Auto Insurance Claim Prediction for Porto Seguro

Oamen Modupe
oamenmodupe@gmail.com

TABLE OF CONTENTS

1. Discover
   1.1 Understanding the problem
   1.2 Why ML over other solutions
   1.3 KPI and metric(s) to track
   1.4 Data Collection

2. Play
   2.1 Git Repo
   2.2 Create Package
   2.3 EDA(1_EDA.ipynb)
   2.4 Model Selection/POC (2_model_selection.ipynb)
   2.5 Results
   2.6 Hyperparameter Tuning, Preprocessing, and Training(3_hyperparameter_tuning_preprocessing_training.ipynb)
   2.7 Decide on Model

3. Develop
4. Deploy

1. Discover

1.1 Understanding the problem

Porto Seguro wants to improve customer satisfaction by tailoring auto insurance prices for drivers.
We need to know which customers are likely to file a claim(bad drivers) and which are likely to not(good drivers). The final goal is to increase insurance prices for bad drivers and reduce the prices for good drivers.
This project aims to use machine learning to predict which customers are likely to file a claim.

1.2 Why ML over other solutions

- Adaptability
- Scalability
- Grasp Complex Patterns

1.3 KPI and metric(s) to track

- KPI: Increase customer satisfaction by 20% over the next year.
- Metric:
- F1: balance between reducing False Positives and False negatives as both are costly and robust to class imbalance
- AUC: Robust to Class Imbalance

1.4 Data Collection

The data was collected from Kaggle and uploaded to AWS S3

1.4.1 About Dataset

In the train and test data, features that belong to similar groupings are tagged as such in the feature names (e.g., ind, reg, car, calc). In addition, feature names include the postfix bin to indicate binary features and cat to indicate categorical features. Features without these designations are either continuous or ordinal. Values of -1 indicate that the feature was missing from the observation. The target columns signifies whether or not a claim was filed for that policyholder.

1.4.1.1 File descriptions

Train.csv: contains the training data, where each row corresponds to a policyholder, and the target column signifies that a claim was filed.
Test.csv: contains the test data.

2. Play (1_EDA.ipynb)

2.1 Git Repo

- Create a Repo in GitHub and open the project folder on the local machine
- Run git init in the terminal

- Create a master branch in GitHub and set it to default(only if the default branch in local and GitHub are different)
- Delete the main branch in GitHub
- Run these commands in Terminal: git remote add origin <git url>, git pull origin master
- Create dev branch: git branch dev => git checkout dev

2.2 Create Package

A package will be created and installed into Jupyter and VSCode for use. It contains reusable functions for tasks like visualization, preprocessing, training, deployment, etc.

Process:

- Create the folder (custom_package)
- Create a sub-folder(package)
- Create a setup.py file within custom_package and input values
- Create modules(py files) and functions for the package
- Run these commands: cd package, pip install build, python -m build, cd dist, pip install <package-file.whl> , cd ../..

2.3 EDA

2.3.1 Connect to AWS S3

1. Create IAM User in AWS:

    a. Open IAM service
    b. Click on users in the left panel > Create user
    c. Input user name(e.g. jupyter_access)
    d. Select policy (e.g. administrator access) > Create user
    e. Select the newly created user and click on create access key
    f. Select use case
    g. Download .csv file

2. Connect to S3 in Jupyter:
    a. Create s3 client object:
    ```
    s3_client = boto3.client('s3',aws_access_key_id=secret['Access key ID'],aws_secret_access_key=secret['Secret access key'])
    ```
    b. Input name of bucket and file:
    ```
    bucket = 'auto-insurance-data-x'
    file_name = "train.csv"
    ```
    c. Get object/file from s3:
    ```
    s3_clientobj = s3_client.get_object(Bucket=bucket, Key=file_name)
    ```

3. Create data_retrieval module in custom package:

a. Create data_retrieval.py module under package/data_retrieval sub-package
b. Within data_retrieval.py, create the function to retrieve data from an s3 bucket
c. Save and update the setup.py file to include the new sub-package

```
from setuptools import setup
setup(
    name = 'package',
    version = '0.1',
    description = 'Useful functions for training and deployment',
    author_email = 'oamenmodupe@gmail.com',
    packages = ['package.data_retrieval],
    install_requires = ['numpy', 'pandas', 'scikit-learn', 'matplotlib', 'mlflow']
```

d. Build the package: cd custom_package> python -m build > cd dist > pip install pip install package-0.1-py3-none-any.whl
e. Import new function in jupyter notebook for use

2.3.2 Replace -1 with np.nan

The dataset came with null values encoded as -1. These values will be replaced with np.nan so they can be handled properly.

2.3.3 Stratified Sampling

In stratified sampling, researchers divide subjects into subgroups called strata based on characteristics that they share. We start with splitting to avoid data snooping bias.
Here, we stratify the target variable and split it into four datasets:

a. Full Train: Full training dataset (reduced train + validation set). This set will be used to train the chosen model.
b. Reduced Train: This dataset will be used for EDA and to experiment and try out different algorithms and feature engineering techniques before deciding on the best-performing one.
c. Test: This will be used to test the performance of the main model
d. Validation: This will be used to evaluate models and for hyperparameter tuning

All datasets will be written to S3 using the write_to_s3 function created within the custom_package

2.3.4 Data Exploration

A random sample of 100,000 was drawn from the reduced train set for faster exploration.

2.3.4.1 Duplicates

The dataset contains no duplicates.

2.3.4.2 Whitespaces

If the column names contain whitespaces, the whitespaces need to be removed for easier feature retrieval. If the dataset has empty strings, it will cause the column type to be inferred wrongly by pandas

as an object type instead of the actual data type. The column names and dataset contain no whitespaces.

2.3.4.3 Missing Values

The dataset is 2% nulls, with columns containing 0-69% nulls.

2.3.4.4 Split Data Types

We split the dataset in types, categorical and numerical, and save the feature names to the config file. This is useful so we can easily explore the dataset according to the data types.

2.3.4.5 Univariate analysis

Our dataset consists mostly of discrete data, which makes it difficult to reduce the number of unique values (cardinality) for some features. High cardinality in discrete features can complicate analysis and model training but we notice a significant class imbalance in the target. We also notice some outliers in the boxplots.

2.3.4.6 Bivariate analysis

Not much to see here as the column names are uninformative. The Chi-Square Test of Independence was performed to see the relationship between the target and categorical variables. This test is used to determine if two categorical variables are independent or if they are in fact related to one another. If two categorical variables are independent, then the value of one variable does not change the probability distribution of the other. If two categorical variables are related, then the distribution of one depends on the level of the other. This test measures the differences in the observed conditional distribution of one variable across levels of the other and compares it to the marginal (overall) distribution of that variable.

2.3.4.7 Linear Separability

We use Seaborn's pairplot to plot variables against each other with the hue set to the target to see if a linear decision boundary can be drawn. This is important as it helps in confirming the assumptions of Linear Models. If there is no linearity, we transform the data or use non-linear models.

2.3.4.8 Stats

We do this especially to get an idea of the skew in our data. It also helps in knowing what type of imputation to use. We can use mean imputation in a case where the data is normal and median imputation when it is not. In this case, the data is skewed, so, we will employ median imputation.

2.3.4.9 Correlation

This is using a custom function in our package. We pass in the data, correlation type, figsize, and annot parameters. In this case, we check for linear/pearson and non-linear/spearman correlation.

2.3.4.10 Normality

We use two methods here to check for normality, The Shapiro-Wilk test and Q-Q plot. Shapiro-Wilk's null hypothesis is that the data is normal. We pass this test for each column and reject or accept the null hypothesis according to the p-value. The Q-Q plot shows quantile points against the theoretical normal line. If all or most values lie on this line, we can assume the data is normal or close to normal.

2.3.4.11 Outliers

We use two methods here, the IQR Proximity Rule and ZScore. For the IQR method, we consider values above the 75th percentile and 25th percentile, outliers.
For the ZScore, we consider values with a z-score outside the threshold (+3, -3), as outliers.

2.3.4.12 Write to Json

We write the config to json

2.4 Model Selection/POC (2_model_selection.ipynb)

2.4.1 Read Config

Open the config file created in 1_EDA so we can use the information stored there and save more.

2.4.2 Read Data

The stratified split data (full_train, red_train, test, validation set) is read from S3 using the s3_retrieval function in the custom package. We filter the data down to columns needed for training and convert assumed categorical columns to str dtype for easier encoding.

2.4.3 Dummy/Baseline Model

Here, we fit a dummy model to our data to get the baseline metrics. We consider any models that do better than the dummy model, 'good'. A Dummy Classifier is a simple, baseline classifier that doesn't try to learn any patterns from the data. It just makes predictions based on simple rules. For example, it might always predict the most frequent class or make random predictions.
Stratified Strategy: The classifier makes predictions by randomly guessing while maintaining the original class distribution.

2.4.4 Model Building

We create this function to load a list of tuples of model names and models, the parameter, class_weight, is used to address class imbalance by setting it to balanced, None, or manual weights.

2.4.5 Class Imbalance

We notice a severe class imbalance in our target variable, 0.963552(class 0) and 0.036448(class 1).

2.4.5.1 Dealing with imbalance with manual estimation class weights

Here, we try to find a good class weight by iterating through several combinations from 0 to 0.99 using LogisticRegression and GridSearchCV. We arrive at {0: 0.06964824120603015, 1: 0.9303517587939698} as the best class weight for this model.

2.4.6 Experiments

Experiment in this sense means we try out different feature engineering techniques, resampling, and feature selection models and evaluate the performance of the models. The end goal is to retrain the full_train set on the best-performing model, i.e. best feature engineering techniques for the data.

2.4.6.1 Experiment 0

Missing Values: We apply listwise deletion, i.e., we drop missing values and columns with more than 60% missing values. We notice a considerable loss of information

Encoding: One-Hot Encoding
Transformation: None
Outliers: None
Scaling: None
Resampling: None
Feature Selection: None

Results: We have very poor results with most models not even grasping the minority class which is to be expected since the data is heavily imbalanced and no resampling was applied.

2.4.6.2 Experiment 1

Missing Values: We apply simple median/mode imputation. This method is simple and preserves the class distribution but it may introduce bias.

Encoding: One-Hot Encoding. It is a technique we use to represent categorical variables as numerical values in a machine-learning model. It can help avoid the problem of ordinality but in can lead to increased dimensionality, sparsity, and overfitting.

Transformation: Log. Log transformation converts the column into its log. It is used in situations of non-linearity, non-normality, and situations where the data doesn't meet the assumptions of a linear model.

Outliers: Winsorize(0.95-0.05). In an attempt to reduce outliers, we create a function in our custom package to replace all values above the 95th percentile and under the 5th percentile with the values of the 95th percentile and 5th percentile respectively.

Resampling: Synthetic Minority Oversampling Technique, or SMOTE for short. It involves duplicating examples in the minority class, although these examples don't add any new information to the model. Instead, new examples can be synthesized from the existing examples. While it helps with balancing, it can lead to overfitting.

Feature Selection: Enet Coefs. Elastic Net is a regularized linear algorithm that is a mixture of Ridge and Lasso Regression. The mixture is controlled with the l1_ratio where 0 is Ridge and 1 is Lasso. Lasso tends to set the weights/coefficients of useless features down to zero and Ridge reduces the weights. The aim here is to fit Elastic Net to our data and select non-zero coefficients. We use ElasticNet instead of Lasso because Lasso tends to behave erratically when there is multicollinearity or when the number of features is not much larger than the number of instances.

Results: We notice better results compared to the previous experiment after applying smote, especially with the minority class. The models are most likely overfitting especially with LightGBM having a Precision of 100%.

2.4.6.3 Experiment 2

Missing Values: Percentile Imputation. This method replaces the null values with the 95th percentile value. It is useful when the data contains outliers but it requires the correct value of the percentile.

Encoding: Ordinal Encoding. It is a natural encoding for ordinal variables. For categorical variables, it imposes an ordinal relationship where no such relationship may exist but it is useful as it doesn't to high dimensionality

Transformation: Square Root. It converts the column into its square root. It is used in situations of non-linearity, non-normality, and situations where the data doesn’t meet the assumptions of a linear model.

Outliers: Winsorize(0.95-0.05). In an attempt to reduce outliers, we create a function in our custom package to replace all values above the 95th percentile and under the 5th percentile with the values of the 95th percentile and 5th percentile respectively.
Scaling: RobustScaler. It is a scaling method that uses quantiles so it’s good in situations where we have outliers. It removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range).

Resampling: ADASYN. Oversample using Adaptive Synthetic (ADASYN) algorithm.
This method is similar to SMOTE but it generates a different number of samples depending on an estimate of the local distribution of the class to be oversampled. It focuses on the minority instances that are difficult to classify correctly, rather than oversampling all minority instances uniformly. It assigns a different weight to each minority instance based on its level of difficulty in classification.

Feature Selection: SelectFromModel. Using the feature importance scores, It automatically determines which features are important by comparing the scores to the set threshold.

Results: We notice better results compared to experiment 0 after applying ADASYN, especially with the minority class. The models are most likely overfitting especially with LightGBM having a Precision of 100%

2.4.6.4 Experiment 3

Missing Values: Random Sample Imputation. This method involves replacing null values with random samples drawn from the data. It doesn't introduce synthetic bias but introduces variability.

Encoding: Count Encoding. It involves replacing categorical values with the number or frequency of occurring values. For example, if we have a feature with 5 As and 5 6s, it replaces the As with 5 and Bs with 6. The count_encoder function was created in the custom package for this purpose and it can be passed into a Sklearn pipeline by wrapping it in the FunctionTransformer class

Transformation: Polynomial. Polynomial Transformation helps to fit linear models to non-linear data. It does this by adding new features as the power of the features. It has the added advantage of finding relationships between features which is something a plain linear model can't do

Outliers: Winsorize(0.95-0.05). In an attempt to reduce outliers, we create a function in our custom package to replace all values above the 95th percentile and under the 5th percentile with the values of the 95th percentile and 5th percentile respectively.

Scaling: MaxAbsScaler. It is useful when the data contains -ve and +ve values as it preserves the sign. It is robust to outliers but may not be good for normal data. It does this by scaling the numerical expression to the range [-1, 1] by dividing with the maximum absolute value

Resampling: Class weight(balanced). It assigns class weights inversely proportional to the class frequencies. It gives good results when the imbalance is not extreme.

Feature Selection: F classification. The F-test tests null hypothesis that are all model parameters are zero. The alternative hypothesis is that at least one parameter is not zero.

Results: We notice a slight increase in performance compared to experiment 0 after applying balanced class weights, especially with the minority class. The models are still underfitting and producing very high False positives. We could try raising the threshold to reduce the False positives as raising the threshold increases the precision.

2.4.6.5 Experiment 4

Missing Values: Median/Mode

Encoding: One-Hot Encoding

Transformation: Log

Outliers: Winsorize(0.95-0.05). In an attempt to reduce outliers, we create a function in our custom package to replace all values above the 95th percentile and under the 5th percentile with the values of the 95th percentile and 5th percentile respectively.

Scaling: StandardScaler

Resampling: Class weight(manual). We use the manual weights in dict dtype extracted earlier

Feature Selection: Permutation feature importance is a model inspection technique that measures the contribution of each feature to a fitted model's statistical performance on a given tabular dataset. This technique is particularly useful for non-linear or opaque estimators, and involves randomly shuffling the values of a single feature and observing the resulting degradation of the model's score. By breaking the relationship between the feature and the target, we determine how much the model relies on such particular feature.

Results: We notice a slight increase in performance compared to experiment 0 after applying manual class weights, especially with the minority class. The models are still underfitting and producing very high False positives. We could try raising the threshold to reduce the False positives as raising the threshold increases the precision.

2.5 Results

After collating all experiments, we notice very poor results, some underfitting the minority class and some overfitting both classes(overfitting is especially noticed in data that was resampled with SMOTE and ADASYN. Nonetheless, we select the top 4 best-performing models(forest_1, forest_2, and lightgbm_1, xgb_1).

These top 4 models are most likely overfitting the training data. Some solutions to this are:

a. Use more training data: We are going to train the final model on the full set of training data not the reduced train set
b. Regularize the model: We are going to tweak the hyperparameters to attempt regularizing the models with Hyperparameter Tuning
c. Remove outliers/noise/redundant features: We already selected the assumed most important features using Elastic Net and RandomForest.

2.6 Hyperparameter Tuning, Preprocessing, and Training(3_hyperparameter_tuning_preprocessing_training.ipynb)

The aim of this process is to decide on the top 4 models. We are going to create search spaces and will tweak their hyperparameter in an attempt to regularize them and select the model that performs best on the validation set as the final model.

For Hyperparameter Tuning, we use HyperOpt. HyperOpt is an open-source Python package for serial and parallel optimization over awkward search spaces which may be of any data type. It comes with three algorithms:

a. Random Search: It selects random combinations of hyperparameters.
b. Tree of Parzen Estimators(TPE): This is a Bayesian optimization approach that models distributions of hyperparameters below and above a threshold for the objective function and then aims to draw more values from the good hyperparameter distribution. For example, if it notices that high values of max_depth or n_estimators lead to good scores, it will focus on getting more values like this and avoid low max_depth and n_estimators.
c. Advanced TPE

2.6.1 Read Config and Data

Here, we read in the config so we can access saved variables and our full_train, validation, and test set from s3 with the s3_retrieval function from our custom package.

2.6.1 Training Preprocessor

Our model_selection phase gave the 4 top models derived from different preprocessing steps. We need to recreate the same preprocessing pipeline, fit in on the full_train, and then, use it to transform the validation and test set later.

2.6.2 Preprocessor - Experiment 1

Repeating the preprocessing pipeline steps from Experiment 1, we fit it to our full train set and transform the validation set. Next, we apply smote to resample the train set and then use ElasticNet to select features with non-zero coefficients(as important features).

For this experiment, we selected three models, Random Forest, xgboost, and LightGBM.
The search space was created with overfitting in mind, to regularize, we are going to tweak the base models' hyperparameters and that of the ensemble. We can regularize the base models by reducing the max_* and increasing the min_* hyperparameters.

- n_estimators: The higher the value, the more the model is regularized.
- ccp_alpha: Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.
- Class_weight: Since our data is imbalanced

Next, we define the objective function. An instance of the model is created using the search_space params and random_state is set to 0 for reproducibility.
cross_val_score: This function performs cross-validation. It trains the model on different subsets of the data and evaluates it on the remaining parts. The cv argument specifies the number of folds (subsets) to use.
scoring: The evaluation metric used here is f1 which is common for classification problems with class imbalance.
error_score: If an error occurs during model training or scoring, it will be raised.

max_score: The highest f1 score among all folds.
loss: Since hyperopt minimizes the objective function, we define the loss as 1 - max_score. A lower loss corresponds to a higher f1 score.
loss: The computed loss.
params: The parameters used.
status: STATUS_OK indicates that the optimization ran successfully.

Next, we perform the optimization.
The fmin function optimizes the hyperparameters to minimize the loss. It uses the TPE algorithm to suggest new sets of hyperparameters.
fn: The objective function to be minimized. partial is used to fix the n_folds, x, and y arguments, so only params need to be optimized.
space: The search space for the hyperparameters. This is defined separately (similar to the search space defined earlier).
algo: The algorithm to use for optimization, tpe.suggest uses the Tree-structured Parzen Estimator (TPE) algorithm.
max_evals: The maximum number of evaluations (trials) to perform.
trials: An instance of hyperopt. Trials are used to store details of each evaluation

2.6.3 Preprocessor - Experiment 2

Repeating the preprocessing pipeline steps from Experiment 2, we fit it to our full train set and transform the validation set. Next, we apply ADASYN to resample the train set and then use SelectFromModel(RandomForest) to select features with a threshold importance greater than 0.0001

For this experiment, we selected one model, Random Forest.
The search space was created with overfitting in mind, to regularize, we are going to tweak the base models' hyperparameters and that of the ensemble. We can regularize the base models by reducing the max_* and increasing the min_* hyperparameters.

- n_estimators: The higher the value, the more the model is regularized.
- ccp_alpha: Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.
- Class_weight: Since our data is imbalanced

Next, we define the objective function. An instance of RandomForestClassifier is created using the search_space params and random_state is set to 0 for reproducibility.
cross_val_score: This function performs cross-validation. It trains the model on different subsets of the data and evaluates it on the remaining parts. The cv argument specifies the number of folds (subsets) to use.
scoring: The evaluation metric used here is roc_auc (Area Under the Receiver Operating Characteristic Curve), which is common for binary classification problems.
error_score: If an error occurs during model training or scoring, it will be raised.
max_score: The highest ROC AUC score among all folds.

loss: Since hyperopt minimizes the objective function, we define the loss as 1 - max_score. A lower loss corresponds to a higher ROC AUC score.
loss: The computed loss.
params: The parameters used.
status: STATUS_OK indicates that the optimization ran successfully.

Next, we perform the optimization.
The fmin function optimizes the hyperparameters to minimize the loss. It uses the TPE algorithm to suggest new sets of hyperparameters.
fn: The objective function to be minimized. partial is used to fix the n_folds, x, and y arguments, so only params need to be optimized.
space: The search space for the hyperparameters. This is defined separately (similar to the search space defined earlier).
algo: The algorithm to use for optimization, tpe.suggest uses the Tree-structured Parzen Estimator (TPE) algorithm.
max_evals: The maximum number of evaluations (trials) to perform.
trials: An instance of hyperopt. Trials are used to store details of each evaluation

2.6.1 Decide on Model

After collating the validation scores, we decide on Xgboost and VotingClassifier Ensemble of random forest, xgboost, lightgbm. This phase is quite iterative as we notice severe overfitting on the minority class. We tried several things:

- BorderlineSmote
- Random Undersampling
- Adasyn
- Adaboost with Smote
- Setting the threshold/Increasing it to increase precision

We left just the best 2 methods in the notebook,

a. This run uses Random Undersmapling with XGBoost: We notice a significant increase in the recall of the minority class and little to no improvement in the precision.
b. This run uses a Voting ensemble with BorderlineSMOTE: We notice little to no increase on both precision and recall

2.7 Decide on Model

We decided to proceed with run 1 even though the precision was bad. This project mostly focused on MLOPs and less on prediction accuracy.

3. Develop

Here, we concentrate on replicating the model training steps in jupyter for production.

3.1 Data_retrieval.py

This module is responsible for retrieving data from an S3 bucket. It handles the loading of environment variables, setting up the S3 client, reading configuration details, and retrieving specified datasets.

Functions:
load_data():
Description: This function retrieves data from the S3 bucket based on the configuration provided. It returns the full training dataset, validation dataset, and test dataset.
Returns:
full_train_df: The full training dataset retrieved from S3.
validation_df: The validation dataset retrieved from S3.
test_df: The test dataset retrieved from S3.

Workflow:
Environment Setup: The module checks the environment variable ENV. If it is set to local, it loads local environment variables using dotenv and retrieves AWS access credentials from environment variables.

S3 Client Initialization: Creates an S3 client object using the AWS credentials.

Configuration Loading: Opens and reads the configuration file (config.json) to get the bucket name and the paths for the datasets.

Data Retrieval: The load_data() function uses the s3_retrieval function from package.data_retrieval.data_retrieval to fetch the specified datasets from the S3 bucket.

3.2 Preprocessor.py

This module handles the training of a data preprocessing pipeline and its registration with MLflow. It includes loading data, defining preprocessing steps, and managing the MLflow experiment lifecycle.

Functions and Classes:

a. split_and_load(full_train_df=None, validation_df=None, test_df=None):
   Description: Splits the input datasets into dependent and independent variables based on the configuration
   Parameters:

   full_train_df: DataFrame, optional.
   validation_df: DataFrame, optional.
   test_df: DataFrame, optional.
   Returns:
   xtrain: Independent variables for training.
   ytrain: Dependent variables for training.
   xval: Independent variables for validation.

yval: Dependent variables for validation.
xtest: Independent variables for testing.
ytest: Dependent variables for testing.

b. load_pipeline():
Description: Creates the preprocessing pipeline based on the configuration.
Returns:
main_pipeline: The constructed preprocessing pipeline.

c. PreprocessorWrapper(mlflow.pyfunc.PythonModel):
Description: A custom wrapper to log the preprocessor model with MLflow.
Methods:
__init__(self, preprocessor): Initializes the wrapper with the given preprocessor.
predict(self, context, model_input): Applies the preprocessing transformation to the input data.
main():
Description: The main function that orchestrates the preprocessing pipeline creation, training, logging, and registration with MLflow.

Steps:

- Loads and splits the data.
- Creates and fits the preprocessing pipeline.
- Logs the preprocessor model to MLflow.
- Registers and transitions the model version in MLflow.
- Updates the configuration file with the new model version details.

Workflow:
Configuration Loading:
Reads the configuration file (config.json) to get settings for data columns, preprocessing, and MLflow experiment details.

Data Splitting: Uses the split_and_load function to load and split the data into training, validation, and test sets.
Pipeline Creation: Constructs the preprocessing pipeline using load_pipeline, which includes transformations like square root, imputation, winsorization, scaling, and encoding.

MLflow Experiment Management:

- Creates or sets the MLflow experiment.
- Logs the preprocessor model using the custom PreprocessorWrapper.
- Registers and transitions the preprocessor model version in MLflow to production.
- Updates and saves the configuration file with new version details.

Execution Control: Ensures that the main() function runs only when the script is executed directly.

3.3 Hyperparameter_tuning.py

This module provides functions for running hyperparameter tuning for machine learning models, specifically using XGBoost, and integrates with MLflow to manage experiments.

Functions:

a. objective(params, n_folds, x, y):
   Description: This function defines the objective for hyperparameter tuning using cross-validation.
   Parameters:
   params: Dictionary of hyperparameters to evaluate.
   n_folds: Number of cross-validation folds.
   x: Training data features.
   y: Training data labels.
   Returns:
   A dictionary containing:
   loss: The loss to be minimized (1 - max F1 score).
   params: The evaluated hyperparameters.
   status: Status of the optimization (STATUS_OK).

b. run_hyperparameter_tuning(xtrain_exp1, ytrain_exp1):
   Description: Runs the hyperparameter tuning process for the XGBoost model using the TPE algorithm.
   Parameters:
   xtrain_exp1: Training data features.
   ytrain_exp1: Training data labels.
   Returns:
   best_loss: The best loss value achieved.
   best_params: The best hyperparameters found during the tuning.

Workflow:

- Configuration and Imports: Sets the configuration for the output of pandas and suppresses warnings. Imports necessary modules and functions from sklearn, hyperopt, and xgboost.
- Hyperparameter Search Space: Defines the search space for the XGBoost hyperparameters using hyperopt.
- Objective Function: The objective function is used to evaluate the model's performance using cross-validation. It converts certain hyperparameters to integers and calculates the loss as 1 - max F1 score.
- Hyperparameter Tuning Function: The run_hyperparameter_tuning function optimizes the hyperparameters using the TPE algorithm with hyperopt. It configures and runs the optimization process, ensuring integer hyperparameters are correctly set.

3.4 mlflow_utils.py

This module contains utility functions to facilitate working with MLflow, particularly for setting or creating experiments.

Functions:

a. set_or_create_mlflow_experiment(experiment_name, artifact_location, tags={'env':'dev','version':'1.0.0'}):
   Description: Creates a new MLflow experiment if it does not exist, or sets an existing experiment as the active experiment. It returns the experiment ID.
   Parameters:
   experiment_name (str): The name of the MLflow experiment.
   artifact_location (str): The location to store artifacts for the experiment.
   tags (dict, optional): A dictionary of tags to associate with the experiment (default is {'env':'dev','version':'1.0.0'}).
   Returns:
   exp_id (str): The ID of the MLflow experiment.

Workflow:

- Creating or Setting an Experiment: The function first tries to create a new experiment with the given name, artifact location, and tags. If the experiment does not exist, it creates a new one and prints "Created Experiment".
- If the experiment already exists, it catches the exception, retrieves the existing experiment ID, and prints "Extracted experiment".
- Finally, it sets the retrieved or newly created experiment as the active experiment using mlflow.set_experiment.

Returning Experiment ID: The function returns the experiment ID, which can be used to track or log runs in the specified MLflow experiment.

3.5 training_model.py

The main event. This module handles the training of a machine learning model, hyperparameter tuning, feature selection, evaluation, and registration of the trained model with MLflow.

Functions and Main Workflow:

a. split_and_load() (from preprocessor):Description: Splits the data into training, validation, and test sets.
b. run_hyperparameter_tuning(xtrain_exp1, ytrain_exp1) (from hyperparameter_tuning):
   Description: Runs hyperparameter tuning for the XGBoost model.
c. evaluate_sets(y_true, y_pred, y_proba, data_type, model_type) (from model_training):
   Description: Evaluates the performance of the model and returns metrics.
d. set_or_create_mlflow_experiment(experiment_name, artifact_location, tags) (from mlflow_utils):
e. Description: Creates or sets the MLflow experiment and returns the experiment ID.

Workflow:

- Configuration and Imports: Sets the configuration for pandas output and suppresses warnings.
- Imports necessary modules and functions from sklearn, xgboost, mlflow, and other packages.
- Loading Configuration: Loads the configuration from config.json and sets up experiment details.
- MLflow Experiment Setup: Uses set_or_create_mlflow_experiment to create or set the MLflow experiment.
- Data Loading and Preprocessing: Loads and splits the data using split_and_load.
- Loads the preprocessor model from MLflow and preprocesses the training, validation, and test sets.
- Data Resampling: Applies undersampling to the training data to address class imbalance.
- Feature Selection: Uses a RandomForestClassifier to select important features and updates the configuration with selected columns.
- Hyperparameter Tuning: Optionally runs hyperparameter tuning to find the best parameters for the XGBoost model.
- Model Training: Initializes the XGBoost model with the best hyperparameters.
- Enables automatic logging with mlflow.autolog().
- Trains the model on the resampled training data.
- Model Evaluation: Evaluates the trained model on the training and test sets using evaluate_sets.
- Logs the evaluation metrics to MLflow.
- Model Logging and Registration:nLogs the trained model to MLflow and registers it as a new version.
- Transitions the new model version to production and the previous version to staging.
- Saving Updated Configuration: Updates the configuration with model details and saves it to config.json.
- Execution Control: Ensures that the main workflow runs only when the script is executed directly.

3.6 monitoring.py

This module sets up monitoring for the machine learning model using EvidentlyAI to check for data drift and target drift.

Workflow:

- Configuration and Imports: Sets the configuration for pandas output and suppresses warnings.
- Imports necessary modules and functions from sklearn, evidently, and pandas.
- Loading Configuration: Loads the configuration from config.json.
- Data Loading: Loads the full training and test datasets using load_data.
- Creating Out-of-Time Sample: Since there is no out-of-time (OOT) sample, it takes a random subset from the test data for drift analysis.
- Setting up Monitoring Configuration: Defines the path for saving the drift monitoring dashboard in the configuration.

Creating and Configuring EvidentlyAI Dashboard:

- Creates instances of DataDriftTab and CatTargetDriftTab from EvidentlyAI.
- Initializes the EvidentlyAI Dashboard with the data drift and categorical target drift tabs
- Calculates the drift metrics using the training data and the OOT sample.
- Saves the drift monitoring dashboard to the specified path.

Key Variables:

- full_train_df: The full training dataset.
- test_df: The test dataset.
- oot: Out-of-time sample created from the test dataset.
- monitor_dashboard_path: Path to save the EvidentlyAI drift monitoring dashboard

EvidentlyAI Components:

- DataDriftTab: Tab to monitor data drift between the training data and the OOT sample.
- CatTargetDriftTab: Tab to monitor target drift for categorical targets.

This module will generate a drift monitoring dashboard and save it to the path specified in the configuration file.

4. Deploy

Here, we expose our model as a REST API using Flask. In the Flask app, we have a home page, predict page and a monitor page. The home page renders an HTML to introduce the user to the app, the predict page is where the user interacts with the model and the monitor page renders the monitoring dashboard.
To deploy this, we create a docker container.

4.1 app.py

This module creates a Flask web application that allows users to interact with the trained machine learning model. Users can upload data for predictions and view monitoring dashboards.

Key Components:
Configuration and Imports: Imports necessary modules from flask, pandas, and mlflow.
Loads the configuration from config.json.

Flask App Setup:
Initializes the Flask app and loads the model and preprocessor using mlflow.
Routes:

a. / (Home):
   Description: Renders the home page.
   Template: home.html.
b. /predict:
   Methods: GET, POST.

Description: Handles file uploads for prediction.

Workflow:

- Reads the uploaded CSV file.
- Preprocesses the data using the loaded preprocessor model.
- Selects the relevant columns.
- Generates predictions using the loaded model.
- Adds the predictions to the original data and saves the output as output_data.csv.
- Returns the output CSV file as a downloadable attachment.

c. /monitor:

Description: Renders the monitoring dashboard for data and target drift.

Template: data_and_target_drift_dashboard.html.

Main Execution: Runs the Flask app in debug mode if the script is executed directly.

Resources

1. https://stackoverflow.com/questions/58402319/connect-jupyter-notebook-locally-to-aws-s3-without-sagemaker
2. https://stackoverflow.com/questions/54855479/s3-object-throws-typeerror-sequence-item-0-expected-str-instance-tuple-foun
3. http://localhost:8888/notebooks/OneDrive/Documents/DATA%20PROJECTS/Project_Determining_Risk_tolerance_category.ipynb
4. https://sites.utexas.edu/sos/guided/inferential/categorical/chi2/#:~:text=If%20two%20categorical%20variables%20are%20independent%2C%20then%20the%20value%20of,on%20the%20level%20the%20other.
5. https://www.linkedin.com/advice/3/how-can-you-remove-outliers-from-dataset-using-wyx2e#:~:text=To%20spot%20outliers%20with%20the,data%20in%20a%20normal%20distribution.
6. https://www.geeksforgeeks.org/reading-and-writing-json-to-a-file-in-python/
7. https://medium.com/analytics-vidhya/appropriate-ways-to-treat-missing-values-f82f00edd9be
8. https://www.learningtree.com/blog/interpret-q-q-plot/
9. https://www.aimspress.com/aimspress-data/dsfe/2023/4/PDF/DSFE-03-04-021.pdf
10. https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/
11. https://www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/
12. https://www.analyticsvidhya.com/blog/2021/08/complete-guide-on-how-to-use-lightgbm-in-python/
13. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.PrecisionRecallDisplay.html
14. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html
15. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html
16. https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html
17. https://scikit-learn.org/stable/modules/permutation_importance.html#:~:text=Permutation%20feature%20importance%20is%20a,on%20a%20given%20tabular%20dataset.
18. https://stackoverflow.com/questions/73361560/getting-roc-auc-score-nan-for-cross-validation-of-multiclass-classification
19. https://stackoverflow.com/questions/72097425/roc-auc-percentage-calculation-giving-nan
20. https://stackoverflow.com/questions/44036193/using-cross-validation-and-auc-roc-for-a-logistic-regression-model-in-sklearn
21. https://www.geeksforgeeks.org/ml-one-hot-encoding/
22. https://medium.com/@penpencil.blr/data-imbalance-how-is-adasyn-different-from-smote-f4eba54867ab#:~:text=ADASYN%20(Adaptive%20Synthetic%20Sampling)%3A&text=It%20focuses%20on%20the%20minority,level%20of%20difficulty%20in%20classification.
23. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html
24. https://www.geeksforgeeks.org/feature-selection-using-selectfrommodel-and-lassocv-in-scikit-learn/

25. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
26. https://hyperopt.github.io/hyperopt/getting-started/search_spaces/
27. https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlmodel-configuration
28. https://stackoverflow.com/questions/60530176/mlflow-how-to-save-a-sklearn-pipeline-with-custom-transformer
29. https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions
30. https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.BorderlineSMOTE.html
31. https://stackoverflow.com/questions/56597812/nested-runs-using-mlflowclient
32. https://stackoverflow.com/questions/65381378/looking-for-ideas-to-lower-the-false-positive-rate-in-machine-learning-classific
33. https://github.com/langchain-ai/langchain/issues/2222
34. https://pylint.readthedocs.io/en/latest/user_guide/messages/convention/missing-module-docstring.html
35. https://www.analyticsvidhya.com/blog/2024/03/complete-guide-to-effortless-ml-monitoring-with-evidently-ai/
36.