Chapter 4: Training Models

1. Linear Regression

1.1 Normal Equation

To find the value of θ that minimizes the cost function, there is a closed-form solution (gives result directly). This is called the Normal Equation

In [ ]:
```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3*X + np.random.rand(100,1)
```

We will now compute the $\hat{\theta}$ using the normal equation where $\hat{\theta} = (X^T X)^{-1} X^T y$

In [ ]:
```
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)
```

In [ ]:
```
theta_best
```

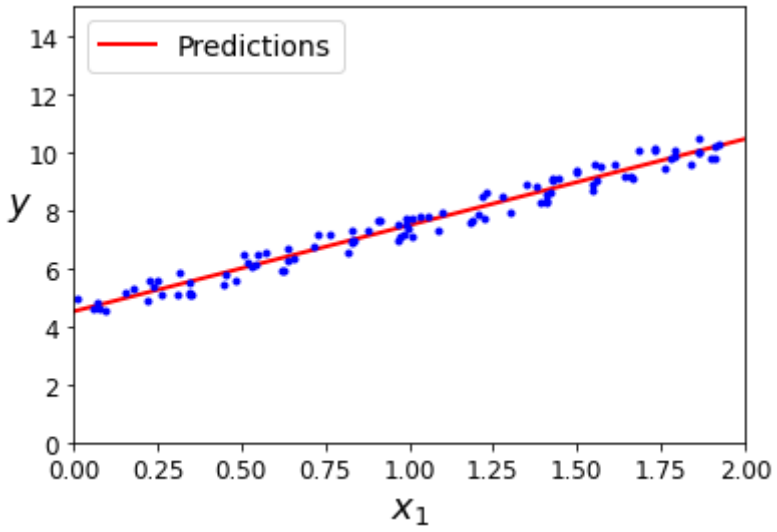
Out[ ]:
```
array([[4.50950269],
       [2.96807747]])
```

Now making predictions using our best $\hat{\theta}$ parameter. $\hat{y} = \hat{\theta} * x$

In [ ]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2,1)), X_new] # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
y_predict
```

Out[ ]:
```
array([[ 4.50950269],
       [10.44565763]])
```

In [ ]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 2, 0, 15])
plt.show()
```



In [ ]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression() # pseudo inverse computed using SVD
lin_reg.fit(X ,y)
lin_reg.intercept_, lin_reg.coef_
```

Out[ ]:
```
(array([4.50950269]), array([[2.96807747]]))
```

In [ ]:
```
lin_reg.predict(X_new)
```

Out[ ]:
```
array([[ 4.50950269],
       [10.44565763]])
```

1.1.1 Psuedo Inverse

The LinearRegression class is based on the scipt.linalg.lstsq() function (least squares).

In [ ]:
```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
theta_best_svd
```

Out[ ]:
```
array([[4.50950269],
       [2.96807747]])
```

This function computes the $\hat{\theta} = X^{+}y$ where X^{+} is the pseudoinverse of X (Moore-Penrose inverse). The normal equation may not work if the matrix $X^{T}X$ is not invertible (due to m < n or some features redundant) but the pseudoinverse is always defined. The pseudoinverse is computed using singular value decomposition method where $X^{+} = V\sum^{T}U^{T}$. $\sum^{T}$ is computed by taking $\sum$ and all values smaller than a tiny threshold value to 0 then replaces all nonzero values with their inverse and finally transposes the resulting matrix.

1.1.2 Computational Complexity

$X^{T}X$ is a (n+1) x (n+1) matrix. Both the normal equation and SVD approach gets very slow when number of features grows large (eg: 100,000). Predictions are very fast since computational complexity is linear. In other words, making predictions on twice as many instances will take roughly twice as much time. As such, other methods of finding parameters such as Gradient Descent might be better.

1.2 Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions. The idea is to tweak parameters iteratively in order to minimize a cost function. It measures the local gradient of the error function with regard to parameter vector θ and it goes in the direction of desceding gradient until minimum

An important parameter in GD is the size of steps. This is determined by the learning rate hyperparameter.

- Learning rate small --> algorithm have to go through many iterations to converge which can take long time.
- Learning rate high --> might overshot the minimum and fail to find a good solution

The MSE cost function of Linear Regression is a convex function (if you pick any 2 points on the curve, the line segment joining them never crosses the curve). This implies that there are no local minimum and just one global minimum. Thus, GD is guaranteed to approach arbitrarily close to the global minimum

If features have very different scale, the cost function will have an elongated bowl shape. Hence, by scaling it, can reach minimum faster. (Since feature 1 is smaller, it takes a larger change in θ_1 to affect the cost function hence the bowl is elongated along θ_1 axis). Therefore, when using GD, we should ensure that all features have similar scale or else it will take much longer to converge.

1.3 Batch Gradient Descent

Uses entire training set to compute gradients at every step

1.3.1 Partial Derivative

To implement GD, we need to compute the gradient of the cost function with regard to each model parameter θ_j. This means how much the cost function will change if you change θ_j by just a little bit. This is called partial derivative.

The gradient vector of MSE cost function of Linear Regression is $\nabla_\theta MSE(\theta)$ = $\frac{2}{m} X^T(X\theta - y)$

Once we have the gradient vector, which points uphill, to go in opposite direction we just have to subtract $\nabla_\theta MSE(\theta)$ from θ.
$\theta^{nextstep}$ = $\theta - \eta \nabla_\theta MSE(\theta)$ where η is the learning rate

In [ ]:
```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100

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

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

In [ ]:
```
theta
```

Out[ ]:
```
array([[4.50950269],
       [2.96807747]])
```

1.3.2 Finding good hyperparameters

- Use grid search to find good learning rate
- To find good number of iterations, set it to a very large number but to interrupt the algorithm when the gradient vector becomes tiny (when its norm because smaller than a threshold ϵ called tolerance)

1.4 Stochastic Gradient Descent

Batch GD can be very slow since it uses entire training set to compute the gradients at every step. SGD picks a random instance in the training set at every step and compute the gradients based only on that single instance. It makes it possible to train on huge training sets.

However, due to the stochastic (random) nature, this algorithm is much less regular than BGD. Instead of gently decreasing until it reaches minimum, the cost function will bounce up and down. Once it gets very close to minimum, it will never settle down and instead continue to bounce up and down. Hence the final parameter values are good but not optimal.

When cost function is very irregular, can help algorithm jump out of local minima.

1.4.1 Learning Schedule

Although the algorithm can never settle at the minimum, a solution is to gradually reduce the learning rate.

- If learning rate is reduced too quickly, get stuck at local minima.
- If learning rate is reduced too slolwy, may jump around minimum for a long time and end up with suboptimal soltion if halt training too early.

When using SGD, the training instances must be indepedent and identically distributed (iid) to ensure that the parameters get pulled toward the global optimum on average. To ensure this, can shuffle the instances during training

1.4.2 SGDRegressor

In [ ]:
```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Out[ ]:
```
SGDRegressor(eta0=0.1, penalty=None)
```

In [ ]:
```
sgd_reg.intercept_, sgd_reg.coef_
```

Out[ ]:
```
(array([4.46886103]), array([2.97290746]))
```

Solution is quite close tot he one returned by the normal equation.

1.5 Mini-batch Gradient Descent

Mini-batch GD computes the gradients on small random sets of instances called mini-batches. Main advantage of Mini-batch GD over SGD is that we get performance boost from hardware optimization of matrix operations especially using GPUs

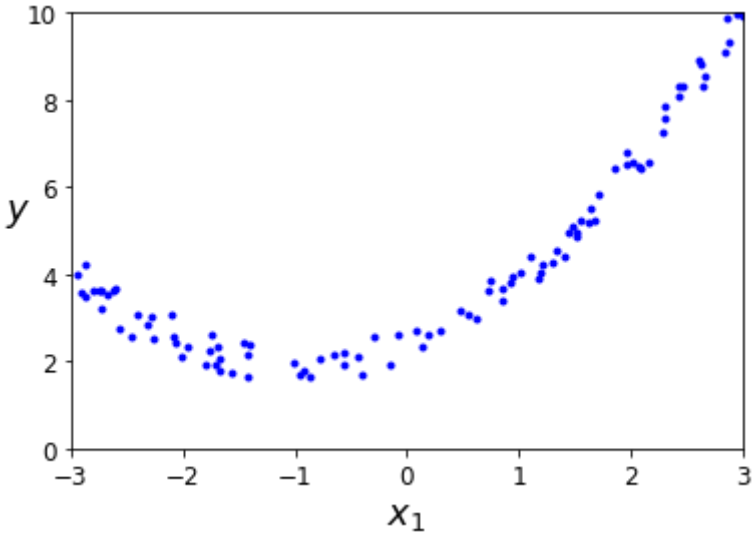
Mini-batch GD will end up walking around a bit closer tot he minimum than SGD but it may be harder for it to escape from local minima.

2. Polynomial Regression

Generate some nonlinear data based on simple quadratic equation $y = \alpha x^2 + bx + c$

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

In [ ]:
```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```



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

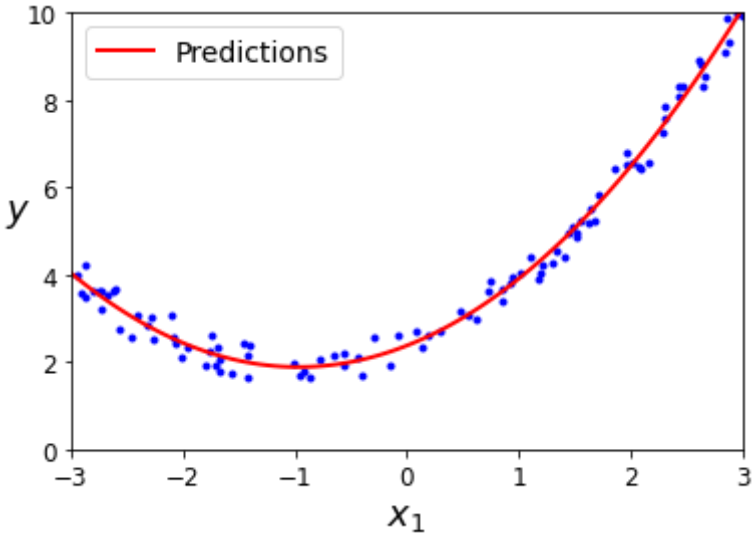
```
[1.00917467]
[1.00917467 1.01843352]
```

X_poly now contains the original features of X + square of this feature.

In [ ]:
```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

Out[ ]:
```
(array([2.37525885]), array([[1.01162951, 0.52105239]]))
```

In [ ]:
```
X_new = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
plt.show()
```



2.1 Learning Curves

How can you tell that your model is overfitting or underfitting the data?

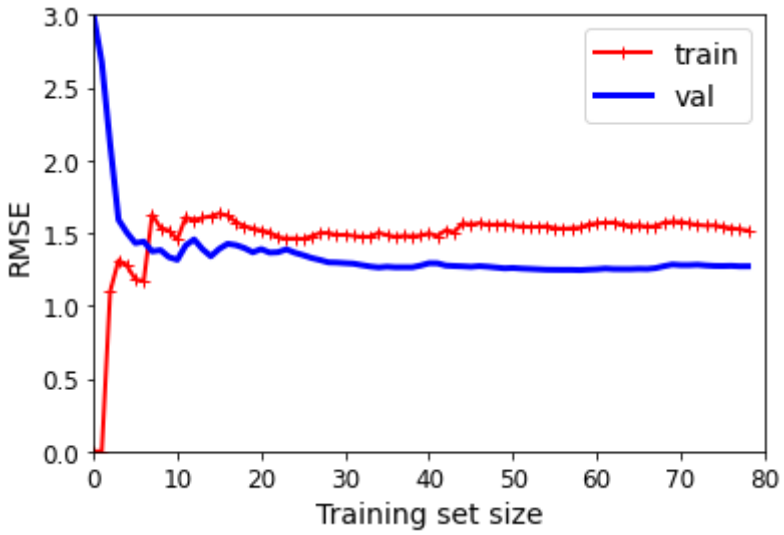
Besides using CV to get an estimate of model's generalization performance (if model performs well on training data but generalizes poorly according to cv metrics), naother way is look at learning curves.

Learning curves are plots of the model's performance on the training set and the validation set as a function of the training set size. To generate these plots, train the model several times on different sized subsets of the training set.

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

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))
```

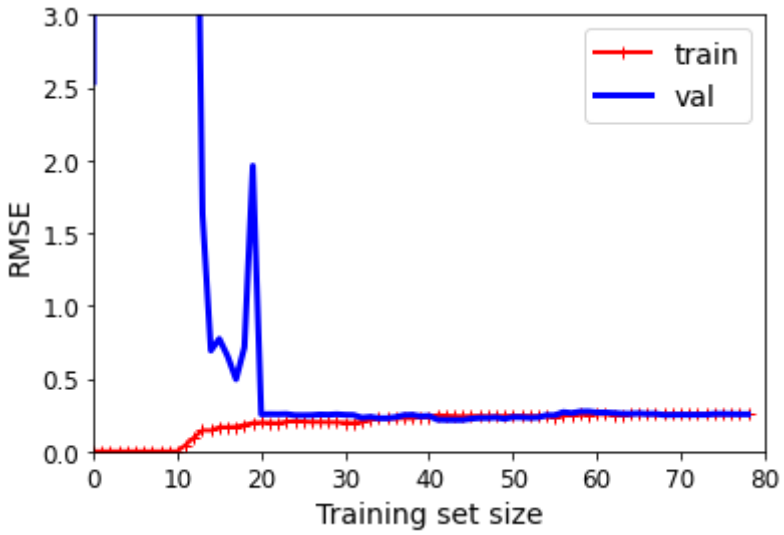
```
        val_errors.append(mean_squared_error(y_val, y_val_predict))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)   # not shown in the book
    plt.xlabel("Training set size", fontsize=14) # not shown
    plt.ylabel("RMSE", fontsize=14)              # not shown
```

In [ ]:
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])                         # not shown in the book
plt.show()
```



Both curves have reached a plateau and is an indication of underfitting. If model is underfitting the training data, adding more training examples will not help. We need more complex model or come up with better features

In [ ]:
```
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)
plt.axis([0, 80, 0, 3])           # not shown
plt.show()
```



For this polynomial regression, the error on the training data is much lower than the linear regression model. There is a gap between the curves. This means that the model performs significantly better on the training data than on validation, which is an indication of an overfitting model. If we used a much larger training set, the two curves would continue to get closer. One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.

3. Bias Variance Tradeoff

A model's generalization error can be expressed as the sum of three different errors:

- Bias: error due to wrong assumptions such as assuming data is linear when it is quadratic. High bias --> underfit
- Variance: due to model's excessive sensitivity to small variations in the training data. A model with many df likely to have high variance and thus overfit.
- Irreducible Error: due to noisiness of data. Only way to reduce this part of error is to clean up the data

4. Regularized Linear Models

A good way to reduce overfitting is to regularize the model (to constrain it). The fewer df, harder it will be for it to verfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees. FOr linear model, we constrain the weights of the model by adding penalties.

4.1 Ridge Regression

Ridge regression is also called Tikhonov regularization. A regularization term $\alpha \sum_{i=1}^{n} \theta_i^2$ is added to the cost function. This forces the learnin algorithm to not only fit the data but also keep the model weights as small as possible. The cost function is defined below:

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

The bias term θ_0 is not regularized. It is important to scale the data before performing Ridge Regression.

α hyperparameter controls how much you want to regularize the model.

- α = 0: ridge regression is just linear regression
- α = large: all weights end up very close to zero and the result is a flat line going through the data's mean.

Increasing α --> increases bias but decreases variance

The closed-form solution of Ridge Regression is $\hat{\theta} = (X^T X + \alpha A)^{-1} X^T y$

In [ ]:
```
from sklearn.linear_model import Ridge
ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X,y)
ridge_reg.predict([[1.5]])
```

Out[ ]:
```
array([[5.70787563]])
```

In [ ]:
```
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

Out[ ]:
```
array([5.69310046])
```

4.2 LASSO Regression

LASSO: Least Absolute Shrinkage and Selection Operator Regression. The cost function is defined below:
$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$

LASSO tends to eliminate the weights of the least important features (i.e sets them to zero). Thus, LASSO automatically performs feature selection and outputs a sparse model

LASSO cost function is not differentiable at $\theta_i = 0$ but GD works fine if we use a subgradient vector instead when any $\theta_i = 0$

In [ ]:
```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
```

Out[ ]:
```
array([5.67145799])
```

4.3 Elastic Net

Elastic Net is a middle ground between Ridge and LASSO Regression. The regulatization term is a mix of both Ridge and LASSO penalty. The cost function is given below:
$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$

Tips:

- It is always preferable to have at least a little bit of regularization
- Ridge is good by default
- If suspect only a few features are useful, use LASSO.
- Elatic Net is preferred over LASSO because LASSO may behave erratically when number of m > n or when several features are strongly correlated.

In [ ]:
```
from sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X, y)
elastic_net.predict([[1.5]])
```

Out[ ]:
```
array([5.67240894])
```

5. Early Stopping

For iterative learning algorithms such as Gradient Descent, we can use early stopping as a regularization technique. Early stopping means stoping the training as soon as the validation error reaches a minimum.

With SGD an Mini-batch GD, the curves are not so smooth and may be difficult to know whether we have reached the minimum. One solution is to stop only after the validation error has been above the minimum for some time, then we roll back the model parameters to the point where the validation error was at a minimum

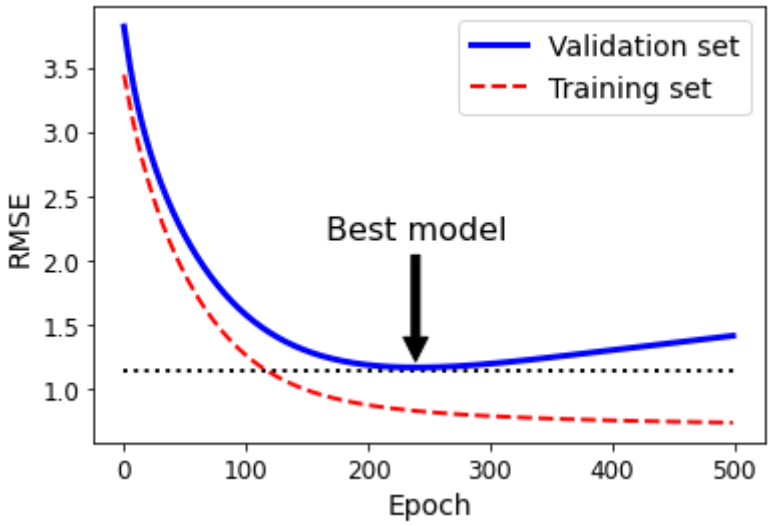
In [ ]:
```
from sklearn.base import clone
from sklearn.preprocessing import StandardScaler

np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 2 + X + 0.5 * X**2 + np.random.randn(m, 1)

X_train, X_val, y_train, y_val = train_test_split(X[:50], y[:50].ravel(), test_siz

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,
                       penalty=None,
                       eta0=0.0005,
                       warm_start=True,
                       learning_rate="constant",
                       random_state=42)
```

In [ ]:
```
n_epochs = 500
train_errors, val_errors = [], []
for epoch in range(n_epochs):
    sgd_reg.fit(X_train_poly_scaled, y_train)
    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    train_errors.append(mean_squared_error(y_train, y_train_predict))
    val_errors.append(mean_squared_error(y_val, y_val_predict))
```

In [ ]:
```
best_epoch = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])
```

In [ ]:
```
plt.annotate('Best model',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )
best_val_rmse -= 0.03  # just to make the graph look better
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
```

```
plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Training set")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel("Epoch", fontsize=14)
plt.ylabel("RMSE", fontsize=14)
plt.show()
```



In [ ]:
```
from sklearn.base import clone
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005, random_state=42)

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)
```

In [ ]:
```
best_epoch, best_model
```

Out[ ]:
```
(239,
 SGDRegressor(eta0=0.0005, learning_rate='constant', max_iter=1, penalty=None,
              random_state=42, tol=-inf, warm_start=True))
```

6. Logistic Regression

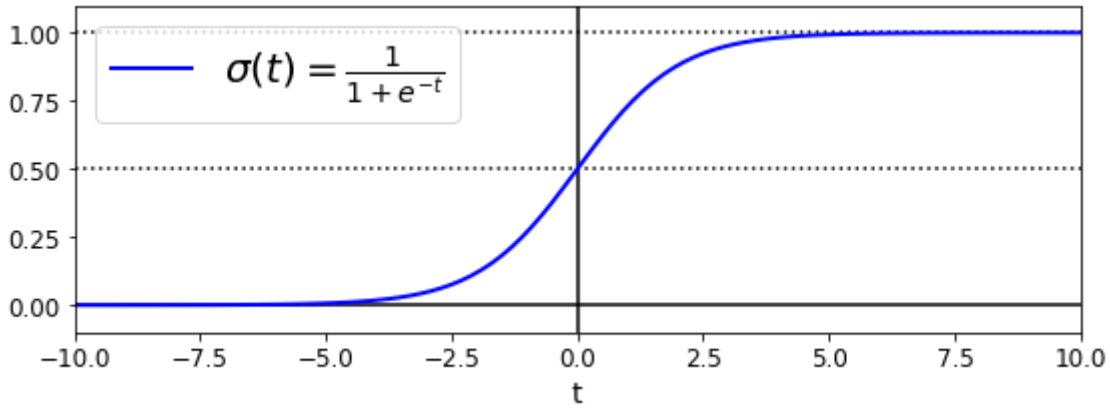
Also known as Logit Regression is commonly used to estimate the probability that an instance belongs to a particular class. Logistic Regression computes a weighted sum of the input features but instead of outputting the result directly, it outputs te logistic of this result.

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

$\sigma(.)$ is a sigmoid function and this function outputs a number between 0 and 1.
$\sigma(t) = \frac{1}{1+exp(-t)}$ The score t is called the logit (log-odds)

In [ ]:
```
t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))
plt.figure(figsize=(9, 3))
```

```
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
```



The cost function of single training instance is given below:

- If y = 1, $c(\theta) = -log(\hat{p})$.
- If y = 0, $c(\theta) = -log(1 - \hat{p})$.

The cost function over the whole training set is the average cost over all training instances:
$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(\hat{p}^{(i)}) + (1 - y^{(i)})log(1 - \hat{p}^{(i)})]$

However, there is no closed-form equation to compute the value of θ that minimizes this cost function. But because this cost function is convex, GD is guaranteed to find the global minimum.

In [ ]:
```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

Out[ ]:
```
['data',
 'target',
 'frame',
 'target_names',
 'DESCR',
 'feature_names',
 'filename',
 'data_module']
```

In [ ]:
```
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(int) # 1 if Iris virginica, else 0
```

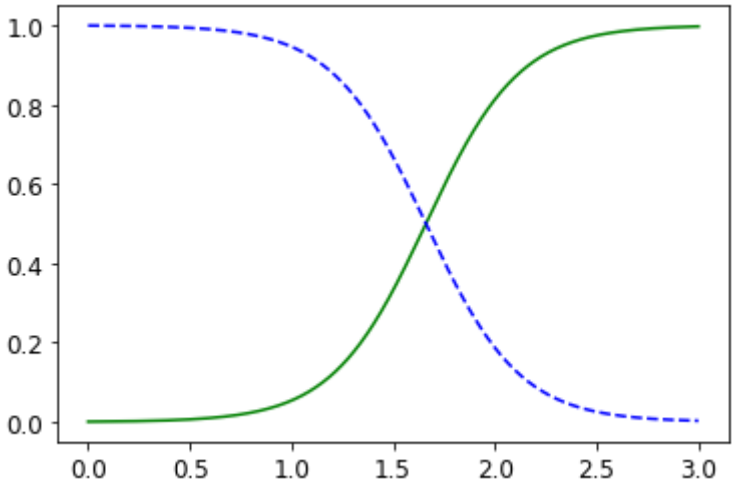
In [ ]:
```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Out[ ]:
```
LogisticRegression()
```

In [ ]:
```
X_new = np.linspace(0, 3, 1000).reshape(-1,1)
```

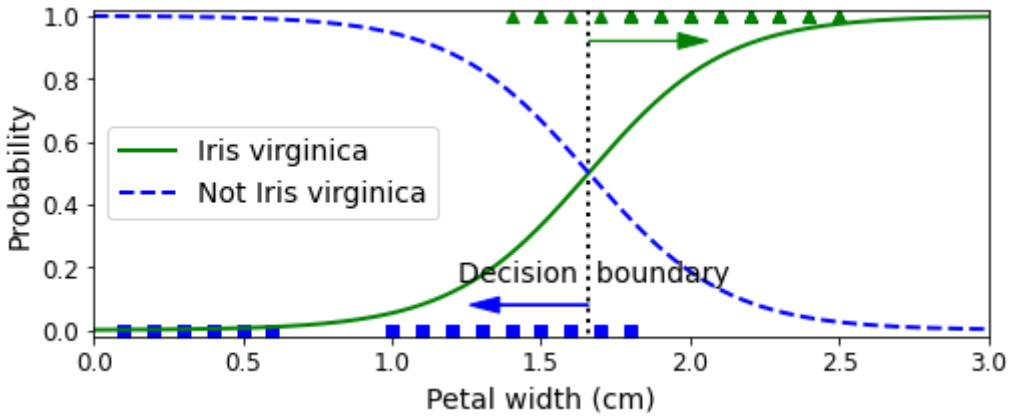
```
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris virginica")
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ab7bc54490>]
```



In [ ]:
```
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]

plt.figure(figsize=(8, 3))
plt.plot(X[y==0], y[y==0], "bs")
plt.plot(X[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2, label="Not Iris virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14, color="k
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02])
plt.show()
```

```
c:\Users\joann\Anaconda3\envs\tensorflow-gpu\lib\site-packages\matplotlib\patches.
py:1444: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequenc
es (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray.
  self.verts = np.dot(coords, M) + [
```



The petal width of Iris virginica flowers are represented by triangles while other flowers are represented by squares. Above 2cm the classifier is highly confident that the flower is an Iris

virginica while below 1cm it is highly confident that it is NOT iris virginica. In between these extremes, the classifier is unsure.

There is a decision boundary at around 1.6cm where both probabilities are equal to 50%. If the petal width > 1.6cm, classifier will predict as Iris Virginica

In [ ]:
```
log_reg.predict([[1.7], [1.5]])
```

Out[ ]:
```
array([1, 0])
```

Logistic Regression models can also be regularized using ℓ_1 or ℓ_2 penalities. Scikit-Learn adds an ℓ_2 penalty by default

7. Softmax Regression/ Multinomial Logistic Regression

Logistic Regression model can be generalized to support multiple classes directly without having to train and combine multiple binary classifiers.

When given an instance x, softmax regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of each class by applying the softmax function (also called normalized exponential) to the scores.

Score is computed by: $s_k(x) = x^T\theta^{(k)}$. Each class has its own parameter vector $\theta^{(k)}$

Once score is computed, probability $\hat{p_k}$ that the instance belongs to class k computed by the following formula:
$\hat{p_k} = \sigma(s(x))_k = \frac{exp(s_k(x))}{\sum_{j=1}^{K} exp(s_j(x))}$

- K: number of classes
- s(x) is a vector containing the scores of each class for the instance x
- $\sigma(s(x))_k$ is the estimated probability that the instance x belongs to class k,given the scores of each class for that instance

The cross entropy cost function is given below:
$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(\hat{p_k}^{(i)})$

$y_k^{(i)}$ is the target probability that the i^{th} instance belongs to class k. It is equal to 0 or 1. When K=2, this cost function is equivalent to the Logistic Regression's cost function

Scikit-Learn's LogisticRegression uses OvR by default when you train on more than two classes but we can set multi*class hyperparameter to "multinomial" to switch to SoftmaxRegression. It also applies $\ell{2}$* regularization by default

In [ ]:
```
X = iris["data"][:, (2,3)] # petal length and petal width
y = iris["target"]
```

In [ ]:
```
softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X,y)
```

Out[ ]:
```
LogisticRegression(C=10, multi_class='multinomial')
```

In [ ]:
```
softmax_reg.predict([[5, 2]])
```

Out[ ]:
```
array([2])
```

In [ ]:
```
softmax_reg.predict_proba([[5, 2]])
```

Out[ ]:
```
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Exercises

1. Which Linear Regression training algorithm can you use if you have a training set with millions of features?

We can use Stochastic Gradient Descent or Mini-batch Gradient Descent and perhaps Batch Gradient Descent if the training set fits in memory. But we cannot use Nomral Equation or the SVD approach because the computational complexity grows quickly with the number of features.

2. Suppose the features in your training set have very different scales. Which algorithms might suffer from this and how? What can you do about it?

If features have different scales, then the cost function will have the shape of an elongated bowl, so the GD algorithms will take a long time to converge. To solve this, we should scale the data before training the model. Normal Equation and SVD will work fine wihtout scaling. Regularized models may converge to a suboptimal solution if the features are not scaled. This because since regularization penalizes large weights, features with smaller vallues will tend to be ignored compared to features with larger values.

3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?

GD cannot get stuck in local minimum when training a Logsitic Regression model because the cost function is convex meaning that it can only one global minimum

4. Do all Gradient Descent algorithms lead to the same model, provided you let them run long enough?

If the optimization problem is convex (Linear Regression or Logsitic Regression), and assuming the learning rate is not too high, then all GD algorithms will approach the global optimium. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge. Instead, they will keep jumping back and forth around the global optimum. This means that even if you let them run for a very long time, the GD algorithms will produce slightly different models.

5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going

on? How can you fix this?

One possibility is that the learning rate is too high and the algorithm is diverging. If the training error also goes up, then this is clealy the problem and we should reduce the learning rate. However, if the training error does not go up, then your model is overfitting the training set and we sholuld stop training.

6. Is is a good idea to stop Mini-Batch Gradient Descent immediately when the validation error goes up?

Due to the random nature, neither Stochastic GD nor Mini-batch GD is guaranteed to make progress at every single training iteration. If we stop training when the validation error goes up, we may stop much too early before the optimum is reached. A better option is to save the model at regular intervals, then when it has not improved for a long time, can rever to the best saved model.

7. Which Gradient Descent algorithm will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?

Stochastic GD has the fastest training iteration since it considers only one training instance at a time. However, only Batch GD will converge, given enough training time. Stochastic GD and Mini-Batch GD will bounce around the optimum unless we gradually reduce the learning rate.

8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?

If validation error is much higher than training error, this is because model is overfitting the training set. To fix this issue, we can reduce the polynomial degree as a model with fewer df is less likely to overfit. Another way to fix this issue is to regularize the model by adding penalty to the cost function. This will also reduce the df on the model. Lastly, we can try to increase the size of the training set.

9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?

The model is likely underfitting the training set which means it has high bias. We should try reducing the regularization hyperparameter α

10(a). Why would you use Ridge Regression instead of plain Linear Regression?

A model with some regularization typically performs better than a model without an regularization so we should generally prefer Ridge Regression over plain Linear Regression

10(b). Why would you use LASSO instead of Ridge Regression?

LASSO regression uses ℓ_1 penalty, which tends to push weights down to exactly zero. This leads to sparse models where all weights are zero except for the most important weights. This is a way to perform feature selection automatically. This is desirable if we suspect that only a few features actually matter. When unsure, use Ridge Regression.

10(c). Why would you use Elastic Net instead of LASSO?

Lasso may behave erratically in some cases (when several features are strongly correlated or when there are more features than training instances). However, it does add an extra hyperparameter to tune. If we want LASSO without the erratic behaviour, we can just use Elastic Net with an l1_ratio close to 1.

11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regresion classifiers or one Softmax Regression classifier?

Since these are not exclusive classes (all four combinations possible), we should train two Logistic Regression classifiers.