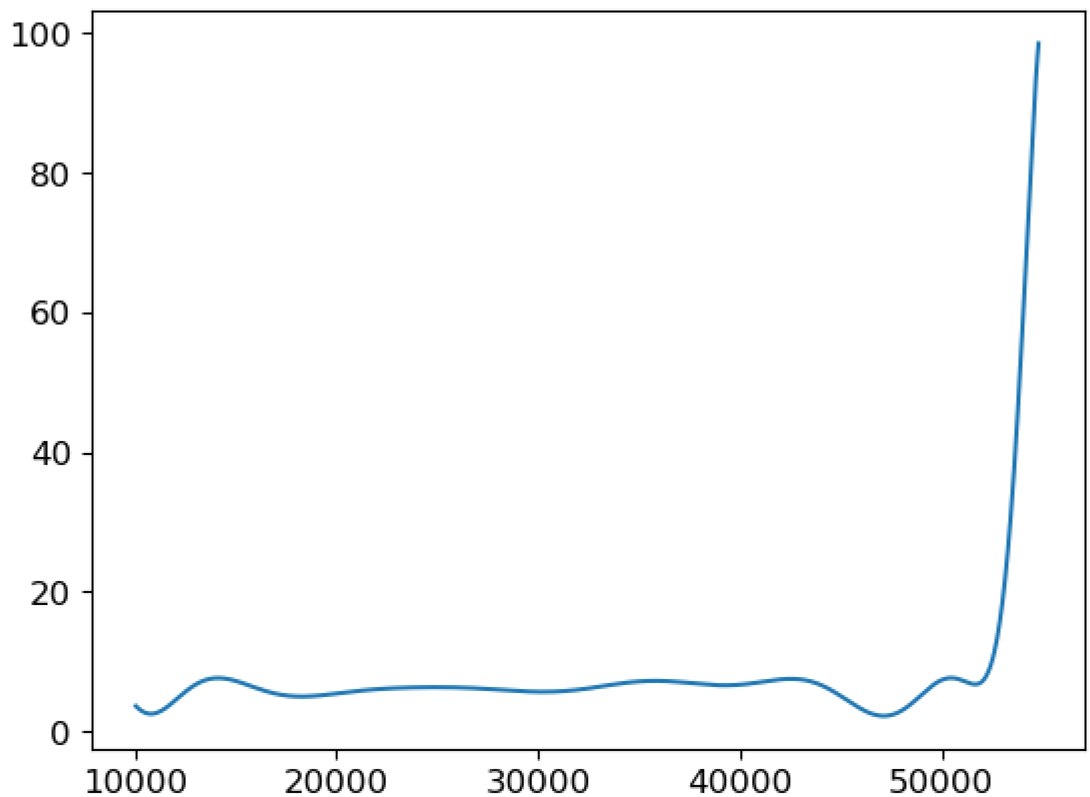
What happened was on the three rich countries (Norway, Switzerland, and Luxembourg) the squared error skyrocketed for the polynomial model.

Looking back at the original picture of the polynomial model points to the source of the problem. The curve of the polynomial model swings wildly up and down in order to thread its way through the life-satisfaction scores for all training data countries, and the training data leaves it on an upswing.

We can observe the polynomial model's model's upward swing by drawing a picture in which the x-axis goes well beyond the GDP values the model was trained on, and that's where the predicted life-satisfaction scores skyrocket.

In [44]:

```
pipeline_reg_full = pipeline.Pipeline([('poly', poly), ('scal', scaler), ('l
# Train on the full GDP dataset for the sake of the picture
pipeline_reg_full.fit(Xsample, ysample)

# Pass in a large set of sample GDP values from poor to rich
Xvals0 = np.linspace(Xsample.min()+1000, Xsample.max()-1000, 1000)
X0 = pd.DataFrame(Xvals0, columns=["GDP per capita"])
# predict expects a 2D array.  Make our 1D array X a 1000x1 2D array
#curve = pipeline_reg.predict(X[:, np.newaxis])
curve = pipeline_reg_full.predict(X0)
plt.plot(Xvals0, curve)
plt.show()
```



The polynomial model minimized the squared error on whatever training set it was given but it missed the general trend. As a result, it performed terribly on unseen data. This is known as **overfitting** or **overtraining**.

This is the danger of very powerful models. In basically memorizing the training set, they sometimes fail to learn the key generalization that will help cope with unseen data.

A Note: Evaluating Regression models (R^2 vs MSE)

A final note on evaluation. We have been using mean squared error (MSE) as our chief tool for evaluating regression models. As a first approximation it is useful. It is intuitive and, so far, it has given us results that make sense.

But in practical applications it is wise to use another measure R^2 (or `r2_score` in the `sckit learn` metrics, or **coefficient of determination** as it's sometimes called).

R^2 avoids this problem. The R^2 score is a number less than 1 that, in the general case, represents the proportion of variability in y that can be "explained" by the regressor variables. $R^2 = 1$ means that all the variability is explained. The model is a perfect fit. A value of 0 results when the sequence of predictors is constant and is always the mean of the true values. The R^2 value can be negative (as it is in our case), when the model performs worse than a model that always predicts the mean.

Unlike `mean_squared_error`, it's not symmetric, and the arguments must be passed in this order: `r2_score(true_values, predicted_values)`.

One drawback of R^2 is that it grows as the number of variables grows, so R^2 with no modification is of no help in evaluating what happens as variables are added to the model. So below where we explore feature choice we've used R^2 to illustrate its use, but we don't rely on it to help evaluate the models.

To keep this discussion tidily in one place, here is the code for evaluating the regression model above with `sckit learn` metrics, using both `mean_squared_error` and `r2_score`.

In [108…

```
from sklearn.metrics import mean_squared_error, r2_score

vals_lin = lin1.predict(missing_data[["GDP per capita"]])
```

In [109…

```
# vals_lin is  a 7x1 2D array.  mean_squared_error wants two 1D arrays or se
# the predicted values and the actual values.  So we pass in the first colum
lin_mse = mean_squared_error (missing_data["Life satisfaction"], vals_lin[:,
# Same for r2 score
lin_r2 =  r2_score(missing_data["Life satisfaction"],vals_lin[:,0])
print(f' Lin Model MSE: {lin_mse:.3e} Lin Model R^2: {lin_r2:.2f}')
```

```
Lin Model MSE: 2.682e+00 Lin Model R^2: -21.43
```

Note that the R^2 value is a negative number less than -1. This means this is a very bad model.

Here are the numbers our evaluation metrics compared:

In [51]:

```
comparison_df = missing_data["Life satisfaction"].to_frame(name="Actual")
comparison_df["Predicted"] = vals_lin
comparison_df["$E^2$"] = (comparison_df["Actual"]-comparison_df["Predicted"]
comparison_df
```

Out[51]:

	Actual	Predicted	E^2
Country			
Brazil	7.0	5.278884	2.962242
Mexico	6.7	5.295548	1.972487
Chile	6.7	5.508297	1.420155
Czech Republic	6.5	5.700634	0.638986
Norway	7.4	8.527974	1.272325
Switzerland	7.5	8.815457	1.730426
Luxembourg	6.9	9.862538	8.776632

Verifying MSE for the DataFrame

In [48]:

```
comparison_df["E^2"].mean()
```

Out[48]: 2.681893248747465

`r2_score` works on `pandas` `Series` arguments.

In [54]:
```
r2_score(comparison_df["Actual"],comparison_df["Predicted"])
```

Out[54]: -21.425387233553884

Choosing features

Regression always starts with choosing features to predict with. Don't skip this step. Don't just always use all the features in your data set. More isn't necessarily better.

Here's an interesting observation about the training data, which a model might "notice" if we had features that represented the letters in the country name. Let's consider all countries with a "w" in their name:

In [55]:
```
full_country_stats.loc[[c for c in full_country_stats.index
                        if "W" in c.upper()]]["Life satisfaction"]
```

Out[55]:
```
Country
New Zealand    7.3
Sweden         7.2
Norway         7.4
Switzerland    7.5
Name: Life satisfaction, dtype: float64
```

Every single one of them has life satisfaction score of over 7! This is fairly high. If this were represented in our data, a learner would certainly pick up on it, and treat it as a feature that increases the likelihood of happiness. Yet it is fairly clear that this is just an accidental feature of our data that will not generalize to other cases.

This example shows how poorly chosen features can hurt. They introduce noise that can be mistaken for meaningful patterns. We note that this feature does not correlate particularly well with GDP in the GDP table, which has a larger sample of countries, and we know that very low GDP makes a high life satisfaction score virtually impossible (because many of the features used to compute the score depend on wealth).

In [46]:
```
# Let's just look at one column of this largish table
gdp_col = gdp_per_capita["GDP per capita"]
gdp_col.loc[[c for c in gdp_col.index if "W" in c.upper()]].head()
```

Out[46]:
```
Country
Botswana        6040.957
Kuwait         29363.027
Malawi           354.275
New Zealand    37044.891
Norway         74822.106
Name: GDP per capita, dtype: float64
```

Note that relatively poor Botswana and Malawi are "W"-countries. This confirms our suspicion that the W "rule" is actually an accident of our sample.

With this warning in mind, let's try training with more features.

lin3: Lifesat feats

Now let's train with more features.

Shall we use all the columns? Here are some we might want to leave out.

They are precisely the features we acquired by merging in GDP data,

In [47]:
```
full_country_stats.columns[-6:]
```

Out[47]:
```
Index(['Subject Descriptor', 'Units', 'Scale', 'Country/Series-specific Not
es',
       'GDP per capita', 'Estimates Start After'],
      dtype='object')
```

Why leave out GDP? Because we used it in the model with one feature, We've demonstrated that it's a powerful predictor. Let's see how we do without it, using the other features, shown below:

In [57]:
```
predictors = list(full_country_stats.columns[:-6])
predicted ='Life satisfaction'
predictors.remove(predicted)
print(len(predictors),"features")
predictors
```

```
23 features
```

Out[57]:
```
['Air pollution',
 'Assault rate',
 'Consultation on rule-making',
 'Dwellings without basic facilities',
 'Educational attainment',
 'Employees working very long hours',
 'Employment rate',
 'Homicide rate',
 'Household net adjusted disposable income',
 'Household net financial wealth',
 'Housing expenditure',
 'Job security',
 'Life expectancy',
 'Long-term unemployment rate',
 'Personal earnings',
 'Quality of support network',
 'Rooms per person',
 'Self-reported health',
 'Student skills',
 'Time devoted to leisure and personal care',
 'Voter turnout',
 'Water quality',
 'Years in education']
```

Here is the code we used above to train and evaluate the linear model with one feature. We're going to adapt it to the case of many features.

In [69]:
```
# Linear Model with one feature
#Data

Xsample = sample_data[["GDP per capita"]]
ysample = sample_data[["Life satisfaction"]]

#Create model
lin1 = linear_model.LinearRegression()
#Train model on data
lin1.fit(Xsample, ysample)

#vals_lin = lin1.predict(missing_data["GDP per capita"].values[:,np.newaxis]
vals_lin = lin1.predict(missing_data[["GDP per capita"]])
lin1_mse = mean_squared_error (vals_lin[:,0], missing_data["Life satisfactic
lin1_r2 = r2_score (missing_data["Life satisfaction"],vals_lin[:,0])
```

These are the changes we need to create a training and test set with the 23 new features. Some code has been copied from above to make it clear what we're doing.

We will use the same training test split to facilitate comparison.

In [59]:
```
# Linear Model with our new predictors

remove_indices = [0, 1, 6, 8, 33, 34, 35]
keep_indices = list(set(range(36)) - set(remove_indices))

#Train on this!
```

```
sample_data3 = full_country_stats[predictors].iloc[keep_indices]
#Test on this!
missing_data3 = full_country_stats[predictors].iloc[remove_indices]

# Scikit learn will accept a Pandas DataFrame for training data!
Xsample3 = sample_data3[predictors]
Xtest_sample3 = missing_data3[predictors]
print(Xsample3.shape)
print(Xtest_sample3.shape)
# Same as with 1 feature model
ysample = np.c_[sample_data["Life satisfaction"]]
print(ysample.shape)
```

```
(29, 23)
(7, 23)
(29, 1)
```

In [83]:
```
#Create model
lin3 = linear_model.LinearRegression()
#Train model on data
lin3.fit(Xsample3, ysample)

vals3_lin = lin3.predict(Xtest_sample3)
lin3_mse = mean_squared_error (vals3_lin[:,0], missing_data["Life satisfacti
lin3_r2 = r2_score (missing_data["Life satisfaction"],vals3_lin[:,0])
```

In [85]:
```
eval_df = pd.DataFrame({"MSE": [lin1_mse,lin3_mse],
                        "R2":[lin1_r2,lin3_r2]},
                       index=["GDP","LifeSat"])

with pd.option_context("display.precision",2):
    print(eval_df.sort_values(by="MSE"))
```

```
          MSE     R2
LifeSat  1.65 -12.81
GDP      2.68 -21.43
```

Substantial improvement in the MSE (it went **down**). Now let's try folding GDP back in.

lin4: Lifesat feats + GDP

In [62]:
```
# Linear Model with our new predictors + GDP

remove_indices = [0, 1, 6, 8, 33, 34, 35]
keep_indices = list(set(range(36)) - set(remove_indices))

new_predictors = predictors + ['GDP per capita']
#Train on this!
sample_data4 = full_country_stats[new_predictors].iloc[keep_indices]
#Test on this!
missing_data4 = full_country_stats[new_predictors].iloc[remove_indices]

# Scikit learm Data
#Xsample4 = np.c_[sample_data4[new_predictors]]
#Xtest_sample4 = np.c_[missing_data4[new_predictors]]
Xsample4 = sample_data4[new_predictors]
```

```
Xtest_sample4 = missing_data4[new_predictors]
print(Xsample4.shape)
print(Xtest_sample4.shape)
```

```
(29, 24)
(7, 24)
```

In [63]:
```
#Create model
lin4 = linear_model.LinearRegression()
#Train model on data
lin4.fit(Xsample4, ysample)

vals4_lin = lin4.predict(Xtest_sample4)
lin4_mse = mean_squared_error (vals4_lin[:,0], missing_data["Life satisfacti
lin4_r2 = r2_score (missing_data["Life satisfaction"],vals4_lin[:,0])
```

The result is that adding GDP to the LifeSatisfaction model makes it worse, both in MSE and R2 score:

In [79]:
```
eval_df = pd.DataFrame({"MSE": [lin1_mse,lin3_mse,lin4_mse],
                        "R2":[lin1_r2,lin3_r2,lin4_r2]},
                       index=["GDP","LifeSat","GDP + LifeSat"])

with pd.option_context("display.precision",3):
    print(eval_df.sort_values(by="MSE"))
```

```
                  MSE      R2
LifeSat         1.651 -12.807
GDP + LifeSat   2.110 -16.642
GDP             2.682 -21.425
```

We saw above that there is some noise in the GDP feature.

The other predictors appear better suited to the task of predicting Life Satisfaction.

The takeaway: More features often help, but not always. When we introduce a noisy feature into a feature set with good prediction capabilities, we're building a model in a new higher dimensional space and the "best" model in that space may actually perform worse.

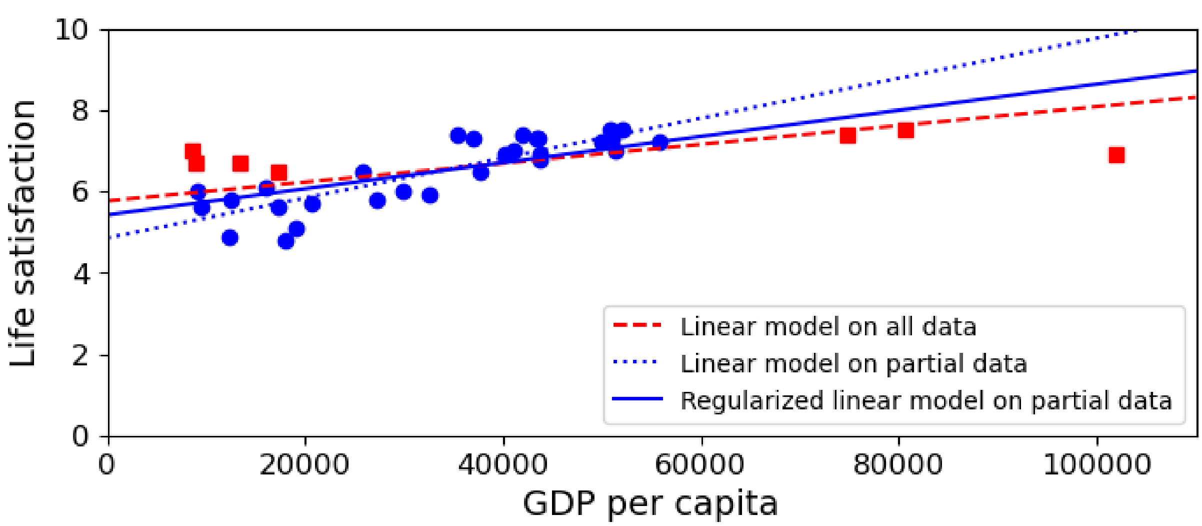
Ridge learning (regularization)

Model building is all about making reasonable generalizations, which in turn is about wisely choosing which patterns in the data will be used to make those generalizations.

One very important kind of constraint on generalization is called smoothing or regularization. This is a family of techniques used to used to discourage "extreme" models, for example, models that forbid certain kinds of patterns because they were never seen in the training sample. Without going into the details here, the picture below shows the intended behavior of regularization, using a case in which it works well. The dashed red line shows a model trained on all the data. For the time being, think of this as the right answer. The dotted blue line shows the model trained on our original training sample, which as we noted before, was not entirely representative of the data. At the GDP extremes, that model drifts quite far from the "right" answer. The solid blue line is the same kind of model trained on the same sample data, but with "ridge" regularization, which penalizes models with many high coefficients; in this case it will help reduce the slope of the line, while still requiring it to reduce the error on the training data.

The result is the solid blue line, which is closer to the "right" answer than the original linear regression model. Note: *this example is a somewhat oversimplified attempt to convey the intuition of what regularization does. One wouldn't normally use ridge regression in a case where there's a single predictor variable, because what makes it useful is that it reduces the influence of features that account for less of the variance.*

In [81]:
```
plt.figure(figsize=(8,3))

plt.xlabel("GDP per capita")
plt.ylabel('Life satisfaction')

plt.plot(list(sample_data["GDP per capita"]), list(sample_data["Life satisfa
plt.plot(list(missing_data["GDP per capita"]), list(missing_data["Life satis

X = np.linspace(0, 110000, 1000)
plt.plot(X, t0full + t1full * X, "r--", label="Linear model on all data")
plt.plot(X, theta0 + theta1*X, "b:", label="Linear model on partial data")

ridge = linear_model.Ridge(alpha=10**9.5)
#ridge = linear_model.Ridge()
Xsample = np.c_[sample_data["GDP per capita"]]
ysample = np.c_[sample_data["Life satisfaction"]]
ridge.fit(Xsample, ysample)
t0ridge, t1ridge = ridge.intercept_[0], ridge.coef_[0][0]
plt.plot(X, t0ridge + t1ridge * X, "b", label="Regularized linear model on p

plt.legend(loc="lower right")
plt.axis([0, 110000, 0, 10])
plt.show()
```



Summary thus far

We'll conclude with some general regression tips.

1. Choose your features wisely.
2. Learn about smoothing and regularization.
3. Explore (cautiously) models of regression other than linear.

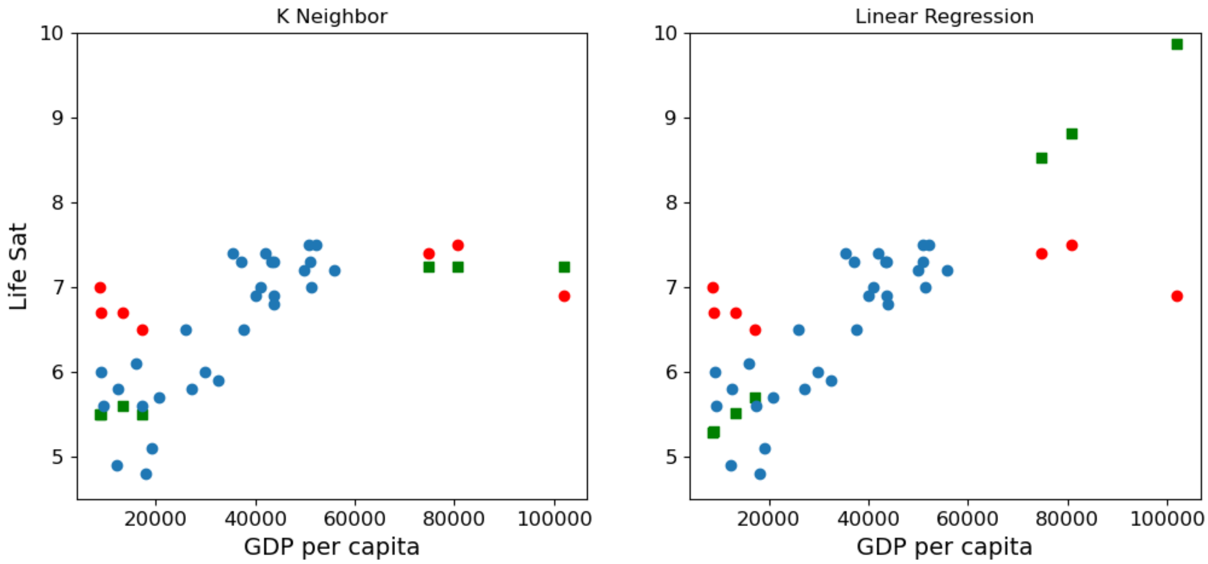
Example modeling code

The code below is for you to use when implementing new kinds of models using `scikit_learn`. It trains a new kind of prediction model, called a `k- neighbors model`. It makes use of the work we did before, so if you're fresh here, you'll have to execute all the cells above.

In [58]:
```
backup = oecd_bli, gdp_per_capita

def prepare_country_stats(oecd_bli):
    """
    This would normally do prep work,including the train/test split.
    For now just redoing the steps whereby we merged gdp info with
    the original life-satisfaction data. and split training and test
    """
    gdp_per_capita = load_gdp_data()
    gdp_per_capita.rename(columns={"2015": "GDP per capita"}, inplace=True)
    # Make "Country" the index column.  We are going to merge data on this c
    gdp_per_capita.set_index("Country", inplace=True)
    full_country_stats = pd.merge(left=oecd_bli, right=gdp_per_capita,
                                  left_index=True, right_index=True)
    full_country_stats.sort_values(by="GDP per capita", inplace=True)
    remove_indices = [0, 1, 6, 8, 33, 34, 35]
    keep_indices = list(set(range(36)) - set(remove_indices))

    #Train on this!
    training_data = full_country_stats[["GDP per capita", 'Life satisfactior
    #Test on this!
    test_data = full_country_stats[["GDP per capita", 'Life satisfaction']].
    return training_data, test_data

# Code example
#########################################################################
from sklearn import neighbors

# Load the data code goes here
# We loaded oecd_bli above.


# Prepare the data
training_data,test_data = prepare_country_stats(oecd_bli)
```

```python
# The np.c_ makes both X and y  2D arrays
X = np.c_[training_data["GDP per capita"]]
y = np.c_[training_data["Life satisfaction"]]

#######################################################################
#
#   Training
#
########################################################################

# Select and train a k-neighbors regression model, supplying values for impo
k_neigh_reg_model = neighbors.KNeighborsRegressor(n_neighbors=3)
# Train the model (always the .fit() method in sklearn)
k_neigh_reg_model.fit(X, y)
# select and train a linear regression model
lin_reg_model3 = linear_model.LinearRegression()
lin_reg_model3.fit(X, y)

#######################################################################
#
#   Testing
#
########################################################################

#. Get test data
Xm,Ym = test_data._values[:,0],test_data._values[:,1]
# Use models to predict y vals on test data.
Yp = k_neigh_reg_model.predict(Xm[:,np.newaxis])
Yp3 = lin_reg_model3.predict(Xm[:,np.newaxis])

#######################################################################
#
#   Plotting
#
########################################################################

# Set up a figure
f = plt.figure(figsize=(12,5))
#  Add 2 plots side by side
ax = f.add_subplot(121)
ax2 = f.add_subplot(122)

#######################################
# Model 1 plot
#######################################
# Put test points in plot, color red
ax.scatter(Xm, Ym, c="r",marker="o")
# Put  predicted test points in plot color grren
ax.scatter(Xm, Yp, c="g",marker="s")
# Put training data in plot; let matplotlib choose the color.
# It will choose a color not yet used in this axis.
ax.scatter(X,y)
# plot parameters
ax.set_ylim([4.5,10])
ax.set_title('K Neighbor')
ax.set_xlabel('GDP per capita')
```

```
ax.set_ylabel('Life Sat')

########################################
# Model 2 plot
########################################
ax2.scatter(Xm, Ym, c="r",marker="o")
ax2.scatter(Xm, Yp3, c="g",marker="s")
ax2.scatter(X,y)
# plot parameters
ax2.set_ylim([4.5,10])
ax2.set_xlabel('GDP per capita')
ax2.set_title('Linear Regression')

# Not usually needed in a NB but okay
plt.show()
##########################################################################

oecd_bli, gdp_per_capita = backup
```



the blue points are the training data and the red points are the test data. The green points are the predicted locations of the red points (using the GDP, the x-value, to predict the Life Satisfaction, the y-value). Comparing the red points with the blue training data points, we see the test points are distinct outliers.

In the linear regression graph, as expected, the green points lie on a line chosen based on the blue points, the training data. The predicted K Neighbor model points do not lie on a line. K Neighbor is a more powerful, more flexible model and does a better job predicting here, in the sense that there is less spread between the red and green points. Despite the fact that the training data lacks high GDP countries, the K Neighbor model seems to capture the fact that life satisfaction levels off with GDPs above 60K.

Because the data does not lie on a line, the linear model ends up having two flaws. It underestimates the life satisfaction of poor folks in the test set, and overestimates that of rich folks.

In [ ]: