Example: predicting blood glucose level

```
import pandas as pd
diabetes_df = pd.read_csv("diabetes.sc")
df.head()

X = diabetes_df.drop("glucose", axis=1).values
Y = diabetes_df["glucose"].values
```

Drop target and store the values as X, we want to predict X

```
X_bmi = X[: 3]
print(y.shape, X_bmi.shape)
```

We can then reshape to be two-dimensional, compatible with sci-kit learn

```
X_bmi = X_bmi.reshape(-1,1)
print(X_bmi.shape)

import matplotlib.pyplot as plt
plt.scatter(X_bmi, y)
plt.show()
```

We see as BMI increases, glucose increases

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_bmi, y)
predictions = reg.predict(X_bmi)
plt.scatter(X_bmi, y)
plt.plot(X_bmi, predictions)
plt.show()
```

Now we see a line of best fit across the scatter plot

Linear Regression Mechanics

y = ax + b
simple linear regression with one feature, one weight, and one bias

Define an error function and reduce its residuals.
We can calculate the Residual Sum of Squares (RSS).
Ordinary Least Squares (OLS) regression minimises RSS.

The default metric is R^2, which is the amount of variance in target values (0, 1)

```
reg.score(X_test, y_test)
```

Or we can use the mean squared error and root mean squared error.

$$MSE = \frac{1}{n}\Sigma(y - y_2)^2$$

$$RMSE = \sqrt{MSE}$$

```
from sklearn.metrics import mean_squared_error
rmse = mean_squared_error(y_test, y_pred, squared=False)
```

This returns RMSE.

Cross Validation

Model performance depends on how we split data.
We can use cross validation to eliminate random splitting.

We can separate data in 5 folds. For each split, pick one fold to test, and the rest to train. This should in five unique splits. This is **5-fold CV** or generally k-fold CV.

More folds = more computationally expensive

```
from sklearn.model_selection import cross_val_score, KFold
kf = KFold(n_splits=6, shuffle=True, random_state=42) # n_split default is 5
reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=kf)
print (cv_results) # returns 6 R^2 values
print(np.mean(cv_results), np.std(cv_results))
print(np.quantile(cv_results, [0.025, 0.975]))
```

Regularized regression

Reduces overfitting
Regularization penalizes large coefficients by a new loss function

Ridge regression

Loss function = OLS function + alpha * sum of a^2

This penalizes large coefficient (a) values

Alpha, a hyperparameter, needs to be chosen.

```
from sklearn.linear_model import Ridge
scores = []
for alpha in [0.1, 1, 10, 100, 1000]:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train)
    y_pred = ridge.predict(X_test)
    scores.append(ridge.score(X_test, y_test))
print(scores) # performance gets worse as alpha increases
```

Lasso regression

Loss function = OLS + alpha * sum of abs.alpha

Lasso can select important features, because it shrinks less important features to zero.

```
from sklearn.linear_model import Lasso
X = diabetes_df.drop("glucose", axis=1).values
y = diabetes_df["glucose"].values
names = diabetes_df.drop("glucose", axis=1).columns
lasso = Lasso(alpha=0.1)
lasso_coef = lasso.fit(X, y).coef_
plt.bar(names, lasso_coef)
plt.show()
```