STAT 243 Problem Set 8

Treves Li

2024-12-04

Collaboration Statement

I did not collaborate with anyone.

Question 1

Consider probit regression, which is an alternative to logistic regression for binary outcomes. The probit model is $Y_i \sim \text{Ber}(p_i)$ for $p_i = P(Y_i = 1) = \Phi(X_i^\top \beta)$ where Φ is the standard normal CDF, and Ber is the Bernoulli distribution. We can rewrite this model with latent variables, one latent variable, z_i, for each observation:

$$\begin{aligned} y_i =& \quad I(z_i > 0) \\ z_i \sim& \quad \mathcal{N}(X_i^\top \beta, 1) \end{aligned}$$

Question 1a

Design an EM algorithm to estimate β, taking the complete data to be Y, Z. You'll need to make use of the mean and variance of truncated normal distributions (see hint below). Be careful that you carefully distinguish β from the current value at iteration t, β^t, in writing out the expected log-likelihood and computing the expectation and that your maximization be with respect to β (not β^t). Also be careful that your calculations respect the fact that for each z_i you know that it is either bigger or smaller than 0 based on its y_i. You should be able to analytically maximize the expected log likelihood. A couple hints:

i. From the Johnson and Kotz 'bibles' on distributions, the mean and variance of the truncated normal distribution, $f(w) \propto \mathcal{N}(w; \mu, \sigma^2) I(w > \tau)$, are:

$$\begin{aligned}
\mathbb{E}(W \mid W > \tau) &= \mu + \sigma\rho(\tau^*) \\
\mathrm{Var}(W \mid W > \tau) &= \sigma^2 \left(1 + \tau^*\rho(\tau^*) - \rho(\tau^*)^2\right) \\
\rho(\tau^*) &= \frac{\phi(\tau^*)}{1 - \Phi(\tau^*)} \\
\tau^* &= \frac{\tau - \mu}{\sigma}
\end{aligned}$$

where $\phi(\cdot)$ is the standard normal density and $\Phi(\cdot)$ is the standard normal CDF. Or see the Wikipedia page on the truncated normal distribution for more general formulae.

ii. You should recognize that your expected log-likelihood can be expressed as a regression of some new quantities (which you might denote as $m_i, i = 1, ..., n$, where the m_i are functions of β^t and y_i) on X.

We can break the algorithm down (from the course notes) into:

1. The E step
2. The M step
3. Continue until convergence

Some of the ideas below were assisted with the help of ChatGPT.

The E Step

In this step, we are interested in computing the expected value of the log-likelihood, conditional on Y, X and the current estimate β^t. Or in other words, we want to replace missing Z with its conditional expectation.

First we note that

$$\begin{aligned}
\rho(\tau_i^*) &= \frac{\phi(\tau_i^*)}{1 - \Phi(\tau_i^*)} \\
\tau_i^* &= \frac{\tau - X_i^\top \beta^t}{\sigma} = \tau - X_i^\top \beta^t
\end{aligned}$$

We can calculate the **mean** with the provided Johnson and Kotz equation, conditional on the above parameters:

$$\begin{aligned}\mathbb{E}(z_i \mid y_i, X_i, \beta^t) &= \mu + \sigma\rho(\tau^*)\\ &= X_I^\top\beta + (1)\rho(\tau^*)\end{aligned}$$

where

$$\rho(\tau^*) = \begin{cases} \frac{\phi(-X_i^\top\beta^t)}{\Phi(-X_i^\top\beta^t)} & \text{if } y_i = 0\\ \frac{\phi(X_i^\top\beta^t)}{1-\Phi(X_i^\top\beta^t)} & \text{if } y_i = 1\end{cases}$$

We can similarly calculate the **variance** based on the provided Johnson and Kotz equation:

$$\begin{aligned}\mathrm{Var}(z_i \mid y_i, X_i, \beta^t) &= \sigma^2 \quad (1 + \tau^*\rho(\tau^*) - \rho(\tau^*)^2)\\ &= (1)^2 \quad (1 + \tau_i^*\rho(\tau_i^*) - \rho(\tau_i^*)^2)\end{aligned}$$

The variance accounts for the uncertainty in z_i, influencing the weights w_i later in the M step. It ensures that observations with higher uncertainty are weighted less in the regression.

We can then compute $Q(\beta \mid \beta^t) = E(\log L(\beta \mid Y) \mid x; \beta^t)$, which is the expected complete-data log-likelihood with respect to Z, and conditional on Y and β^t:

$$\begin{aligned}Q(\beta \mid \beta^t) &= \mathbb{E}(\log L(\beta \mid Y) \mid x; \beta^t)\\ &= \mathbb{E}(\log L(\beta \mid Y, Z))\\ &= \mathbb{E}\left(\sum_{i=1}^n -\frac{1}{2}(z_i - X_i^\top\beta)^2 + \text{constant terms}\right)\\ &= \sum_{i=1}^n -\frac{1}{2}\mathbb{E}\left[(z_i - X_i^\top\beta)^2\right]\\ &= \sum_{i=1}^n -\frac{1}{2}\left[\mathbb{E}[z_i^2] - 2X_i^\top\beta\mathbb{E}[z_i] + (X_i^\top\beta)^2\right]\end{aligned}$$

From here, we can calculate $\mathbb{E}[z_i]$ and $\mathbb{E}[z_i^2] = \mathrm{Var}(z_i) + (\mathbb{E}[z_i]^2)$, and substitute those values into $Q(\beta \mid \beta^t)$.

The M Step

In this step, we are trying to maximise $Q(\beta \mid \beta^t)$ with respect to β, by finding β^{t+1}. We do this by updating Z^* to solve a weighted least squares problem for β.

From the E step above, we can simplify the function $Q(\beta \mid \beta^t)$, such that:

$$Q(\beta \mid \beta^t) = -\frac{1}{2}\sum_{i=1}^{n} w_i (z_i^* - X_i^\top \beta)^2$$

This is a standard weighted least squares form, where:

- $z_i^* = \mathbb{E}(z_i \mid y_i, X_i, \beta^t)$ (which was computed in the E step)
- $w_i = 1/\mathrm{Var}(z_i \mid y_i, X_i, \beta^t)$

The solution to minimising $Q(\beta \mid \beta^t)$ is then:

$$\beta^{t+1} = (X^\top W X)^{-1} X^\top W Z^*$$

where

- $Z^* = (\mathbb{E}[z_1], ..., \mathbb{E}[z_n])^\top$ which is the vector of conditionally expected values of z_i, computed in the E step, and based on the current estimate of β^t
- W is a diagonal matrix with entries $w_i = 1/\mathrm{Var}(z_i)$.

At each iteration, W is recomputed based on the updated $\mathrm{Var}(z_i \mid y_i, X_i, \beta^t)$ from the E step. This makes the E and M steps iterative, with $Q(\beta \mid \beta^t)$ improving at every step.

Convergence

We can repeat the E and M steps until β^t converges to a stable value, or alternatively:

$$|\beta^{t+1} - \beta^t| < \epsilon$$

for some tolerance ϵ.

Question 1b

Propose how to get reasonable starting values for β.

Based on the course notes, below are some ideas on getting reasonable starting values.

Random initialisation

Randomly sample values for β from a normal distribution with mean 0 and a small variance σ^2 near 0, which would reduce the risk of divergence, such that:

$$\beta_j^{(0)} \sim \mathcal{N}(0, \sigma^2)$$

Multiple random starts

Generate multiple random starting points for β, and run the algorithm for each of those points. We could then select the β that gives the highest likelihood or best objective function value. According to the course notes, this is good for multimodal problems.

Scaling data before initialisation

We can centre and scale X and Y to mean 0 and variance 1 before computing initial values, which would reduce the sensitivity of the algorithm to bad starting values.

Question 1c

> Write a Python function to estimate the parameters. Make use of the initialization from part (b). You may use existing regression functions for the update steps. You'll need to include criteria for deciding when to stop the optimization.

Although the question says to make use of the initialisation from part (b), I implemented the random initialisation later in Question 2b (which uses the function defined in this part), since it made the coding more practical. In my function, I take `beta_init` as an input, which assumes randomly initialised values from later parts of this problem set.

I computed the conditional expectation z and $\mathrm{Var}(z)$ using the current estimate of β. The formulas for these were derived from the Johnson and Kotz equations, with $\rho(\tau)$ being computed based on whether $y_i = 0$ or $y_i = 1$.

I then updated β using weighted least squares, using the conditional expectation z as the target, and weights derived from the inverse of $\mathrm{Var}(z)$. The update in the M-step was performed using `scikit-learn`'s `LinearRegression` class.

The algorithm was designed to stop if the norm difference between successive β values was less than `tol=1e-6`, or after 100 iterations.

```python
import numpy as np
from sklearn.linear_model import LinearRegression, LogisticRegression
from scipy.stats import norm

def em_algorithm(X, y, beta_init, max_iter=100, tol=1e-6):
  beta = beta_init # Initialise beta based on function inputs
  n = X.shape[0]   # Number of samples

  for num_iter in range(max_iter):
    # E Step
    z_exp = np.zeros(n)
    var_z = np.zeros(n)

    for i in range(n):
      # Calculate tau based on current beta
      tau = (y[i] - 0.5) - X[i, :].dot(beta)

      # Compute conditional EXPECTATION of z, given y, X, beta
      if y[i] == 1:
        rho = norm.pdf(tau) / (1 - norm.cdf(tau))
      else:
        rho = -norm.pdf(tau) / norm.cdf(tau)

      z_exp[i] = X[i, :].dot(beta) + rho

      # Compute conditional VARIANCE of z, given y, X, beta
      var_z[i] = 1 + tau * rho - rho**2

    # M Step
    W = np.diag(1 / var_z)
    Z = z_exp  # Z is conditional expectations of z

    # Update beta using weighted least squares
    # We don't care about intercepts
    regressor = LinearRegression(fit_intercept=False)
    regressor.fit(X, Z, sample_weight=1/var_z)

    # Update beta
    beta_new = regressor.coef_

    if np.linalg.norm(beta_new - beta) < tol:
      return beta_new, num_iter + 1

    # Update beta (final)
    beta = beta_new

  return beta, max_iter
```

Question 1d

Try out your function using data simulated from the model. Take $n = 100$ and the parameters such that $\hat{\beta}_1/\text{se}(\hat{\beta}_1) \approx 2$ and $\beta_2 = \beta_3 = 0$. In other words, I want you to choose β_1 such that the signal to noise ratio in the relationship between x_1 and y is moderately large. You can do this via trial and error simply by simulating data for a given β_1 and fitting a logistic regression to get the estimate and standard error. Then adjust β_1 as needed.

In the code below, I assumed there were going to be three predictors, since three coefficients were mentioned (β_1, β_2, β_3). The `target_ratio` variable defines $\hat{\beta}_1/\text{se}(\hat{\beta}_1) \approx 2$. The number of max iterations was set at 50.

An initial β_1 value of 0.5 was selected; if there was no convergence based on the resulting standard error, then the value was incremented by 5%.

At the end of the code block, I run the EM algorithm with the simulated data to get the β values.

```python
# Simulate data from given info
n = 100
np.random.seed(42)
X = np.random.randn(n, 3)  # Assume 3 predictors
beta_2 = beta_3 = 0

# Settings for beta_1
beta_1 = 0.5  # Initial guess
tol = 0.1
target_ratio = 2
max_iter = 50

print(f"Initial beta_1:   {beta_1}")

for i in range(max_iter):
  # Generate response with fixed beta_2 and beta_3
  beta = np.array([beta_1, beta_2, beta_3])
  eps = np.random.normal(0, 1, n)  # Noise/error term
  y = X @ beta + eps

  # Convert to binary response for logistic regression
  y_binary = (y > 0).astype(int)
```

```
    # Fit logistic regression model using only X[:, 0]
    model = LogisticRegression(fit_intercept=False, solver='liblinear')
    model.fit(X[:, :1], y_binary)
    beta_est_1 = model.coef_[0][0]

    # Bootstrap for SE estimation
    bootstrap_ests = []
    for _ in range(100):
      indices = np.random.choice(n, n, replace=True)
      X_sample = X[indices, :1]  # Use only X[:, 0] to lock beta_2, beta_3 at 0
      y_sample = y_binary[indices]
      model.fit(X_sample, y_sample)
      bootstrap_ests.append(model.coef_[0][0])
    se_beta_1 = np.std(bootstrap_ests)

    # Calculate signal-to-noise ratio
    sn_ratio = beta_est_1 / se_beta_1

    if abs(sn_ratio - target_ratio) < tol:
      print(
        f"Converged beta_1: {beta_1:.2f} "
        f"(signal-to-noise ratio = {sn_ratio:.2f})"
      )
      break

    # Adjust beta_1 for better convergence
    beta_1 += (target_ratio - sn_ratio) * 0.05

# Run EM algorithm with the simulated data
beta_est, _ = em_algorithm(X, y, beta_init=np.array([beta_1, beta_2, beta_3]))
print(f"Estimated beta: {beta_est}")
```

```
Initial beta_1:   0.5
Converged beta_1: 0.44 (signal-to-noise ratio = 1.93)
Estimated beta: [ 0.24761583  0.38264982 -0.08607219]
```

We see that we end up selecting β_1 such that the signal-to-noise ratio is 1.93, which is close to the target of 2. The non-zero estimates for β_2 and β_3 may arise from slight correlations between the predictors and random noise introduced by ϵ, despite their intended values being zero in the data generation process.

Question 2

A different approach to this problem just directly maximizes the log-likelihood of the observed data under the original probit model (i.e., without the zs).

Question 2a

Write an objective function that calculates the negative log-likelihood of the observed data using JAX or PyTorch syntax (for use in part (d)).

To do this task, I assumed that the model would follow probit regression, such that it involves the CDF of the stanard normal distribution. The negative log-likelihood is thus given by the sum below. Note that since I have no stats background, I had to consult ChatGPT for the equation.

$$\mathrm{NLL}(\beta) = -\sum_{i=1}^{n} \left(y_i \log\left(\Phi(x_i\beta)\right) + (1 - y_i)\log\left(1 - \Phi(x_i\beta)\right)\right)$$

where:

- Φ is the CDF of the standard normal distribution
- x_i are the predictor values
- β consists of the regression coefficients

After discussing with Chris, we realised that using JAX's default 32-bit floating point numbers would cause problems for the BFGS optimisation later in part (b) of the homework. Because of this, I had to explicitly set JAX to use 64-bit floating points via `jax.config.update`. I also modified the `JAX_PLATFORM_NAME` environment variable to suppress warnings about GPU compatibility.

```python
import os
import jax
import jax.numpy as jnp
from jax.scipy.stats import norm

jax.config.update("jax_enable_x64", True)
jax.config.update('jax_platform_name', 'cpu')

def nll(beta, X, y):
```

```
# Compute the probit link function (i.e. the cumulative normal dist)
probit = norm.cdf(jnp.dot(X, beta))

# Compute log-likelihood for the probit model
log_likelihood = y * jnp.log(probit) + (1 - y) * jnp.log(1 - probit)

# NLL is the negative sum of log-likelihoods
return -jnp.sum(log_likelihood)
```

Question 2b

Estimate the parameters for your test cases using `scipy.optimize.minimize()` with the BFGS option. Compare how many iterations EM and BFGS take. Note that this provide a nice test of your EM derivation and code, since you should get the same results from the two optimization approaches. Calculate the estimated standard errors based on the inverse of the Hessian. Note that the `hess_inv` returned by `minimize` is probably NOT a good estimate of the Hessian as it seems to just be the approximation built up during the course of the BFGS iterations and not a good numerical derivative estimate at the optimum. Try using `numdifftools` and compare to what is seen in `hess_inv`. If you get warnings about loss of precision, you may need to tell JAX or PyTorch to use 64-bit rather than 32-bit floating point numbers.

Below, I defined a function `optimise_bfgs` that takes the `nll` objective function defined in part (a) and applies Scipy's `minimize` function. I nested the `nll` function inside the wrapper function `objective` for convenience. This time, the guesses for β are initialised with a random guess (to comply with the requirements from Question 1 part (c)), as well as scaling X and β to reduce the sensitivity of the algorithm to bad/excessively big starting values.

Since we were advised in the problem formulation that the `hess_inv` returned by Scipy is not a good estimate of the Hessian, the standard errors calculated from `numdifftools` are also outputted below.

```python
import scipy.optimize as opt
import numdifftools as ndt

np.random.seed(42)

# BFGS optimisation using Scipy
def optimise_bfgs(X, y, beta_init):
  # Define objective function (negative log-likelihood)
  def objective(beta):
    return nll(beta, X, y)

  # Optimise
  result = opt.minimize(objective, beta_init,
    method='BFGS', options={'gtol': 1e-6, 'maxiter': 100}
    )
  return result
```

```python
# Function to calculate standard errors from the Hessian
def est_std_err(hess_inv):
  return np.sqrt(np.diag(hess_inv))

# Initialise values/guesses for beta
n = 100
X = np.random.randn(n, 3)
y = np.random.randint(0, 2, n)
beta_init = np.random.randn(X.shape[1]) * 0.1
X = (X - X.mean(axis=0)) / X.std(axis=0)

# Run EM algorithm
beta_em, em_iter_count = em_algorithm(X, y, beta_init=beta_init)

# Run BFGS optimization; get results
result_bfgs = optimise_bfgs(X, y, beta_init)
beta_bfgs = result_bfgs.x
hess_inv_bfgs = result_bfgs.hess_inv

# Compute BFGS standard errors from the Hessian
scipy_stderr = est_std_err(hess_inv_bfgs)

# Compute numdifftools standard errors for comparison
ndt_hessian = ndt.Hessian(lambda beta: nll(beta, X, y))
ndt_stderr = est_std_err(np.linalg.inv(ndt_hessian(beta_bfgs)))

# Print results
print(f"EM estimated betas:    {beta_em} ({em_iter_count} iterations)")
print(f"BFGS estimated betas:  {beta_bfgs} ({result_bfgs.nit} iterations)")
print(f"EM vs BFGS diff:       {beta_em - beta_bfgs}\n")

# Print standard errors
print(f"Scipy std errs (BFGS): {scipy_stderr}")
print(f"numdifftools std errs: {ndt_stderr}\n")
```

```
An NVIDIA GPU may be present on this machine, but a CUDA-enabled jaxlib is not installed. Fa

EM estimated betas:    [-0.05521041 -0.0419608   0.12660694] (15 iterations)
BFGS estimated betas:  [-0.0849735  -0.06446407  0.19541074] (6 iterations)
EM vs BFGS diff:       [ 0.02976309  0.02250327 -0.06880381]
```

```
Scipy std errs (BFGS): [0.12840681 0.12878378 0.13086401]
numdifftools std errs: [0.12839881 0.12879622 0.13079944]
```

The results above show that the estimated betas between the EM and BFGS are nearly equal, which validates the EM derivation from Question 1. Nevertheless, the difference between the betas from the two methods range in order of magnitude from 10^{-2}, which could be explained by their different approximation strategies. Specifically, EM relies on iterative steps that may be sensitive to initialisation values, while BFGS uses gradient-based techniques that may differ slightly due to numerical precision.

It seems that the BFGS method was able to get to the optimum in fewer iterations than the EM method (by more than half), which could be explained by its use of the gradient to more efficiently navigate the parameter space, whereas the EM method relies on slower iterative updates that may converge to local rather than a global optima.

It's also worth noting that the estimated standard errors from the `hess_inv` returned by Scipy's `minimize` resulted in more or less the same values as those returned from `numdifftools`'s numerical differentiation method, despite what is mentioned in the problem description. This could be because the Scipy developers may have updated the method recently to be more numerically robust, or alternatively that the problem has a well-conditioned Hessian near the optimum.

No precision warnings were encountered, likely due to scaling of X and β, and using 64-bit floating points from Question 2 part (a).

Question 2c

> As part of this, try a variety of starting values and see if you can find ones that cause the optimization **not** to converge using BFGS. Also try them with Nelder-Mead.

By simply changing the scaling factor, I can get different β values that do not converge. I also defined a Nelder-Mead optimise function, and then the convergence is compared at the end with BFGS.

```
# Nelder-Mead optimiser
def optimise_nm(X, y, beta_init):
  def objective(beta):
      return nll(beta, X, y)

  result = opt.minimize(objective, beta_init,
    method='Nelder-Mead', options={'maxiter': 100, 'xatol': 1e-6}
    )
  return result

# Initialise data based on Q2, part (c)
np.random.seed(52)
n = 100
X = np.random.randn(n, 3)
X = (X - X.mean(axis=0)) / X.std(axis=0)
y = np.random.randint(0, 2, n)

# Get a range of scaling factors
scale_list = [0.1, 0.2, 1, 1.1, 1.2]
starting_values = [np.random.randn(X.shape[1]) * scale for scale in scale_list]
results = []

for beta_init in starting_values:
  # BFGS
  result_bfgs = optimise_bfgs(X, y, beta_init)
  beta_bfgs = result_bfgs.x
  bfgs_converged = result_bfgs.success

  # Nelder-Mead
  result_nm = optimise_nm(X, y, beta_init)
  beta_nm = result_nm.x
  nm_converged = result_nm.success

  # Store results
  results.append({
    'beta_init': beta_init,
    'beta_bfgs': beta_bfgs,
    'bfgs_converged': bfgs_converged,
    'beta_nm': beta_nm,
    'nm_converged': nm_converged
  })

# Analyze results
for res in results:
  print(f"Starting betas: {res['beta_init']}")
  print(f"  BFGS Converged?:          {res['bfgs_converged']}")
  print(f"  Nelder-Mead Converged?:   {res['nm_converged']}\n")
```

```
Starting betas: [0.10127787 0.04615036 0.01965955]
  BFGS Converged?:          True
  Nelder-Mead Converged?:   False

Starting betas: [ 0.03671261 -0.05328842  0.01673415]
  BFGS Converged?:          True
  Nelder-Mead Converged?:   True

Starting betas: [ 2.01712544  1.41154122 -0.25794999]
  BFGS Converged?:          False
  Nelder-Mead Converged?:   False

Starting betas: [-0.698568   -1.46341842  1.23318375]
  BFGS Converged?:          False
  Nelder-Mead Converged?:   True

Starting betas: [-2.63794375  0.45967707 -0.61528112]
  BFGS Converged?:          True
  Nelder-Mead Converged?:   False
```

The results show that BFGS and Nelder-Mead are both sensitive to starting values, with different initializations leading to varied convergence results. For some initialisations, BFGS converges successfully while Nelder-Mead fails (and vice versa). What seems to be common to both methods is that larger starting values lead to failure (maybe because it starts the algorithm in parts of the parameter space with steep gradients or where it's ill-conditioned?).

Question 2d

Now use JAX or PyTorch automatic differentiation (AD) functionality to create a gradient function. Set up your objective and gradient functions to use just-in-time compilation (you can use `jit()` or `@jit` for JAX and `torch.compile` or `@torch.compile` for PyTorch). Use these functions to find the parameters using BFGS. Check that you get the same results as in (b) and compare the number of iterations and timing to using BFGS without providing the gradient function (and thereby relying on scipy using numerical differentiation). Finally, use JAX or PyTorch functionality to create a Hessian function and use it to calculate the Hessian at the optimum. Compare the inverse of the Hessian and the estimated standard errors to what you got in part (b).

In the code below, I use JAX's own `scipy.optimize` function, and use it together with a gradient function I define as `grad_nll` and a Hessian function `hess_nll`, both using just-in-time compilation. The objective function was already defined in part (a) to use JAX.

The initialisation is the same as in part (b) to allow a fair comparison.

```
import jax.scipy.optimize as jopt
from jax import jit, grad, hessian
import time

# Define gradient function with jit
grad_nll = jit(grad(nll))

# Define NLL Hessian with JAX/jit
hess_nll = jit(hessian(nll))

# BFGS optimization function using JAX, with JAX gradient function
@jit
def optimise_bfgs_jax(X, y, beta_init):
  start_time = time.time()
  result = jopt.minimize(
    fun=lambda beta: nll(beta, X, y),
    x0=beta_init,
    method='BFGS',
    options={'gtol': 1e-6, 'maxiter': 100}
  )
  end_time = time.time()
```

```
    return result, end_time - start_time

# Initialise values/guesses for beta, same as part (b)
np.random.seed(42)
n = 500
X = np.random.randn(n, 3)
y = np.random.randint(0, 2, n)
beta_init = np.random.randn(X.shape[1]) * 0.1
X = (X - X.mean(axis=0)) / X.std(axis=0)

# Run "normal" scipy optimisation (with timing)
start_time = time.time()
result_bfgs = optimise_bfgs(X, y, beta_init)
scipy_time = time.time() - start_time
beta_bfgs = result_bfgs.x
hess_inv_bfgs = result_bfgs.hess_inv

# Run BFGS optimisation using JAX
result_bfgs_jax, jax_time = optimise_bfgs_jax(X, y, beta_init)
beta_bfgs_jax = result_bfgs_jax.x
hess_inv_bfgs_jax = np.linalg.inv(hess_nll(beta_bfgs_jax, X, y))

# Compute and compare scipy vs JAX standard errors
scipy_stderr = est_std_err(hess_inv_bfgs)
jax_stderr = est_std_err(hess_inv_bfgs_jax)

# Print results
print(f"BFGS (Scipy) estimated betas: {beta_bfgs}")
print(f"BFGS (Scipy) iterations:      {result_bfgs.nit}")
print(f"Time taken: {scipy_time:.4f} seconds\n")

print(f"BFGS (JAX) estimated betas:   {beta_bfgs_jax}")
print(f"BFGS (JAX) iterations:        {result_bfgs_jax.nit}")
print(f"Time taken: {jax_time:.4f} seconds\n")

# Print standard errors
print(f"JAX std errs:   {jax_stderr}")
print(f"Scipy std errs: {scipy_stderr}")
```

```
BFGS (Scipy) estimated betas: [-0.11045854 -0.05704121  0.10345738]
BFGS (Scipy) iterations:      6
Time taken: 0.3525 seconds
```

```
BFGS (JAX) estimated betas:   [-0.11045853 -0.0570412   0.10345739]
BFGS (JAX) iterations:        6
Time taken: 0.2381 seconds

JAX std errs:   [0.0564515  0.05657476 0.05655903]
Scipy std errs: [0.05644834 0.05657087 0.05655296]
```

The results confirm that the JAX AD calculations are getting the same result as the "normal" scipy results without the gradient function in part (b). There are some minor differences in precision, but this can be attributed to how JAX and Scipy each implement their floating-point operations.

The number of iterations is the same for both methods (since they're using the same BFGS algorithm), but the JAX AD is able to converge much faster due to the more efficient jitted compilation. Furthermore, JAX takes advantage of the gradient function, thus avoiding the overhead that would affect "standard" numerical differentiation.

The results from the JAX Hessian function agree largely with the values from part (b), confirming the accuracy of JAX AD method and showing that AD can be a good (and quicker!) alternative to standard numerical approaches in optimisation.