Linear Regression Prediction

Prediction in Linear Regression

- Linear regression equation

$$\hat{y} = \theta_0 + \theta_1\, x_1 + \theta_2\, x_2 + \ \ldots + \theta_n x_n$$

- $\hat{y}$ is the predicted value
- n is the number of features
- θ_j is the j^{th} model parameter, which represents the feature weights from 1 to n
- θ_0 is the bias term

Linear Equation in Vector Form

- Prediction of linear regression model in vectorized form

$$\hat{y} = h_0(\boldsymbol{x}) = \boldsymbol{\theta} \cdot \boldsymbol{x}$$

- θ is the model's parameter vector
- $\boldsymbol{X}$ is the instance's feature vector
- $\boldsymbol{\theta} \cdot \boldsymbol{x}$ is the dot product of vectors Theta and x

Minimizing the Cost Function

- Training a model tweaks the parameters (θ) to obtain the best fit on the training set
- Training the data involves coming up with a performance measure to determine how well the model is doing
- A common performance measure for regression is mean squared error (MSE)

The Linear Cost Function: MSE

- The MSE cost function for linear regression

$$MSE(\theta) = \sum_{i=1}^{m} \frac{1}{m} (\theta^T x^{(i)} - y^{(i)})^2$$

- $\theta^T x^{(i)}$ is our prediction, $\hat{y}$ on the previous slide
- In layman's terms: we want to reduce the error

The End

Solutions for Minimizing Error in Linear Regression

Possible Solutions

- There are several different ways to minimize the cost function during training:
  - The normal equation
  - Calculation of the pseudoinverse
  - Gradient descent
- Let’s talk about each of these in more detail

The Normal Equation

- If there are more examples than features, (a narrow and tall table) there is an eloquent vector solution to solving for the minimized parameters directly
- Here is the vector solution for the normal equation:

$$\hat{\theta} = (x^T x)^{-1} x^T y$$

Calculating the Intercept

- Recall that θ_0 represents the intercept of our linear equation
- In order to solve for the coefficients and the intercept, we add a column of ones to our feature vector
- The column of ones will solve the intercept when we do the dot product of the normal equation

Scikit-learn make_regression

- Scikit-learn has a make_regression function that creates a linear relationship between dependent and independent variables
- It is part of the sklearn.datasets module

Using make_regression

```
: X, y, coef = make_regression(n_samples=100,n_features=1,noise=1.0,bias=4.0,coef=True)
  print(coef)
  print(X.shape)
  print(y.shape)
```

```
61.99638134842075
(100, 1)
(100,)
```

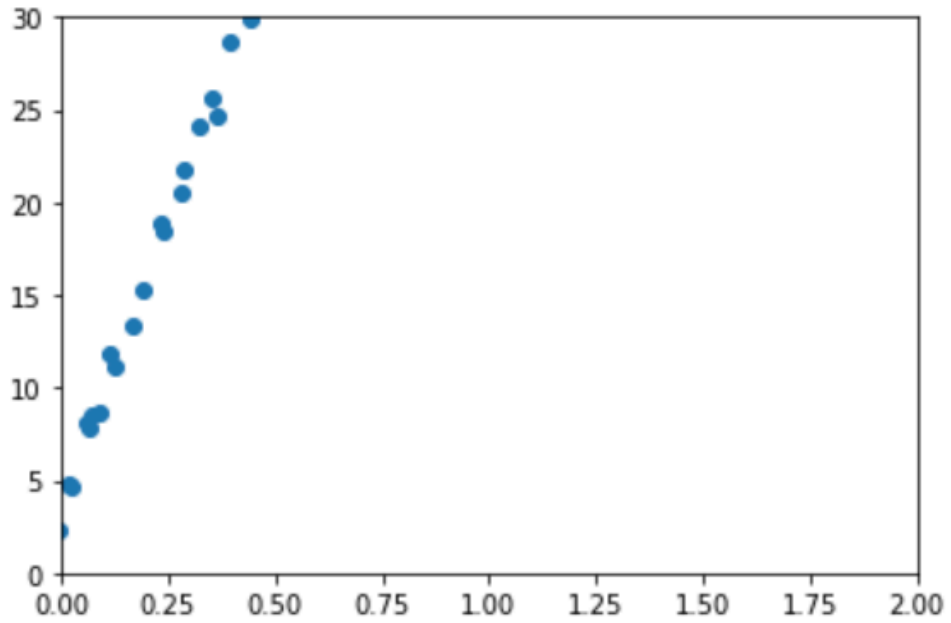
- n_samples is the number of samples
- n_features is the number of features
- noise is the standard deviation of the noise
- bias is the y intercept
- coef stores the coefficients

Visualizing the Data

- Here is the data shown in a matplotlib scatter plot

```
plt.scatter(X,y)
plt.ylim(0,30)
plt.xlim(0,2)
```

```
(0, 2)
```



Steps to Implement the Normal Equation

- First, add an extra column of ones:
- X_b = np.c_[np.ones((100,1)), X]

```
X_b = np.c_[np.ones((100,1)),X]
print(X_b[:5])
print(X_b.shape)
```

```
[[ 1.          -1.02580973]
 [ 1.          -1.96484304]
 [ 1.           1.3420512 ]
 [ 1.           0.06893198]
 [ 1.          -3.35234614]]
(100, 2)
```

Steps to Implement the Normal Equation (cont.)

- Now, implement the normal equation as theta_best:

```
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

- The output of theta_best is as such:

```
: theta_best

: array([ 4.00651602, 61.94686523])
```

What Happens if We Don't Add a Column of Ones?

- If we don't add a column of ones and just use X in the normal equation we only get the output for the coefficients, not the intercept:

```
theta1_best
```

```
array([62.29277014])
```

Let's Make Predictions Using theta_best

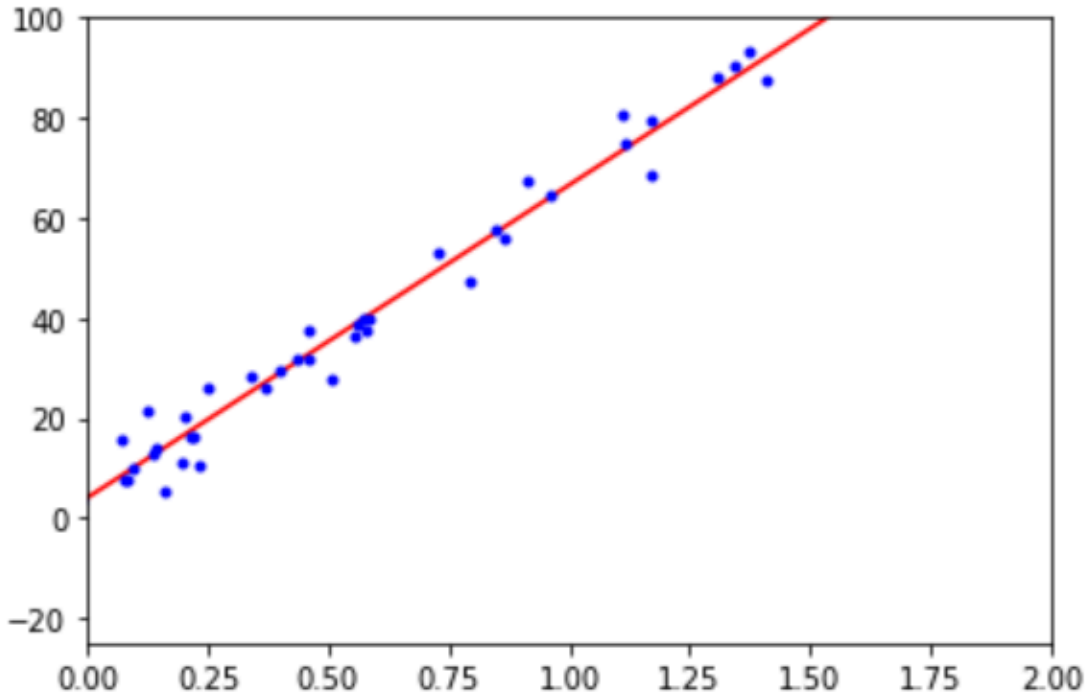
```
X_new = np.array([[0],[2]])
print(X_new)
X_new_b=np.c_[np.ones((2,1)), X_new]
print(X_new_b)
y_predict = X_new_b.dot(theta_best)
print(y_predict)
```

```
[[0]
 [2]]
[[1. 0.]
 [1. 2.]]
[  4.04034882 129.13347309]
```

And Plot the Regression Line

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X,y,"b.")
plt.xlim(0,2)
plt.ylim(-25,100)
```

```
(-25, 100)
```



Linear Regression in sckit-learn

- From sklearn.linear_model import Linear Regression
- lin_reg = LinearRegression()
- Lin_reg.fit(X,y)
- View the resulting intercept and coef by calling intercept_ and coef_

```
lin_reg.intercept_, lin_reg.coef_
```

```
(4.040348816028617, array([62.54656214]))
```

Psuedoinverse of X

- $(X^TX)^{-1}\ X^T$ is called the pseudoinverse of X
- The linear regression model of scikit-learn uses singular value decomposition to calculate the pseudoinverse
- This is more efficient than computing the normal equation and even works when you have more features than examples
- But what if you can't fit your data in memory?

The End

Gradient Descent Introduction

Gradient Descent

- Tweak parameters iteratively to minimize a cost function
- Finds which direction is down and takes a step that direction
- The gradient will be 0 at the bottom

The Learning Rate

- Start by filling theta with random values
- Minimize the cost function gradually
- How quickly you go down hill is determined by the learning rate
- Can’t be too small or it will take too long
- Can’t be too big or it will jump right over the minimum
- This is a hyperparameter you can set in your model

Challenges

- Local minimum
- Plateaus
- Good news! MSE is convex
- But you might need to scale if features are of very different sizes

The End

Types of Gradient Descent

Batch Gradient Descent

- Based on partial derivatives
- How much will the cost function will change if you change θ just a little?
- Batch gradient descent does this across the entire data set during every step
- To go down you go the opposite of the gradient vector (subtract it from theta) multiplied by the learning rate
- But it is still faster than calculating the normal equation

Stochastic Gradient Descent

- As stated before, batch gradient descent does matrix calculation of the entire training set at every step making it slow
- Stochastic gradient descent picks a random example in the training set and computes gradients on that one instance
- Only problem is that it will bounce up and down as it decreases on average

Finding the Global Minima

- The fact that stochastic gradient descent bounces around can help it jump out of a local minima if it encounters one
- To combat jumping around the global minima once it is reached, one can gradually reduce the learning rate
- This is done by specifying a learning schedule for each iteration

Implementing Stochastic Gradient Descent in scikit-learn

- Using get_params you can see the settings for the SGD Regressor
- eta0 is the initial learning rate

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor()
sgd_reg
sgd_reg.get_params()
```

```
{'alpha': 0.0001,
 'average': False,
 'early_stopping': False,
 'epsilon': 0.1,
 'eta0': 0.01,
 'fit_intercept': True,
 'l1_ratio': 0.15,
 'learning_rate': 'invscaling',
 'loss': 'squared_loss',
 'max_iter': 1000,
 'n_iter_no_change': 5,
 'penalty': 'l2',
 'power_t': 0.25,
 'random_state': None,
 'shuffle': True,
 'tol': 0.001,
 'validation_fraction': 0.1,
 'verbose': 0,
 'warm_start': False}
```

SGD Hyperparameters

- max_iter is the maximum number of epochs the model will run for
  - Default is 1000
- Penalty is the regularization hyperparameter
  - Default is l2

Use set_params

- Let's set eta0 to 0.1 and penalty to none

```
sgd_reg.set_params(eta0=0.1,penalty=None)
```

```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=None,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

Fitting the Data and Viewing the Result

- Here is the fit statement:

```
sgd_reg.fit(X, y)
```

```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.1, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty=None, power_t=0.25, random_state=None,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

- And here is the result:

```
sgd_reg.intercept_, sgd_reg.coef_
```

```
(array([3.99472681]), array([62.72131102]))
```

Mini-Batch Gradient Descent

- Mini-batch strikes middle ground between batch gradient descent and stochastic gradient descent
- The gradients are computed on small random sets of instances called mini-batches

The End

Polynomial Regression

Polynomial Regression

- Polynomial regression uses a linear model to fit nonlinear data by adding powers of each feature as new features in the training set
- You then train a model on the extended set of features

A Nonlinear Example

- Lets' build some nonlinear data we can test on
- We'll plot 100 points

  m = 100

- Make x a linear equation:

  X= 5 * np.random(m,1) - 2

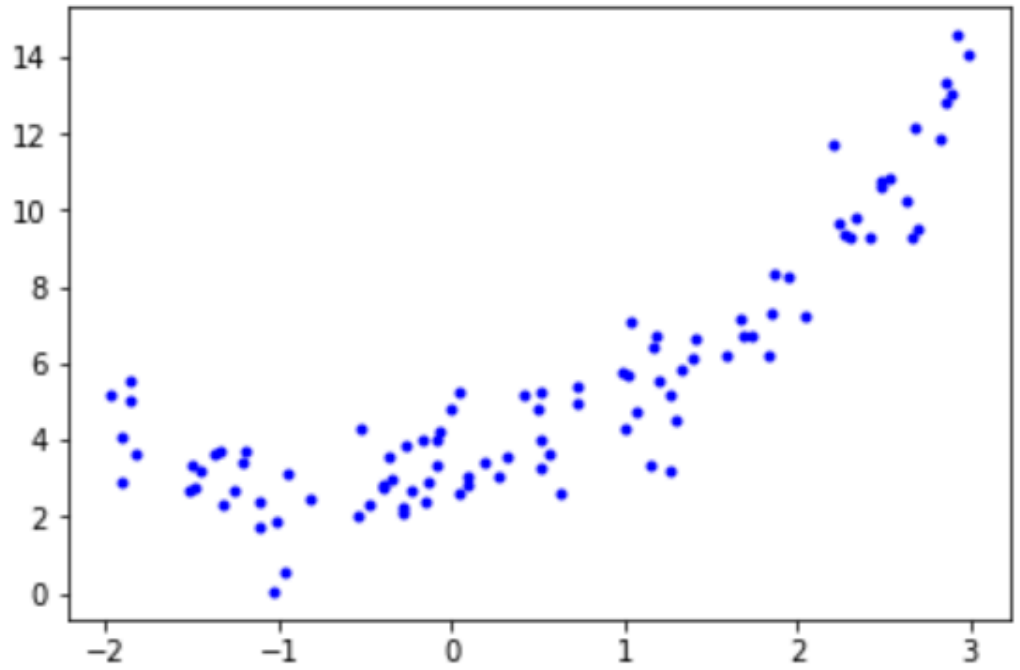
- Make y a quadratic equation

  y = 0.5 * X**2 +X+2 + np.randmn(m,1)

Now Graph It:

```
plt.plot(X,y, "b.")
```

```
[<matplotlib.lines.Line2D at 0x1a1fff0630>]
```



Estimating Polynomial Features

- Import your packages:

  ```
  from sklearn.preprocessing import PolynomialFeatures
  ```

- Set your degree to 2 since we are trying to solve a quadratic equation:

  ```
  poly_features = PolynomialFeatures(degree=2,
  include_bias=False)
  ```

- Fit and transform on X

  ```
  X_poly = poly_features.fit_transform(X)
  ```

  - X_poly returns two values: one for x and one for x^2

Fit a Linear Regression on Extended Training Data

- Instantiate a linear regression model and fit it on our extended featureset:

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

- Let's look at the intercept and coefficients

```
lin_reg.intercept_,lin_reg.coef_
```

```
(array([3.14492534]), array([[1.04800273, 0.77090152]]))
```

Predicting on New Data

- Create a new set of points to predict on:

  X_new = np.linspace(-3,3,100).reshape(100,1)

- Transform X-new into a quadratic:

  X_new_poly=poly_features.transform(X_new)

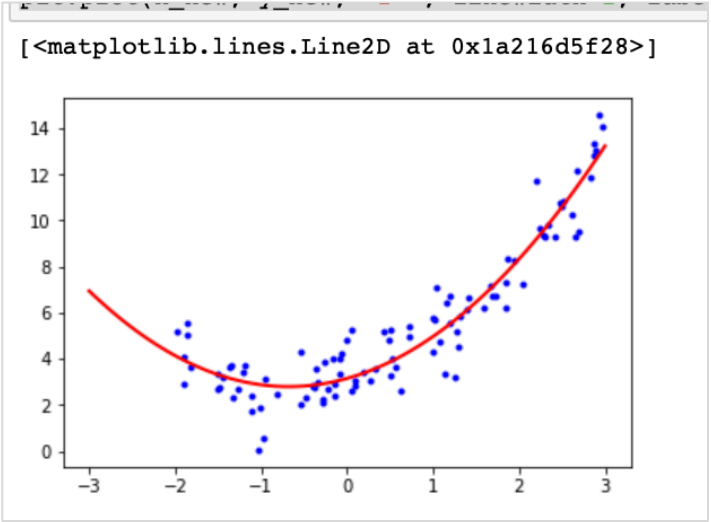
  - This returns an array with 2 features
  - Notice we are only doing a transform, not fit_transform

- Predict with our linear regressor:

  y_new=lin_reg.predict(X_new_poly)

What Does It Look Like?

- Let's plot our initial data again:
  - plt.plot(X,y, "b.")
- Let's plot our prediction:
  - plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
- Here is the chart



The End

Introduction to Learning Curves

Learning Curves

- Plots a model's performance on training and validation sets as a function of training set size
- The model is trained several times on different sized subsets of data
- The error on the training set starts low, because the problem is easy to solve

Learning Curves (cont.)

- The error increases as the size of the subset grows until it plateaus because it can’t be solved with a linear equation
- On the validation set the RMSE starts high and goes down as the model learns
- Again you will see the error plateau as it becomes clear the linear model can’t minimize it any more

Interpreting Learning Curves

- A learning curve where both the training and evaluation curves plateau but are close and fairly high is a sign of underfitting
- Overfitting on the training set causes its curve to be much lower than the error on the validation data
- Also, the gap between the curves on the chart will be much larger

The End

Regularized Linear Models

Regularized Linear Models

- Regularization in linear models seeks to limit the size of the weights of the model
- There are three types of regularization for linear models
  - Ridge regression
  - Lasso regression
  - Elastic net
- Another way to implement regression is to implement early stopping

Ridge Regression

- Ridge regression adds a new term to the loss function
- The term squares the sum of the parameter vector and scales the amount of shrinkage with a new hyperparameter called alpha (α)
- The regularization term is only added to the loss function during training

Notes on Ridge Regression

- Ridge regression is also called l2 normalization (because we are squaring the vector)
- Scaling the data is important because squaring the parameter vector makes it susceptible to outliers
- Increasing α leads to flatter predictions
- Increasing α reduces variance but increases bias

Training in Ridge Regression

- The vector-based closed form solution of Ridge Regression adds an identity matrix in the inverse portion of the normal equation

New Term

$$\hat{\theta} = (X^T X + \alpha A)^{-1} X^T \mathrm{y}$$

- Scikitlearn has an implementation of ridge regression in the linear_model module

Lasso Regression

- Another type of linear regression is least absolute shrinkage and selection operator, known as LASSO for short
- It uses the absolute value of the l1 norm of the weight vector as a regularization term in the cost function

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Notes on LASSO

- The second part of LASSO standards for selection operator
- As α approaches infinity LASSO regression will totally eliminate weights of unimportant features
- The result is a sparse model with weights only for the most important features
- A subgradient vector is required to differentiate 0s in your weight vector for gradient descent

Elastic Net

- Elastic net is a regression model that constrains the weight vector by combining LASSO and ridge with a mix ratio implemented as an r parameter
- When r is 0 elastic net is the same as Ridge Regression
- When r is 1 elastic net is the same as LASSO

Regularization Best Practices

- Always try to use a little regression instead of a straight linear model
- Ridge regression is good default
- If only a few features are important, try lasso or elastic net

Early Stopping

- Another way to prevent overfitting is to stop when the validation error on your training set reaches a specified minimum
- This is called early stopping
- You configure your training model to stop as soon as the validation error goes back up
- You roll the parameters back to the point where your validation error was at its minimum

The End