Project Overview

Goal

- Model and predict exoplanet radius based on the following properties:
  - Planet Mass
  - Star Effective Temperature
  - Equilibrium Temperature
  - Stellar Mass
  - Distance
  - Stellar Metallicity
- Use Bayesian Linear Regression to allow users to incorporate prior knowledge or understanding of astronomy to get more meaningful results. This might look like "coefficient can only be positive" or "coefficient is usually around X."
- Bayesian linear regression also allows users to understand the uncertainty in the model as the posterior distribution can create credible intervals for coefficients and predictions.
- Uses Gibbs Sampling and conjugate priors for computation efficiency. Beta is from multivariate normal distribution and variance is from inverse gamma distribution.

Dataset

- Source: NASA Exoplanet Archive of confirmed planets
- Size: 38421 x 42
- Link: https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=PS

Data Processing

Loading into Rust

- Used the csv crate to read each row and return a 2D array of the features and a 1D array of the target variable.
- First created vectors (vector of vector for the features, vector of floats for the target) and then populated them as I iterated through each record.
- Converted the vectors to array at the end of reading the csv file

Cleaning

- Loaded the full data into Python first and performed some exploratory data analysis
- Reduced the 42 columns down to 7, including the target variable.
- Fit a linear regression model to the data first to decide which variables are statistically significant. I also analyzed the columns to deduce which variables would be most important. For example, planet names likely have no relation to the radius.
- Removed NaN rows as well, reducing it down to ~1400 rows
  - While Bayesian Linear Regression handles missing rows very well, I decided to just drop them all together as I did not quite understand how to properly encode them and have them work with the model

Code Structure

Modules

- main.rs - The main app for interacting with the model and running the tests. It is short and not overly complicated so users do not get overwhelmed.
- blr.rs - Contains all the functionality for my model, including the struct and all the respective methods. Separates the core logic from the main to keep it clean and so the main can just bring it into scope.
- tests.rs - Contains the tests for my code that I separate from the main.rs file and it calls on it

Key Functions and Types

- <u>struct BayesLinearRegressor</u>
    - x: training data features
    - y: training data target
    - beta: array of the estimated coefficients
    - variance: variance term for normal distribution of error terms
    - beta_prior_mean: prior of the beta values
    - beta_prior_cov: prior of the covariance matrix of the beta values
    - a: shape for the inverse gamma (for variance)
    - b: scale for the inverse gamma (for variance)
    - beta_samples: vector of arrays containing the samples from Gibbs sampling
    - variance_samples: vector of floats containing variance samples from Gibbs sampling
    - num_samples: total number of samples the model should generate
- <u>new() method:</u>
    - What it does: Initializes a new instance of the model with required parameters and optional parameters for users to input with defaults if not inputted of the data
    - Inputs: Data and optionally, parameters for prior, likelihood, and Gibbs Sampling settings
    - Outputs: Returns an instance of the model
    - High-level logic: Unwrapping with defaults and adding intercept term
- <u>sample_variance() method</u>
    - Purpose: Generates samples of variance from the inverse gamma distribution
    - Inputs: Parameters of the distribution, a and b since they can differ
    - Outputs: A sample from the inverse gamma distribution with the inputted params
    - Core logic and key components:
        - Using RNG and Gamma struct from rand_distr crate to generate samples.
        - Inverse gamma is taken by doing 1 / gamma sample

- sample_prior() method
    - What it does: Generates samples of beta and variance to initialize the Gibbs Sampling
    - Inputs: Reference to self
    - Outputs: None as it just updates the model attributes
    - High-level logic:
        - Sample_beta function:
            - Populates an array of beta values by sampling each value from N(0,1) and transforming them into multivariate normal distribution samples with N(beta_prior_mean, beta_prior_cov)
            - $\beta_{multivariate} = \beta_{prior} + \sigma^2 Lz$ where L is lower triangular matrix from Cholesky decomposition and z is the beta values sampled from N(0, 1)
        - Updates beta and variance by calling the sample_beta and sample_variance methods and passes in the model attributes to do so
- sample_beta_posterior() method
    - What it does: Samples beta values from the posterior distribution
    - Inputs: Reference to self
    - Outputs: 1D Array where each element is a Beta value from a sample
    - High-level logic:
        - Transforms samples from N(0,1) to multivariate normal with $\beta_{multivariate} = \beta_{prior} + \sigma^2 Lz$
        - Calculating covariance matrix: $\Sigma_n = (\frac{X^T X}{\sigma^2} + \Sigma_0)^{-1}$ where X is the data, σ^2 is the variance, Σ_0 is the prior covariance matrix.
        - Calculating mean vector: $\mu_n = \Sigma_n(\frac{X^T y}{\sigma^2} + \Sigma_0 \mu_0)$ where y is the target variable, and $\mu_0$is the prior mean vector
        - Uses rand_distr::Normal to generate random samples from N(0,1)
- sample_variance_posterior() method
    - What it does: Generates a sample of variance from its posterior distribution
    - Inputs: Reference to self
    - Outputs: Sample from the posterior distribution
    - High-level logic:
        - Calculates residuals from $r = y - y_{pred}$
        - Calculates SSR: $||r||^2$
        - Uses conjugate priors so the Inverse Gamma posterior has the following updated parameters
            - $a' = a + \frac{n}{2}$ where a is the prior inverse gamma shape and n is the number of observations in the data

                - $b' = b + \frac{SSR}{2}$ where b is the prior inverse gamma scale and SSR is the sum of squared residuals
            - Passes in those updated param values to the sample_variance to generate a new sample from the posterior distribution
- gibbs_sample() method
    - What it does: Updates the values of beta and variance for each step in Gibbs Sampling
    - Inputs: Reference to self
    - Outputs: None as we just update the model attributes
    - High-level logic:
        - Updates beta and variance by calling the respective methods of sampling from the posterior distribution
- run_gibbs_sampling() method
    - What it does: Uses Gibbs Sampling to populate the beta and variance sample vectors
    - Inputs: Reference to self
    - Outputs: None as it just updates the model attributes
    - High-level logic:
        - For loop to iterate through the number of samples and calls the gibbs_sample method to generate the samples and populate them
        - Update the current beta and variance at the end of the iterations to be the MMSE estimator of the samples; the average
        - Beta update
            - Iterates through the beta_samples to get the total betas and then apply closure of dividing by number of samples to get the mean
        - Variance update
            - Iterates through the variance samples, gets the sum, and divides by the number of samples to get the mean
- empirical_beta_covariance() method
    - -- Asked ChatGPT how to get the covariance matrix from the sampled beta from my Gibbs sample –
    - What it does: Calculates the covariance matrix from the beta samples
    - Inputs: Reference to self
    - Outputs: 2D covariance matrix
    - High-level logic:
    - Variables
        - n: number of samples from Gibbs sampling
        - d: number of regression coefficients
        - mean: MMSE estimator of the beta coefficients
    - Initializes an d x d covariance matrix of 0s
    - Converts the delta row vector into column vector with insert_axis
    - Iterates through each sample from the beta samples vector, calculates the difference from the mean and squares it by taking the dot product

    - Adds the dot product to the covariance matrix and then divide by n - 1 to get the sample variance.
- predict_single_with_credible_interval() method
    - What it does: Makes a prediction on a new data point and provides the 95% credible interval
    - Inputs: Reference to self, new data point
    - Outputs: Tuple of (mean, lower_bound, upper_bound)
    - High-level logic:
        - Adds intercept term of 1 to the data point
        - Calculates mean prediction by computing dot product of Beta and the data point
        - Calculates the prediction variance and since the linear regression model assumes the data to be normally distributed, gets the upper and lower bounds with 1.96 (z-score) of the prediction variance
        - $\sigma^2_{prediction} = \sigma^2 + x\Sigma_{beta}x$ where Σ_{beta} is the empirical covariance of the beta regression coefficient samples. **Why am i using empirical covariance?**
        - $\mu \pm 1.96 \cdot \sigma^2_{prediction}$ where μ is the predicted value using the MMSE estimator for beta and variance
- predict_multiple_with_mean_beta() method
    - What it does: Makes prediction on multiple data points and returns the predictions
    - Input: Reference to self, test data
    - Outputs: Array of the predictions
    - High-level logic:
        - Adds an column of 1s for the intercept term with concatenate
        - Computes the prediction by doing the dot product of the Beta coefficients and the test data
- get_beta_credible_interval() method
    - What it does: Returns a vector of tuples where each tuple is the 95% credible interval for the beta coefficients
    - Inputs: Reference to self
    - Outputs: Vector of (f64, f64) tuple
    - High-level logic:
        - Initializes empty vector
        - Uses for loop to update each tuple for the predictor
        - Gets a vector of the ith column of the beta samples with into_iter() and then map with closure of the ith column and then collects it into vector
        - Uses sort_by since f64 does not support normal comparison
        - Gets the lower_index and upper index by getting the 2.5 and 97.5 percentiles of the data and then rounds them to nearest 3 decimal points and pushes them to the tuple and to the vector

- export_beta_samples_to_csv() method
  - -- Asked ChatGPT how to export the beta samples as a CSV file –
  - What it does: Exports the beta samples as a CSV file to use for post-processing or analysis
  - Inputs: Reference to self, file path
  - Outputs: None but it creates the CSV file in the project root folder
  - High-level logic:
    - Creates a file and wraps it in the BufWriter
    - Iterates through each row in the beta_samples, converts them to strings with closure and then collects them into a vector of strings and joins them with "," for CSV notation
- r_squared() method
  - What it does: Calculates the R^2 for the data
  - Inputs: Reference to self
  - Outputs: f64 R^2 value
  - High-level logic:
    - Computes predicted target: $y_{pred} = X\beta$ where X is the training data features and β is the beta regression coefficients
    - Computes residual: $r = y - y_{pred}$
    - Computes Sum of Squared residuals: $SSR = r^2$
    - Computes the total sum of squares: $TSS = \sum_{1}^{n} (y - \mu_y)^2$ where n is the number of data points, y is the ith target value, μ_y is the mean of the target values
    - $R^2 = 1 - \frac{SSR}{TSS}$

Main Workflow

1) Users create a new instance of the model by passing in the data. In the new() method, there are defaults for the other parameters, and users can pass in optional arguments like prior knowledge.
   a) In my new() method, I do some checks such as making sure the data are compatible and I also add in an intercept term to the data.
2) Then, they call the run_gibbs_sampling() method which first calls the sample_prior() method to generate the first sample and update the beta and variance to use for the sampling, and then at every iteration, calls the gibbs_sample() method to begin generating samples and updating the beta and variance values.
   a) sample_prior() calls on sample_beta and sample_variance to generate the first beta and variance samples
   b) gibbs_sample() calls on sample_beta_posterior() and sample_variance_posterior() to sample from the posterior distribution and update beta and variance of the model
   c) At the end of the algorithm, I update beta and variance to be their MMSE estimator of the posterior distribution by taking the average of the posterior.
3) Then, they can predict on a single data point and get its 95% credible intervalwith predict_single_with_credible_interval()
   a) This method calls on the empirical_covariance() method to estimate the beta covariance matrix by approximating it using the estimates from the Gibbs sampling algorithm
4) Users can also predict on several values using predict_with_multiple() method which just adds a column of 1s and takes the dot product with the beta regression coefficients.
5) Users can also get the credible intervals for each predictor with get_beta_credible_intervals() to get the 95% credible interval which sorts the predictors and gets their 2.5 and 97.5 percentiles.
6) Users can also calculate the R_squared value from using the r_squared() method.
7) Users can also export the samples from Gibbs sampling with export_beta_samples_to_csv() method to use for post-processing.

Tests

- **test_new_with_defaults()**: Checks the new() method works and that it is storing the proper data and model parameters. Matters as the model needs to be properly created
- **test_sample_variance_positive()**: Tests if the sampled variances are positive. Matters as you can't have negative variances and this is used extensively throughout the model.
- **test_sample_prior_updates_model()**: Test if the sample prior method provides a positive variance and the size of beta is correct. Matters as variance cannot be negative and beta has to be the correct size to predict on new data.
- **test_run_gibbs_sampling_produces_samples()**: Tests if the gibbs sampling algorithm is producing samples. Matters as we need samples to get an MMSE estimator to predict on new data

Test Output.
running 4 tests
test tests::test_new_with_defaults ... ok
test tests::test_sample_variance_positive ... ok
test tests::test_sample_prior_updates_model ... ok
test tests::test_run_gibbs_sampling_produces_samples ... ok

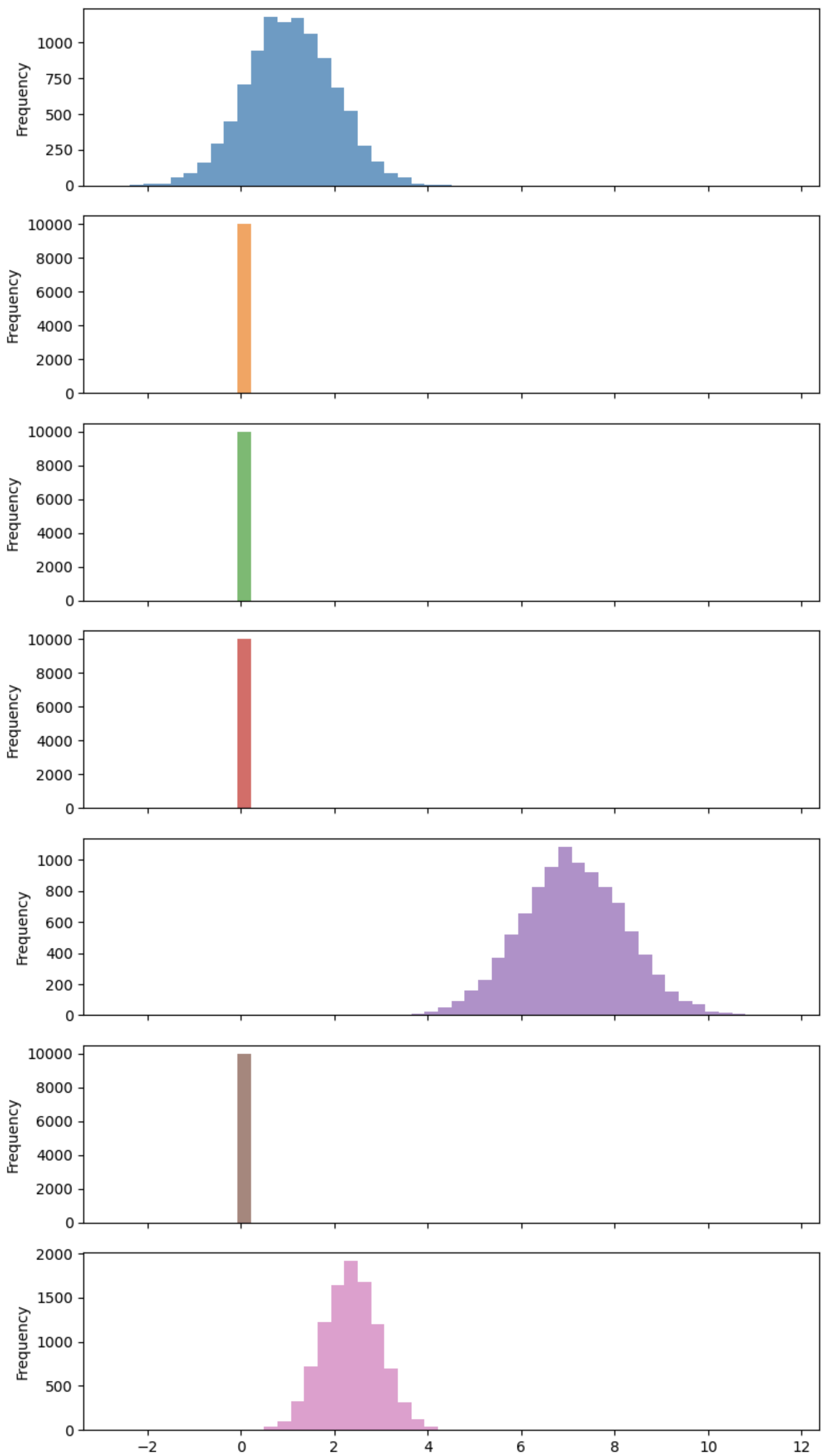
test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Results

```
Features:
[[7.81, 5234, 1958, 0.905, 12.5855, 0.31],
 [16.3, 5766, 546, 0.961, 179.461, -0.15],
 [3932, 6935, 2001, 1.41, 589.423, -0.34],
 [2002.3189641, 3406, 434, 0.37, 10.8864, 0],
 [327.3649, 5950, 1915, 0.95, 787.909, -0.3],
 ...,
 [344.52772, 5370, 1203, 0.914, 276.211, 0.05],
 [1398.452, 6720, 1577, 1.47, 235.479, -0.07],
 [225.34147, 6250, 1743, 1.405, 234.149, 0.432],
 [4.52, 5870, 1147, 1.02, 18.2702, 0.05],
 [4.82, 6037, 1169.8, 1.094, 18.2702, 0.08]]
Targets:
[2.08, 2.23, 21.59, 12.44196858, 17.385159, ..., 12.206601, 23.20263, 15.389957, 2.06, 2.042]
-----------------------------------------------
Prediction for first data point: 10.071, 95% Credible Interval: (2.012, 18.129)

Predictions for multiple data points:
[10.070804659795549, 4.921172634151426, 18.595335702323315, 4.901617437286318, 11.817497393316428, ..., 8.45275822382056, 13.451180902313515, 13.500951911506561, 7.0
295458008435405, 7.55165740896255]

MMSE Estimates for Beta: [1.0828657493942682, 0.0013929319637889011, -0.0009085303189175933, 0.003321717828089244, 7.128487487605853, 0.004128633848187097, 2.3391683
74194763]

[(-0.839, 2.984), (0.001, 0.002), (-0.002, -0.0), (0.003, 0.004), (4.894, 9.355), (0.003, 0.005), (1.155, 3.536)]

R^2 for the model: 0.5300892613921606
```

- R squared of 0.53; 53% of the variance in the exoplanet radius can be explained by the variance in the predictors
- MMSE estimates for the predictors are shown in the image

Posterior Distribution of the Parameters



Usage Instructions

Usage

- Download the code and do cargo run –release. You can do regular cargo run as well

Toml file

```
[package]
name = "finalproject"
version = "0.1.0"
edition = "2024"

[dependencies]
ndarray = "0.15"
ndarray-linalg = { version = "0.16", features = ["openblas-system"] }
csv = "1.1"
rand = "0.8"
rand_distr = "0.4"
```