Report - Group J

Ana Teresa Maia, 20201562 | Diogo Barros, 20230555 | Henrique Vences, 20201519

Mariana Oliveira, 20230429 | Rodrigo Fazendeiro, 20230756

1. Introduction

In today's competitive market, organizations must optimize their recruitment and retention strategies to attract and retain top talent, especially in fields like big data and data science. To address this, our machine learning operations project aims to **predict job change intentions among data scientists** who have completed training programs at a specific company. This project uses various concepts and techniques from our Machine Learning Operations classes to build predictive models, providing valuable insights into the factors influencing job changes.

The company offers specialized data science training courses that attract many applicants. Knowing which candidates are likely to seek new job opportunities post-training is crucial for optimizing training efforts, reducing costs, and improving resource allocation. Accurate job changes predictions help the company focus on retaining candidates more likely to stay, enhancing overall recruitment and training efficiency.

1.1 Success Criteria:

To assess the success of the organization in implementing new strategies, several common industry metrics can be identified and used as measures. These KPIs are used to create a comparative analysis, allowing for the benchmarking of current performance results against those achieved following the implementation of the project.

- **Model Accuracy:** It is important to ensure high accuracy in distinguishing between candidates who are likely to change jobs and those who are not.
- **F1 Score:** It is useful to have a single metric that **balances both precision and recall**, it considers the unbalancing level of the data.

2. Methodology

2.1 Data Understanding and Exploration

The dataset used for this project is "HR Analytics: Job Change of Data Scientists", obtained from Kaggle. It comprises **19158 rows and 14 features** related to demographics, education, experience, and more, aiming to predict whether a candidate will look for a new job or stay with the company after completing a training program.

The dataset comprises a mix of numerical and categorical features. The binary variable 'target', indicating whether the candidate is looking to change jobs, serves as the dependent variable in this study.

To provide a more in-depth analysis of the dataset, **some visualizations were created using Power BI** (Figure 1 - City specific analysis, Figure 2 - Workers profile analysis), which offer valuable insights regarding data distribution and patterns. The dataset reveals several insights about the candidates. Most candidates are **male (69%)**, followed by **female (6%)** and **other/unknown** categories. Most of the candidates are **not enrolled in any university course**, with **smaller proportions enrolled in full-time or part-time courses**. The dataset predominantly consists of **graduates**, with a significant number holding a master's degree, while other educational levels such as PhD, high school, and primary school are less represented. Most individuals have **relevant experience, with many having over ten years of experience**. A significant number of candidates come from **'city_103'**. The majority work in **private limited companies**, followed by the public sector and startup companies. There is a considerable number of candidates who have **never worked before**, as indicated by the substantial number of null values in the 'company_size' feature. Approximately **25%** of individuals are looking for a **job change**.

During the exploratory data analysis phase, a thorough investigation was conducted to identify potential data gaps and ensure data quality and integrity. No duplicates were identified, confirming that each row represents a unique individual. Outliers were examined to ensure they did not significantly distort the analysis. An inconsistency was found in the 'company_size' data, where '10/49' was corrected to '10-49' to maintain integrity across values. Missing values were identified in the following features: 'gender', 'enrolled_university', 'education_level', 'major_discipline', 'experience', 'company_size', 'company_type', and 'last_new_job'.

2.2 Ingestion

This section outlines the process of ingesting data into our environment, ensuring it is ready for subsequent analysis and modeling. The ingestion pipeline is initiated by setting up *Kedro* and loading the necessary configuration and credentials, thereby ensuring a structured and modular approach. The main steps involved in the ingestion process are as follows: loading the data, validating it using Great Expectations, where we define **expectation suites for numerical and categorical features, and storing it in a feature store repository.** Great Expectations is utilized for validating the ingested data. By defining expectation suites for numerical and categorical features, we ensure data quality and integrity before storing it in the feature store. This step involves checking for data anomalies, inconsistencies, and missing values to prevent data quality issues from propagating through the pipeline.

3. Data Unit Tests

Unit tests are a fundamental practice to ensure data quality and adherence to predefined criteria, establishing confidence in the dataset's integrity for subsequent machine learning modeling and analysis tasks. Key steps include: **Validation setup,** where expectation suites are established to verify various aspects of the dataset, such as categorical values (the expected values for each feature), numerical ranges (for example the city development index is expected to be between 0 and 1, and column formats; **Execution**, where various expectations are defined to validate different columns in the dataset. These expectations check if the column values fall within specified sets. Using Great Expectations, the dataset undergoes validation against the defined criteria; **Results Compilation,** where validation results are collected and summarized to provide insights into any discrepancies found during testing, such as unexpected counts and values; **Assertions**, where additional assertions are implemented to validate critical data properties directly within the code; **Logging,** where logging mechanisms are employed to track the outcome of unit tests, confirming whether the dataset meets the defined expectations.

4. Data Preparation

This section outlines the **initial preprocessing steps**, which include cleaning, transforming, and feature engineering the data. These preprocessing steps are essential to prepare the data for reliable analysis and effective modeling, setting a solid foundation for subsequent stages in the machine learning pipeline.

4.1 Data Preprocessing

The preprocessing steps included setting ‘enrolle_id’ column as the index for easier data manipulation. Outliers were removed, specifically where 'city_development_index' was smaller than 0.4. Missing values that had a logic behind existing were appropriately filled in, such as: ’gender’ is filled with "Unknown" because there could be people that didn’t identify as the provided options, ‘experience’ with 0 as some people could have never worked before, and ‘company_size’ and ‘company_type’ with "Not Applicable" for the same reason. Additionally, typographical errors in the data, such as the one found in the ‘company_size’ where ‘10/49’ was changed to ‘10-49’, were corrected to ensure clarity and consistency.

4.2 Transformation and Feature Engineering

Some columns were categorized into bins to simplify the analysis. The 'experience' column was categorized into the following intervals: '0-5', '6-10', '11-15', '16-20', and '>20'. The ‘city_development_index’ was divided into '<0.4', '0.4-0.5', '0.5-0.6', '0.6-0.7', '0.7-0.8', '0.8-0.85', '0.85-0.9', '0.9-0.95', and '0.95-1.0'. The ‘training_hours’ were categorized into '0-20', '20-40', '40-60', '60-80', '80-100', '100-150', '150-200', '200-250', '250-300', '300-350', and '>350'. After creating these new features, the original columns used for binning were dropped. Additionally, lists of numerical and categorical features were created to be used in further analysis and modeling.

5. Additional Data Unit Tests

Following the initial preprocessing steps, additional data unit tests are crucial to validate the newly created bins and ensure data consistency, when taking into consideration missing values and outliers’ treatment. These tests provide an extra layer of assurance that the data transformations have been executed correctly and that the resulting dataset adheres to expected standards. The key steps are the same as those previously outlined, however now we consider the transformations applied at the initial preprocessing stage.

6. Train Validation Split and Additional Preprocessing

This section details the essential steps of splitting data into training and validation sets and the subsequent preprocessing applied to each split. These steps are crucial for preparing the data for machine learning model training and evaluation.

6.1 Train-Validation Split:

Before splitting the dataset, we removed the target variable, 'target,' to ensure an accurate division. In a specific processing step, the dataset is then divided into training and validation sets, with the test set comprising 20% of the entire dataset. This split is performed in a way that maintains the distribution of the target variable across both sets, which is crucial for reliable model evaluation and performance assessment.

6.2 Additional Preprocessing (Train/Validation):

To prevent data leakage, additional preprocessing steps were performed separately on the training and validation datasets after the initial split.
Categorical Encoding: Categorical features were encoded using target encoding to transform categorical values into numerical representations. This ensures that categorical variables are appropriately represented for the algorithms. The parameter handle_missing='return_nan' was included to ensure that missing values do not interfere with the target encoding process. The target encoding was trained on the training dataset and then applied to both the training and validation datasets.
Scaling: At this stage, all features are numerical. These features were scaled using Min-Max scaling to normalize their values within the range of 0 to 1. This scaling technique prevents features with larger numerical ranges from dominating the model training process, ensuring balanced model training. The Min-Max scaling parameters were learned from the training dataset and subsequently applied to both the training and validation datasets.
Missing Value Imputation: Missing values in selected columns 'enrolled_university', 'education_level', 'major_discipline', 'last_new_job' were imputed using the K-Nearest Neighbors (KNN) imputation method. This approach replaces missing values based on the values of neighbouring data points, ensuring imputed values are realistic and reflective of the dataset's structure. Given the context of these variables, we determined that KNN imputation was the most suitable method for treating their missing values as the existence of them didn’t make much sense since the possible values for those columns covered all the reasonable possible real-world choices. The KNN imputer was trained on the training dataset and then applied to both the training and validation datasets.

7. Feature Selection

Feature selection is pivotal in the machine learning pipeline for identifying the most relevant features that contribute to model performance while reducing complexity and computational cost.

We have created a comprehensive node function **that integrates multiple selection methods**, such as: variance analysis, Spearman correlation analysis with a 0.7 threshold, recursive feature elimination analysis applied with logistic regression, lasso regression analysis and decision tree analysis with a 0.035 importance threshold for feature importance. Each method is implemented as a function that identifies features to exclude. The features marked for elimination are stored in lists, and any feature **appearing in at least two different methods' lists is selected for removal.** This analysis is performed on the training dataset 'X_train', and the results are then applied to the validation dataset 'X_val'. At the end, the removed features were 'gender', 'relevent_experience', 'major_discipline' and 'city_development_index_bin'.

8. Preprocess Batch (Transformations on Test Set)

The test set was subjected to preprocessing steps to ensure alignment with the transformations applied to the training data, which is crucial for accurate model evaluation and deployment.

Data Cleaning: The 'enrollee_id' column was designated as the index to ensure a unique identifier for each entry. Duplicates based on 'enrollee_id' were removed to prevent redundant information. Outliers were removed based on 'city_development_index' (values below 0.4). Any missing values in categorical columns, such as 'gender', 'experience', 'company_size', and 'company_type', were imputed with 'Unknown', 0, 'Not Applicable', and 'Not Applicable', respectively.

Feature Engineering: The process of binning continuous features was carried out in the same manner as previously used for the training set. The original columns used for binning were removed to avoid any potential issues with redundancy.

Additional Preprocessing: Categorical features were transformed into numerical representations using the 'TargetEncoder' that was fitted on the training data to ensure consistent encoding between the training and test sets. All numerical features were scaled using the 'MinMaxScaler' trained on the training data, which normalizes the feature values to a range between 0 and 1. The KNN Imputer was used to impute missing values in 'enrolled_university', 'education_level', 'major_discipline', 'last_new_job' columns. As in the training set, the 'gender', 'relevant_experience', 'major_discipline', and 'city_development_index_bin' columns have been excluded to reduce dimensionality and focus on the most relevant features.

9. Model Evaluation

This section describes the process of training and selecting the best-performing model for the project, incorporating both initial model comparison and hyperparameter tuning.

Model Training: **Logistic Regression** was chosen as the baseline model (Logistic Regression) for its simplicity and effectiveness in binary classification tasks. It was trained using the preprocessed training dataset. During training, predictions were generated for both the training and validation sets to evaluate model performance.

Model Interpretability with SHAP: SHAP (SHapley Additive exPlanations) values were computed to interpret the Logistic Regression model's predictions. This technique provides insights into feature importance, aiding in understanding the model's decision-making process. The **SHAP summary plot** *Figure 4* **illustrates the impact of each feature on the model's predictions, highlighting key factors influencing the outcome.**

Initial Model Comparison: After training the Logistic Regression model, an initial comparison was performed with **Gradient Boosting Classifier** as a challenger model. Each model's performance was assessed based on its **accuracy** score on the validation set. This comparison helps in identifying which model performs better initially, establishing a benchmark for further optimization. The best-performing model is identified as the initial champion model.

Hyperparameter Tuning: The champion model from the initial comparison undergoes hyperparameter tuning using **RandomizedSearchCV**, a technique that explores various combinations of hyperparameters to find the optimal configuration that maximizes model performance, evaluated using the **F1 Score** during cross-validation. The tuned model with **the highest F1 score on the validation set is selected as the final model.**

Champion Model Selection: After hyperparameter tuning, the model's performance was reassessed on the test set using accuracy as the metric. If the refined model outperformed the current champion model (previously selected best model), it would replace it, which was the case, Gradient Boosting Classifier was defined as the new champion model.

Logging and Experimentation: Throughout the training and selection process, **MLflow is employed for tracking and logging experiments.** Model performance metrics, hyperparameters, and SHAP values were logged to facilitate reproducibility and comparison across different iterations. This logging process validates model selection decisions and ensures transparency and traceability in model development.

This iterative approach of comparing models and fine-tuning parameters guaranteed that the final model used for prediction was optimized based on the data's characteristics, ensuring robustness and reliability in model deployment.

10. Model Predict

The final predictions were made with the application of the champion model previously stated. The **'model_predict' node** returned the batch data frame with the final predictions as well as some descriptive statistics regarding it.

Looking at the **final descriptive values** it is possible to understand that most of the values predicted were '0' (1351 over 2129, about 63%), meaning that we are predicting that most of the individuals do not have the intention of changing the current work.

11. Data Drift Tests

Data Drift, or data change detection, is a critical aspect of data analytics that identifies changes in data characteristics over time. These changes can arise from shifts in the business environment, user behavior, system updates, or seasonal effects. Detecting data drift is essential for maintaining the accuracy and reliability of model performance.

Initially, **drift was detected in the feature 'city'** which makes sense since it has a lot of u**nique different values** and the representation across datasets is **normally different**. However, to validate our detection mechanisms, we artificially created a drift scenario. Specifically, we added the value **'family business' to the 'company_type' column and values of 30, 40 and 50 to the 'experience'** column in the current dataset. This ensured our tools and processes effectively identified data drift.

Using *NannyML*, a specialized library for data drift detection, we conducted **univariate and multivariate tests**. A constant threshold of 0.2 was set for the categorical drift test, utilizing the **Jensen-Shannon method**, which measures distribution differences for categorical features. We analyzed features including 'city',' gender', 'relevant experience', 'enrollment in university', 'education level', 'major discipline', 'experience', 'company size', 'company type' and 'last new job'. Data drift calculations were performed in chunks of 50 records.

We initialized the **Univariate Drift Calculator** from *NannyML*, specifying the categorical features and methods to be used. The calculator was fitted on the **reference dataset** (historical data) and then applied to the analysis **dataset** (new data). Results were filtered for the analysis period and specific categorical features, and a drift plot was generated and saved as an HTML file for visualization. In the *Figure 5* there's an example of the output of this method offering a comprehensive analysis of the feature 'city'.

For numerical features, we used *Evidently AI*. **Evidently AI's Data Drift Report** was configured to use the **Kolmogorov-Smirnov (KS)** test for numerical features and the **Population Stability Index (PSI)** for categorical features, with a statistical test threshold set at 0.05. A report was saved as an HTML file, providing a comprehensive view of any detected data drift, as we can see in the Figure 6.

The advantages of the mentioned technologies include their ability **to detect and analyze data drift** with precision, enhancing the overall reliability of data models. However, there are risks involved, such as the additional overhead and complexity introduced by these tools. They might also have limitations in scalability if not configured properly. To mitigate these risks, it is essential to regularly review and optimize the configuration settings for these tools. If performance issues arise, we should consider scaling the infrastructure or exploring alternative tools with better performance at scale.

12. Pytest Validation

Pytest validation is crucial for ensuring the functionality and integrity of functions within operational pipelines. By validating key stages of our project pipeline, we guarantee accurate predictions and facilitate strategic decision-making to optimize recruitment, retention, and training efforts efficiently.

- **Initial Data Preprocessing Validation:** Tests in the 'clean_data' function ensuring correct data type and absence of null values post-processing. Additionally, 'feature_engineering' function is validated by checking for expected columns in the final transformed dataset.
- **Data Splitting:** Tests in the 'split_data' function, confirming proper dataset splitting and integrity of train-validation sets based on predefined parameters.
- **Model Selection:** focuses on validating the 'model_selection' function. Tests ensure the selection of an appropriate champion model by checking its type and performance metrics against a production model.

Unfortunately, our implementations didn't work due to an error related to a Kedro dependency that couldn't be resolved in time.

13. Considerations for Model Improvement

There is the possibility of improving the model's performance results. For this purpose, our team discussed which changes or implementations could be valuable to achieve higher performance.

Firstly, having **additional data** regarding the phenomenon under study would be valuable in the sense that new variables could have higher relevance for model prediction results.

Secondly, it should be noticed that after all our team experimented some **resampling techniques** ('SmoteTomek'), without any success as the results obtained in the modeling stage led to higher values of overfitting (*Figure 6*). For this reason, we ended up not using resampling techniques. However, there is the possibility to conduct research and evaluate other resampling possibilities that could be more adequate to our approach.

Exploring advanced techniques like Generative Adversarial Networks (GANs) for data augmentation might provide more effective ways to handle class imbalance and improve model robustness.

Moreover, for a more professional implementation of a MLOps job, we could develop a **Kubernetes structure** for real-time execution. This would involve **containerizing the Kedro pipeline using Docker**. Integrating the full functionalities of Great Expectations would automate the validation process for new data through data contexts and checkpoints. Additionally, to scale up for big data scenarios, adapting the project structure to operate with PySpark, for example.

14. Conclusions

The project's overall results met our expectations. It underscored the **importance of using 'Kedro' and 'Git'** for organizing pipelines into reproducible and easily maintainable modular code, as well as facilitating collaboration with team members. **Integration with MLflow proved invaluable** for tracking pipeline execution, monitoring grid search experiments during hyperparameter tuning, and comparing model performances and scores. **The effort dedicated to data unit tests is also a key point**, as it is extremely important to ensure the quality and validation of the data we are using for a proper resolution of the problem we are addressing as weel. Therefore, for this purpose we not only addressed data unit tests for raw data but also for the data preprocessed to double check the quality of our data. **The feature store utility** was also proven during the development of the project as it seems extremely useful to store features in a repository and consider them for future reuse, for example for an additional study regarding the problem we are addressing right now, or even other related problems.

15. References:

[1] Run a pipeline — kedro 0.19.6 documentation. (n.d.). Docs.kedro.org. Retrieved June 26, 2024, from https://docs.kedro.org/en/stable/nodes_and_pipelines/run_a_pipeline.html#load-and-save-asynchronously

[2] bank_project_example – Class contents

[3] Trevisan, Vinícius. "Is Your ML Model Stable? Checking Model Stability and Population Drift with PSI and CSI." *Medium*, Towards Data Science, 28 Mar. 2022, towardsdatascience.com/checking-model-stability-and-population-shift-with-psi-and-csi-6d12af008783.

16. Annex

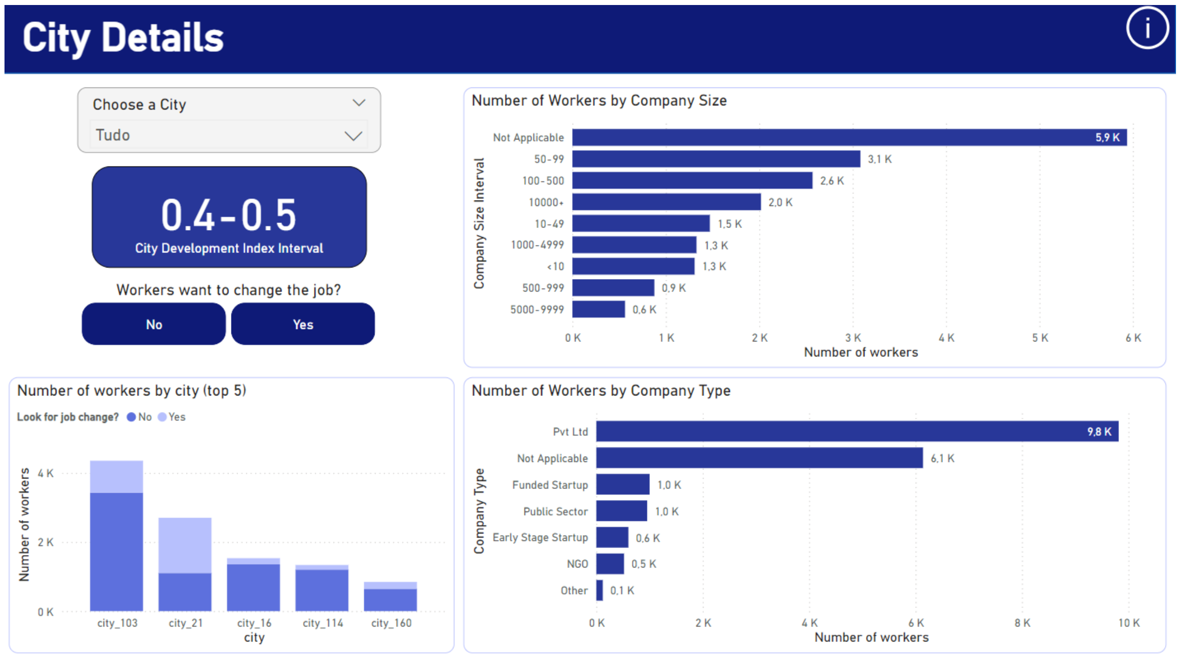


Figure 1 - City specific analysis

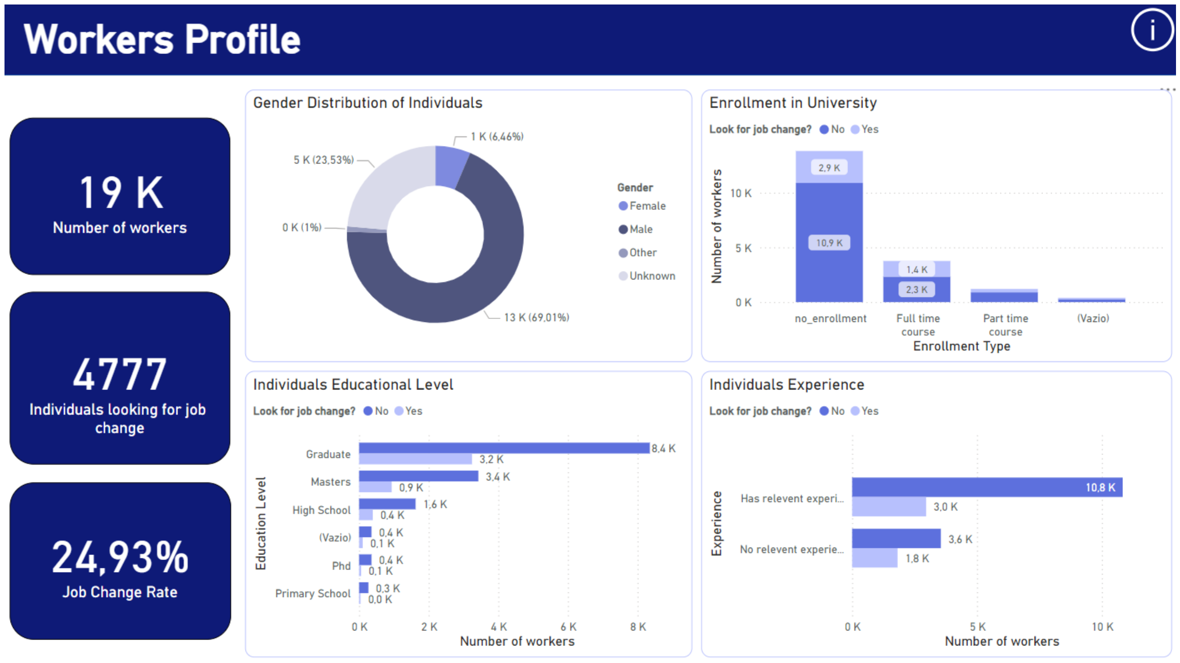


Figure 2 – Workers' profile analysis

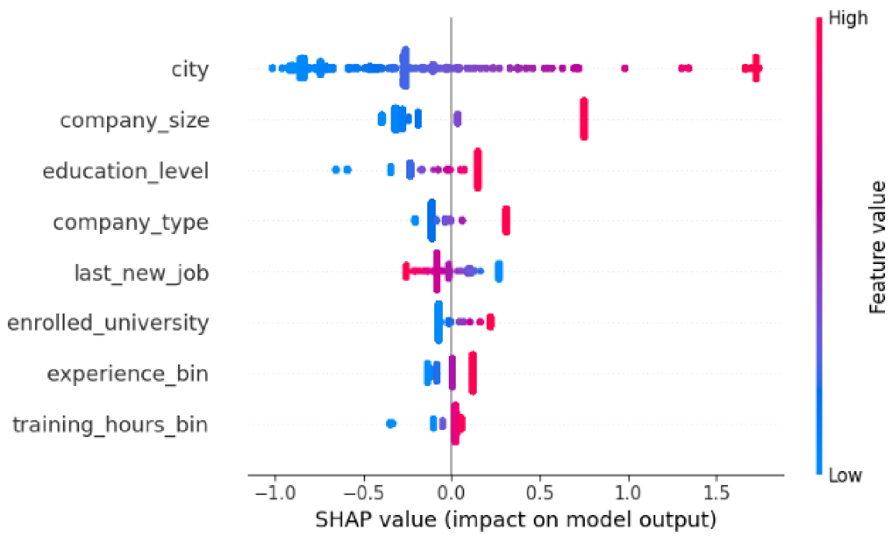


Figure 3 - SHAP for Logistic Regression

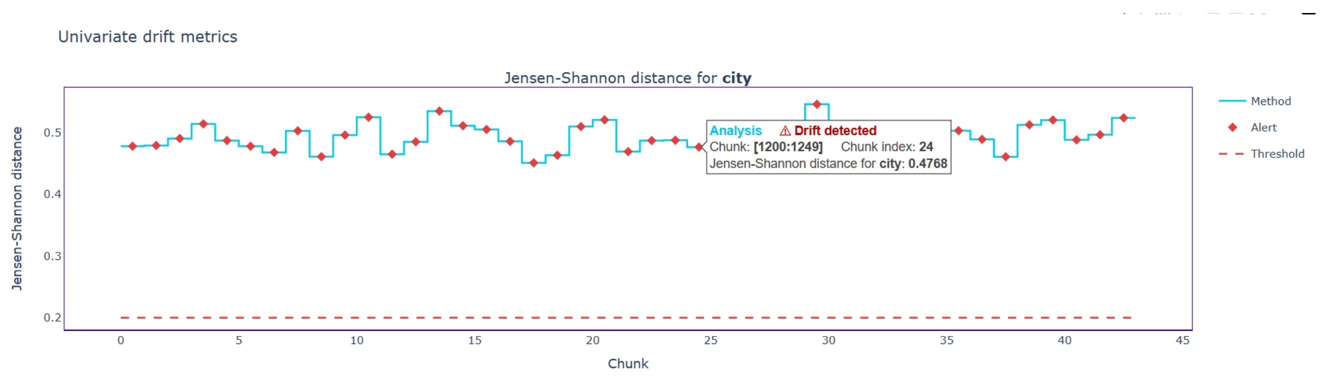


Figure 4 - Univariate drift metrics for 'City'

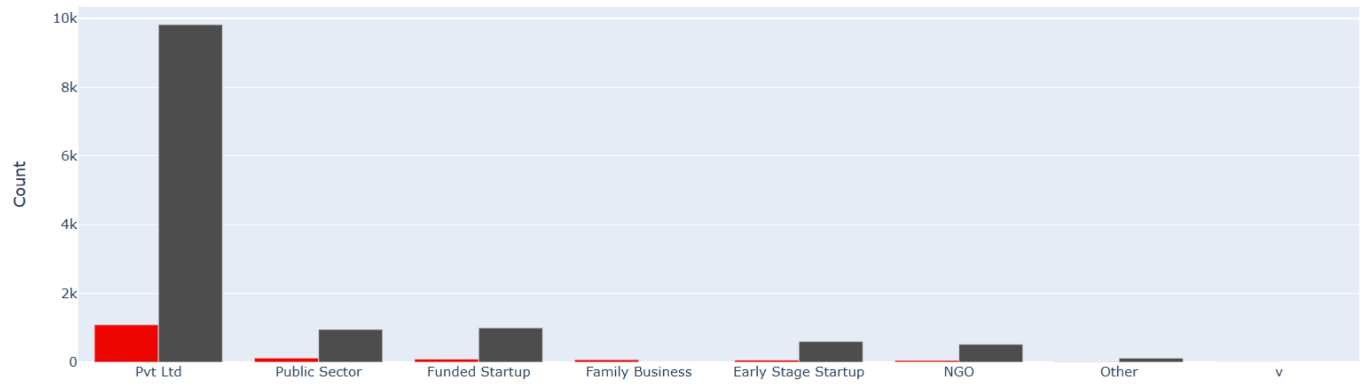
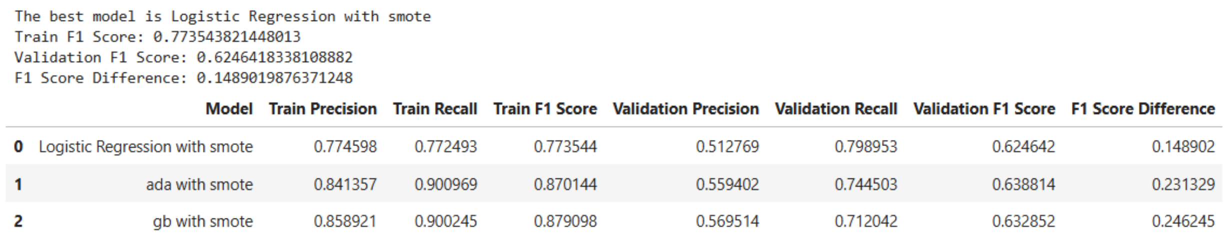


Figure 5 - Company Type drift identification



```
The best model is Logistic Regression with smote
Train F1 Score: 0.773543821448013
Validation F1 Score: 0.6246418338108882
F1 Score Difference: 0.1489019876371248
```

	Model	Train Precision	Train Recall	Train F1 Score	Validation Precision	Validation Recall	Validation F1 Score	F1 Score Difference
0	Logistic Regression with smote	0.774598	0.772493	0.773544	0.512769	0.798953	0.624642	0.148902
1	ada with smote	0.841357	0.900969	0.870144	0.559402	0.744503	0.638814	0.231329
2	gb with smote	0.858921	0.900245	0.879098	0.569514	0.712042	0.632852	0.246245

Figure 6 - Models scores after applying Smote (Registration of overfitting)

aiofiles==23.2.1

aiomysql==0.2.0

alembic==1.13.1

altair==4.2.2

aniso8601==9.0.1

annotated-types==0.6.0

antlr4-python3-runtime==4.9.3

anyio==3.7.1

appdirs==1.4.4

APScheduler==3.10.4

argon2-cffi==23.1.0

argon2-cffi-bindings==21.2.0

arrow==1.3.0

asttokens==2.4.1

async-lru==2.0.4

attrs==23.2.0

avro==1.11.3

Babel==2.14.0

backoff==2.2.1

beautifulsoup4==4.12.3

binaryornot==0.4.4

bleach==6.1.0

blinker==1.7.0

boto3==1.34.88

botocore==1.34.88

build==1.2.1

cachetools==5.3.3

category-encoders==2.6.3

certifi==2024.2.2

cffi==1.16.0

chardet==5.2.0

charset-normalizer==3.3.2

click==8.1.7

click-default-group==1.2.4

cloudpickle==3.0.0

colorama==0.4.6

colorlog==6.8.2

comm==0.2.2

confluent-kafka==2.3.0

contourpy==1.2.1

cookiecutter==2.6.0

coverage==7.5.3

cryptography==42.0.5

cycler==0.11.0

dacite==1.8.1

debugpy==1.8.1

decorator==5.1.1

defusedxml==0.7.1

distro==1.9.0

dnspython==2.6.1

docker==7.0.0

dynaconf==3.2.5

email_validator==2.2.0

entrypoints==0.4

et-xmlfile==1.1.0

evidently==0.4.25

exceptiongroup==1.2.1

executing==2.0.1

Faker==25.9.1

fastapi==0.111.0

fastapi-cli==0.0.4

fastavro==1.8.4

fastjsonschema==2.19.1

featuretools==1.30.0

filelock==3.15.3

FLAML==1.2.4

Flask==3.0.3

fonttools==4.51.0

fqdn==1.5.1

fsspec==2024.3.1

furl== 2.1.3

future==1.0.0

gcsfs==2022.11.0

gitdb==4.0.11

GitPython==3.1.43

graphene==3.3

graphql-core==3.2.3

graphql-relay==3.2.0

great-expectations==0.17.23

greenlet==3.0.3

h11==0.14.0

holidays==0.47

hopsworks==3.7.0

hsfs==3.7.2

hsml==3.7.0

htmlmin==0.1.12

httpcore==1.0.5

httptools==0.6.1

httpx==0.27.0

idna==3.7

ImageHash==4.3.1

importlib_metadata==7.1.0

importlib_resources==6.4.0

iniconfig==2.0.0

ipykernel==6.29.4

ipython==8.23.0

ipywidgets==8.1.2

isoduration==20.11.0

iterative-telemetry==0.0.8

itsdangerous==2.2.0

javaobj-py3==0.4.4

jedi==0.19.1

Jinja2==3.0.3

jmespath==1.0.1

joblib==1.2.0

json5==0.9.25

jsonpatch==1.33

jsonpointer==2.4

jsonschema==4.21.1

jsonschema-specifications==2023.12.1

jupyter_client==8.6.1

jupyter_core==5.7.2

jupyter-events==0.10.0

jupyter-lsp==2.2.5

jupyter_server==2.14.0

jupyter_server_terminals==0.5.3

jupyterlab==4.1.6

jupyterlab_pygments==0.3.0

jupyterlab_server==2.26.0

```
jupyterlab_widgets==3.0.10
jwt==1.3.1
kaleido==0.2.1
kedro==0.19.4
kedro-datasets==3.0.0
kedro-mlflow==0.12.2
kedro-telemetry==0.4.0
kedro-viz==9.1.0
kiwisolver==1.4.5
lazy_loader==0.4
lightgbm==3.3.2
litestar==2.9.0
llvmlite==0.42.0
makefun==1.15.2
Mako==1.3.3
Markdown==3.6
markdown-it-py==3.0.0
MarkupSafe==2.1.5
marshmallow==3.21.1
matplotlib==3.8.4
matplotlib-inline==0.1.7
mdurl==0.1.2
mistune==3.0.2
mkl-fft==1.3.8
mkl-random==1.2.4
mkl-service==2.4.0
mlflow==2.11.3
mock==5.1.0
more-itertools==10.2.0
msgspec==0.18.6
multidict==6.0.5
multimethod==1.11.2
munkres==1.1.4
mypy-extensions==1.0.0
nannyml==0.10.6
nbclient==0.10.0
nbconvert==7.16.3
nbformat==5.10.4
nest-asyncio==1.6.0
networkx==3.3
nltk==3.8.1
notebook==7.1.3
notebook_shim==0.2.4
numba==0.59.1
numpy==1.24.3
omegaconf==2.3.0
openpyxl==3.1.4
opensearch-py==2.4.2
optuna==3.6.1
orderedmultidict==1.0.1
orjson==3.10.5
overrides==7.7.0
packaging==23.2
pandas==1.5.3
pandocfilters==1.5.1
parse==1.20.1
parso==0.8.4
patsy==0.5.6
phik==0.12.4
pillow==10.3.0
platformdirs==4.2.0
plotly==5.21.0
pluggy==1.3.0
polyfactory==2.16.0
pre-commit-hooks==4.6.0
prometheus_client==0.20.0
prompt-toolkit==3.0.43
protobuf==4.25.3
psutil==5.9.8
psycopg2-binary==2.9.9
pure-eval==0.2.2
pyaml==24.4.0
pyarrow==14.0.2
pyasn1==0.6.0
pyasn1_modules==0.4.0
pycparser==2.22
pycryptodomex==3.20.0
pydantic==2.7.1
pydantic_core==2.18.2
pyfiglet==0.8.post1
Pygments==2.17.2
PyHopsHive==0.6.4.1.dev0
pyhumps==1.6.1
pyjks==20.0.0
PyMySQL==1.1.0
pyparsing==2.4.7
pyproject_hooks==1.0.0
pytest==7.4.4
pytest-cov==3.0.0
pytest-mock==1.13.0
python-dateutil==2.8.2
python-dotenv==0.21.1
python-json-logger==2.0.7
python-multipart==0.0.9
python-slugify==8.0.4
pytoolconfig==1.3.1
pytz==2024.1
PyWavelets==1.6.0
pywin32==306
pywinpty==2.0.13
PyYAML==6.0.1
pyzmq==26.0.2
querystring-parser==1.2.4
referencing==0.34.0
regex==2024.5.15
requests==2.32.0
retrying==1.3.4
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rich==12.6.0
rich-click==1.8.3
rope==1.13.0
rpds-py==0.18.0
ruamel.yaml==0.17.17
ruff==0.1.15
s3fs==2022.11.0
s3transfer==0.10.1
scikit-learn==1.3.0
scikit-optimize==0.8.1
scipy==1.11.3
seaborn==0.13.2
secure==0.3.0
```

```
segment-analytics-python==2.3.2
Send2Trash==1.8.3
setuptools==68.2.2
shap==0.45.1
shellingham==1.5.4
six==1.16.0
slicer==0.0.8
smmap==5.0.1
sniffio==1.3.1
soupsieve==2.5
SQLAlchemy==2.0.31
sqlmodel==0.0.15
sqlparse==0.5.0
stack-data==0.6.3
starlette==0.37.2
statsmodels==0.13.2
strawberry-graphql==0.235.0
tenacity==8.2.3
termcolor==2.4.0
terminado==0.18.1
text-unidecode==1.3
threadpoolctl==3.4.0
thrift==0.20.0
tinycss2==1.2.1
toml==0.10.2
tomli==2.0.1
toolz==0.12.1
toposort==1.10
tornado==6.4
tqdm==4.66.2
traitlets==5.14.3
twofish==0.3.0
typeguard==4.2.1
typer==0.12.3
types-python-dateutil==2.8.19.20240311
types-PyYAML==6.0.12.20240311
typing_extensions==4.11.0
typing-inspect==0.9.0
tzdata==2024.1
tzlocal==5.2
ucimlrepo==0.0.6
ujson==5.9.0
uri-template==1.3.0
urllib3==1.26.18
uvicorn==0.29.0
vaderSentiment==3.3.2
visions==0.7.6
waitress==3.0.0
watchdog==3.0.0
watchfiles==0.22.0
watchgod==0.8.2
wcwidth==0.2.13
webcolors==1.13
webencodings==0.5.1
websocket-client==1.7.0
websockets==12.0
Werkzeug==3.0.2
wheel==0.41.2
widgetsnbextension==4.0.10
woodwork==0.30.0
wordcloud==1.9.3
xgboost==2.0.3
ydata-profiling==4.7.0
zipp==3.18.1
```