In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

Regression

In this notebook we will cover regression and regression metrics, using Linear Regression as our motivating example.

Regression is the general term for supervised learning problems where we try to predict the value of a continuous variable. That is, we want to build a model f that approximates the relationship between features X and labels y so that

$$f(X_j) \approx y_j$$

for each observation (X_j, y_j). Here the y_j are real numbers, because y is continuous.

Linear Regression

In a linear regression model, f is simply a linear function of the features. If X_{ji} is the ith feature of observation X_j, then

$$f(X_j) = \sum_{i=1}^{p} X_{ji}\beta_i + \beta_0$$

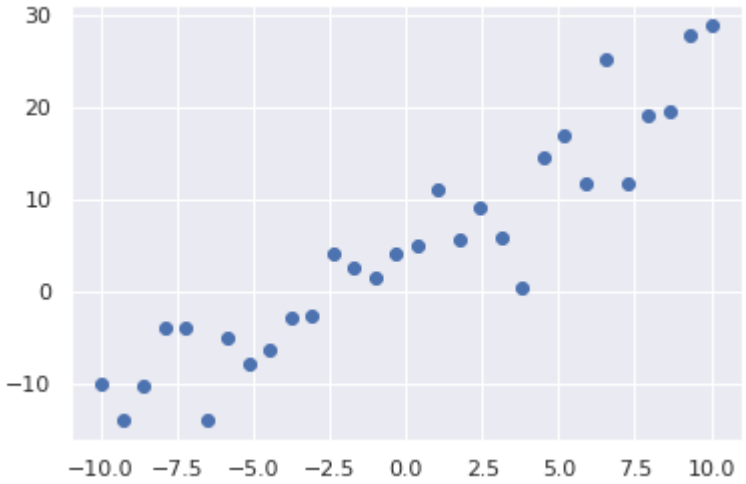
where the coefficients β_i and intercept term β_0 are values learned by the model during training, and p is the number of features. This is sometimes written more compactly as

$$f(X) = X \cdot \beta + \beta_0$$

where $X = (X_{ji})$ is a matrix, $\beta = (\beta_i)$ is a column vector, and β_0 is added elementwise to the result of the product.

Let's look at a simple example. First we'll generate random data with a single feature.

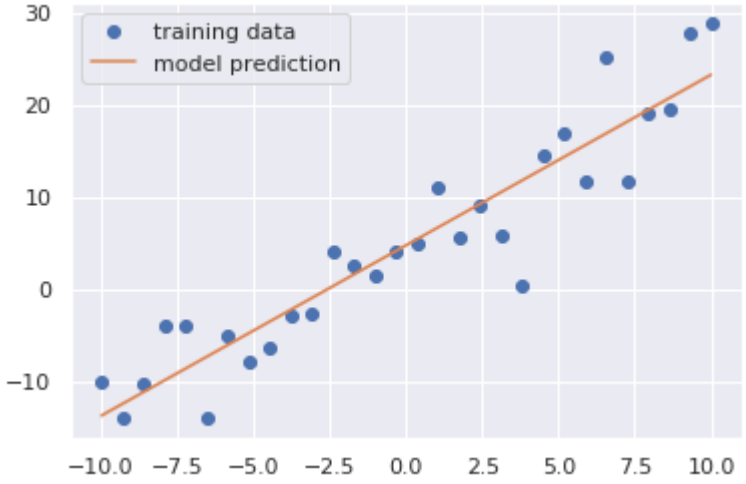
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0) # for consistency

X = np.linspace(-10,10,30)
y = 2 * X + 3 + 4*np.random.randn(X.shape[0])
plt.plot(X, y, 'o');
```



Training a linear model on this data means finding the coefficient β and intercept β_0 so that $f(X) = X \cdot \beta + \beta_0$ best approximates our label variable y. This is equivalent to finding the line $y = mx + b$ that best fits the plotted points.

In [3]:
```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X.reshape(-1,1), y) # reshape to column vector
print('Intercept = {:.2f}'.format(lr.intercept_))
print('Slope = {:.2f}'.format(lr.coef_[0]))

y_pred = lr.predict(X.reshape(-1,1))

plt.plot(X, y, 'o', label='training data')
plt.plot(X, y_pred, label='model prediction')
plt.legend();
```

```
Intercept = 4.77
Slope = 1.85
```



Regression Metrics

In the preceding discussion, we used phrases like "best approximates" and "best fits". In order to actually do machine learning, we need to be more concrete. That is, we need to define metrics that quantify how well a model fits a given set of data. Metrics provide a cost function to minimize during training and they act as a benchmark to evaluate trained models.

Mean Squared Error is the usual metric:

$$\frac{1}{n}\sum_j \left[f(X_j) - y_j\right]^2.$$

Unfortunately, this is susceptible to outliers. When this is an issue, **Mean Absolute Error** can be better:

$$\frac{1}{n}\sum_j \left|f(X_j) - y_j\right|.$$

You've probably heard of R^2 or the **Coefficient of Determination**. Although it's usually defined in a linear regression context, it's actually a very general idea: it measure the fraction of the error explained by the model f versus the fraction of the error explained by a naive model that assumes the mean value of y (i.e. the variance of y):

$$1 - \frac{\sum_j \left[f(X_j) - y_j\right]^2}{\sum_j \left(\overline{y} - y_j\right)^2} \qquad \text{where} \qquad \overline{y} = \frac{1}{n}\sum_j y_j\,.$$

Questions:

1. For a list of scalar values $z_1, \ldots, z_n$, the **mean** $\overline{z}$ is the quantity that minimizes the squared error:

$$\frac{d}{dz}\sum_j |z - z_j|^2 = 0$$

$$\frac{d}{dz}\left(Nz^2 - 2z(z_1 + z_2 + \ldots) + z_1^2 + z_2^2 + \ldots\right) = 0$$

$$2Nz - 2(z_1 + z_2 + z_3 + \ldots) = 0$$

$$z = \frac{z_1 + z_2 + z_3 + \ldots}{N} = \overline{z}$$

   Do you know what quantity comes from minimizing the absolute error?

$$\text{argmin}_z \sum_j |z - z_j|$$

   Does this help explain why Absolute Error is less susceptible to outliers?
2. How does each of these metrics scale as you scale the labels (y's) in our data set?

In [4]:
```
# Here are those metrics in scikit learn

from sklearn import metrics

print("Mean Absolute Error:", metrics.mean_absolute_error(y, y_pred))
print("Mean Squared Error:", metrics.mean_squared_error(y, y_pred))
print("R^2:", metrics.r2_score(y, y_pred))
```

```
Mean Absolute Error: 3.270344695208926
Mean Squared Error: 17.875032901714572
R^2: 0.8716155448895739
```

Note that Mean Absolute Error (MAE) and Mean Squared Error (MSE) are both difficult to interpret without context because they depend on the scale of the data. However, because R^2 has a fixed range, a value close to 1 always means that the model is fitting the data fairly well.

Optimization

As is typical in machine learning setups, training our model is equivalent to solving an optimization problem. For linear regression, we are trying to find the model parameters β_i which jointly minimize the Squared Error (or equivalently the Mean Squared Error) of our predictions. In theory we could do this using Gradient Descent, but in practice we take advantage of a direct mathematical solution.

To simplify the math, we'll absorb the intercept term β_0 into the coefficient vector β by pretending that our feature matrix X has an extra column (a dummy 0th feature) where every value is 1. Then the linear model can be written as $f(X) = X\beta$ and our optimization problem is

$$\min_{\beta} \|y - X\beta\|^2$$

where $\|z\|^2 = \|z\|_2^2 = \sum_i |z_i|^2$ is the square of the L^2 norm. This problem has a closed form solution given by

$$\hat{\beta} = (X^T X)^{-1} X^T y\,.$$

Questions:

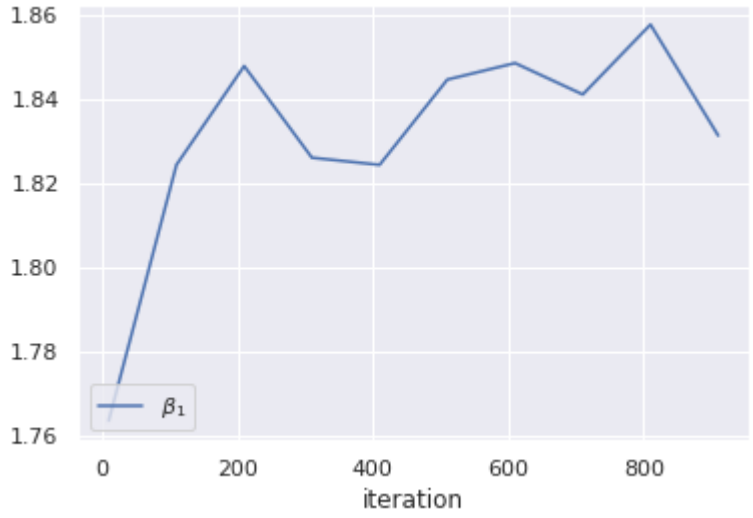
1. What about the intercept term?
2. Prove that the solution $\hat{\beta}$ actually minimizes the Mean Squared Error. (Hint: $X(X^T X)^{-1} X^T$ is the projection operator onto the subspace spanned by the columns of X).
3. What happens if $X^T X$ is singular, e.g. X has two columns that are co-linear. What does this mean in terms of identification? When might this occur in the data in real life?
4. What happens when $p \gg n$? How do you deal with this? (n is the number of observations and p is the number of features, so X is a $n \times p$ matrix.)
5. What is the effect of outliers? How do you deal with them?
6. What if y values are always positive? What if y values are in a fixed range $[a, b]$?

Stochastic Gradient Descent

Stochastic Gradient Descent is a optimization method which considers each training observation individually, instead of all at once (as normal gradient descent would). Instead of calculating the exact gradient of the cost function, it uses each observation to estimate the gradient and then takes a step in that direction. While each individual observation will provide a poor estimate of the true gradient, given enough randomness the parameters will converge to a good global estimate.

Because it need only consider a single observation at a time, stochastic gradient descent can handle data sets too large to fit in memory. Additionally, the training cost is essentially linear in the size of the training set.

In [5]:
```
from sklearn import linear_model

X = X.reshape(-1,1)

coefs = []
iterations = range(10,1000,100)
for n_iter in iterations:
    sgd_regressor = linear_model.SGDRegressor(random_state=42, max_iter=n_iter
).fit(X, y)
    coefs.append(sgd_regressor.coef_)

plt.plot(iterations, [c[0] for c in coefs], label=r'$\beta_1$')
plt.legend(loc=3)
plt.xlabel('iteration');
```

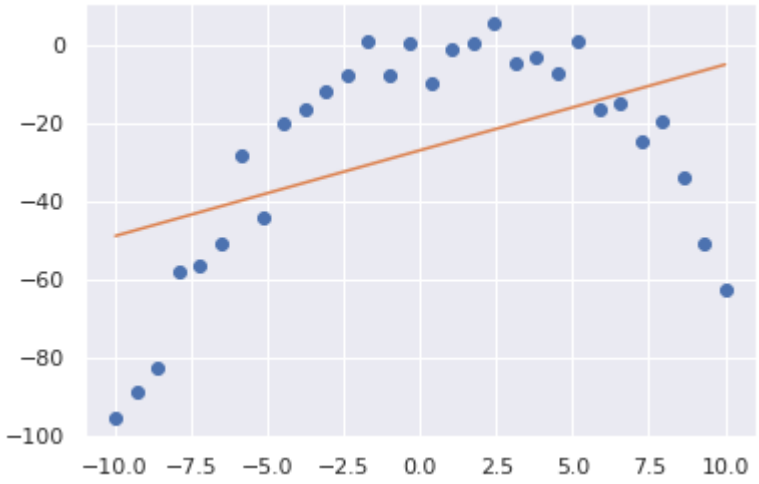


Adding Features

One apparent limitation of linear models is that they cannot capture non-linear behavior. For example, suppose we try to fit a linear model to data with a quadratic shape:

In [6]:
```
X = np.linspace(-10,10,30)
y = -0.8 * X * X + 2 * X + 3 + 8*np.random.randn(X.shape[0])

lr = LinearRegression()
lr.fit(X.reshape(-1,1), y) # reshape to column vector

y_pred = lr.predict(X.reshape(-1,1))

print("R^2:", metrics.r2_score(y, y_pred))

plt.plot(X, y, 'o')
plt.plot(X, y_pred);
```

```
R^2: 0.21264305016982188
```

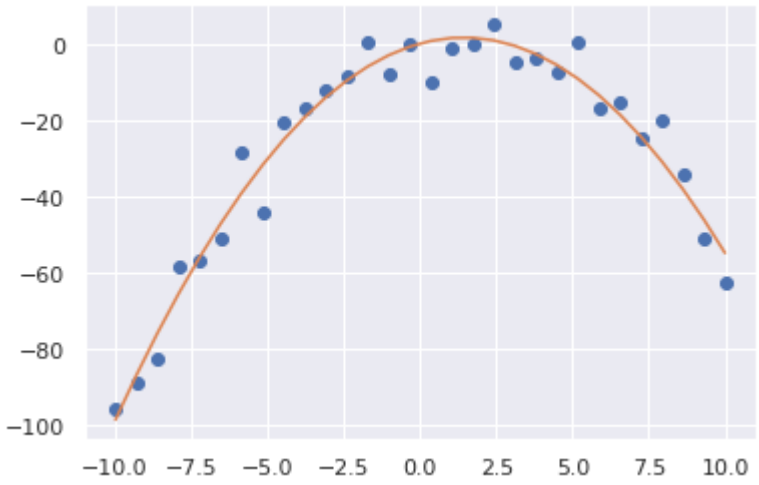


As expected, the model fits the data poorly. However, if we have a vague idea of what sort of pattern the data follows, then we can help our model by creating artificial features that encode the corresponding nonlinearities. In this case, it seems that the shape of the data is quadratic, so we can add a feature whose value is always equal to X^2.

In [7]:
```
X_quad = np.vstack((X,X**2)).T # Augmented feature matrix with columns X, X**2

lr.fit(X_quad, y)

y_pred = lr.predict(X_quad)

print("R^2:", metrics.r2_score(y, y_pred))

plt.plot(X, y, 'o')
plt.plot(X, y_pred)
```

```
R^2: 0.9581764214454389
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x7f5ced8444e0>]
```



Regularization

As we add complexity to a linear model by increasing the number of features, we also increase the danger of overfitting. One way to combat this tendency is to add an extra term to our cost function that penalizes model complexity. We judge model complexity by looking at the sizes of the model coefficients. One formulation, called **Ridge Regression** uses the combined cost function

$$||y - X\beta||^2 + \alpha||\beta||^2$$

where $||z||^2 = \sum_i |z_i|^2$ and α is a constant (called the **regularization parameter**) chosen by the modeler. Minimizing this cost function means finding a balance between fitting the training data and keeping model complexity low. Lower values of α emphasize the first of these priorities, while higher values emphasize the latter.

The penalty term used in Ridge may seem arbitrary at first. There are alternatives worth considering (for example Lasso Regression (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html)), but it turns out that Ridge can be rigorously motivated on statistical grounds and that the corresponding optimization problem has a closed form solution. See the reference section at the end of this notebook for more details.

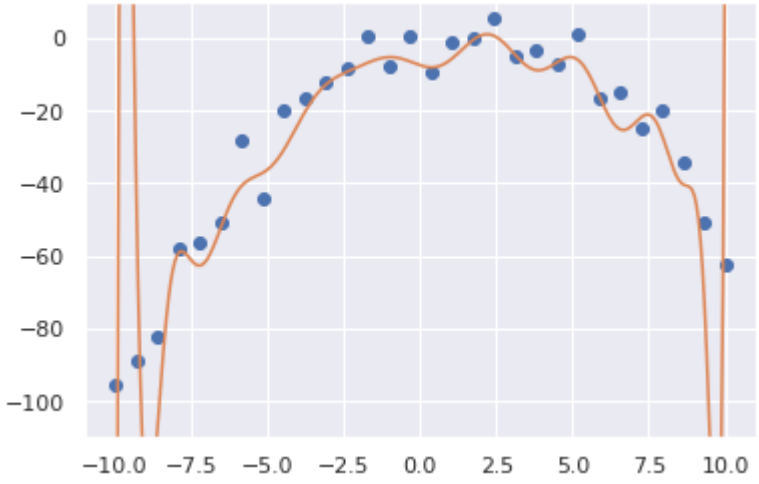
Questions:

1. What may go wrong if the features do not have comparable scales? What should you do to prevent this?
2. Compared with linear regression, how to you expect the β's to behave? How does this behavior change as you vary α?

Let's look at an example. If we add more polynomial features to the quadratic model discussed above, then we may grossly overfit the data:

In [9]:
```
from sklearn.preprocessing import StandardScaler

def extra_features(array_, max_power=20):
    '''
    Generates higher order polynomial features X, X**2, ... , X**N
    Then scales the columns to have mean 0 and standard deviation 1
    '''
    new_features = np.vstack(array_**i for i in range(1,max_power)).T
    return StandardScaler().fit_transform(new_features)

X_fine = np.linspace(-10,10,300) # Additional X values so that we can see pred
iction behavior between data points

lr = LinearRegression()
lr.fit(extra_features(X),y)
y_pred = lr.predict(extra_features(X_fine))

plt.ylim(-110,10)
plt.plot(X, y, 'o')
plt.plot(X_fine, y_pred);
```

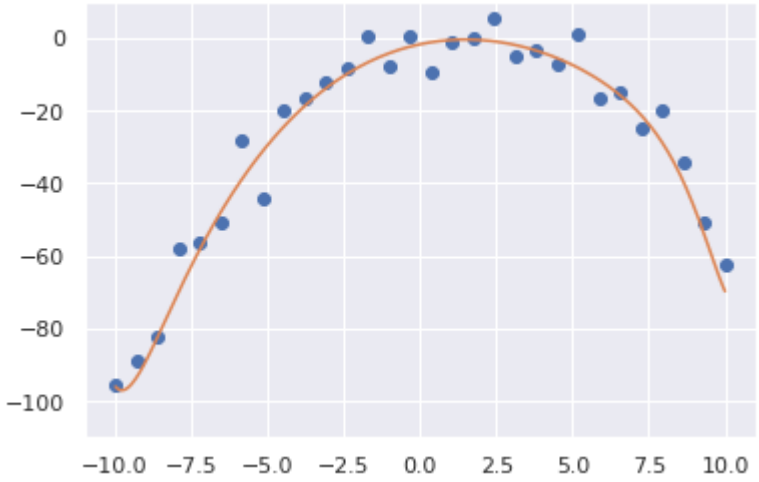
```
/opt/conda/envs/data3/lib/python3.6/site-packages/ipykernel_launcher.py:8: Fu
tureWarning: arrays to stack must be passed as a "sequence" type such as list
or tuple. Support for non-sequence iterables such as generators is deprecated
as of NumPy 1.16 and will raise an error in the future.
```



However, we can address this overfitting by using Ridge regression, implemented as `sklearn.linear_model.Ridge` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html) in Scikit-Learn.

In [10]:
```
from sklearn.linear_model import Ridge

ridge_est = Ridge(alpha=0.1)
ridge_est.fit(extra_features(X),y)
y_pred = ridge_est.predict(extra_features(X_fine))

plt.ylim(-110,10)
plt.plot(X, y, 'o')
plt.plot(X_fine, y_pred);
```

```
/opt/conda/envs/data3/lib/python3.6/site-packages/ipykernel_launcher.py:8: Fu
tureWarning: arrays to stack must be passed as a "sequence" type such as list
or tuple. Support for non-sequence iterables such as generators is deprecated
as of NumPy 1.16 and will raise an error in the future.
```



Example: California Housing Data Set

Now that we have a thorough understanding of Linear Regression, let's work through a real world example with multiple features.

We are going to explore a well-known data set called the *California Housing Data*. This is a useful data set for building and benchmarking models. The data set contains aggregated data on housing values and characteristics by census block. Researchers can study the data to predict the median value of homes based on the other variables.

This is a standard data set and comes with Scikit Learn. However, we have already downloaded it and put it in a json file just in case of connectivity issues.

In [11]:
```
import gzip
import json

cali_data = json.load(gzip.open('small_data/cal_house.json.gz'))

X = cali_data['data']
y = cali_data['target']
print(cali_data['DESCR'])
```

```
California housing dataset.

The original database is available from StatLib

    http://lib.stat.cmu.edu/

The data contains 20,640 observations on 9 variables.

This dataset contains the average house value as target variable
and the following input variables (features): average income,
housing average age, average rooms, average bedrooms, population,
average occupation, latitude, and longitude in that order.

References
----------

Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
Statistics and Probability Letters, 33 (1997) 291-297.
```

Let's add the data to a DataFrame to make it easier to use.

In [12]:
```
from pandas import DataFrame, Series

names = cali_data['feature_names']

data_dict = dict(zip(names, ['Median income', 'House age', 'Average # of room
s', 'Average # of bedrooms', 'Population', 'Average occupancy', 'Latitude', 'L
ongitude']))

cali_df = DataFrame(cali_data['data'], columns=names)

home_values = Series(cali_data['target'])
```

In [13]:
```
cali_df.head()
```

Out[13]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitu
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

In [14]:
```
home_values.head()
```

Out[14]:
```
0    4.526
1    3.585
2    3.521
3    3.413
4    3.422
dtype: float64
```

We can familiarize ourselves with the data by plotting it. Let's have some fun exploring data using `IPython` widgets.

Experiment with the dropdown to plot each column vs. Median home value.

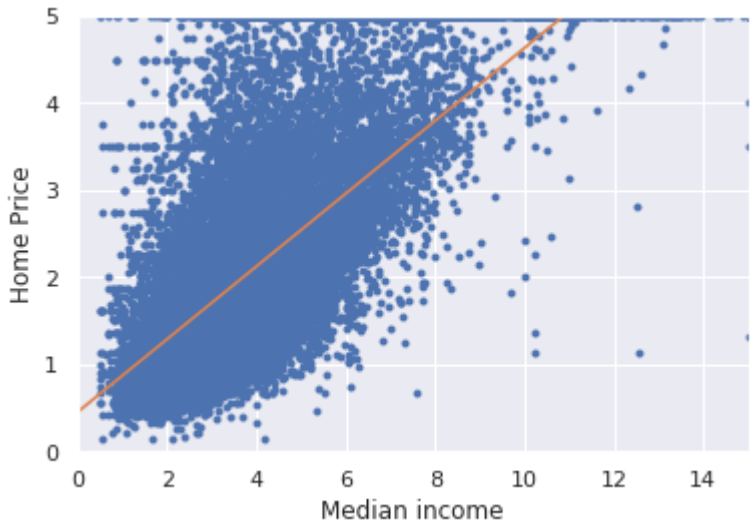
Question: Which columns seem to have somewhat of a linear relationship with home values?

In [15]:
```
from ipywidgets import widgets

def cali_plot(column):
    plt.plot(cali_df[column], home_values, '.')
    plt.xlabel(data_dict[column])
    plt.ylabel('Home Price')

dropdown_values = {"{0}: {1}".format(k, v):k for k, v in data_dict.items()}

widgets.interact(cali_plot, column=dropdown_values);
```

The median income feature in particular seems to have a linear relationship with home price.

Let's use Linear Regression to find the line that best fits this trend.

In [16]:
```
linreg = linear_model.LinearRegression(fit_intercept=True)  # fit_intercept=True is the default value
linreg.fit(cali_df[['MedInc']], home_values)
x = np.linspace(-1, 15).reshape(-1,1)

plt.plot(cali_df['MedInc'], home_values, '.')
plt.plot(x, linreg.predict(x), '-')
plt.ylim(0, 5)
plt.xlim(0, 15)
plt.xlabel(data_dict['MedInc'])
plt.ylabel('Home Price');
```



We can get the R^2 score of this model using the `.score` method.

In [17]:
```
linreg.score(cali_df[['MedInc']], home_values)
```

Out[17]:
```
0.473447491807199
```

This means that the model based on median income explains about 47% of the variance in home prices.

Next, let's build a more complicated linear model by using all of the features instead of just median income.

In [18]:
```
linreg = linear_model.LinearRegression()

linreg.fit(cali_df, home_values)
```

Out[18]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

This model learns a coefficient for each feature as well as an intercept term, and then makes predictions by returning the corresponding linear combination.

In [19]:
```
print("coefficients: ", linreg.coef_, "\n")
print("intercept:", linreg.intercept_, "\n")
print(("prediction = " +
       "{0} +\n".format(linreg.intercept_) +
       " +\n".join(["{1} * {0}".format(n, f) for n, f in zip(names, linreg.coe
f_)])))
```

```
coefficients:  [ 4.36693293e-01  9.43577803e-03 -1.07322041e-01  6.45065694e-
01
 -3.97638942e-06 -3.78654265e-03 -4.21314378e-01 -4.34513755e-01]

intercept: -36.94192020720651

prediction = -36.94192020720651 +
0.43669329313386046 * MedInc +
0.009435778033236638 * HouseAge +
-0.10732204139025008 * AveRooms +
0.6450656935165835 * AveBedrms +
-3.9763894212358625e-06 * Population +
-0.003786542654959176 * AveOccup +
-0.4213143775275795 * Latitude +
-0.43451375467510645 * Longitude
```

Note that the R^2 score has improved:

In [20]:
```
linreg.score(cali_df, home_values)
```

Out[20]:
```
0.6062326851997475
```

This means that the new model fits the data better, but that's not necessarily a cause for celebration.

- Adding too many features to a model can lead to overfitting. The fact that our model fits the training data well doesn't necessarily mean that it will generalize to new data. The proper way to check for this issue is to do a train-test split on the data before training the model (see the overfitting notebook for more details). Although we omitted this step here for simplicity, you can take our word that this particular model is not overfit.
- Linear models aren't able to make good use of features that don't vary linearly with the target variable. This means that some features (like house age) don't make a meaningful contribution to the model and that some features (like latitude and longitude) may have non-linear information that's being wasted.

One way to incorporate non-linear information is to create new features. For example, if you take another look at the plots for latitude and longitude, you'll see that there are two prominent bands in each. These correspond to the locations of Los Angeles and San Francisco, and we can create new features which reflect the distance from each city.

In [21]:
```
import numpy as np

def distance_from(cit, lat, lon):
    if cit == "LA":
        lat_0, lon_0 = 34.05, -118.24
    if cit == "SF":
        lat_0, lon_0 = 37.77, -122.42
    return np.sqrt((lat-lat_0)**2 + (lon-lon_0)**2)

for cit in ["LA","SF"]:
    cali_df[cit] = distance_from(cit, cali_df["Latitude"], cali_df["Longitude"
])

cali_df.head()
```

Out[21]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitu
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

It turns out that the minimum of these distances has some correlation with home prices; homes which are further away from a city are cheaper on average.

In [22]:
```
cali_df["Closest"] = cali_df[["LA", "SF"]].min(axis=1)

plt.plot(cali_df["Closest"], home_values, '.')
plt.xlabel("Distance To Closest City")
plt.ylabel('Home Price');
```

We can do almost as well as our full model from before by using only median income and distance to closest city as features.

In [23]:
```
linreg = LinearRegression()
linreg.fit(cali_df[["MedInc", "Closest"]], home_values)
linreg.score(cali_df[["MedInc", "Closest"]], home_values)
```

Out[23]: 0.5566109677731299

And a full model based on our expanded feature set performs marginally better than before.

In [24]:
```
linreg = linear_model.LinearRegression()
linreg.fit(cali_df, home_values)
linreg.score(cali_df, home_values)
```

Out[24]: 0.6169511095245312

A few notes:

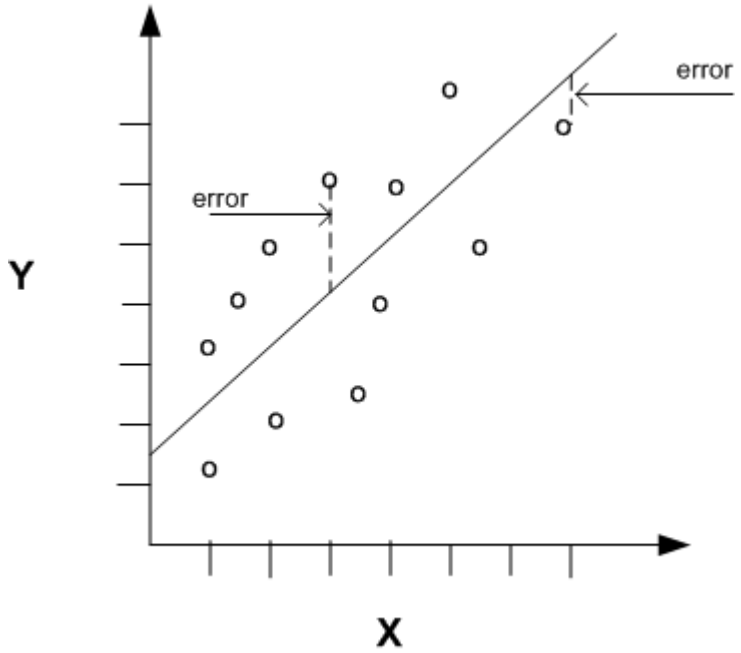
1. A good baseline is to see how well a mean model performs. That is, take a model that predicts `y.mean()` and whose MSE is going to be `y.var()`. The mean model always has an R^2 of 0, but that doesn't automatically mean it makes poor predictions.
2. How many (original) features have a correlation coefficient > .6? These explain the majority of the error (compared with the baseline model).
3. One way to prevent this overfitting is to choose only those features $X_{\cdot i}$ that are highly correlated with y. This can lead much better models.

Question:

1. We tried to predict y but since it is non-negative, it might make sense to predict $\log(y)$. What metric would you use to be able to evaluate which one is better?

Reference: Statistical Motivation

We can also think of Linear Regression as arising from a statistical model in which the true behavior of data is linear, but each observation has some noise added in the form of an error term.



Different error distributions give us different classes of the General Linear Models (GLM)s. To learn more about GLMs, there are a good set of notes available here (http://data.princeton.edu/wws509/notes/a2.pdf).

Here we'll assume that the errors are independent and normally distributed with standard deviation σ. Our goal is to use Maximum Likelihood Estimation to estimate the coefficients of the underlying linear function $f(X) = X\beta$.

If the coefficient vector $\beta = (\beta_i)$ is known, then the probability of measuring $y = (y_j)$ is simply

$$P(y \mid \beta) = \prod_j \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\left(\frac{X_j \cdot \beta - y_j}{2\sigma} \right)^2 \right] .$$

However, we don't know β. Instead we want to find it, given y, by finding the β that maximize $P(\beta \mid y)$. Thanks to Bayes' Rule, we know

$$P(\beta \mid y) = P(y \mid \beta) \frac{P(\beta)}{P(y)} .$$

We know the first term on the right hand side, and $P(y)$ is independent of β, leaving only $P(\beta)$ unknown. In linear regression, we suppose we have no *a priori* knowledge of the expected coefficients and take $P(\beta)$ to be constant as well. Thus, the most probable model is determined by maximizing the likelihood function

$$L(\beta) = \prod_j \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\left(\frac{X_j \cdot \beta - y_j}{2\sigma} \right)^2 \right] \propto P(\beta \mid y) .$$

Since $\log$ is monotonic, we can also maximize the log-likelihood. A few calculations show us that the negative log-likelihood (up to a linear transformation) is

$$-\log(L(\beta)) \sim \|y - X\beta\|^2 .$$

Here, $\|z\|^2 = \|z\|_2^2 = \sum_i |z_i|^2$ is the square of the L^2 norm. The objective is to minimize this quadratic:

$$\min_\beta \|y - X\beta\|^2 .$$

This is the familiar expression for squared error, which means that we've recovered the optimization objective we were using before. Of course, this means that we can use the same closed form solution:

$$\hat{\beta} = (X^T X)^{-1} X^T y .$$

In addition to theoretically motivating the use of Mean Squared Error, we can motivate Ridge Regression by revisiting the assumption that $P(\beta)$ is constant.

In the California example, we used an *ad-hoc* criteria to select features. Essentially, this reflects an expectation that most coefficients should be zero. A more principled approach is to choose a prior distribution for $P(\beta_i)$ that is peaked about $\beta_i = 0$, instead of uniform. We'll start by taking the coefficients to be identical independently normally distributed about 0 with a standard deviation of $\sigma/\sqrt{\alpha}$, where α is a hyperparameter:

$$P(\beta) \propto \prod_i \exp\left[-\frac{\alpha}{2}\left(\frac{\beta_i}{\sigma}\right)^2\right],$$

so

$$L(\beta) \propto \prod_j \exp\left[-\frac{1}{2}\left(\frac{X_j \cdot \beta - y_j}{\sigma}\right)^2\right] \prod_i \exp\left[-\frac{\alpha}{2}\left(\frac{\beta_i}{\sigma}\right)^2\right].$$

Then the negative log-likelihood is (up to a linear transformation)

$$-\log(L(\beta)) \sim \|y - X\beta\|^2 + \alpha\|\beta\|^2 \,.$$

As promised, this is the cost function used in Ridge Regression. The corresponding optimization problem has a closed form solution:

$$\hat{\beta} = (X^T X + \alpha I)^{-1} X^T y \,.$$

To get some motivation for what's happening, use the *singular value decomposition*

$$X = U\Sigma V^T$$

We can see that

$$\hat{\beta} = VDU^T y$$

where

$$D_{ii} = \frac{\Sigma_{ii}}{\Sigma_{ii}^2 + \alpha} \,.$$

When $\alpha = 0$, $D_{ii} = \frac{1}{\Sigma_{ii}}$ and it decreases to 0 as $\alpha \to \infty$. The smaller Σ_{ii}, the faster this decrease to 0 (for a given level of α). So smaller Σ_{ii} are "shrunk" faster than larger Σ_{ii} and we get the "significant values" are left.

Questions:

1. Can you prove the formula for $\hat{\beta}$ for Ridge Regression from ordinary Linear Regression?
2. What is the corresponding prior for plain-vanilla linear regression?

Exit Tickets

1. How would you assess whether a relationship is actually linear?
2. If instead of being able to observe y, you observe a noisy estimate of $y \pm \epsilon$ with unbiased normally distributed noise. What is the effect on your estimates β?
3. When you loaded your data, you unwittingly loaded each row of the data (both X and y) twice and performed the same regression. What is the effect on your estimates β?
4. When you loaded your data, you unwittingly loaded each column of the features (just X) twice and performed the same regression. What is the effect on your estimates β?
5. Everything we've talked about so far involves loading all the data into memory. What if you have more data than you can fit into memory?

Spoilers

Answers

Optimization

1. Add a column of ones, or subtract off the average value of y.
2. The hint pretty much gives you the answer since a projection of a point onto a plane is the closest you can get to that point while still remaining on the plane. That being said, you can project the global minimum onto the accessible subspace. Alternatively, you can differentiate the matrix expression and take the first-order condition and find the zero.
3. The matrix can't be inverted. In practice, this shows up as numerical instabilities. This will happen if two columns are measuring the same thing, or if one column is a linear combination of two others.
4. When $p > n$, X will be degenerate, so it can't be (pseudo-)inverted and you no longer have a unique β. To deal with this, you can reduce the number of features with PCA or use regularization.
5. Outliers can really skew the results of β because of the quadratic penalty. Remember, that minimizing the least squares is essentially looking for a mean, which is affected by outliers. You can transform the model via quantiles to reduce the effect of noise, bin the data, or use floors and caps on the data.
6. For non-negative y, try using the $\log(y)$. If y is always within a fixed $[a, b]$, use
$$\frac{y - a}{b - a}.$$
Alternatively, scale by the mean / range.

Regularization

1. The largest features will be under penalized because the corresponding coefficients are smaller relative to how much they impact the model. Scale the features beforehand, so that coefficient size gives importance.
2. Increasing α shrinks the terms of β towards zero, with smaller values of β shrunk faster.

Example: California Housing Data Set

1. Look at coefficient of determination; plot residuals and look for a pattern.

Reference: Statistical Motivation

1. The formula for $\hat{\beta}$ can be deduced by completing the square. Then the problem looks exactly like an ordinary Least Squares problem with a different X matrix.
2. Recall that the prior is
$$\exp\left[-\left(\alpha\frac{\beta}{2\sigma}\right)^2\right]$$
when $\alpha = 0$, this is a flat "improper" prior (it's not really a distribution). This is often what a Bayesian calls improper.

Exit Tickets

1. To assess if the relationship is linear, plot the distribution of the residuals as a function of x. If there's a systematic bias, take a look at it and see what's going on.
2. With extra (unbiased) noise, the estimate of β does not change (on average), but the the confidence goes down.
3. Loading rows twice has no effect on β but it does artificially increase your confidence (dividing it by a factor $\sqrt{2}$)
4. The problem becomes degenerate and β_j is now split between $\beta_{j'}$ and $\beta_{j''}$ such that $\beta_j = \beta_{j'} + \beta_{j''}$.
5. All of these problems can be solved using gradient descent, which only requires a *stream* of data, rather than the entire data set. Linear regression (with either L^2, Huber penalty, epsilon insensitive) can be solved using `sklearn.linear_model.SGDRegressor` and logistic regression can be solved using `sklearn.linear_model.SGDClassifier`. These methods implement a `partial_fit` method, which can iteratively updates the coefficients on small chunks of data. In this case, you are no longer ram constrained, but constrained in the amount of time it takes to read data from disk.