Working of Basic Machine Learning Algorithms

Richeek Das

April 12, 2020

Chapter 1

Working of Basic Machine Learning Algorithms

Things we will look at here :

- **Linear Regression**
- **Gradient Descent**
- **Polynomial Regression**
- **Learning Curves**
- **Regularized Linear Models**
- **Logistic Regression**

1.1 Linear Regression

A linear model makes a prediction by simply computing a weighted **sum** of the **input features**, plus a constant called the bias term (also called the **intercept term**)

Linear Regression model prediction :

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

Here :

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the i^{th} feature value
- θ_j is the j^{th} model parameter, i.e the feature weights.

It can be further elegantly written as :

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^\intercal \cdot \mathbf{x}$$

Here :

- θ is the model's *parameter column vector* containing the bias term θ_0 and the feature weights.
- $\mathbf{x}$ is the instance's *feature vector* containing x_0 to x_n, with x_0 always equal to 1
- Needless to say the $\theta \cdot \mathbf{x}$ is the usual dot product
- h_θ is the hypothesis function, using the model parameters θ

So, let's come to the algorithms handling the training part of Linear Regression

The most common performance measure of a regression model is the **Root Mean Square Error (RMSE)**. Therefore, to train a **Linear Regression** model, we need to find the value of θ that minimizes the **RMSE**. It is simpler to minimize the **mean squared error (MSE)** than the **RMSE**, and it leads to the same result.

The **MSE** of a **LR** hypothesis h_θ on a training set $\mathbf{X}$ is calculated using :

$$\mathbf{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^\intercal \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

This function mentioned above is called the **cost function** for a **Linear Regression model**

To find the value of θ that minimizes the cost function, there is a mathematical tool that gives the result directly. Its called the *Normal Equation*

1.1.1 The Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^\intercal \mathbf{X} \right)^{-1} \mathbf{X}^\intercal \mathbf{y}$$

Here :

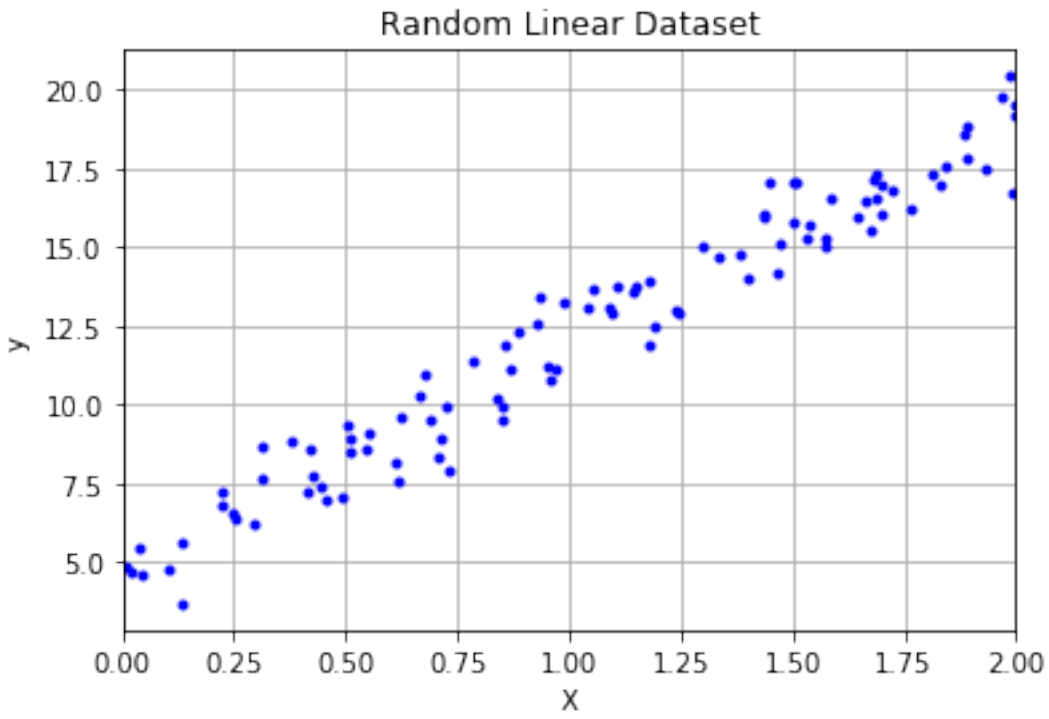
- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

Let's test this normal equation with some code !

[1]:
```
import numpy as np

X = 2 * np.random.rand(100,1)
y = 5 + 7 * X + np.random.randn(100,1)
## Generates a random linear dataset with some Gaussian Noise
```

Let's plot it :

[2]:
```
import matplotlib.pyplot as plt

plt.plot(X, y, " bo", markersize = 3)
plt.xlim(0.00, 2.00)
plt.grid(True)
plt.xlabel("X")
plt.ylabel("y")
plt.title("Random Linear Dataset")
plt.show()
```



We will use the `inv()` function from *NumPy*'s linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication :

[3]:
```
X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Function intended for : $y = 5 + 7x$

Expression found :

[4]:
```
print(theta_best[[0]]) ## Intercept term
print(theta_best[[1]]) ## First weight term
```

```
[[4.80833324]]
[[7.12658528]]
```

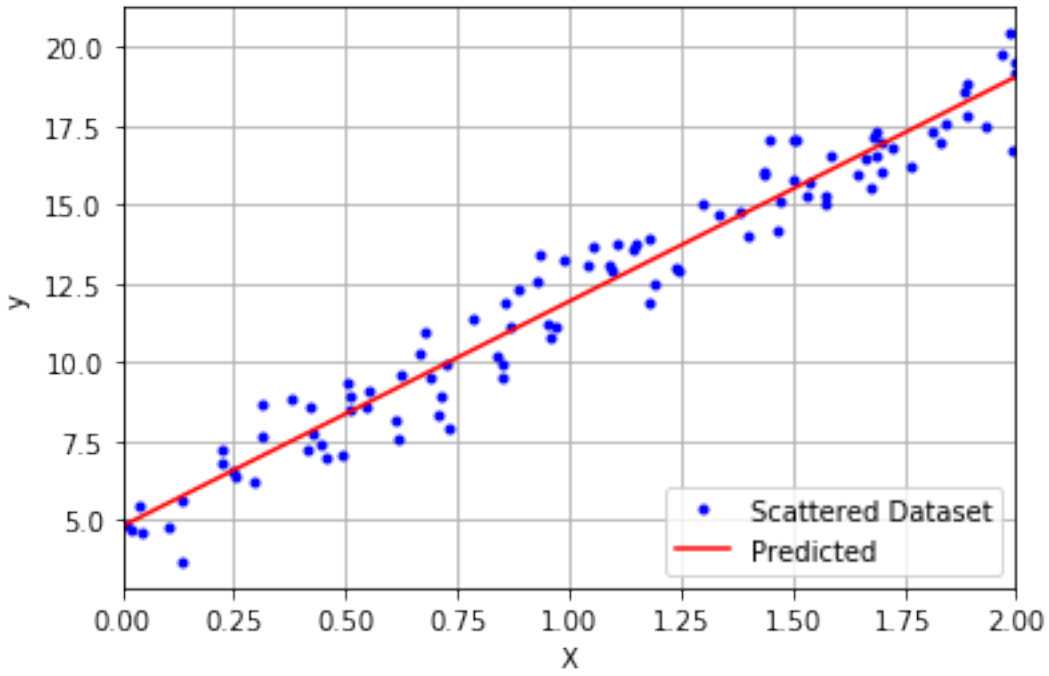
Now, let's get the predicted linear function using the $\hat{\theta}$ found using normal equations :

[5]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
```

Let's plot the prediction :

[6]:
```
plt.plot(X, y, " bo", markersize = 3, label = "Scattered Dataset")
plt.xlim(0.00, 2.00)
plt.grid(True)
plt.xlabel("X")
plt.ylabel("y")
```

```
plt.plot(X_new, y_predict, "r-", label = "Predicted")
plt.legend(loc = "lower right")
plt.show()
```



In *NumPy* we usually calculate $\hat{\theta}$ using :

$$\hat{\theta} = \mathbf{X}^{+} y$$

where, $\mathbf{X}^{+}$ is the **pseudoinverse** of $\mathbf{X}$, aka the **Moore-Penrose inverse**. The pseudoinverse is calculated using a standard matrix factorization technique called *Singular Value Decomposition(SVD)* > Can be found in the mathematical preliminary part

This approach is **more efficient** than computing the *Normal Equation*, plus it also handles the edge cases. The *Normal Equation* may not work if the matrix $\mathbf{X}^{\intercal}\mathbf{X}$ is not invertible, such as if its not a square matrix or if some features are redundant, but the pseudoinverse is always defined.

REMEMBER TO PUT THE LINK HERE

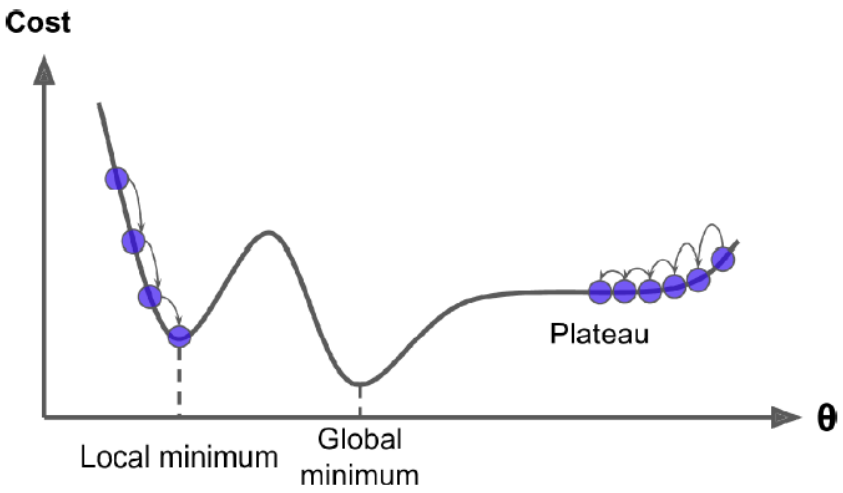
1.1.2 Computational Complexity

The **Normal Equation** computes the inverse of $\mathbf{X}^{\intercal}\mathbf{X}$, which is an $(n+1) \times (n+1)$ matrix (where n is the number of features). The computational complexity of inverting such a matrix is typically about $O(n^3)$, depending on the implementation.

The **SVD** approach has $O(n^2)$. Both these approaches are linear with regard to the number of instances in the training set (they are $O(m)$), so they handle large training sets efficiently.

1.2 Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.



Here, we start by filling θ with random values (*random initialization*). Then we improve it gradually, taking one $d\theta$ step at a time, each step making an attempt to reduce the cost function till it converges to a minimum.

Learning Rate hyperparameter : It speaks about the size of the steps taken. if learning rate is too small then the algorithm will make a huge amount of iterations to converge(in-efficient). On the other hand, if its too high, then it might even make the algorithm diverge, and fail to reach a good solution.

Well, one problem that arises at once, is the fact that there can be multiple local minimas, thus making it difficult to find the global minima with *random initialization* plan, we previously had.

Fortunately, the **MSE cost function** for a **Linear Regression** model happens to be a convex function (we can easily verify it) ! This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly. These two facts have a great consequence: **Gradient Descent** is guaranteed to approach arbitrarily close the global minimum(well, but excluding other factors).

When using Gradient Descent, we should ensure that all features have a similar scale (e.g in **Python**, using `Scikit-Learn StandardScaler` class)(there are other scalers as well, which suit to different occasions), or else it will take much longer to converge.

1.2.1 Batch Gradient Descent

To implement *GD* we nee dto compute the gradient of the *cost function* with regard to each model parameter θ_j. Therefore, we need to calculate the *partial derivative* with respect to θ_j. The equation looks something like this :

$$\frac{\partial}{\partial \theta_j}\mathbf{MSE}\Big(\theta\Big) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Gradient vector of the cost function :

$$\nabla_\theta \mathbf{MSE}\Big(\theta\Big) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathbf{MSE}\Big(\theta\Big) \\ \frac{\partial}{\partial\theta_1}\mathbf{MSE}\Big(\theta\Big) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathbf{MSE}\Big(\theta\Big) \end{pmatrix} = \frac{2}{m}\mathbf{X}^\intercal\Big(\mathbf{X}\theta - y\Big)$$

The only problem(or benefit, as you see it) of this algorithm is that it involves calculations over the full training set **X** at each step of *Gradient Descent*. That is why it is called the *Batch/Full Gradient Descent*. This makes this algorithm terribly slow but kind of accurate.

However, Gradient Descent scales well with the number of features. Training a *Linear Regression* model when there are a lot of features is much faster using *Gradient Descent* than using the *Normal Equation* or *SVD decomposition* :)

Finding the gradient, gives us the uphill direction, so quote trivially, the *Gradient Descent Step* looks like this :

$$\theta^{(next)} = \theta - \eta\nabla_\theta \mathbf{MSE}\Big(\theta\Big)$$

Here, η is the learning rate of the model.

Now let's try the same example we used with *Normal Equation* :

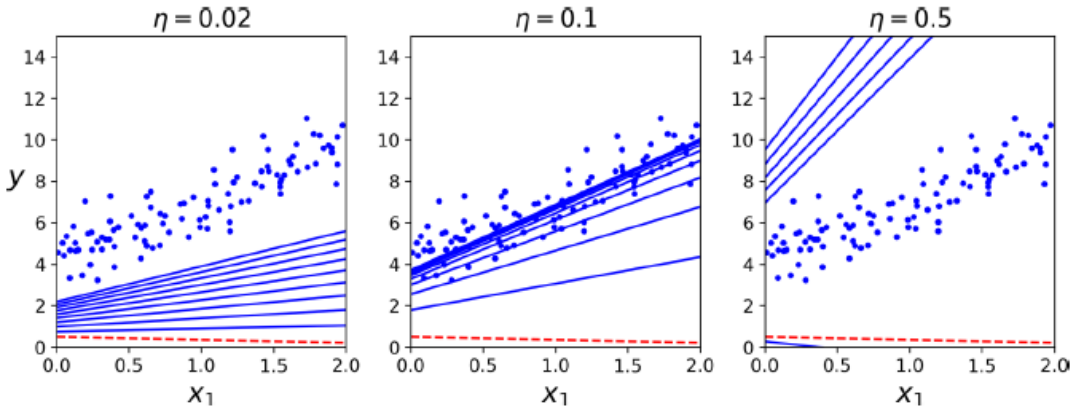
[7]:
```
learning_rate = 0.1
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) #Random initialization to start with

for i in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - learning_rate*gradients
```

The final θ we got:

[8]:
```
theta
```

[8]:
```
array([[4.80833324],
       [7.12658528]])
```

That's the same as what the Normal Equation found ! But the case would be different if we had used a different *learning rate*.

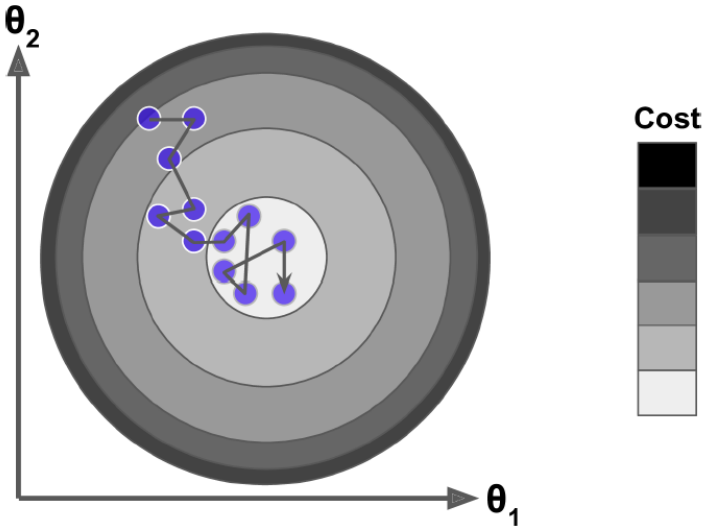


If its too low, then it'll take a lot of time to reach the optimal solution, whereas if its too high, the algorithm diverges and we won't even reach the solution. We can use `Scikit GridSearchCV` for finding the optimal hyperparameter.

For the number of iterations : Best practice is to set a very large number of iterations, and to interrupt the algorithm, when the **gradient vector** becomes smaller than a certain *tolerance*(ϵ).

1.2.2 Stochastic Gradient Descent

"Stochastic" means "random". *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration.

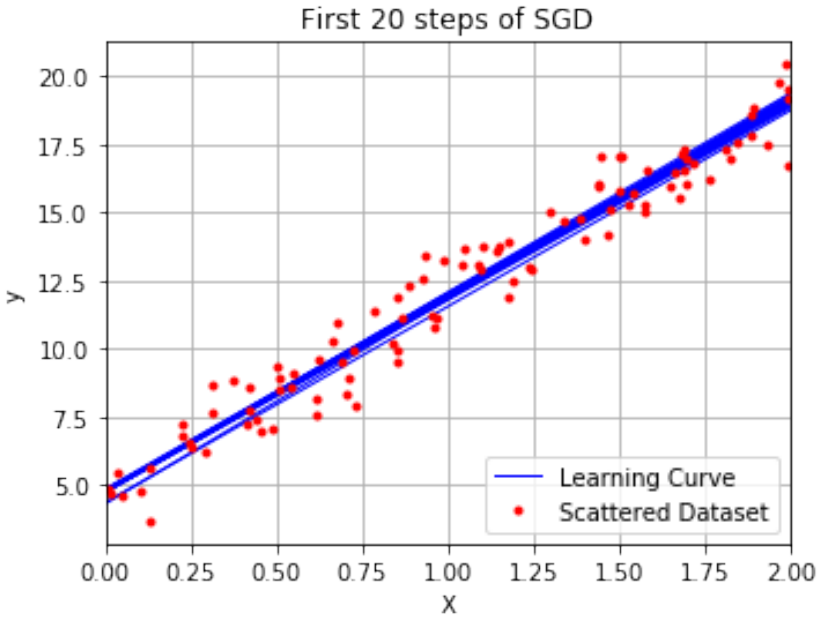


Due to its randomness, **SGD** has a better chance of finding the global minima, than *Batch Gradient Descent*. So, the randomness is good for escaping the local optima, but bad because it can never settle at the minima. One solution would be to gradually decrease the learning rate. The steps start out large, then it dies down and settles at the global minima. This process is also known as ***simulated annealing***. the function which determines the learning rate at each

iteration is called the ***learning schedule***.

Code implementation of *SGD* :

[9]:
```
n_epochs = 50
t0, t1 = 5, 50 # Learning Schedule Hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

fig = plt.figure()
ax = fig.add_subplot(111)

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients

    if epoch < 20 :
        X_new = np.array([[0], [2]])
        X_new_b = np.c_[np.ones((2, 1)), X_new]
        y_predict = X_new_b.dot(theta)
        if epoch == 1:
            ax.plot(X_new, y_predict, "b-", linewidth = 1, label = "Learning␣
 ↪Curve")
        else :
            ax.plot(X_new, y_predict, "b-", linewidth = 1)

ratio = 0.6
xleft, xright = ax.get_xlim()
ybottom, ytop = ax.get_ylim()
ax.set_aspect(abs((xright-xleft)/(ybottom-ytop))*ratio)

plt.xlabel("X")
plt.ylabel("y")
plt.xlim(0.00,2.00)
plt.plot(X, y, "ro", markersize = 3, label = "Scattered Dataset")
plt.grid(True)
plt.legend(loc="lower right")
plt.title("First 20 steps of SGD")
plt.show()
```



By convention we iterate by rounds of m iterations; each round is called an *epoch*. While the *Batch Gradient Descent code* iterated **1000 times** through the whole training set, this code goes through the training set only **50 times** and reaches an acceptable solution :

[10]:
```
theta
```

[10]:
```
array([[4.81314772],
       [7.16962854]])
```

1.2.3 Mini-batch Gradient Descent

In this algorithm, at each step, instead of computing the gradients based on the full training set (as in *Batch GD*) or based on just one instance (as in *Stochastic GD*), *Mini-batch GD* computes the gradients on small random sets of instances called **mini-batches**.

The algorithm's progress in parameter space is less erratic than with *Stochastic GD*, especially with fairly large mini-batches but both *Stochastic GD and Mini-batch GD* will reach the minimum if you use a good learning schedule.

1.3 Polynomial Regression

We can actually use a linear model to fit non-linear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called ***Polynomial Regression***.

Let's generate some noise in some non-linear quadratic data :

[11]:
```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

Now, we'll use `Scikit-Learn PolynomialFeatures` class to transform our training data :

[12]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X_poly[0]
```

[12]:
```
array([2.86334939, 8.19876974])
```

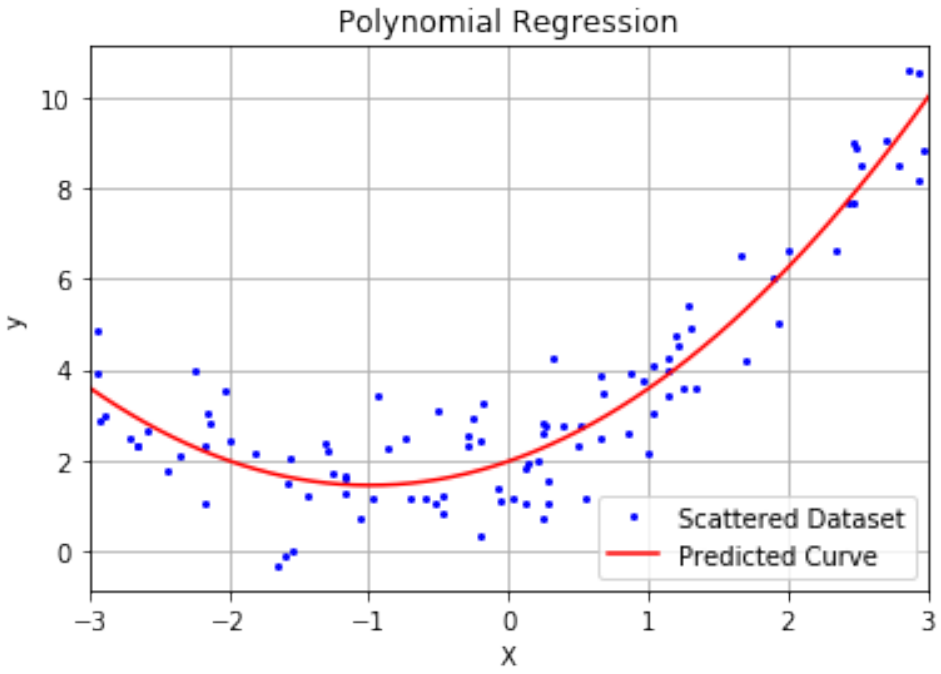
Now let's fit a ***Linear Regression*** model to this extended training data(we are using `Scikit-Learn LinearRegression` here :

[13]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

[13]:
```
(array([1.98031152]), array([[1.06214383, 0.53454417]]))
```

Let's plot and see what we get !

[14]:
```
X_new = np.arange(-3,3.2,0.1)
X_new_b = np.c_[X_new, X_new**2]
y_predict =   np.c_[np.ones((62,1)).dot(lin_reg.intercept_) + X_new*lin_reg.
 ↪coef_[[0][0]][0] + (X_new**2)*lin_reg.coef_[[0][0]][1]]

plt.plot(X, y, "bo", markersize = 2, label = "Scattered Dataset")
plt.plot(X_new, y_predict, "r-", label = "Predicted Curve")
plt.title("Polynomial Regression")
plt.grid("True")
plt.xlim(-3,3)
plt.xlabel("X")
plt.ylabel("y")
plt.legend(loc="lower right")
plt.show()
```



1.4 Learning Curves

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression.

But a very-high degree polynomial has chances of severely *overfitting* the training data, while a linear model might have chances of *underfitting* the same data. So we can use **cross-validation** to get an estimate of the model's performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then the model is overfitting. If it performs poorly on both, then it is underfitting.

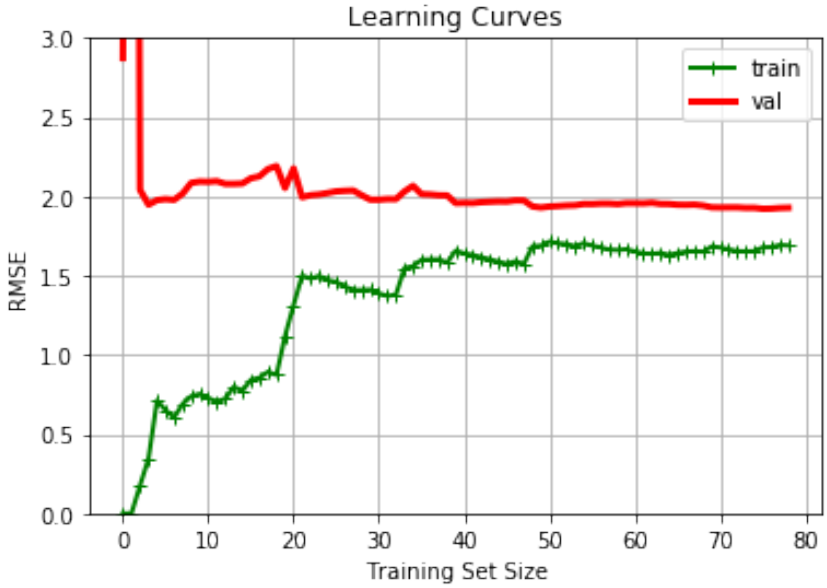
Another way to tell it, is to look at the *learning curves* : These are plots of the model's performance on the training set and the validation set as a function of the training set size. Let's generate the *Learning Curves* here :

[15]:
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

##A function that, given some training data, plots the learning curves of a␣
 ↪model:
def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m],
        y_train_predict))
```

```
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "g-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "r-", linewidth=3, label="val")
    plt.grid(True)
    plt.legend(loc = "upper right")
    plt.xlabel("Training Set Size")
    plt.ylabel("RMSE")
    plt.title("Learning Curves")
    plt.ylim(0.00,3.00)
    plt.show()
```

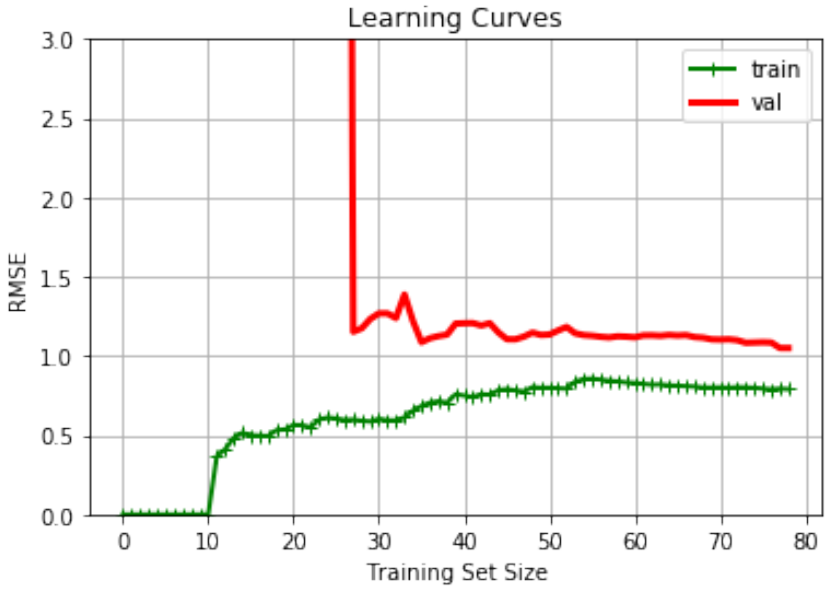
For a plain *Linear Regression* model :

[16]:
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



This is a typical example of underfitting. Both training and validation curves have reached a plateau, they are close and high. Let's plot the *learning curve* for a 10th-degree polynomial :

[17]:
```
##Create a pipeline for Scikit first :)

from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([
                        ("poly_features", PolynomialFeatures(degree=10,
                        include_bias=False)),
                        ("lin_reg", LinearRegression()),
                        ])

plot_learning_curves(polynomial_regression, X, y)
```



Differences :

- The error on the training data is much lower than with the Linear Regression model.
- There's a gap between the curves, which indicates that this model is the heavily overfitting :(

Note : Its good to notice that, as the train set size increases the model performs much better, just like any other overfitting scenario.

1.5 Regularized Linear Models

The fewer degrees of freedom a model has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, *regularization* is typically achieved by constraining the weights of the model. We will now look at **Ridge Regression, Lasso Regression, and Elastic Net**, which implement three different ways to constrain the weights.

1.5.1 Ridge Regression

Ridge Regression cost function :

$$J(\theta) = \mathbf{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Here :

- $\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$ is the regularization term. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. We apply the regularization term only during the model training. Once the model is trained, we use the unregularized performance.

- The hyperparameter α controls how much we want to regularize the model. If $\alpha = 0$, then *Ridge Regression* is just *Linear Regression*. If α is very large, then all weights end up very close to zero and the result is a flat line going through the mean.

Ridge Regression closed-form solution :

$$\hat{\theta} = \left(\mathbf{X}^\intercal \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \mathbf{X}^\intercal \mathbf{y}$$

We can perform *Ridge Regression* either by computing a closed-form equation or by performing *Gradient Descent*. The pros and cons are the same.

1.5.2 Lasso Regression

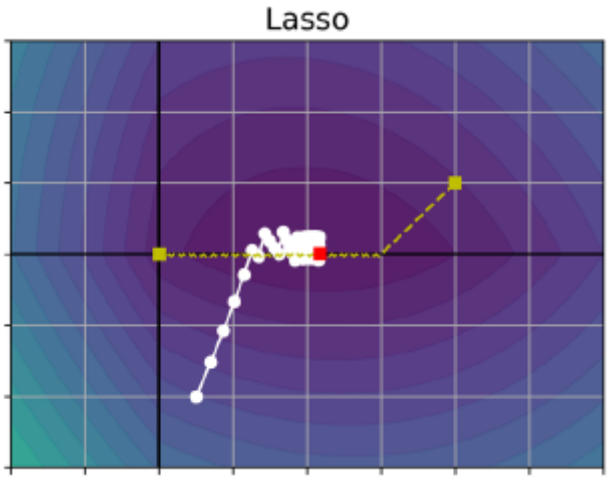
Least Absolute Shrinkage and Selection Operator Regression (usually simply called Lasso Regression) is another regularized version of Linear Regression. But it uses the norm of the weight vector instead of half the square of the norm as the regularization term.

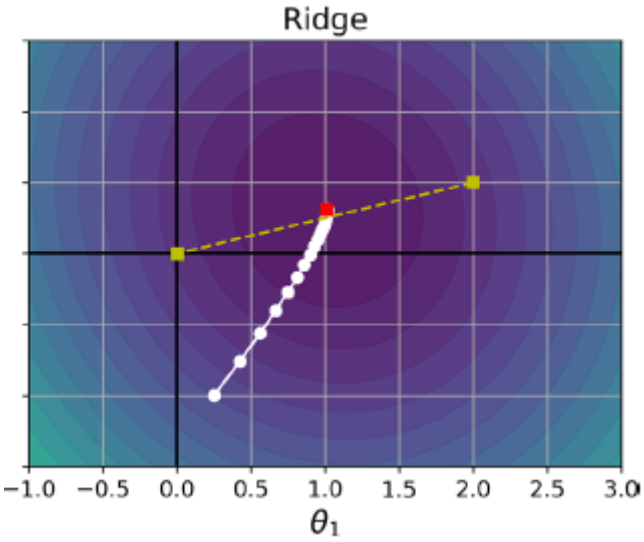
Lasso Regression cost function :

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

An important characteristic of *Lasso Regression* is that it tends to eliminate the weights of the least important features. *Lasso Regression* automatically performs feature selection and outputs a *sparse model*.

Comparison of **Lasso and Ridge models** :





The axes represent two model parameters, and the background contours represent different loss functions. The small white circles show the path that Gradient Descent takes to optimize some model parameters that were initialized.

1.5.3 Elastic Net

Elastic Net is a middle ground between *Ridge Regression* and *Lasso Regression*. The regularization term is a simple mix of both *Ridge* and *Lasso's regularization terms*, and you can control the mix ratio r. When $r = 0$, *Elastic Net* is equivalent to *Ridge Regression*, and when $r = 1$, it is equivalent to *Lasso Regression*.

Elastic Net cost function :

$$J(\theta) = \mathbf{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

1.5.4 Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called **early stopping**.

As the algorithm learns, the prediction error(RMSE) on the training set goes down, along with its prediction error on the validation set. After a number of *epochs* the **validation error stops decreasing and starts to go back up**. This is an indication of premature overfitting. We need to stop right here.

It functions as a simple and efficient regularization technique. We just need to find the minima of the *validation error* :)

Basic implementation of early stopping :

[ ]:
```
from sklearn.base import clone
from sklearn.preprocessing import StandardScaler
# prepare the data
poly_scaler = Pipeline([
```

```
                        ("poly_features", PolynomialFeatures(degree=90,
                                                             include_bias=False)),
                        ("std_scaler", StandardScaler())
                        ])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None

for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)

    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

With `warm_start=True`, when the `fit()` method is called it continues training where it left off, instead of restarting from scratch.

1.6 Logistic Regression

Logistic Regression is commonly used to estimate the probability that an instance belongs to a particular class. Therefore, it is a binary classifier.

Just like a Linear Regression model, a *Logistic Regression* model, computes a weighted sum of the input features and it outputs the *logistic* of this result.

Logistic Regression model **estimated probability** :

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\mathbf{x}^\intercal \theta)$$

σ is the *sigmoid function*. It takes the $\mathbf{x}^\intercal\theta$ and squeezes its value to a number between 0 and 1.

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a *Logistic Regression* model predicts 1 if $\mathbf{x}^\intercal\theta$ is positive and 0 if it is negative.

1.6.1 Training and Cost Function

The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0).

Such a cost function which implements this concept for a single training instance :

$$c(\theta) = \begin{cases} -\mathbf{log}(\hat{p}) & \text{if y = 1} \\ -\mathbf{log}(1-\hat{p}) & \text{if y = 0} \end{cases}$$

The cost function over the whole training set is the average cost over all training instances :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})\right]$$

This is known as, *Logistic Regression cost function(log loss)*.

Well, here we doon't have a generalised Normal Equation, but we can easily check that the cost function(log loss) is actually convex, so we can apply our second option, the **Gradient Descent**.

The partial derivatives of the cost function with respect to the j^{th} model parameter is :

$$\frac{\partial}{\partial\theta_j}\mathbf{J}(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma(\theta^{\intercal}\mathbf{x}^{(\mathbf{i})}) - y^{(i)}\right)x_j^{(i)}$$

That's it, we can now apply *Batch, Stochastic or Mini-Batch Gradient Descent* to use this *Logistic Regression model* :)

1.7 Conclusion

We saw the working of quite a few, Machine Learning **Regressors** and **Classifiers**. We will now move on to more interesting stuff :)