Overview

Submit your writeup including any code and plots as a PDF via Gradescope.[1] We recommend reading through the entire homework beforehand and carefully using functions for testing procedures, plotting, and running experiments. Taking the time to reuse code will help in the long run!

Data science is a collaborative activity. While you may talk with others about the homework, please write up your solutions individually. If you discuss the homework with your peers, please include their names on your submission. Please make sure any handwritten answers are legible, as we may deduct points otherwise.

1. GLM for Dilution Assay

Being able to reformulate problems as generalized linear models (GLMs) enables you solve a wide variety of problems with existing packages. We recommend reviewing the examples of GLMs from Lectures 10 and 11. In particular, make sure you understand that formulating a GLM involves choosing an 1) output distribution and 2) link function that are appropriate for the application at hand.

In this problem, you'll retrace the footsteps of the statistician R. A. Fisher and develop one of the very first applications of GLMs. In a 1922 paper, Fisher formulated a GLM he used to estimate the unknown concentration ρ_0 of an infectious microbe in a solution. Without specialized technology to directly measure ρ_0 from the solution, Fisher devised the following procedure: we will progressively dilute the original solution, and after each dilution, we'll pour out some small volume v onto a sterile plate. If zero microbes land on the plate, it will remain sterile, but if any microbes land on a plate, they will grow visibly on it (we call this an "infected plate"). By observing whether or not the plate is infected at each dilution, and by formulating the relationship between this data and ρ_0 as a GLM, we can estimate ρ_0 from this data.

Specifically, let ρ_t denote the concentration at dilution t. Each time, we dilute the solution to be half its concentration, such that

$$\rho_t = \frac{\rho_0}{2^t} \tag{1}$$

for $t = 0, 1, \ldots$. When we pour out volume v of the solution onto the plate, and wait awhile to allow for microbe growth, we can observe whether a plate was infected (*i.e.*, has a non-zero number of microbes) or is sterile (*i.e.*, has zero microbes). Therefore, our data $Y_t \in \{0, 1\}$ is whether or not the plate is infected at each dilution.

We'll formulate a GLM that relates ρ_0 and t to the data Y_t. Estimating the parameters of this GLM will then allow us to estimate ρ_0, as will become clear in the last part.

[1] In Jupyter, you can download as PDF or print to save as PDF

(a) (2 points) At dilution t, the data $Y_t \in \{0, 1\}$ indicates whether or not the plate is infected. The chance that a plate gets infected is denoted by $\mu(t) := \mathbb{E}[Y_t]$. Write down an output distribution for Y_t that is appropriate for the values it takes on, using $\mu(t)$ as a parameter. (We'll derive what $\mu(t)$ should be in the next part).

(b) (3 points) At dilution t, we pour out volume v onto a plate, so the expected number of microbes on the plate is $\rho_t v$. The actual number of microbes is distributed as a Poisson random variable with this mean $\rho_t v$:

$$\#\text{ microbes on plate at dilution } t \sim \text{Poisson}(\rho_t v). \tag{2}$$

Using this fact, write out an expression for $\mu(t) := \mathbb{E}[Y_t]$. Start with

$$\mu(t) = \mathbb{P}(\text{plate is infected at dilution } t) \tag{3}$$
$$= 1 - \mathbb{P}(\text{there are 0 microbes on plate at dilution } t). \tag{4}$$

(c) (3 points) Use your findings from part (b), along with Equation (1), to find a link function g such that

$$g(\mu(t)) = \beta_0 + \beta_1 t \tag{5}$$

for some constants β_0 and β_1. (Remember that in class, we talked about the inverse link function g^{-1}, such that $\mu(t) = g^{-1}(\beta_0 + \beta_1 t)$).

(d) (2 points) Choosing an appropriate output distribution and link function as we've done in Parts (a) and (c) completes the GLM specification. Now, suppose you've estimated β_0 and β_1 (*e.g.*, using maximum-likelihood estimation). Write down an estimate of ρ_0.

Hint: For this question, you do not need to estimate β_0 and β_1: assume you know them, and find a way to estimate ρ_0 from them.

2. Image Denoising with Gibbs Sampling

In this problem, we derive a Gibbs sampling algorithm to restore a corrupted image [1]. A grayscale image can be represented by a 2-dimensional array X of shape $n \times m$, where the intensity of the (i, j)-th pixel is X_{ij}. In this problem, we are given an image X whose pixels have been corrupted by noise, and the goal is to recover the original image Z.

(a) (2 points) Load the grayscale image `X.pkl` as a numpy array X. Visualize the image.

From plotting the image X, it is clear that it has been corrupted with noise. Let Z denote the original image, which we also represent as an $n \times m$ array. Let $\mathcal{I} = \{(i, j) : 1 \le i \le n \text{ and } 1 \le j \le m\}$ denote the collection of all pixels in the image, represented by the corresponding index of the array. Given a pixel (i, j), define the set of *neighboring pixels* to be

$$N_{(i,j)} = \{(i', j') \in \mathcal{I} : (i = i' \text{ and } |j - j'| = 1) \textbf{ or } (|i - i'| = 1 \text{ and } j = j')\}.$$

To capture the fact that, in natural images, neighboring pixels are likely be similar, we consider the following prior over the original image:

$$p(Z) \propto \exp\left(-\frac{1}{2}\sum_{(i,j)\in\mathcal{I}}\left[aZ_{ij}^2 - b\sum_{(i',j')\in N_{(i,j)}} Z_{ij}Z_{i'j'}\right]\right).$$

Assuming the image has been corrupted with Gaussian noise $X_{(i,j)} \mid Z_{(i,j)} \sim \mathcal{N}(Z_{(i,j)}, \tau^{-1})$ (independently across pixels $(i,j) \in \mathcal{I}$), the complete posterior can be written as

$$p(X \mid Z) \propto \exp\left(-\frac{1}{2}\sum_{(i,j)}\left[(a+\tau)Z_{ij}^2 - 2\tau Z_{ij}X_{ij} - b\sum_{(i',j')\in N_{(i,j)}} Z_{ij}Z_{i'j'}\right]\right) \quad (6)$$

Let $S_{ij} = \sum_{(i',j')\in N_{(i,j)}} Z_{i'j'}$. By completing the square in the posterior (6), we have

$$Z_{ij} \mid (Z_{i'j'})_{(i',j')\neq(i,j)}, X \sim \mathcal{N}\left(\frac{\tau X_{ij} + bS_{ij}}{a+\tau}, \frac{1}{a+\tau}\right) \quad (7)$$

(b) (2 points) Fill in the missing line of pseudocode for a Gibbs sampler of the posterior, $p(Z|X)$. **Be specific with each conditioned variable and sub/superscript!**

- Initialize $Z^{(0)} = X$.
- For $t = 1, \ldots, T$:
  - Sample $Z_{1,1}^{(t)} \sim p(Z_{1,1} \mid Z_{1,2} = Z_{1,2}^{(t-1)}, Z_{1,3} = Z_{1,3}^{(t-1)}, \ldots, Z_{n,m} = Z_{n,m}^{(t-1)}, X)$.
  - Sample $Z_{1,2}^{(t)} \sim p(Z_{1,2} \mid Z_{1,1} = Z_{1,1}^{(t)}, Z_{1,3} = Z_{1,3}^{(t-1)}, \ldots, Z_{n,m} = Z_{n,m}^{(t-1)}, X)$.
  - Sample $Z_{1,3}^{(t)} \sim$ *# TODO: fill this in.*
  - ...
  - Sample $Z_{n,m}^{(t)} \sim p(Z_{n,m} \mid Z_{1,1} = Z_{1,1}^{(t)}, Z_{1,2} = Z_{1,2}^{(t)}, \ldots, Z_{n,m-1} = Z_{n,m-1}^{(t)}, X)$

(c) (3 points) Write the pseudo-code from Part (b) more explicitly both by using a double for-loop over $(i,j) \in \mathcal{I}$ and by being explicit about the conditional distributions of the form $p(Z_{1,1} \mid Z_{1,2} = Z_{1,2}^{(t-1)}, Z_{1,3} = Z_{1,3}^{(t-1)}, \ldots, Z_{n,m} = Z_{n,m}^{(t-1)}, X)$. In your pseudo-code, use `np.random.randn()` to generate a $\mathcal{N}(0,1)$ random variable at each step.

(d) (5 points) Implement the Gibbs sampler from Part (c) with $a = 250, b = 62.5$, and $\tau = 0.01$. Run your code for $T = 1$ iteration, i.e. update each coordinate exactly once. Visualize the resulting image $Z^{(1)}$. Time your code and estimate how long it would take to compute $Z^{(100)}$.

(e) (2 points) The bottleneck in running the Gibbs sampler from Part (d) is sampling a single pixel Z_{ij} with the values of all others held fixed. Fortunately, it is possible to speed up the sampling process with an improvement known as *blocked Gibbs sampling*. Specifically, define two subsets of the pixels $\mathcal{I}_{\text{even}} = \{(i,j) : i+j \text{ is even}\}$ and $\mathcal{I}_{\text{odd}} = \{(i,j) : i+j \text{ is odd}\}$. The blocked Gibbs sampler proceeds as follows:

- Initialize $Z^{(0)} = X$.
- For $t = 1, \ldots, T$:
  - Let $Z = Z^{(t-1)}$.
  - Let Δ be an $n \times m$ matrix with $\mathcal{N}(0, \frac{1}{a+\tau})$ entries.
  - For $(i, j) \in \mathcal{I}_{\text{even}}$:
    - Let $S_{ij} = \sum_{(i',j') \in N_{(i,j)}} Z_{i'j'}$
  - Update $Z_{\mathcal{I}_{\text{even}}} = \frac{\tau}{a+\tau} X_{\mathcal{I}_{\text{even}}} + \frac{b}{a+\tau} S_{\mathcal{I}_{\text{even}}} + \Delta_{\mathcal{I}_{\text{even}}}$.
  - For $(i, j) \in \mathcal{I}_{\text{odd}}$:
    - Let $S_{ij} = \sum_{(i',j') \in N_{(i,j)}} Z_{i'j'}$
  - Update $Z_{\mathcal{I}_{\text{odd}}} = \frac{\tau}{a+\tau} X_{\mathcal{I}_{\text{odd}}} + \frac{b}{a+\tau} S_{\mathcal{I}_{\text{odd}}} + \Delta_{\mathcal{I}_{\text{odd}}}$.
  - Let $Z^{(t)} = Z$.

The advantage of this approach is that the inner for-loops can be *vectorized*. Explain why updating half the variables $Z_{\mathcal{I}_{\text{even}}}$ (and then $Z_{\mathcal{I}_{\text{odd}}}$) at once is justified.
Hint: if you're not sure why, try drawing out a small grid of pixels and label each one with $i + j$.

(f) (1 point) Implement the Gibbs sampler from Part (e) using $a = 250, b = 62.5$ and $\tau = 0.01$. Run your code for $T = 100$ iterations, and visualize the resulting image $Z^{(100)}$. Time your code and report how long it took.
Hint: Compute the entire $n \times m$ matrix S at once using matrix operations on Z. You may find it helpful to pad the matrix Z with a border of zeros using `Z_bar = np.pad(Z, 1)`. Then use slicing on the $(n+2) \times (m+2)$ matrix `Z_bar` to compute S.

3. Bayesian GLM

In this problem, we'll apply Gaussian linear regression to election data, and use PyMC3 to explore the effect of what prior we choose.
Suppose $x_1, \ldots, x_n \in \mathbb{R}^d$ are fixed feature vectors. Assume the linear model, where we observe

$$y_i = \beta^\top x_i + \varepsilon_i, \quad i = 1, \ldots, n,$$

where $\varepsilon_i \sim N(0, \sigma^2)$ are independent of each other, and $\beta \in \mathbb{R}^d$ and $\sigma^2 > 0$ are unknown. Let $y = (y_1, \ldots, y_n)$, $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)$, and let X denote the matrix whose i-th row is equal to x_i. Using this notation, we may more succinctly write the linear model as

$$y = X\beta + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2 I_n).$$

We model the regression weights as a random variable with the following prior distribution:

$$\beta \sim N(0, \sigma_0^2 I_d).$$

where $\sigma_0^2 > 0$ is hyperparameter we choose.

Using the file `us_elections.csv`, we'll try to predict the outcome of the 2020 election using information from previous elections[2]. Specifically, for each Congressional district, we'll try to predict how much Democrats won by (`house_dem20_margin` column) using the current officeholder's ideology score (`govtrack_ideology` column) and the result from the 2018 election (`house_dem18_margin` column), with **no intercept**. Because we're predicting using only two variables, we can easily visualize each value for the 2-dimensional β. When making visualizations below, your x-axis should have values for the coefficient that corresponds to `govtrack_ideology`, and the y-axis should have values for the coefficient that corresponds to `house_dem18_margin`.

(a) (4 points) Use the documentation of PyMC3 to figure out how to choose a prior for a GLM, and then obtain 1000 samples from the posterior distribution for β, given the model and data as described above. Make three scatter plots showing the posterior samples for different values of $\sigma_0^2 = \{1, 0.01, 10^{-4}\}$. All three scatter plots should be plotted with the same axis range (for example, if one scatter plot has an x-axis that goes from -0.3 to 0.1, then all three of them should too).

Some helpful hints:

- To set up a GLM with no intercept in PyMC3, you must specify the formula using something like `y ~ 0 + x1 + x2`.
- For a GLM specified as above, you can get posterior samples as a dataframe using `trace.posterior[['x1', 'x2']].to_dataframe()`
- If your PyMC3 code seems to be hanging or freezing, try setting the `cores` argument to 1 instead of 2 when you call `pm.sample`.
- While debugging your code, it might help to restart the kernel between each time you try to run it.
- Make sure you don't mix up standard deviation and variance!

(b) (2 points) Explain why using different values of σ_0^2 produces such different results.

(c) (2 points) Explain in plain English what assumptions we are making when we use a prior with a very small value of σ_0.

References

[1] Stuart Geman and Donald Geman (1984). *Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images*. IEEE Transactions on Pattern Analysis and Machine Intelligence, (6), 721-741.

[2]For more on this dataset, see here

Q.1)

Collaborator: Riya Berry

a) $Y_t \in \{0,1\}$ indicates whether or not the plate is infected.

Since this is a boolean, 0/1 variable, Bernoulli would be a good choice.

Question tells us to use $\mu(t)$ as a parameter.

$$\therefore\ Y_t \sim \text{Bernoulli}(\mu(t))$$

b) $\mu(t) := E[Y_t]$

$= P(\text{plate is infected at dilution } t)$

$= 1 - P(\underbrace{\text{there are 0 microbes on plate at dilution } t}_{\text{this is Poisson}(p_t V)})$

$= 1 - e^{-p_t V} \cdot \frac{p_t V^0}{0!}$ (density of Poisson$(\mu) = e^{-\mu}\frac{\mu^k}{k!}$)

$= 1 - e^{-p_t V} \cdot 1$

$$= 1 - e^{-p_t V}$$

c) Trying to get $\mu(t)$ in the form $\beta_0 + \beta_1 t \ldots$

$$\mu(t) = 1 - e^{-p_t V}$$

$$1 - \mu(t) = 1 - 1 + e^{-p_t V}$$ Subtracting both sides from 1 to isolate e

$$1 - \mu(t) = e^{-p_t V}$$

$$\log(1-\mu(t)) = \log(e^{-p_t V})$$ Taking log to get t out of the exponent

$$\log(1-\mu(t)) = -p_t V$$

$$-\log(1-\mu(t)) = p_t V$$

$$\log(-\log(1-\mu(t))) = \log(p_t V)$$ Taking log to convert product to sum

$$\log(-\log(1-\mu(t))) = \log(p_t) + \log(V)$$

$$\log(-\log(1-\mu(t))) = \underbrace{\log(p_t)} + \log(v)$$

We know that concentration is diluted by half each time

$$p_t = \frac{p_0}{2^t}$$

$$\log(-\log(1-\mu(t))) = \log\left(p_0/2^t\right) + \log(v)$$

$$\log(-\log(1-\mu(t))) = \log(p_0) - \log(2^t) + \log(v)$$

$$\log(-\log(1-\mu(t))) = \log(p_0) - t\cdot\log(2) + \log(v)$$

$$\log(-\log(1-\mu(t))) = \underbrace{\log(p_0) + \log(v)}_{\beta_0} \underbrace{- \log(2)}_{\beta_1} \cdot t$$

$$\therefore \quad \beta_0 = \log(v) + \log(p_0)$$

$$\beta_1 = -\log(2)$$

d) We know from (c) that $\beta_0 = \log(v) + \log(p_0)$

$$\beta_0 - \log(v) = \log(p_0)$$

$$e^{\beta_0 - \log(v)} = p_0$$

$$\Rightarrow \quad p_0 = e^{\beta_0 - \log(v)}$$

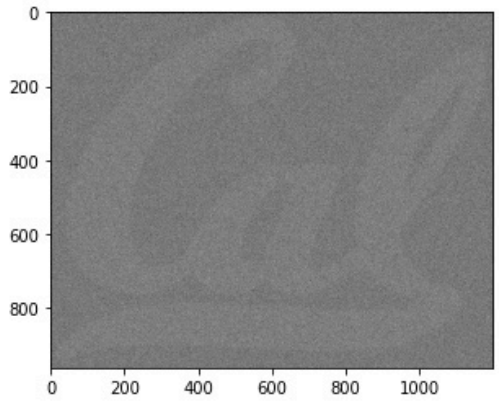
Q2)

a) **Part a**

```
X = np.load('X.pkl', allow_pickle = True)
```

```
plt.imshow(X, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f7b78003700>
```



b) Following the pattern, we get:

- Sample $Z_{1,3}^{(t)} \sim \boxed{p\left(Z_{1,3} \mid Z_{1,1}=Z_{1,1}^{(t)},\ Z_{1,2}=Z_{1,2}^{(t)},\ Z_{1,4}=Z_{1,4}^{(t-1)},\ \cdots,\ Z_{n,m}=Z_{n,m}^{(t-1)},\ X\right)}$

c) Pseudocode...

- Initialize $Z^{(0)} = X$ } given
- For $t = 1, \ldots, T$:
  - * Let $Z = Z^{(t-1)}$ Initialize curr value to prev timestep
  - * For $i = 1, \ldots, n$: } For all x, y combinations...
    - For $j = 1, \ldots, m$:
      - $N_{i,j} = [(i-1,j), (i+1,j), (i,j-1), (i,j+1)]$ get neighboring pixels
      - $S_{i,j} = \sum_{i',j' \in N_{i,j}} Z_{i',j'}$ use the fact that $\sum_{(i',j') \in N(i,j)} Z_{i',j'}$
      - Sample $Z_{i,j}$ from $N\left(\frac{\tau X_{ij} + b S_{ij}}{a+\tau}, \frac{1}{a+\tau}\right)$ given
  - $Z^{(t)} = Z$ Set the Z for this timestep

d) We know, $N(\mu, \sigma^2) = \sigma \cdot N(0,1) + \mu$

As a result, $N\left(\frac{\tau X_{ij} + b S_{ij}}{a+\tau}, \frac{1}{a+\tau}\right) = \sqrt{\frac{1}{a+\tau}} \cdot N(0,1) + \frac{\tau X_{ij} + b S_{ij}}{a+\tau}$

Hence..

Part d

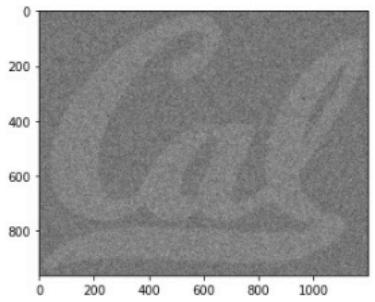
```
a = 250
b = 62.5
tau = 0.01
T = 1
n=X.shape[0]
m=X.shape[1]
```

```
start_time = time.perf_counter()

Z=X
for i in range(n):
    for j in range(m):
        N_ij = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
        S_ij = sum([Z[i_prime, j_prime] for (i_prime, j_prime) in N_ij if 0 <= i_prime < n and 0 <= j_prime < m])
        Z[i, j] = (np.sqrt(1/(a+tau)) * np.random.randn()) + ((tau*X[i,j] + b*S_ij) / (a+tau))

end_time = time.perf_counter()

print('Time taken:', end_time-start_time, 'seconds')
```

```
Time taken: 6.584930977027398 seconds
```

```
plt.imshow(Z, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f3bdaa1f580>
```



According to our timer, it took ~6.6 seconds to compute $Z^{(1)}$. As a result, it would take roughly $6.6 \times 100 = 660$ seconds to compute $Z^{(100)}$. That is approximately 11 minutes.

e) Doing so is justified because if $i+j$ is odd, all the neighbors $N_{i,j}$ are even. Similarly, when $i+j$ is even, all the neighbors $N_{i,j}$ are odd. Hence, updating half the variables at once is justified.

f)

Part f

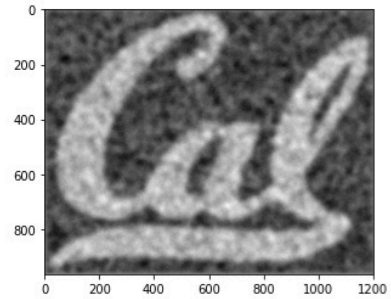
```
a = 250
b = 62.5
tau = 0.01
T = 100
n=X.shape[0]
m=X.shape[1]
```

```
start_time = time.perf_counter()

rows = np.indices((n, m))[0]
cols = np.indices((n, m))[1]
indices = rows+cols

even_unpadded = indices%2==0
odd_unpadded = indices%2!=0

Z = np.pad(X, 1)
even = np.pad(even_unpadded, 1)
odd = np.pad(odd_unpadded, 1)

for t in range(1, T+1):
    S = Z[2:, 1:-1] + Z[:-2, 1:-1] + Z[1:-1, 2:] + Z[1:-1, :-2]
    delta = np.sqrt(1/(a+tau)) * np.random.randn(n, m)
    temp = (tau/(a+tau)) * X[even_unpadded] + (b/(a+tau)) * S[even_unpadded] + delta[even_unpadded]
    Z[even] = temp

    S = Z[2:, 1:-1] + Z[:-2, 1:-1] + Z[1:-1, 2:] + Z[1:-1, :-2]
    delta = np.sqrt(1/(a+tau)) * np.random.randn(n, m)
    temp = (tau/(a+tau)) * X[odd_unpadded] + (b/(a+tau)) * S[odd_unpadded] + delta[odd_unpadded]
    Z[odd] = temp

end_time = time.perf_counter()
print('Time taken:', end_time-start_time, 'seconds')
```

```
Time taken: 15.01666986499913 seconds
```

```
plt.imshow(Z, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f80528e0bb0>
```



Based off of this, it took roughly 15 seconds.

Q3)

a)

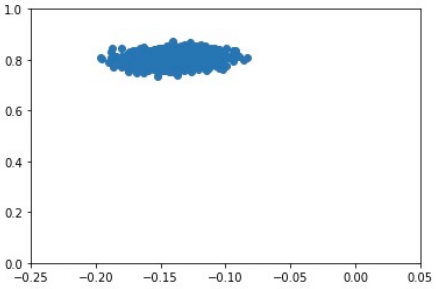
Part a

```
sigma_squared = [1, 0.01, 10**(-4)]
```

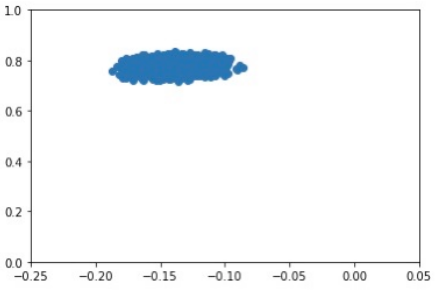
```
x1 = np.array(elections['govtrack_ideology'])
x2 = np.array(elections['house_dem18_margin'])
y = np.array(elections['house_dem20_margin'])
d = {'x1': x1, 'x2': x2, 'y':y}
df = pd.DataFrame(data=d)
```

```
import pymc3 as pm

with pm.Model() as linear_model:

    my_priors = {"Regressor": pm.Normal.dist(mu=0, sigma=np.sqrt(sigma_squared[0]))}

    pm.glm.GLM.from_formula('y ~ 0 + x1 + x2', df, priors=my_priors)

    trace = pm.sample(1000, cores=1)
```

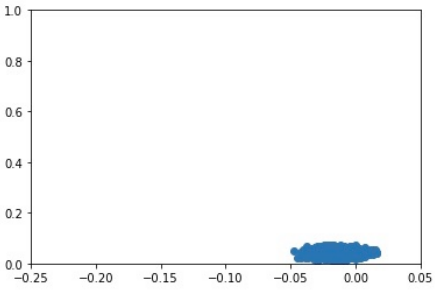
```
plt.scatter(trace['x1'], trace['x2'])
plt.xlim(-0.25, 0.05)
plt.ylim(0, 1)
```



$\sigma_0^2 = 1$



$\sigma_0^2 = 0.01$



$\sigma_0^2 = 10^{-4}$

b) This happens because as we make σ_0^2 smaller and smaller, our prior becomes very heavily centered around its mean. Due to this, the coefficients of our regressors became as close to 0 as possible. So our prior doesn't affect much, we rely on the data more.

c) If σ_0 is small, then we are assuming that the ideology and the 2018 margin are not affecting the 2020 margin much.