Darian Irani

Insurance policy premium modelling - Final Report

Project Description

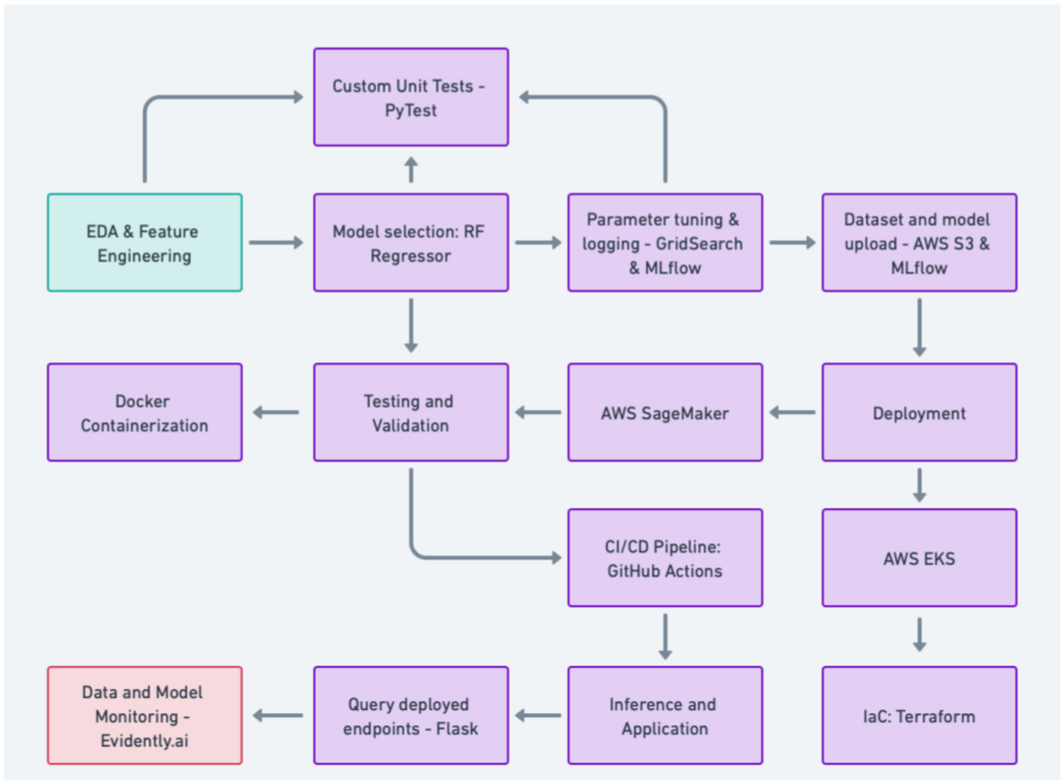
The client operates in the health insurance sector and has partnered with C5i to gain data-driven insights into their existing product. Due to privacy concerns and GDPR regulations, the client's identity is confidential, and the data provided for model training was synthesized to resemble the original dataset. Despite this, the synthesized data retains the same statistical characteristics, enabling senior team members with access to the confidential information to replicate my work on model development in the future.

In the insurance industry, accurately predicting claim likelihood is crucial for effective risk assessment and policy pricing. In this latter part of my internship, I developed a model to assess the risk profiles of previous policyholders to help determine the optimal premium that the client can charge future policyholders. The main objective of this partnership is to help insurance underwriters optimize loss ratios for profitability. The loss ratio is a key metric that measures the ratio of claims paid by an insurer to the premiums collected. The model uses various input features, including policyholder demographics and lifestyle factors, to predict the target variable, 'policyholder expenses' using supervised learning techniques.

My daily activities and overall responsibility are outlined below, this formed the backbone of my end-to-end project deliverables:

- **Exploratory Data Analysis** of the synthesized raw dataset to understand distributions and datatypes of data features.
- **Data Cleaning** to handle any missing values and outliers that may skew model performance and **Feature Engineering** to encode categorical variables for model interpretability.
- **Model Selection** based on what type of target variable I am trying to predict **Hyperparameter Tuning** to achieve the optimal model for the task.
- **PyTest** to write custom unit tests that evaluate whether the logic of my code is working as it should throughout the entire model development and deployment lifecycle.
- **AWS SageMaker and EKS** for deployment of model once completed experimentally on the local machine, developed using both the AWS UI and IaC (Terraform) methods.
- **CI/CD Pipeline** created via GitHub Actions to automate different parts of the lifecycle such as installing dependencies in Docker environment and automatically running test cases.
- **Flask** application created to query deployed endpoints and host model on a server.
- **Evidently.ai** library used to monitor Data drift and Model drift using inference data.

Figure 1 displays a visual schematic of my project workflow which highlights the tech stack I used and the order of tasks I carried out during the internship. In this section of the report, I will outline the work completed and the rationale behind it, explore the approaches taken and those not pursued, and provide insights on future directions for the project, including how my contributions can be further developed.


Custom Unit Tests - PyTest
EDA & Feature Engineering
Model selection: RF Regressor
Parameter tuning & logging - GridSearch & MLflow
Dataset and model upload - AWS S3 & MLflow
Docker Containerization
Testing and Validation
AWS SageMaker
Deployment
CI/CD Pipeline: GitHub Actions
AWS EKS
Data and Model Monitoring - Evidently.ai
Query deployed endpoints - Flask
Inference and Application
IaC: Terraform
Figure 1: Schematic of end-to-end project workflow

I started the project after being given the client's raw dataset (after synthesizing). At this stage I was trying to first understand the data, what input features I can extract for the model, and whether any features need to be created or transformed (encoded). Using primarily the Pandas library, I performed Data Cleaning to replace any null values and outliers. I used two different methods to detect and replace null and outlier values: Z-score method and the IQR range. Equation 1 outlines the formula to calculate the Z-score, which measures a data point's relationship to the mean of a group of values, measured in terms of standard deviation from the mean.

$$Z = \frac{(X - \mu)}{\sigma} \tag{1}$$

I used a common threshold of > 3 and < -3 to filter out outliers in the data. I also carried out further exhaustive filtering using the IQR method which measures the statistical dispersion of data. I replaced null and outlier values with that respective column's median value to maintain the dataset's statistical properties as well as possible.

Next, I identified a subset of categorical features that needed to be encoded to numerical values for the model, I performed two types of encoding: One-Hot and Label. Features which had a binary value such as the 'smoker' feature was encoded using One-Hot encoding. Features that had multiple categories such as the 'region' feature was Label encoded. During this Feature Engineering stage, I was also able to exclude any redundant features and as a result, the final subset of features I chose contributed the model towards the dependent variable.

After Feature Engineering, I plotted a Correlation Matrix, as seen in Figure 2, to visualize the relationship between various features in the dataset. This plot helps me compare multiple features at once while checking for multicollinearity, which can make it difficult to isolate the individual effect of each predictor on the dependent variable, leading to model reliability issues.

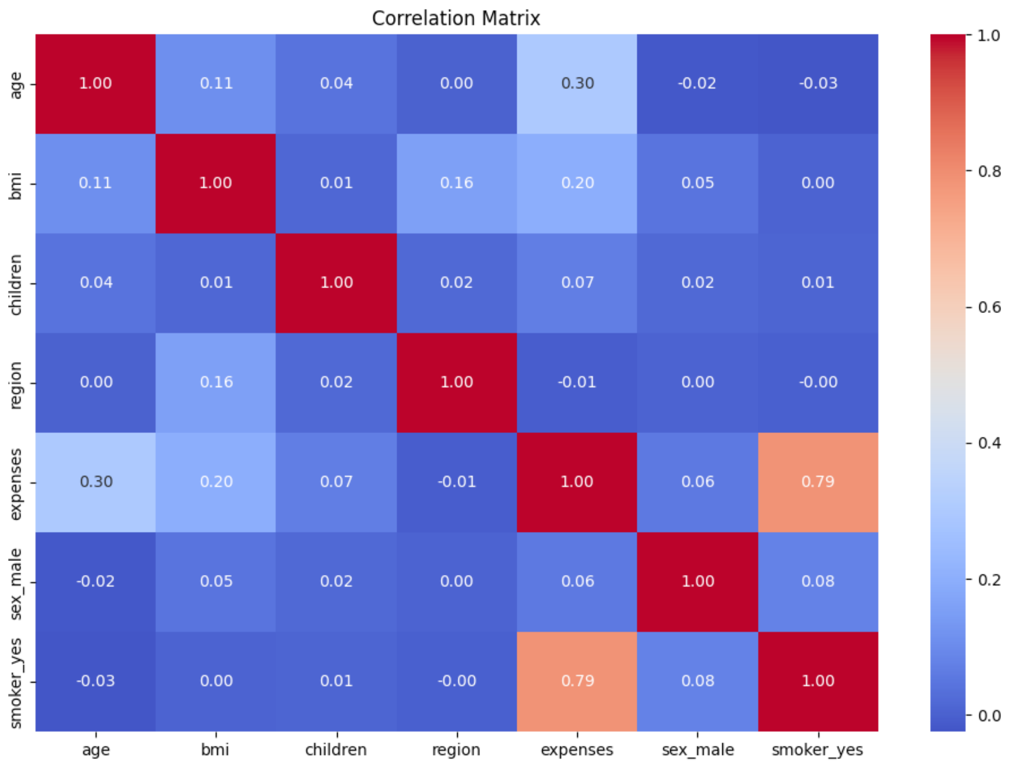


Figure 2: Correlation Matrix

The matrix implies that the strongest correlation lies between 'smoker' and 'expenses' with a correlation of **0.79**, indicating that being a smoker strongly influences higher medical expenses. We can also see that 'age' plays a big enough role in predicting the target variable. This matrix was also plotted as a presentation deliverable for the client to use in the future in case they want a high-level understanding of what specific policyholder feature contributes the most to the dependant variable (expenses).

Lastly, I plotted the distribution of multiple numerical features to observe their statistical trends and gauge whether any form of standardization or normalization was required. After examining these plots, I applied either standardization or normalization based on which features required it. Figure 3 displays a normalized plot of the policyholders' BMI, evidently displaying that the distribution is unimodal and symmetric.

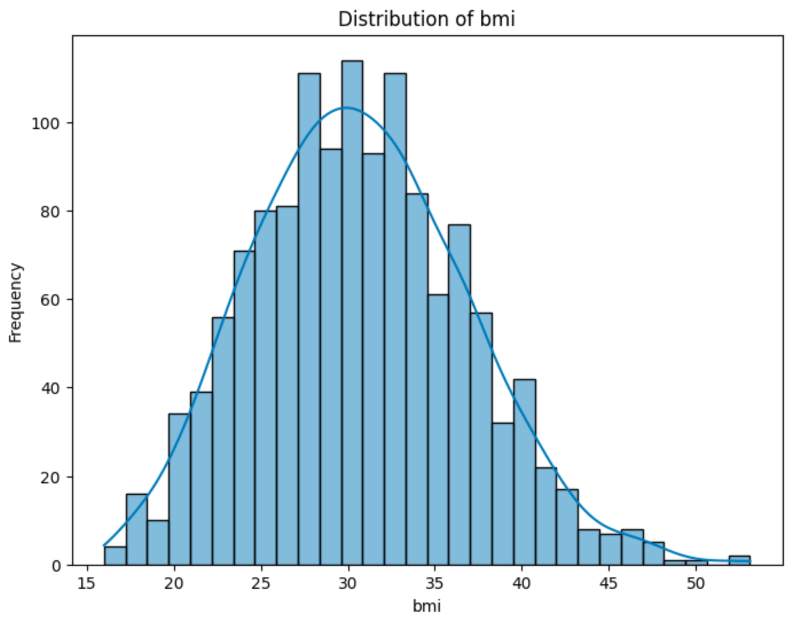


Figure 3: BMI histogram distribution after normalization

During this stage I also established that my target variable is continuous and possesses a linear relationship with the input features, which indicates the need for a regression model to be fit with the data. I will now discuss the systematic process I followed in selecting, evaluating, and fine-tuning different regression models.

To select the most optimal regression model, I defined a range of regression models to fit and test on the data using their base parameters as a start. The dataset was split into train, test, and validation subsets. Each of these models were run sequentially on the training data using a simple 'for loop' and their performance was evaluated using common regression metrics. These metrics included: Root Mean Squared Error (RMSE), R-squared (R^2), Mean Absolute Percentage Error (MAPE), Explained Variance Score (EVS). After evaluating each regression model's performance, I narrowed down two best performing models: Random Forest Regressor and XGBoost.

I then performed hyperparameter tuning using GridSearchCV for both models, this tool exhaustively iterated through a pre-defined hyperparameter grid to output the best model parameters. This resulted in the Random Forest Regressor outperforming the XGBoost model slightly even though Gradient Boosting Regressors use powerful ensemble methods that model complex relationships which normally perform better than random forests. The hyperparameter tuning process was then taken a step further by evaluating the model performance using MLflow. This platform was used to visualize with permutations and combinations of parameter values output perform the best for all metrics consistently. Figure 4 displays a graph, generated with MLflow, that was used to evaluate how the parameter 'n_estimators' performs for each metric, this process was repeated for each feature. As seen in the plot, each metric has its own optimal 'n_estimators' value, and hence the best combination of parameters was selected via this method. This ultimately resulted in confidently, the best model with the best performing parameters as per the respective metrics defined, given the training data supplied.

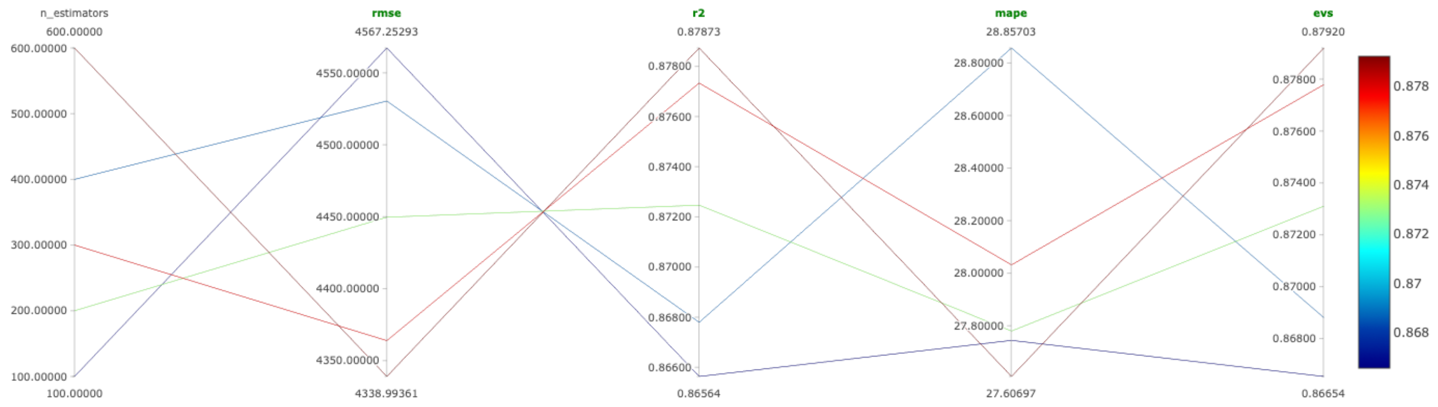


Figure 4: Hyperparameter comparison with regression metrics

Table 1 displays the performance of the fine-tuned Random Forest Regressor model fitted to the training data supplied by the client.

Table 1: Model performance with regression metrics

RMSE	R^2	EVS	MAPE
4349	0.88	0.88	28%

The RMSE score reflects the average magnitude of error in the model's predictions. Considering the range of values and overall dispersion of the target variable, this error can be deemed reasonable and is justified. The R^2 metric indicates that 88% of the variance in the actual premium values is explained by the model's features. This can be considered as a strong fit, indicating that the model captures most of the variability in the data. Similarly, an EVS score of 88% reinforces the fact that the model is performing well in explaining most of the target variable's variability. Lastly, the MAPE value suggests that the model's predictions are off by 28%, which is useful to understand how large the prediction errors are in relation to the true values. Figure 5 shows a scatter plot comparison between the predicted premium and actual premium values.

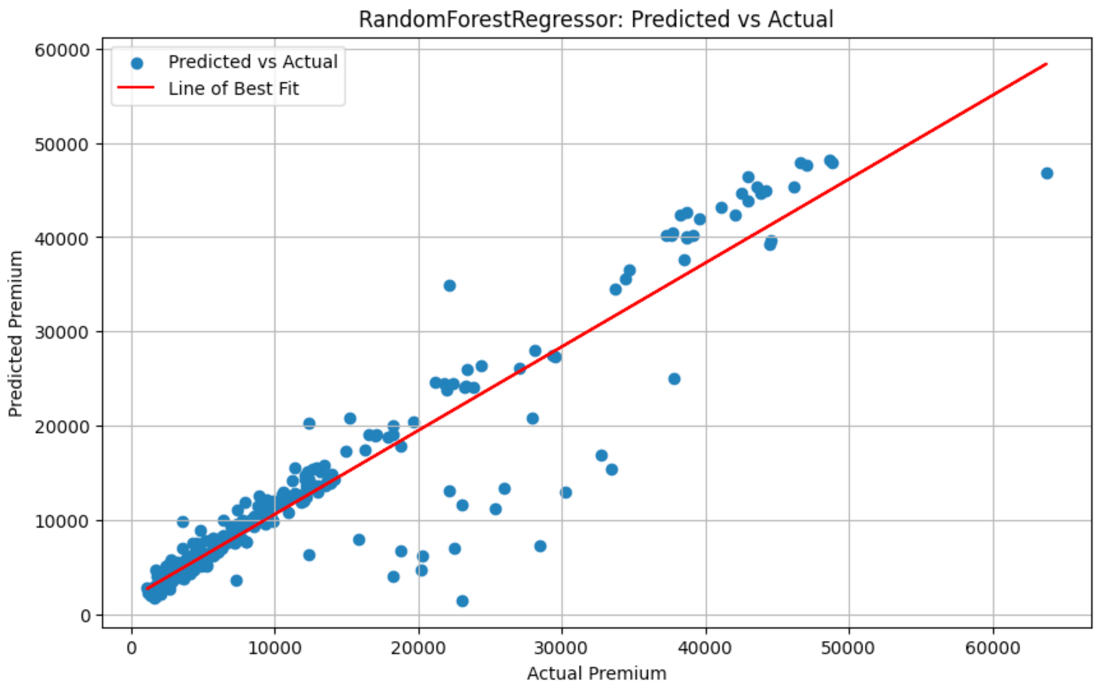


Figure 5: Predicted vs Actual Premium values

The line of best fit indicates that the predicted values align closely with the actual values as it follows the y=x trendline, which is a visual indicator of good model performance. We can also make further observations from the plot: firstly, most of the points under the 15,000 premium value are clustered close to the trendline indicating that the model is good at predictions in the lower premium range. The outliers are present in the higher premium range (above 30,000) therefore concluding that the model is most accurate at the lower to mid-range premiums.

The next stage of the project was to create a reusable deployment architecture for the Senior Data Scientists and ML Engineers to later continue working with as per client needs and new inference data. For AWS EKS and SageMaker deployments, I deployed both using the AWS web console and then also using Infrastructure as Code (IaC) methods for the repeatability of my projects as C5i normally uses IaC methods.

AWS Elastic Kubernetes Service (EKS) is a fully managed service allowing for Kubernetes to automate the deployment, scaling, and management of containerized applications, including ML models. First an EKS Cluster is created which contains an API which provides the main interface to interact with the Kubernetes services and servers. Next, a Docker Image is built to containerize the model, install dependencies, and run any test cases created in the future and then pushed to the AWS Elastic Container Registry (ECR). Custom .yaml configuration files are then written which contain a set of instructions for the deployment and service stages which are used to deploy the model to EKS. The seniors in the team intend to use an ingress controller to expose the service externally to HTTP/s requests using the existing EKS architecture I built.

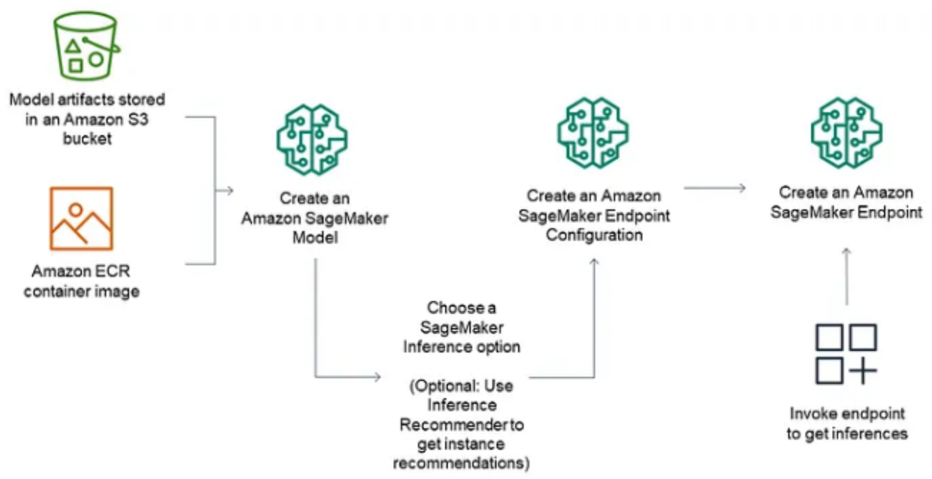
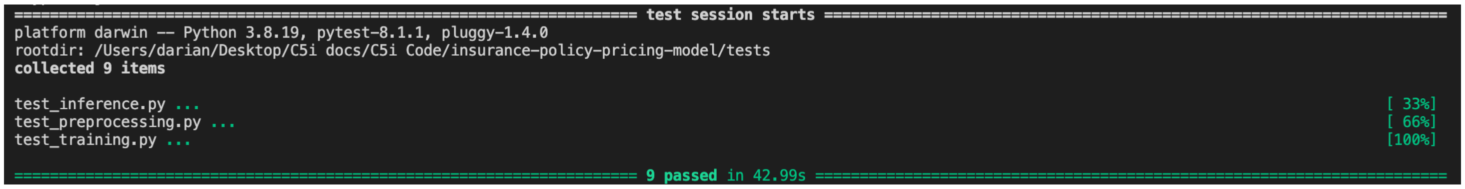

Model artifacts stored in an Amazon S3 bucket
Amazon ECR container image
Create an Amazon SageMaker Model
Choose a SageMaker Inference option
(Optional: Use Inference Recommender to get instance recommendations)
Create an Amazon SageMaker Endpoint Configuration
Create an Amazon SageMaker Endpoint
Invoke endpoint to get inferences

Figure 6: AWS SageMaker deployment pipeline [1]

Figure 6 outlines the step-by-step processes involved in the AWS SageMaker deployment which chosen as the main deployment method for this project offering tools and infrastructure to support the full ML lifecycle. Most importantly, SageMaker automatically scales the compute resources allocated based on the load, resulting in great scalability should the team gather large amounts inference data. At first, all the model artifacts and datasets were added to an S3 bucket and, for future use, these are only accessed by the code via AWS credential configuration and access of environmental variables for collaboration within the C5i team. After setting up the necessary instance types and endpoint configurations the endpoint was deployed so that requests can be sent using an API. SageMaker was configured to automatically scale the endpoint based on traffic and this was thoroughly tested by the team and I using test (inference) data to monitor model performance.



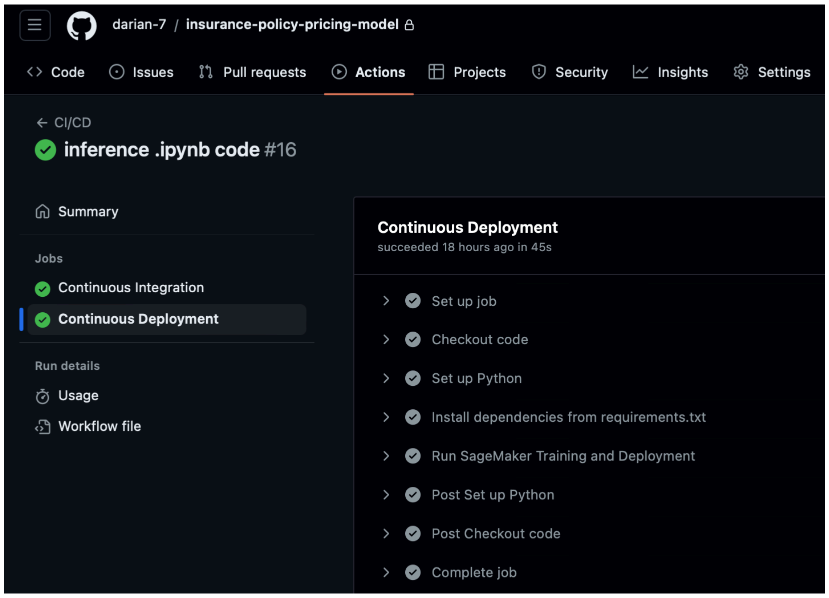
```
====================================================================== test session starts ======================================================================
platform darwin -- Python 3.8.19, pytest-8.1.1, pluggy-1.4.0
rootdir: /Users/darian/Desktop/C5i docs/C5i Code/insurance-policy-pricing-model/tests
collected 9 items

test_inference.py ...                                                                                                                                     [ 33%]
test_preprocessing.py ...                                                                                                                                 [ 66%]
test_training.py ...                                                                                                                                      [100%]

====================================================================== 9 passed in 42.99s =======================================================================
```

Figure 7: PyTest unit tests

Throughout the model development and deployment process, I wrote custom unit tests that test for successful model training, data ingestion, AWS configuration, model performance, and successful deployment. The code in the notebook was split into preprocessing, training, and inference .py files which were then tested by their respective PyTest files, as seen in Figure 7. These tests are automatically run by a CI/CD pipeline developed on GitHub Actions, visualized in Figure 8. This pipeline ultimately reduces the deployment time by **30%** by sequentially running the set of tasks written in the .yaml file I developed.


darian-7 / insurance-policy-pricing-model

inference .ipynb code #16

- Continuous Integration
- Continuous Deployment

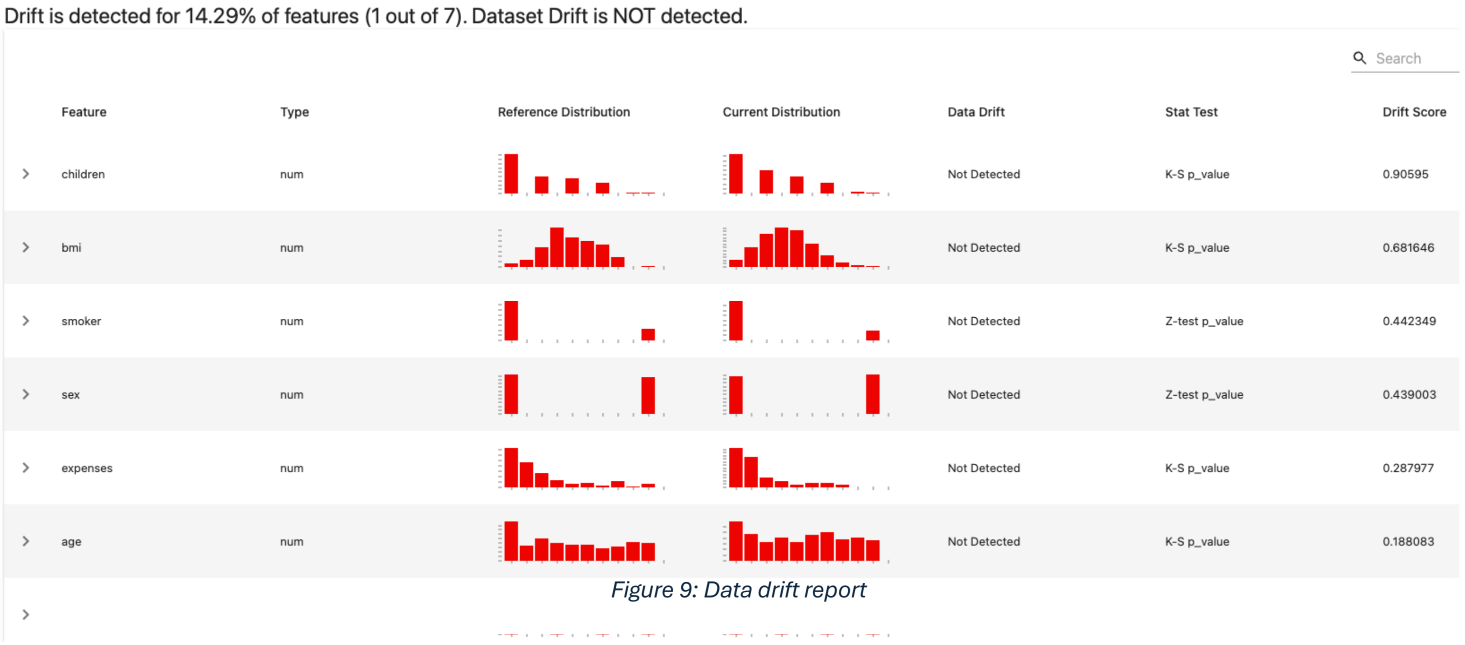
Continuous Deployment
succeeded 18 hours ago in 45s

- Set up job
- Checkout code
- Set up Python
- Install dependencies from requirements.txt
- Run SageMaker Training and Deployment
- Post Set up Python
- Post Checkout code
- Complete job

Figure 8: CI/CD pipeline created using GitHub Actions

The final part of the project involves setting up infrastructure to monitor the model for data drift and model drift. Data drift can occur when the statistical properties of the new inference data fitted to the model starts to get different than the properties of the data the model was trained on. Model drift occurs when the relationship between features and the target variable evolves. Hence, these two inevitable occurrences need to be accounted for and managed appropriately to ensure optimal model performance. Evidently is a library that creates dashboards and reports of how your model and data performs with new inference data.

Figure 9 outlines one of the Evidently data drift dashboards that presents an analysis of error bias for the top 5% of errors in the model's predictions.


Drift is detected for 14.29% of features (1 out of 7). Dataset Drift is NOT detected.

Feature	Type	Reference Distribution	Current Distribution	Data Drift	Stat Test	Drift Score
children	num			Not Detected	K-S p_value	0.90595
bmi	num			Not Detected	K-S p_value	0.681646
smoker	num			Not Detected	Z-test p_value	0.442349
sex	num			Not Detected	Z-test p_value	0.439003
expenses	num			Not Detected	K-S p_value	0.287977
age	num			Not Detected	K-S p_value	0.188083

Figure 9: Data drift report

The model was fed with mock inference data that was synthesized through unseen subsets of the original dataset to set up this infrastructure for future work. The table shows that, overall, the dataset as a whole has not experienced significant drift. The library assigns a drift score based on in-built statistical tests such as chi-square, z-test, and K-S p-values to determine this. The existence of drift in the 'region' feature can indicate a change in geographical distribution of the data. This drift could impact model performance if 'region' possesses a heavy weighting to the model's results. Figure 10 outlines an example from one of the many graphs from the model drift evaluation.

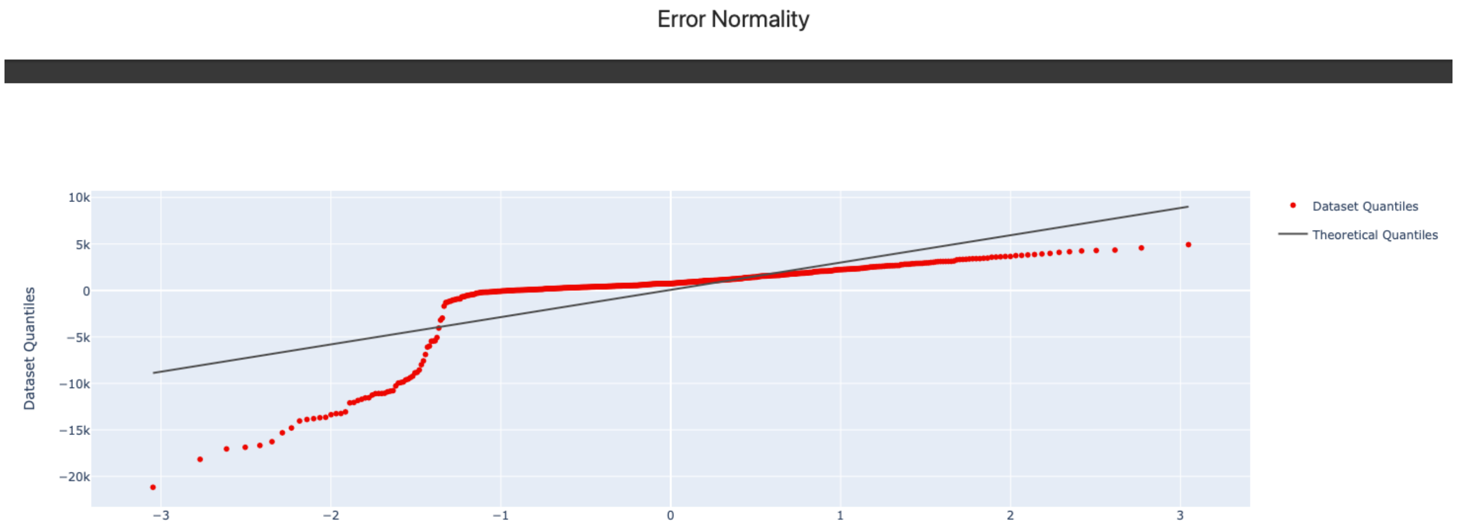


Figure 10: Model drift report

This graph is a Q-Q plot (Quantile-Quantile) plot that assesses the normality of the errors (residuals) in the model's predictions. It helps to check whether the errors follow a normal distribution. If the points lie on or close to the black line, then it is a good indicator that the model's assumptions hold. This graph shows the importance of monitoring a ML model after deployment as this model is no longer performing in the way it should – given that the residuals do not follow a normal distribution. This infrastructure for model and data drift had hence been set up for the senior members of the team to use in the future when new inference data is received from the client.