Exercise 2

File name convention: For group 42 and memebers Richard Stallman and Linus Torvalds it would be "Exercise2_Goup42_Stallman_Torvalds.pdf".

Submission via blackboard (UA).

Feel free to answer free text questions in text cells using markdown and possibly $\LaTeX$ if you want to.

Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

In [ ]:

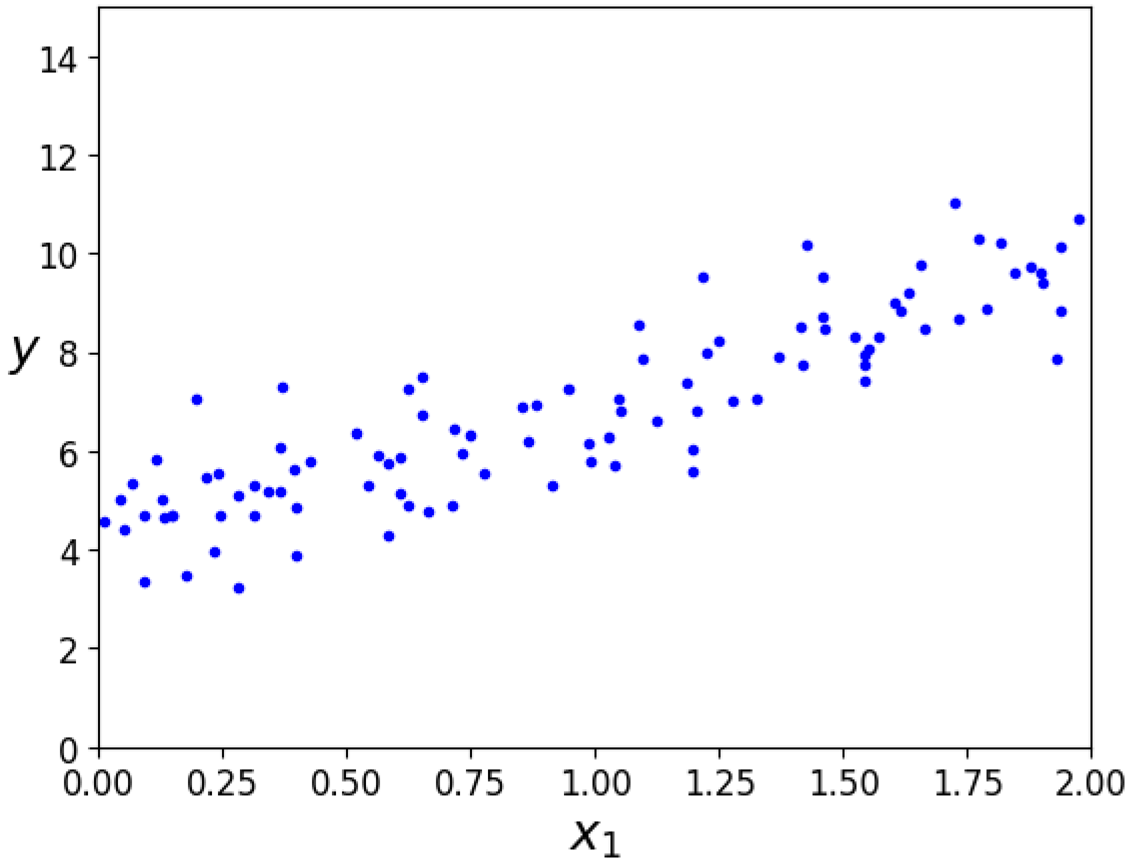
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

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

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

In [ ]:

```
import numpy as np

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

In [ ]:

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```



Tasks 1-3

Task 1

Fit a linear regression model `lin_reg` and print out its intercept and coefficient.

To start with, create a LinearRegression model object then use `lin_reg.fit(X, y)` to fit the model to the data sets X and y.

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

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

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

```
[4.21509616]
[[2.77011339]]
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

Task 2

With the model you created above, predict the new values `X_new`.

In [ ]:
```
X_new = np.array([[0], [2]])
```

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

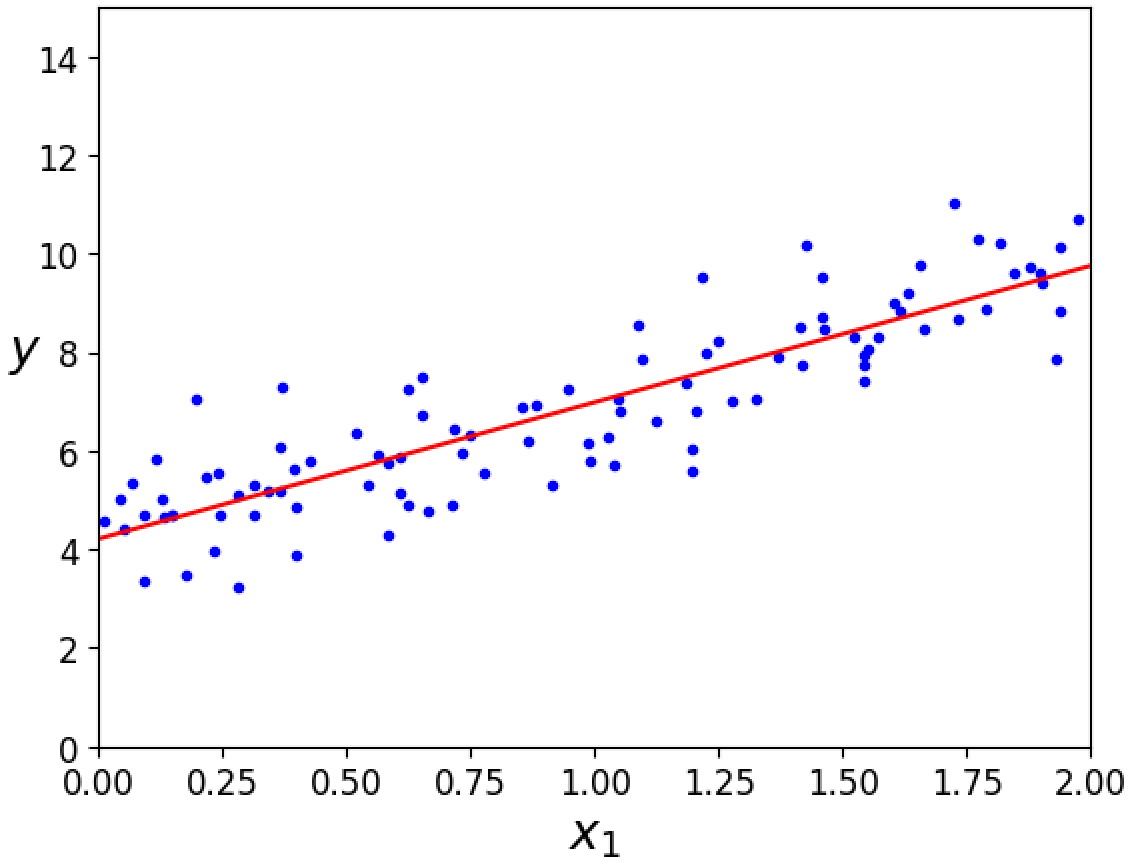
In [ ]:
```
y_predict = lin_reg.predict(X_new)
print(y_predict)
```

```
[[4.21509616]
 [9.75532293]]
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

Let's plot the data (blue) and our predictions (red) using matplotlib.

In [ ]:
```
plt.plot(X, y, "b.")
plt.plot(X_new, y_predict, "r-", markersize = 20)
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```



Task 3

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"). In essence, it attempts to fit a line by minimize the sum of squares of residuals (true - pred) for all points in a data set.

Basically we fit to the form $y = 1\Theta_0 + \Theta_1 x = x_b \cdot \Theta$ with $x_b = (1, x)^T$ and $\Theta = (\Theta_0, \Theta_1)^T$. So we need to add a 1 to each instance of `X`. See slide 13 of Lecture 4.

Look up `np.linalg.lstsq()`, call it directly on `X_b`, `y` and print out the Θ you found this way. Compare the output with the output from your `LinearRegression` model above. Set `rcond=1e-6` in `np.linalg.lstsq`.

In [ ]:
```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
print(X_b)
```

```
[[1.         0.74908024]
 [1.         1.90142861]
 [1.         1.46398788]
 [1.         1.19731697]
 [1.         0.31203728]
 [1.         0.31198904]
 [1.         0.11616722]
 [1.         1.73235229]
 [1.         1.20223002]
 [1.         1.41614516]
 [1.         0.04116899]
 [1.         1.9398197 ]
 [1.         1.66488528]
 [1.         0.42467822]
 [1.         0.36364993]
 [1.         0.36680902]
 [1.         0.60848449]
 [1.         1.04951286]
 [1.         0.86389004]
 [1.         0.58245828]
 [1.         1.22370579]
 [1.         0.27898772]
 [1.         0.5842893 ]
 [1.         0.73272369]
 [1.         0.91213997]
 [1.         1.57035192]
 [1.         0.39934756]
 [1.         1.02846888]
 [1.         1.18482914]
 [1.         0.09290083]
 [1.         1.2150897 ]
 [1.         0.34104825]
 [1.         0.13010319]
 [1.         1.89777107]
 [1.         1.93126407]
 [1.         1.6167947 ]
 [1.         0.60922754]
 [1.         0.19534423]
 [1.         1.36846605]
 [1.         0.88030499]
 [1.         0.24407647]
 [1.         0.99035382]
 [1.         0.06877704]
 [1.         1.8186408 ]
 [1.         0.51755996]
 [1.         1.32504457]
 [1.         0.62342215]
 [1.         1.04013604]
 [1.         1.09342056]
 [1.         0.36970891]
 [1.         1.93916926]
 [1.         1.55026565]
 [1.         1.87899788]
 [1.         1.7896547 ]
 [1.         1.19579996]
 [1.         1.84374847]
 [1.         0.176985  ]
 [1.         0.39196572]
 [1.         0.09045458]
 [1.         0.65066066]
 [1.         0.77735458]
 [1.         0.54269806]
 [1.         1.65747502]
 [1.         0.71350665]
 [1.         0.56186902]
 [1.         1.08539217]
 [1.         0.28184845]
 [1.         1.60439396]
 [1.         0.14910129]
 [1.         1.97377387]
 [1.         1.54448954]
 [1.         0.39743136]
 [1.         0.01104423]
 [1.         1.63092286]
 [1.         1.41371469]
 [1.         1.45801434]
 [1.         1.54254069]
 [1.         0.1480893 ]
 [1.         0.71693146]
 [1.         0.23173812]
 [1.         1.72620685]
 [1.         1.24659625]
 [1.         0.66179605]
 [1.         0.1271167 ]
 [1.         0.62196464]
 [1.         0.65036664]
 [1.         1.45921236]
 [1.         1.27511494]
 [1.         1.77442549]
 [1.         0.94442985]
 [1.         0.23918849]
 [1.         1.42648957]
```

```
[1.         1.5215701 ]
[1.         1.1225544 ]
[1.         1.54193436]
[1.         0.98759119]
[1.         1.04546566]
[1.         0.85508204]
[1.         0.05083825]
[1.         0.21578285]]
```

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

In [ ]:
```
import numpy as np

theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)

print(theta_best_svd)
```

```
[[4.21509616]
 [2.77011339]]
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

In [ ]:
```
# these are the values from the linear regression model
# compare them to what you got above
lin_reg.intercept_, lin_reg.coef_
```

Out[ ]:
```
(array([4.21509616]), array([[2.77011339]]))
```

Linear regression using batch gradient descent

Just click through this section.

The first part shows how to manually do gradient descent. The formula for the gradients has been calculated by hand and we just plug in `X` and `theta`.

The second part (function `plot_gradient_descent`) does the gradient descent and plots the first 10 steps.

Here is a nice visualization of different gradient descent methods.

In [ ]:
```
eta = 0.1  # learning rate
n_iterations = 1000 # number of iterations
m = 100 # the number of items in X_b

theta = np.random.randn(2,1)  # random initialization
print(theta) # printing what our initial theta values look like

# The following loop is the entire gradient descent algorithm for this
# linear regression example. As a bonus question below you can attempt to
# describe what is actually happening here.
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
[[0.01300189]
 [1.45353408]]
```

Notice the change in our value for theta after running this algorithm.

In [ ]:
```
theta
```

Out[ ]:
```
array([[4.21509616],
       [2.77011339]])
```

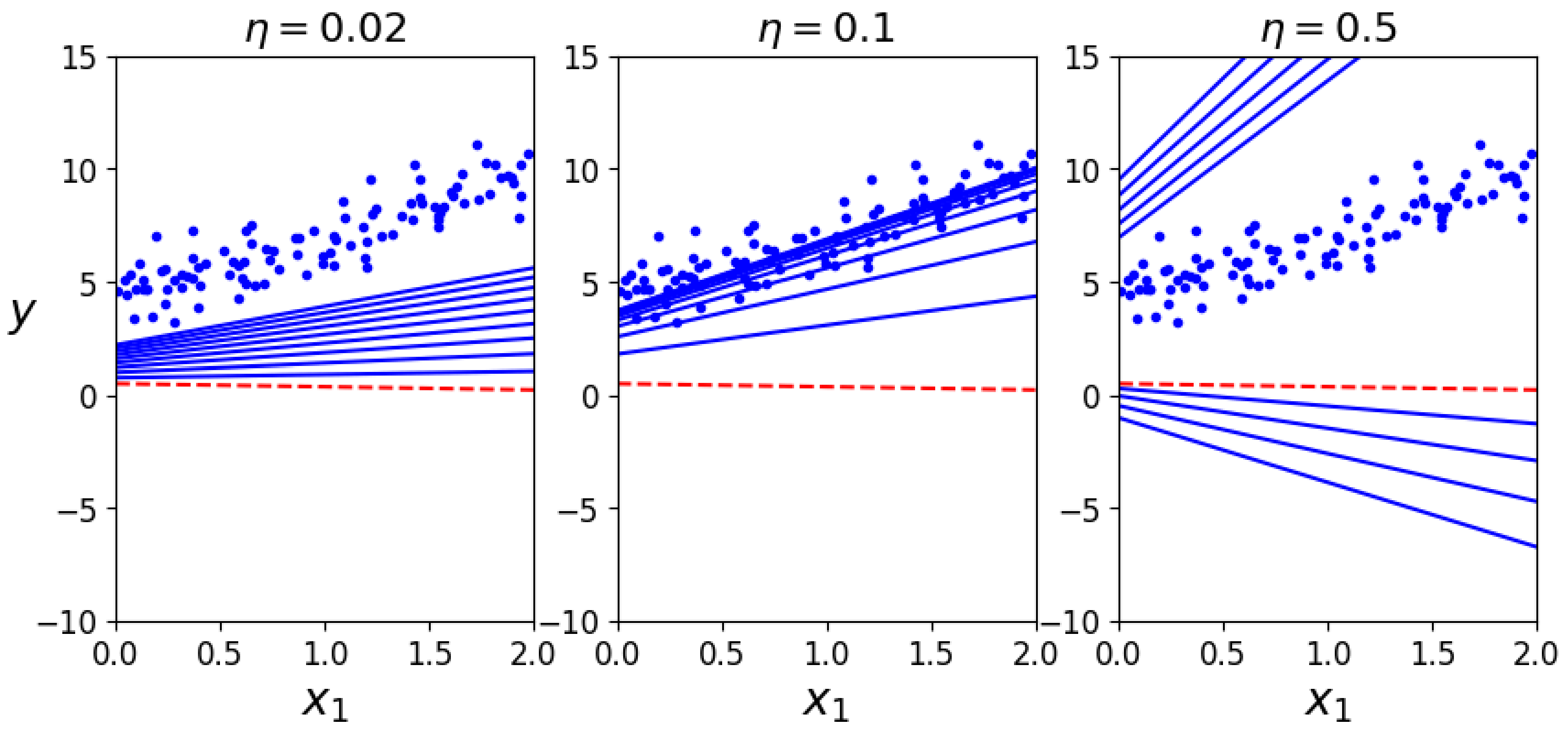
Don't worry too much about what all of the code below is doing. In short, it's **running the gradient descent algorithm and updating a plot** with increasingly better-fitted regression lines.

In [ ]:
```
theta_path_bgd = []

def plot_gradient_descent(theta, eta, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    X_new = np.array([[0], [2]])
    X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)   # this line
        theta = theta - eta * gradients   # and this one
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, -10, 15])
    plt.title(r"$\eta = {}$".format(eta), fontsize=16)
```

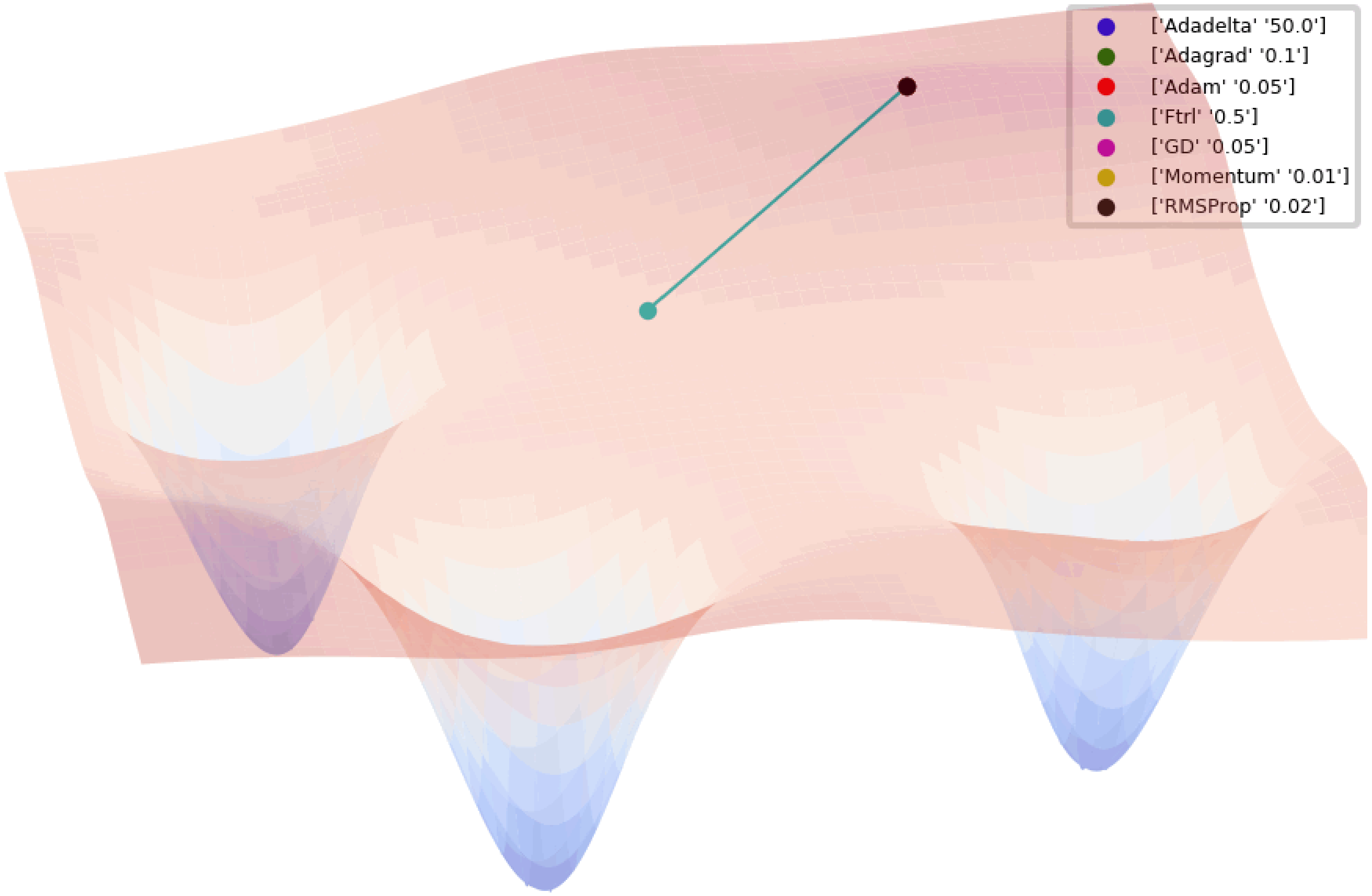
Now that we've made a function to do gradient descent and plot our regressions, we can **test out a few different learning rate hyper-parameters**.

In [ ]:

```
np.random.seed(42) # this keeps your random outputs consistent each time
theta = np.random.randn(2,1)  # random initialization
theta_path_bgd = []

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, eta=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, eta=0.1, theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, eta=0.5)

plt.show()
```



For a learning rate η that is too small we approach the minimum too slowly and for a too large learning rate we jump around the minimum like the purple curve in this animation from this repo on GitHub. Note that this animation does not correspond to our data.



In the above plot the axis of the plane could represent two different model **parameters** and the vertical axis represents your **"Loss"** or measure of error. As you move the points around across the plane you might **increase** or **decrease** the loss. So when you are at the bottom of one of the valleys you've found a **combination of parameters** (a point along the plane) that result in a low loss. Gradient Descent and other optimization algorithms help you to take steps in a direction that gets this point to the **lowest valley possible**. The point along the plane that corresponds to being in the **deepest valley** possible represents the **best** combination of model parameters.

Task 3.5 (Bonus)

Explain what happens in the two lines of code

```
gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)   # this line
theta = theta - eta * gradients   # and this one
```
in the code above. Also explain where the first line comes from.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 3.5 (bonus) answer:

The first line computes the gradient of the cost function with respect to the model parameters. It comes from the partial derivatives of the cost function. The second line performs the update of the model parameters theta by subtracting the scaled gradients from the current value of theta. The scaling is done by the learning rate eta.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Stochastic Gradient Descent

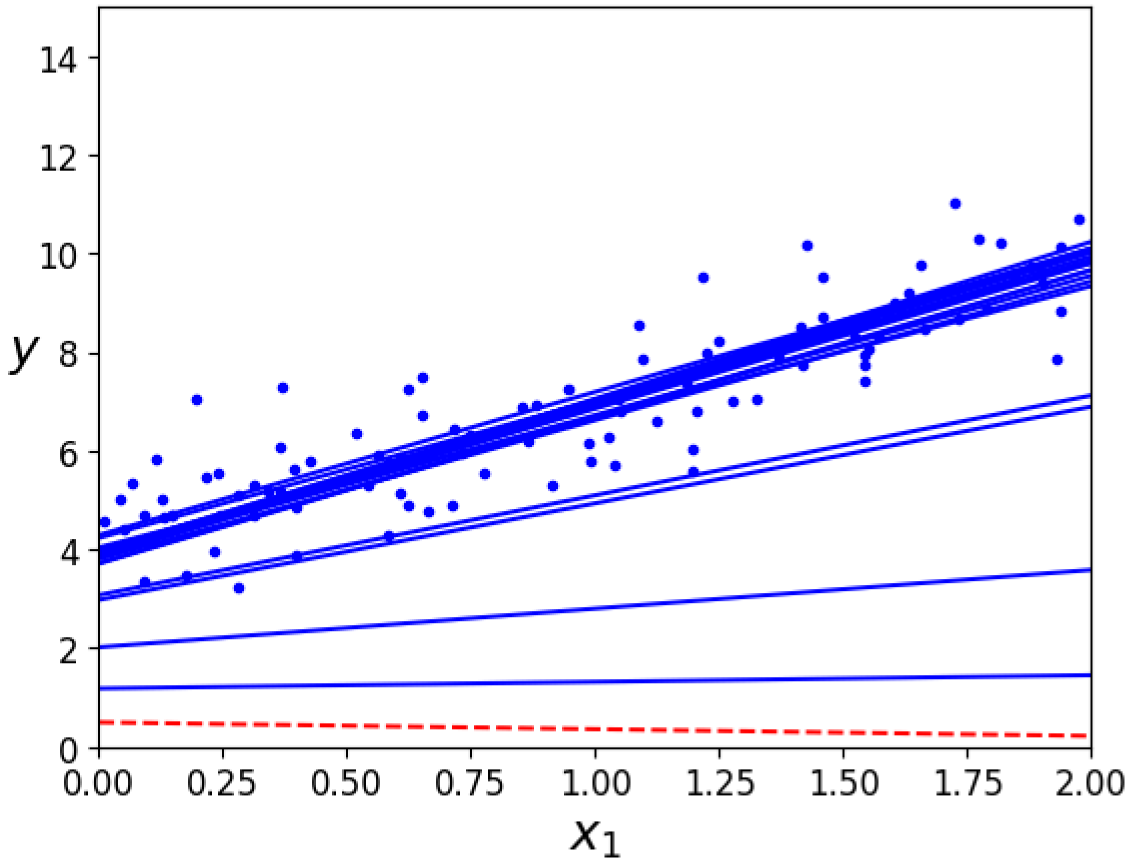
This section shows how one can implement Stochastic Gradient Descent by hand. You might notice that **this looks a lot like the gradient descent method** from above. The difference here, is that **we're adding stochasticity or randomness**. We do this by **performing gradient descent on a random subset** of all data points for each step **rather than on the entire data set** every time.

In [ ]:
```
theta_path_sgd = [] # a list where we can store parameters of our fit
m = len(X_b) # the number of items in our data set
```

```
np.random.seed(42)
```

Notice the lines below that are marked **#here**. Those lines are where **we introduce stochasticity by picking out a random single data point**. **Don't worry too much about whether you understand what's happening in this code**. In practice, we'll usually use other tools to implement this automatically rather than code it by hand.

In [ ]:
```
n_epochs = 50 # number of times to fully iterate over our data set
t0, t1 = 5, 50  # learning schedule hyperparameters

# Sometimes we want to be able to change our learning rate over time.
# If we start with a learning rate of 1. Our next value would be 5/(1+50).
def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance

for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        random_index = np.random.randint(m)   #here
        xi = X_b[random_index:random_index+1] #here
        yi = y[random_index:random_index+1]   #here
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i) # reduce our learning rate
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
plt.show()
```



```
In [ ]: theta
```

```
Out[ ]: array([[4.21076011],
               [2.74856079]])
```

Task 4

- Build an SGD Regressor and assign it to `sgd_reg`.
- Fit the SGD Regressor `sgd_reg` to `X` and `y` and print out its intercept `sgd_reg.intercept_` and coefficients `sgd_reg.coef_`.
- Compare the values you find to the ones from the stochastic gradient descent above.

Use `max_iter=50, tol=None, penalty=None, eta0=0.1, random_state=42` as parameters for the SGD Regressor.

```
In [ ]: from sklearn.linear_model import SGDRegressor
        from sklearn.pipeline import make_pipeline
        from sklearn.preprocessing import StandardScaler
```

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

```
In [ ]: sgd_reg = SGDRegressor(max_iter=50, tol= None, penalty= None, eta0=0.1, random_state=42)
        reg = sgd_reg.fit(X, y.ravel())
        print(reg.intercept_)
        print(reg.coef_)
```

```
[4.16782089]
[2.72603052]
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

Mini-batch gradient descent

The following code shows how one could implement mini-batch gradient descent by hand. **Remember, the stochastic part of SGD means we picked out a random subset of our data**. Above, that was only a single point. How do you think mini-batch gradient descent differs from stochastic gradient descent? As a hint, look for `#pay close attention to this line`.

```
In [ ]: theta_path_mgd = []

        n_iterations = 50
        minibatch_size = 20

        np.random.seed(42)
        theta = np.random.randn(2,1)  # random initialization

        t0, t1 = 200, 1000
        def learning_schedule(t):
            return t0 / (t + t1)

        t = 0
        for epoch in range(n_iterations):
            shuffled_indices = np.random.permutation(m)
            X_b_shuffled = X_b[shuffled_indices]
            y_shuffled = y[shuffled_indices]
```

```
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size] #pay close attention to this line
        yi = y_shuffled[i:i+minibatch_size]   #pay close attention to this line
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)
```

In [ ]:

```
theta
```

Out[ ]:

```
array([[4.25214635],
       [2.7896408 ]])
```

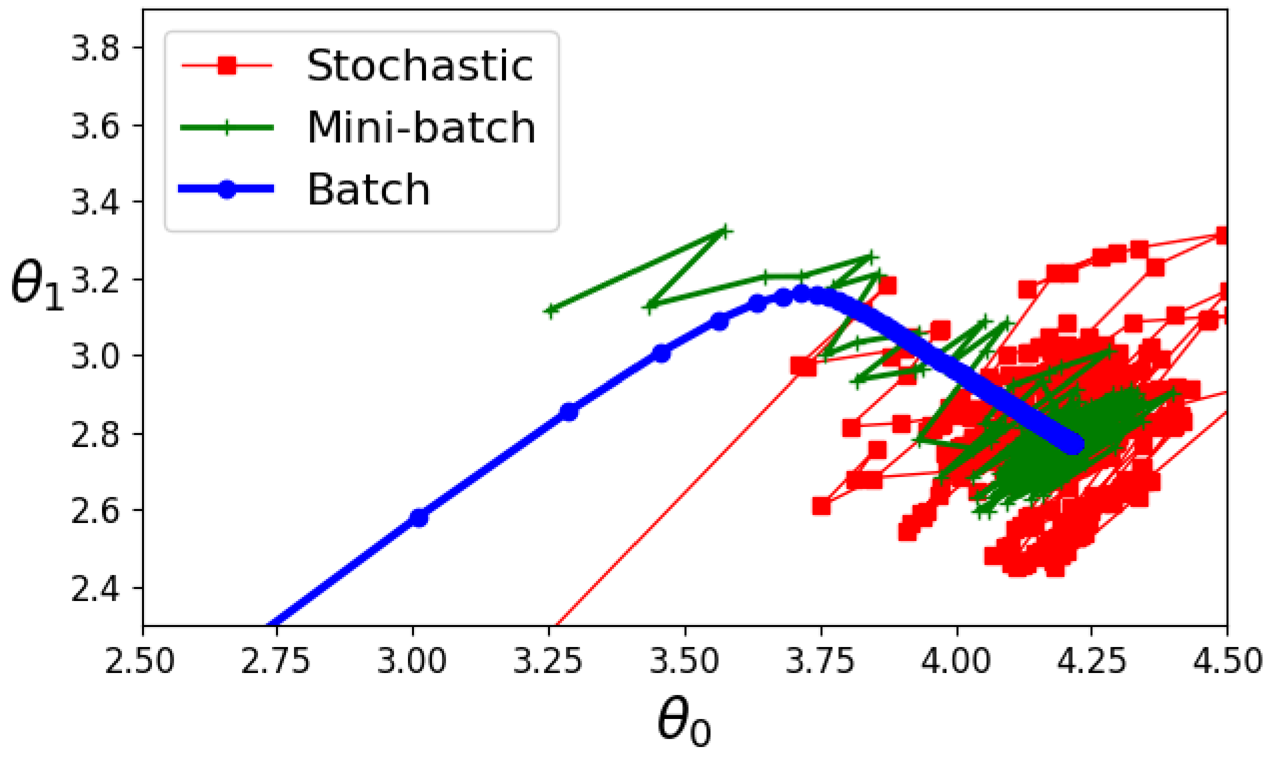
In [ ]:

```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

Let's see how these three different approaches to gradient descent compare with one another.

In [ ]:

```
plt.figure(figsize=(7,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1, label="Stochastic")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", linewidth=2, label="Mini-batch")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", linewidth=3, label="Batch")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
plt.show()
```



Task 5

Explain which Linear Regression training algorithm you can use if you have a training set with millions of features? Why?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 5 answer:

Mini-Batch Gradient Descent might be the best choice. It is more efficient than Batch GD since it does not require computing the full dataset at each step. Additionally, it is more stable than SGD (less oscillation due to batch averaging). However, if memory constraints are extreme, SGD is another option, but it requires careful learning rate to ensure smooth convergence.

Batch GD is not recommended, as it is computationally expensive and infeasible for datasets with millions of features.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Polynomial regression

In [ ]:
```
import numpy as np
import numpy.random as rnd

np.random.seed(42)
```

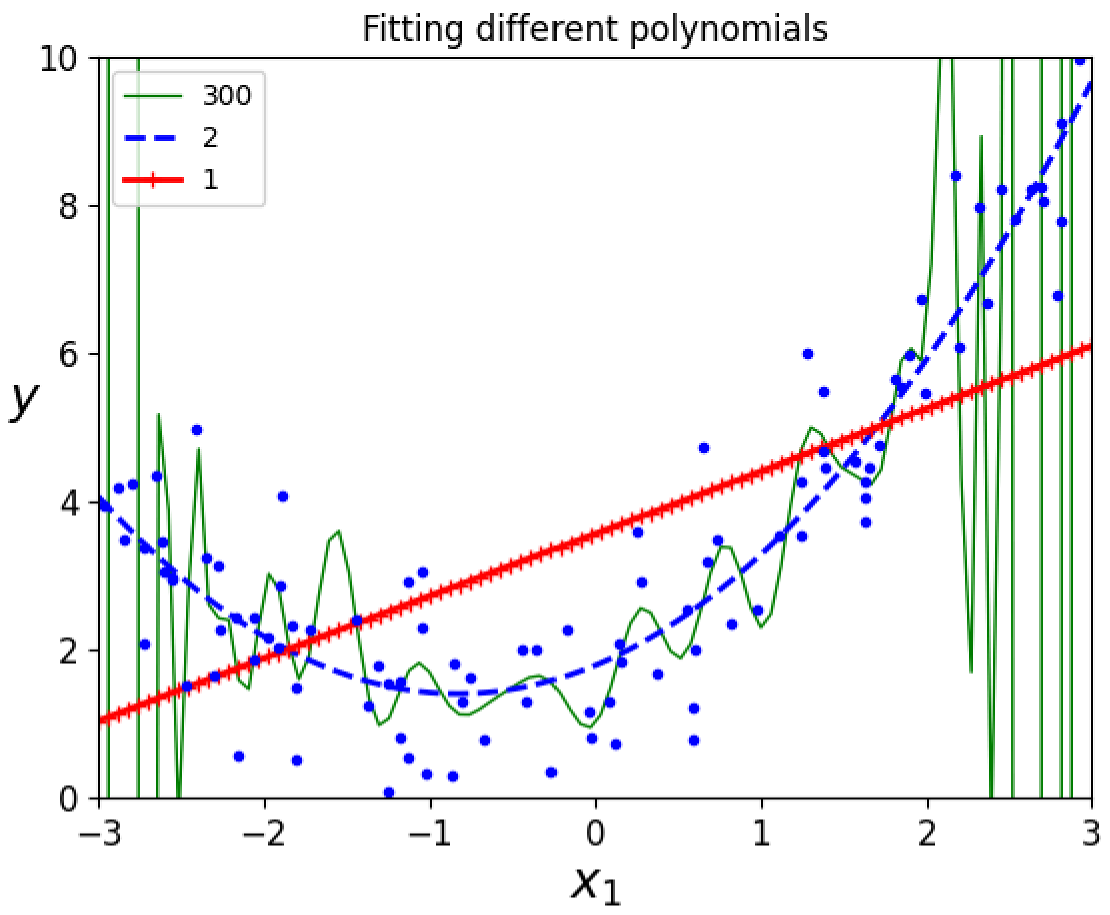
With a little trick Linear Regression turns out to be rather powerful also for data that is not linear!
The trick is to **add "new features" to `X`**. In our case we need a second feature that is just x^2. **Then we fit again**, but because **our feature vector is now** $(1, x, x^2)$, the **dot product with** $(\Theta_0, \Theta_1, \Theta_2)$ **gives the form for the parabola** (check this if you don't see it!).

Here we show an example of how to fit a parabola with Linear Regression.

You do not need to understand the code in detail but we've added comments if you want to understand it better.

In [ ]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

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

# From the documentation for PolynomialFeatures:
# Generate a new feature matrix consisting of all polynomial combinations of
# the features with degree less than or equal to the specified degree. For
# example, if an input sample is two dimensional and of the form [a, b], the
# degree-2 polynomial features are [1, a, b, a^2, ab, b^2].
poly_features = PolynomialFeatures(degree=2, include_bias=False)

# We can then fit to and transform our 100 X values to look like a polynomial.
X_poly = poly_features.fit_transform(X)

# LinearRegression fits the polynomials with least squares fitting.
```

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

# We can also transform other sets of X values without needing to fit first
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)

# With some new polynomial X values that the model hasn't seen, we can get
# some predicted values.
y_new = lin_reg.predict(X_new_poly)

# The following for loop is going to repeat for some polynomials of different
# degrees and create a regression  plot with corresponding design elements.
for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    # Pipeline is a tool that lets you apply multiple sklearn layers to data
    # in a sequential way. Here, we're transforming our data to be polynomial,
    # then scaling the data using standard normalization, and finally running
    # a linear regression on the data.
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    # with our Pipeline constructed, we can fit our model to some data.
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.title("Fitting different polynomials")
plt.axis([-3, 3, 0, 10])
plt.show()
```



Task 6

- Which curve (red, blue, green) fits the data (blue dots) the best?
- Explain what happens to the green curve.
  Hint: ("g-", 1, 300) = style, width, degree

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 6 answer:

1. The blue curve fits the data the best.

2. The green curve oscillates wildly between the data points, especially in regions without data, as it tries to minimize the error for every individual point.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Training and Validation Error

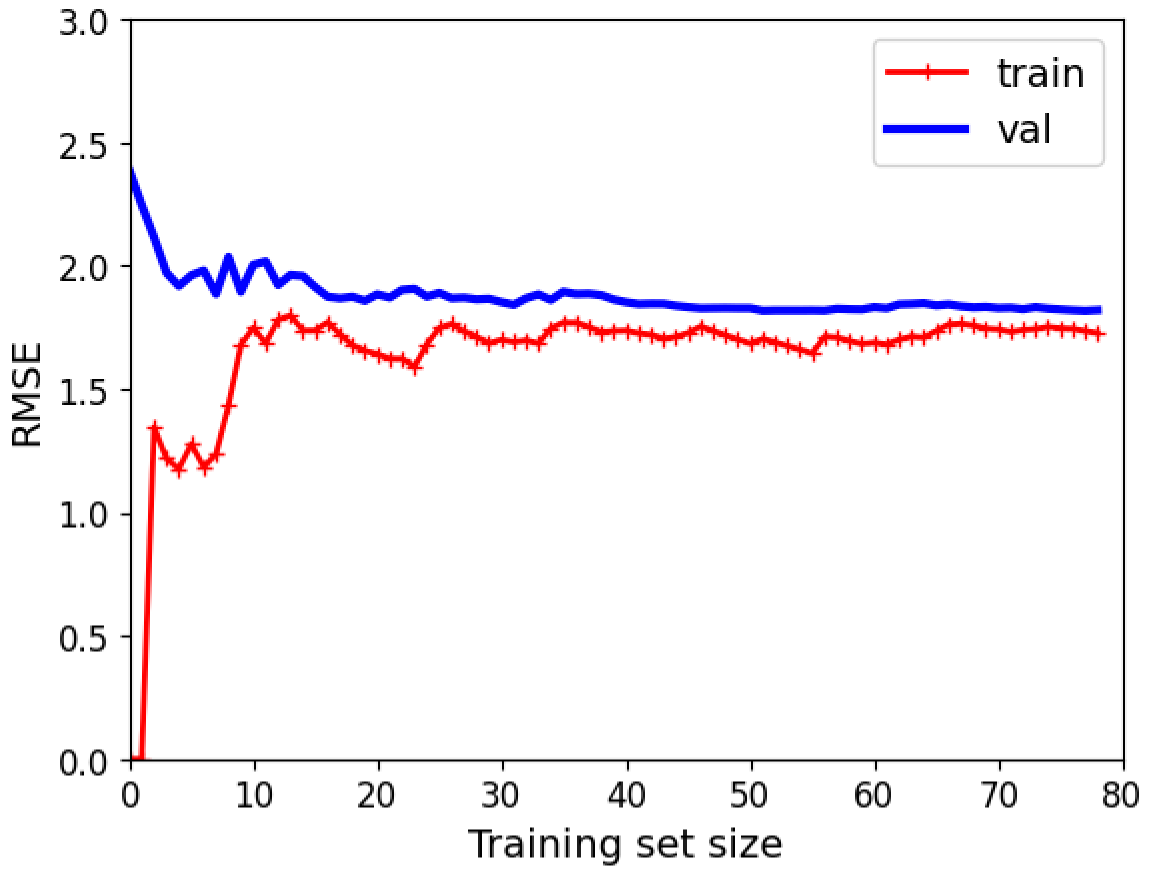
Learning curves are a visualization of your error over time. Here we're looking at plots of mean-squared error (true-pred)^2 for both our training data and our validation data over the course of training. By **checking BOTH training and validation curves**, we can verify that our model is **learning AND generalizing**.

Not all machine learning toolkits keep track of your loss history automatically. Here's an example doing it by hand with lists.

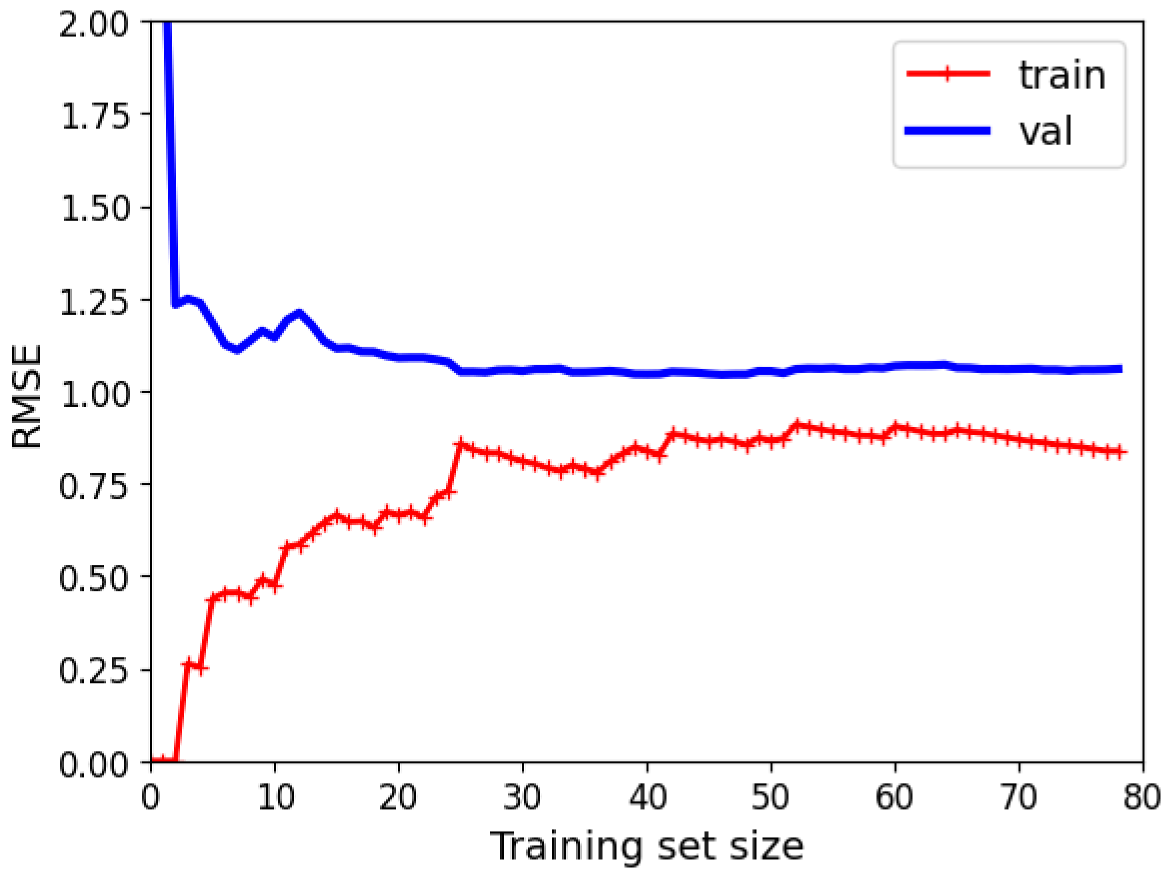
In [ ]:
```python
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    # train_test_split splits data into train and test sets automatically
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=10)
    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)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```

Let's compare different degrees of polynomials for fitting. We're going to **generate some learning curves for increasingly complicated polynomial forms. Degree = 1, 2, 30**.

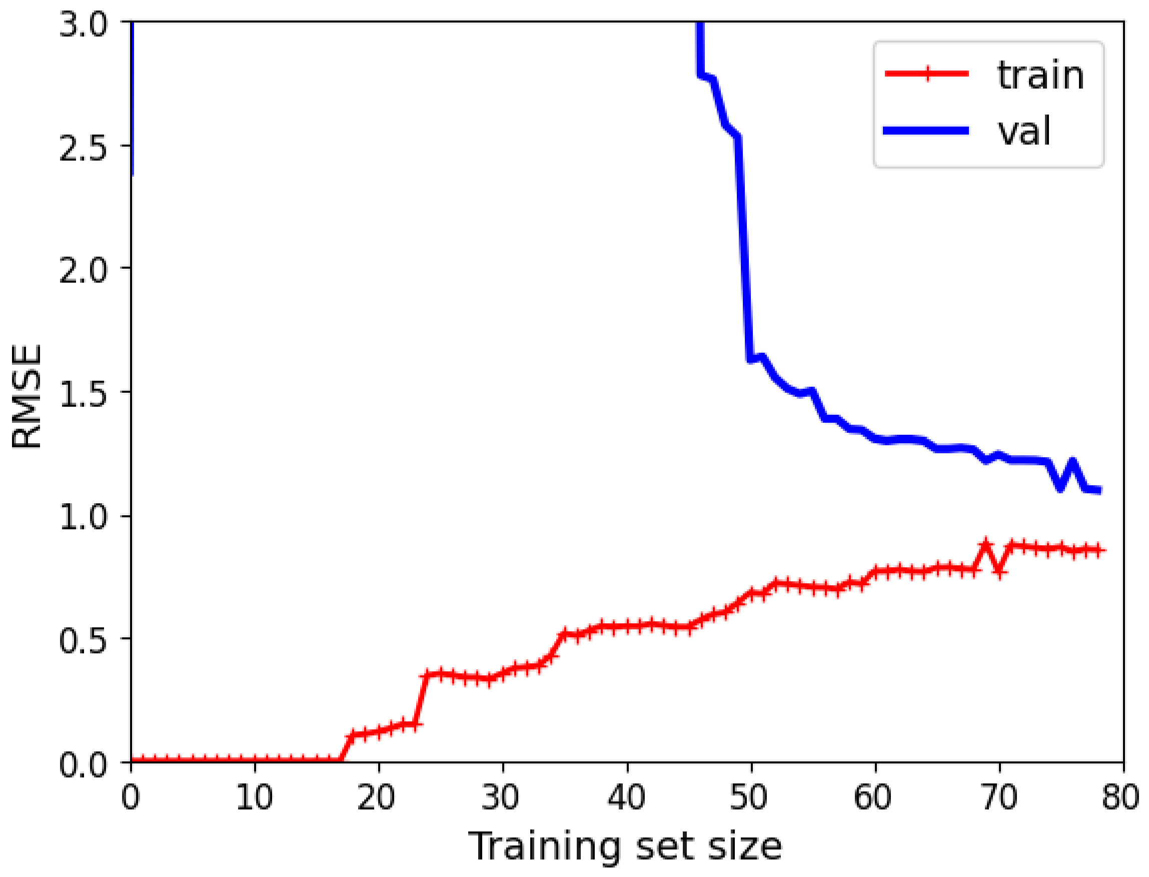
In [ ]:
```python
# Linear Regression (degree 1)
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])
plt.show()
```



In [ ]:
```python
from sklearn.pipeline import Pipeline

# Degree 2
polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=2, include_bias=True)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 2])
plt.show()
```



In [ ]:
```
# Degree 20
polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=30, include_bias=True)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])
plt.show()
```



Task 7

Look at the three plots above that show the training and validation errors for linear regression, quadratic regression and order 20. Compare them to the plot above called "Fitting different polynomials".

Describe which one is likely overfitting and which one is likely underfitting and why.

Hint: Also take the values of the errors into consideration when comparing the different polynomials and not just the shape of the train and validation loss.

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 7 answer:

Linear regression is underfitting, RMSE are high for both training and validation, indicating the model is too simple.

Degree 30 is overfitting. Very low training error but much higher validation error, indicating poor generalization

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Regularized models

Regularization is one approach to help prevent overfitting. The general idea is to **apply a penalty to model coefficients which discourages overly-complex models**.

In [ ]:
```
import numpy as np
np.random.seed(42)
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + X*X + 1.5 * np.random.randn(m, 1) / 2
X_new = np.linspace(0, 3, 1000).reshape(1000, 1)
```

Task 8

Ridge regression is one such form of regularized regression.

Create a ridge linear model with `alpha=1` and Stochastic Average Gradient descent (`solver="sag"`) solver to `X` and `y`.
Fit (`.fit()`) your model to `X` and `y`.
Check which value it predicts (`.predict()`) for `X=[[1.5]]`.

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

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

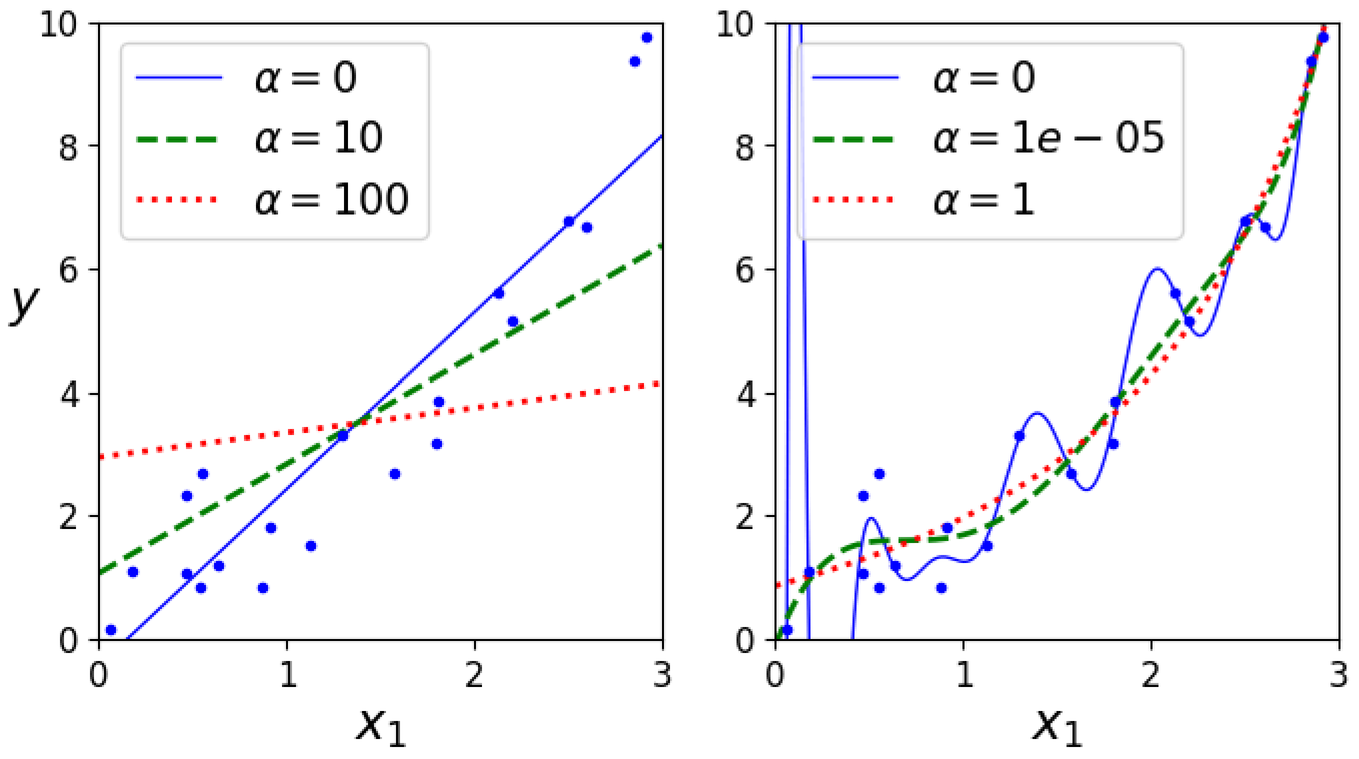
In [ ]:
```
ridge_reg_sag = Ridge(alpha=1.0, solver="sag")
ridge_reg_sag.fit(X, y)
ridge_reg_sag.predict([[1.5]])
```

Out[ ]:
```
array([3.83288259])
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

Now we fit a linear model (left plot) and a degree 15 polynomial model
(right plot) with ridge regression for different values of alpha.

In [ ]:
```
from sklearn.linear_model import Ridge
import matplotlib.pyplot as plt

# Most of the following function should loook familiar from other fit and plot
# loops in this hands-on
def plot_model(model_class, polynomial, alphas, **model_kargs):
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=15, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 10])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)

plt.show()
```



Task 9

- Describe the effect of the regularization parameter alpha in the plot on the right hand side.
- Which curve (blue, green, red) would you expect to have the smallest training loss?
- Which curve would you expect to generalize the best to new data?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 9 answer:

1. Blue curve: no regularization is applied. This model overfits the training data.

Green curve: a small amount of regularization is applied, slightly smoothing the curve compared to the blue one.

Red curve: strong regularization is applied, resulting in a much smoother curve.

2. The blue curve will have the smallest training loss.

3. The red curve is expected to generalize the best to new data.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

We can also add regularization penalties as arguments to other sklearn model classes. **Below we'll show a couple of other ways to add regularization to your model training**.

L2 = Ridge regularization = square value regularization
L1 = Lasso regularization = absolute value regularization

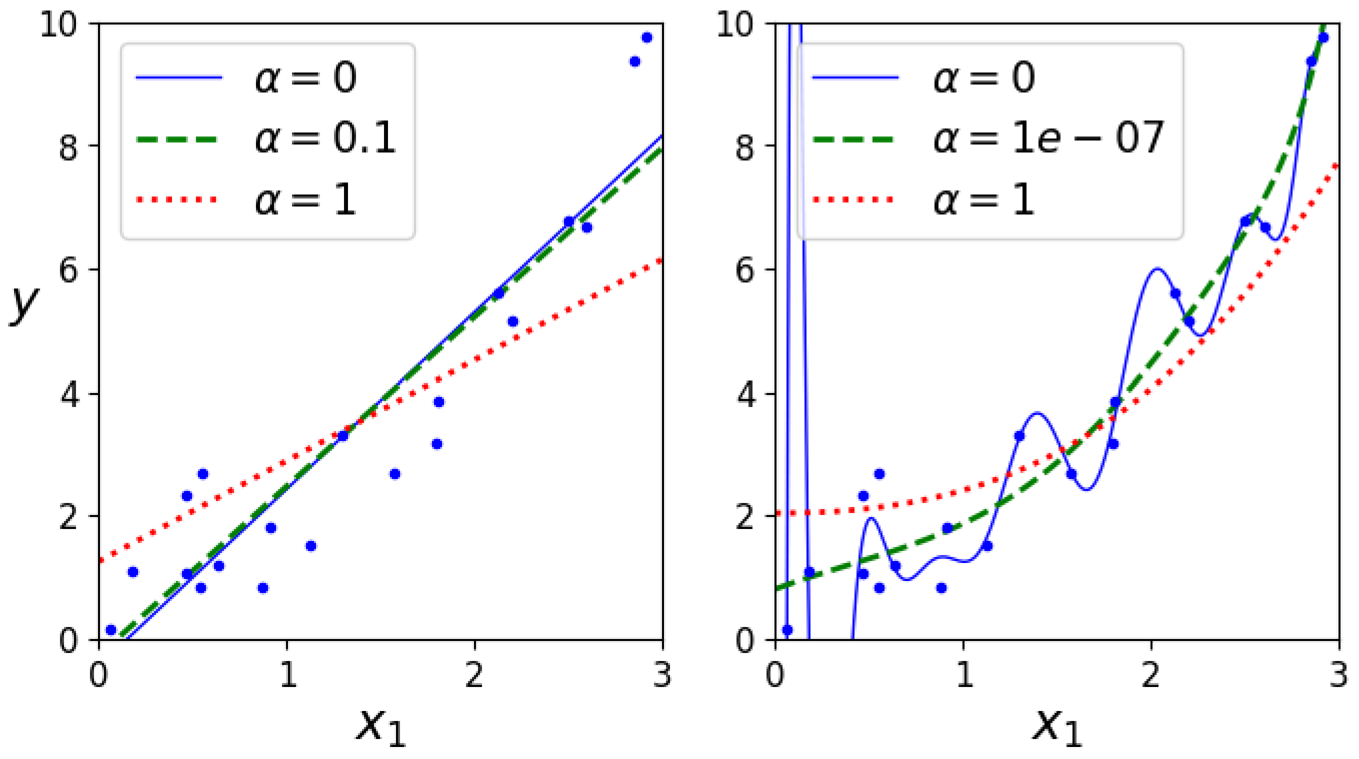
In [ ]:
```
sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42)
sgd_reg.fit(X, y.ravel())
sgd_reg.predict([[1.5]])
```

Out[ ]:
```
array([3.95689619])
```

The same, but with Lasso:

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

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), random_state=42)

plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_coordinate_descent.py:695: ConvergenceWarning: Objective did not c
onverge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisat
ion. Duality gap: 3.442e+00, tolerance: 1.539e-02
  model = cd_fast.enet_coordinate_descent(
```



Task 10

create a Lasso linear model with `alpha=1` and assign it to `lasso_reg`. Fit the model to X and y using `.fit()`. Check which value it predicts for `X=[[1.5]]`.

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

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below

```
In [ ]: lasso_reg = Lasso(alpha=1)
        lasso_reg.fit(X, y)
        lasso_reg.predict([[1.5]])
```

```
Out[ ]: array([3.6972954])
```

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

Early stopping example

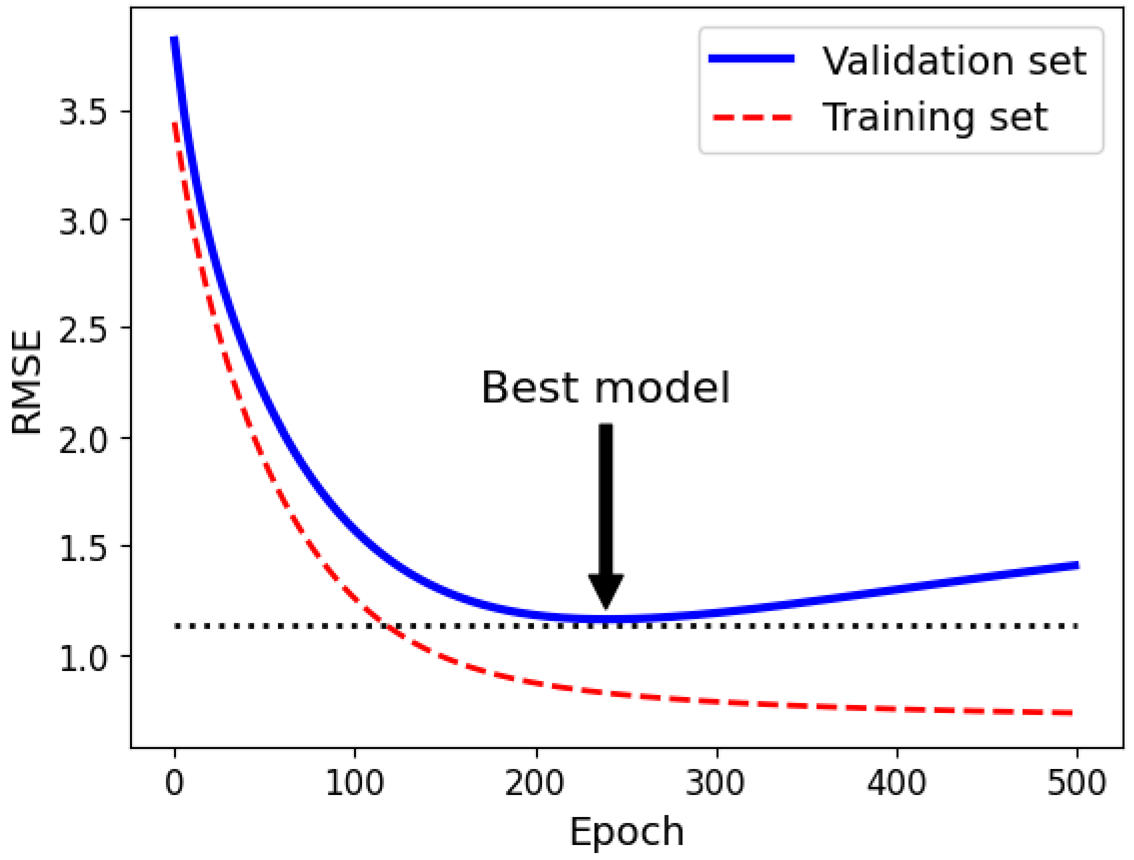
Sometimes our final model at the end of training is not as good as the best model from the whole training procedure. This could be true if, for example, your model overfits to the training data. **Below we'll show how you can save the best model from all of training with sklearn**.

In [ ]:
```
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_size=0.5, random_state=10)
```

In [ ]:
```
from copy import deepcopy

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=None, 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: # Check if this is our best model
        minimum_val_error = val_error # Overwrite the minimum error
        best_epoch = epoch # Record the best epoch

        # Notice how we save a deep copy of the model instead of just writing
        # best_model = sgd_reg. This is because assigning a variable to a
        # different variable can sometimes cause python to treat them as one
        # object; meaning changes made to one would affect the other.
        best_model = deepcopy(sgd_reg)
```

We can also do something similar when searching for good hyper-parameters (e.g. number of epochs) by **saving our loss history instead of saving the best model**.

In [ ]:
```
sgd_reg = SGDRegressor(max_iter=1, tol=None, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005, random_state=42)

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))

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

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()
```



Task 11

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?
What would be options for fixing this?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 11 answer:

I guess my learning rate is too high. set a smaller learning rate

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Task 12

We want to find at which exact epoch our model is the best.
For that we go through 1000 "epochs".

We have already prepared part of the code for this task. You only have to fill in the last part which should do the following:

- if the validation error (of the current epoch) is smaller than the smallest validation error until now (`minimum_val_error`), then:
  - set the `minimum_val_error` to be the current validation error
  - set the `best_epoch` to be the current epoch
  - set the `best_model` to be the current model (`clone(sgd_reg)`)
- else: next epoch (you can also just leave out `else`)

You can check the early stop example above if you're unsure how to do this.

In [ ]:
```
from sklearn.base import clone
sgd_reg = SGDRegressor(max_iter=1, tol=None, 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)
    # ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your code goes below
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
    # ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your code goes above this

print("Best epoch: ", best_epoch)
```

```
Best epoch:  239
```

Task 12.5 bonus question: Why do we need `clone(sgd_reg)` here and not just `sgd_reg`?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 12.5 (bonus) answer:

If we assign sgd_reg to best_model directly , best_model will reference the same object as sgd_reg. Since sgd_reg.fit() modifies the object , any further training will also modify best_model. This means that best_model will no longer correspond to the model at the epoch where the validation error was lowest.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

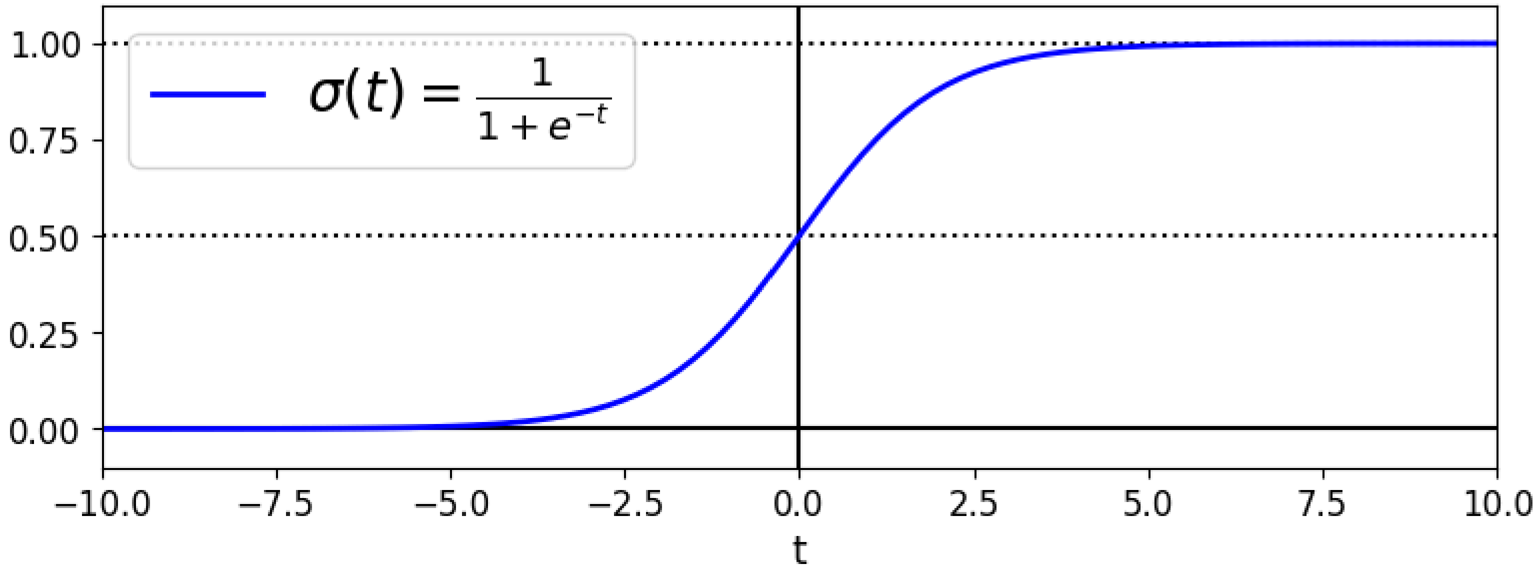
Logistic regression

Below we'll see an example of a model that's sometimes referred to as a **logistic regression classifier**. The model essentially tries to map multiple variables to the log-odds of a particular class being selected in such a way that the log-odds is a linear combination of the variables.

Note: logistic regression models include:

- maximum entropy models (the multi-class case)
- binomial logistic regression models (the binary case)

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()
```



From here on we will work with the Iris Flower Data Set.

First, let's import that dataset from sklearn's collection of datasets.

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 [ ]:
```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

:Number of Instances: 150 (50 in each of three classes)
:Number of Attributes: 4 numeric, predictive attributes and the class
:Attribute Information:
    - sepal length in cm
    - sepal width in cm
    - petal length in cm
    - petal width in cm
    - class:
            - Iris-Setosa
            - Iris-Versicolour
            - Iris-Virginica

:Summary Statistics:

============== ==== ==== ======= ===== ====================
                Min  Max   Mean    SD   Class Correlation
============== ==== ==== ======= ===== ====================
sepal length:   4.3  7.9   5.84   0.83    0.7826
sepal width:    2.0  4.4   3.05   0.43   -0.4194
petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
============== ==== ==== ======= ===== ====================

:Missing Attribute Values: None
:Class Distribution: 33.3% for each of 3 classes.
:Creator: R.A. Fisher
:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
:Date: July, 1988

The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken
from Fisher's paper. Note that it's the same as in R, but not as in the UCI
Machine Learning Repository, which has two wrong data points.

This is perhaps the best known database to be found in the
pattern recognition literature.  Fisher's paper is a classic in the field and
is referenced frequently to this day.  (See Duda & Hart, for example.)  The
data set contains 3 classes of 50 instances each, where each class refers to a
type of iris plant.  One class is linearly separable from the other 2; the
latter are NOT linearly separable from each other.
```

```
.. dropdown:: References

  - Fisher, R.A. "The use of multiple measurements in taxonomic problems"
    Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
    Mathematical Statistics" (John Wiley, NY, 1950).
  - Duda, R.O., & Hart, P.E. (1973) Pattern Classification and Scene Analysis.
    (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
  - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
    Structure and Classification Rule for Recognition in Partially Exposed
    Environments".  IEEE Transactions on Pattern Analysis and Machine
    Intelligence, Vol. PAMI-2, No. 1, 67-71.
  - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
    on Information Theory, May 1972, 431-433.
  - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
    conceptual clustering system finds 3 classes in the data.
  - Many, many more ...
```

Let's only use one feature: The petal width

`iris["data"][:, 3:]` is getting the values for petal width.

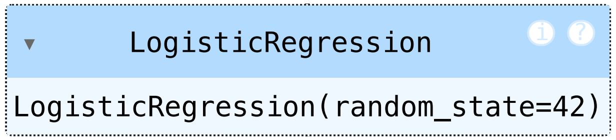
We need to convert our labels to integers (they're strings right now).

In [ ]:
```python
X = iris["data"][:, 3:]  # petal width
y = (iris["target"] == 2).astype(np.int32)  # 1 if Iris virginica, else 0
```

Let's create a logistic regression model and fit it to our data.

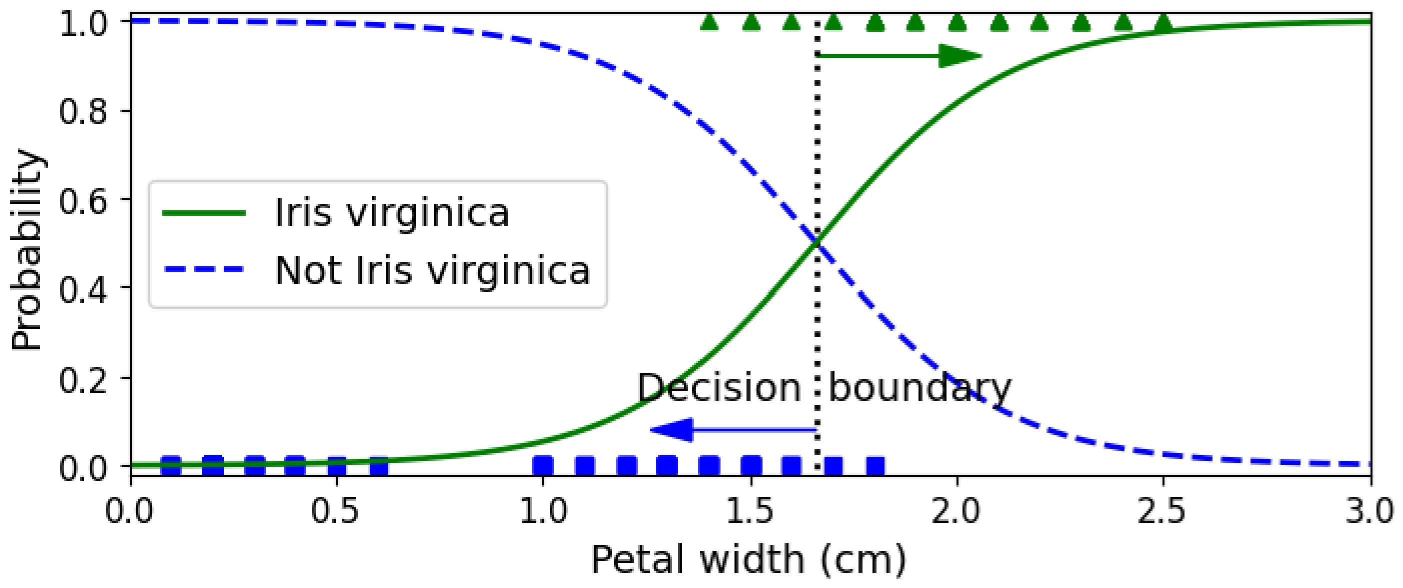
In [ ]:
```python
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver="lbfgs", random_state=42)
log_reg.fit(X, y)
```

Out[ ]:



```
LogisticRegression
LogisticRegression(random_state=42)
```

In [ ]:
```python
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0][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", ha="center")
plt.arrow(decision_boundary, 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='b', ec='b')
plt.arrow(decision_boundary, 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g', ec='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()
```



In [ ]:
```
decision_boundary
```

Out[ ]:
```
1.6606606606606606
```

The decision boundary is at Petal width = 1.66cm.
Let's see what we get if we predict the class of an instance slightly
to the left (1.5) and one slightly to the right (1.7).

In [ ]:
```
# if you are motivated: Try to understand this output
log_reg.predict_proba([[1.7], [1.5]])
```

Out[ ]:
```
array([[0.45713982, 0.54286018],
       [0.66699864, 0.33300136]])
```

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

Out[ ]:
```
array([1, 0], dtype=int32)
```

Task 13

What is the class prediction (iris virginica or not iris virginica?)
for petal width = 1.7 and what is the prediction for petal width = 1.5?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 13 answer:

For petal width = 1.7, the class prediction is iris virginica. For petal width = 1.5, the prediction is "not iris virginica".

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Now let's use two features for predicting the class:

The petal length and the petal width.
We also use all classes this time and not just two like above.

Extension to 2 dimensions

There are no tasks for this part. You can try to understand what
happens or skip if you want.

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

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10, random_state=42)
softmax_reg.fit(X, y)
```

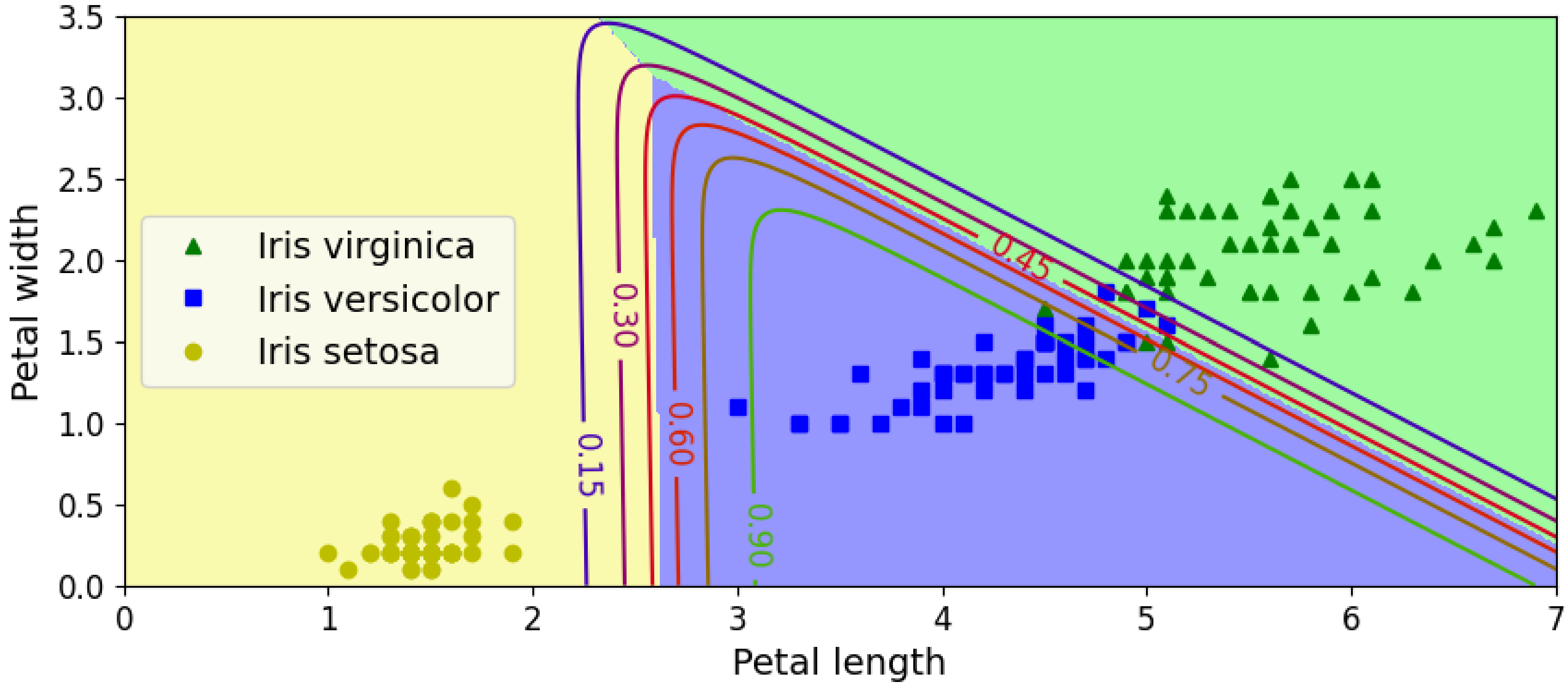
```
/usr/local/lib/python3.11/dist-packages/sklearn/linear_model/_logistic.py:1247: FutureWarning: 'multi_class' was deprecated in v
ersion 1.5 and will be removed in 1.7. From then on, it will always use 'multinomial'. Leave it to its default value to avoid th
is warning.
  warnings.warn(
```

Out[ ]:

```
▾ LogisticRegression
LogisticRegression(C=10, multi_class='multinomial', random_state=42)
```

In [ ]:

```python
x0, x1 = np.meshgrid(
        np.linspace(0, 8, 500).reshape(-1, 1),
        np.linspace(0, 3.5, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]


y_proba = softmax_reg.predict_proba(X_new)
y_predict = softmax_reg.predict(X_new)

zz1 = y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris setosa")

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 7, 0, 3.5])
plt.show()
```



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

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

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

```
Out[ ]: array([[6.21626375e-07, 5.73689803e-02, 9.42630398e-01]])
```

General Questions

Task 14

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

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 14 answer:

No, Logistic Regression cost function is convex.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this

Task 15

Which Gradient Descent algorithm (among those we discussed) will reach
the vicinity of the optimal solution the fastest? Which will actually
converge? How can you make the others converge as well?

↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ your answer goes below

Double Click Here

Task 15 answer:

SGD will reach the vicinity of the optimal solution the fastest. SGD updates the model parameters after computing the gradient on a single training example, allowing for frequent updates.

Batch GD will actually converge to the exact optimal solution for convex functions.

To make SGD or MBGD converge more effectively to the optimal solution, we can gradually reduce the learning rate over time.

↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ your answer goes above this