Introduction

In the case of Netflix, When they were expecting to add another 2.7 million subscribers, they lost around 200,000 of their user base in the United States and Canada, ultimately leading to a loss of 50 billion dollars in market value. Other companies are also facing similar problems in this competitive market.

One can’t simply solve this problem by creating a model that can detect churn; user behaviour is quite dynamic, and based on that, the model constantly needs to adjust to it. Hence, an MLOps-based solution will automatically adjust the model to the changing user behaviour and provide a consistent prediction.

You looked at all the modules and their learning objectives in the previous video. The primary learning objectives are:

- Identifying the business context
- Experimenting with data and models using MLflow
- Automating and orchestrating pipelines with Airflow
- Building continually learning infrastructure
- Creating a monitoring pipeline for model observability

Broadly, there are four core parts in this course:

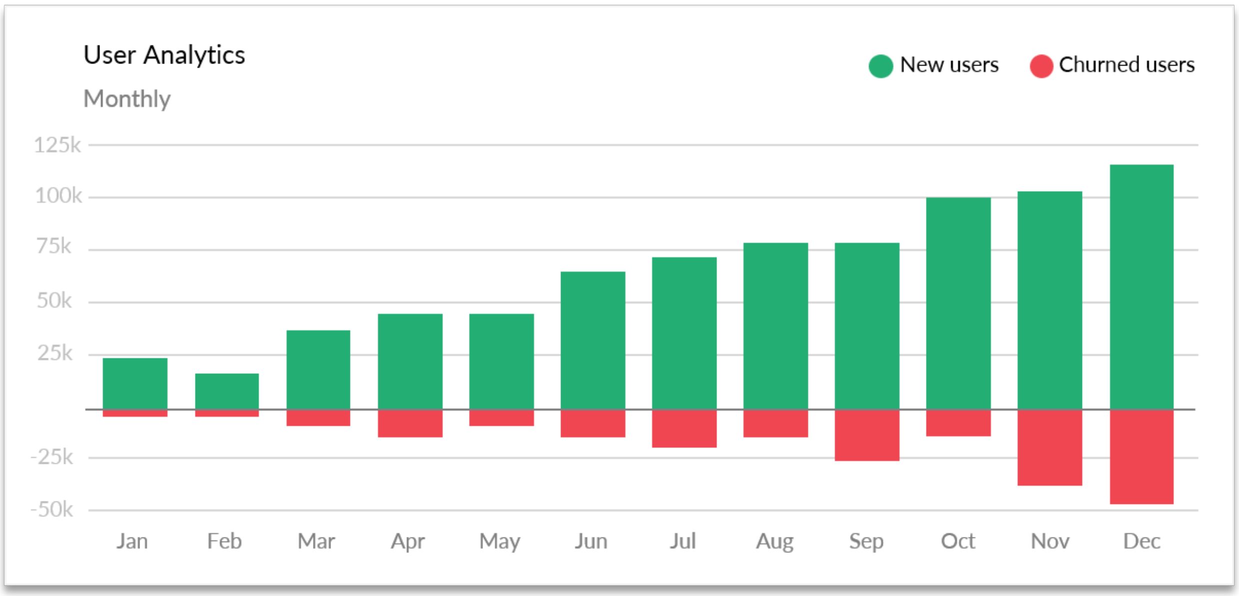
- **Designing Machine Learning Systems**: Here, you will learn the context settings for business. To do that, you will learn about the Product Requirements Document (PRD) and then create the system design, which will give you a solution that meets all the requirements. You will also do conventional ML work such as data analysis, feature engineering and modelling. To bring automation to the traditional model-building approach, you will use tools such as PyCaret, pandas profiling and MLflow. You will also analyse and compare all the models created during experimentation on MLflow that will help you in selecting the model that best serves the requirements of evaluation metrics and model stability.
- **Automating and Orchestrating Pipelines with Airflow**: Here, you will learn Airflow and convert the raw code and functions into directed acyclic graphs (DAGs) that can be executed automatically based on a schedule or an event.
- **Building Continually Learning Infrastructure**: Here, you will learn to build training pipelines and testing scenarios for functions and Airflow graphs. You will also build a dashboard to visualise everything.
- **Data and Model Monitoring**: The main task here is to monitor the model's performance on the live/production data. For this, you will build Airflow graphs that automatically check for deviations in data to help you measure and estimate how the model performs

in a live environment. Significant deviations would trigger the data and model training pipeline you built earlier, thus closing the MLOps loop.

Understanding Business Objectives

You will start with understanding the business context and then defining the project's scope.

You looked at the company **beatit.ai**, a music streaming company that makes money when its customers purchase subscriptions or through advertisements. But the problem that the company is facing is that the churn rate is increasing continuously. This is leading to a loss of revenue and could be detrimental to the company. The following graph shows the number of new, retained and churned customers across different months of the year.



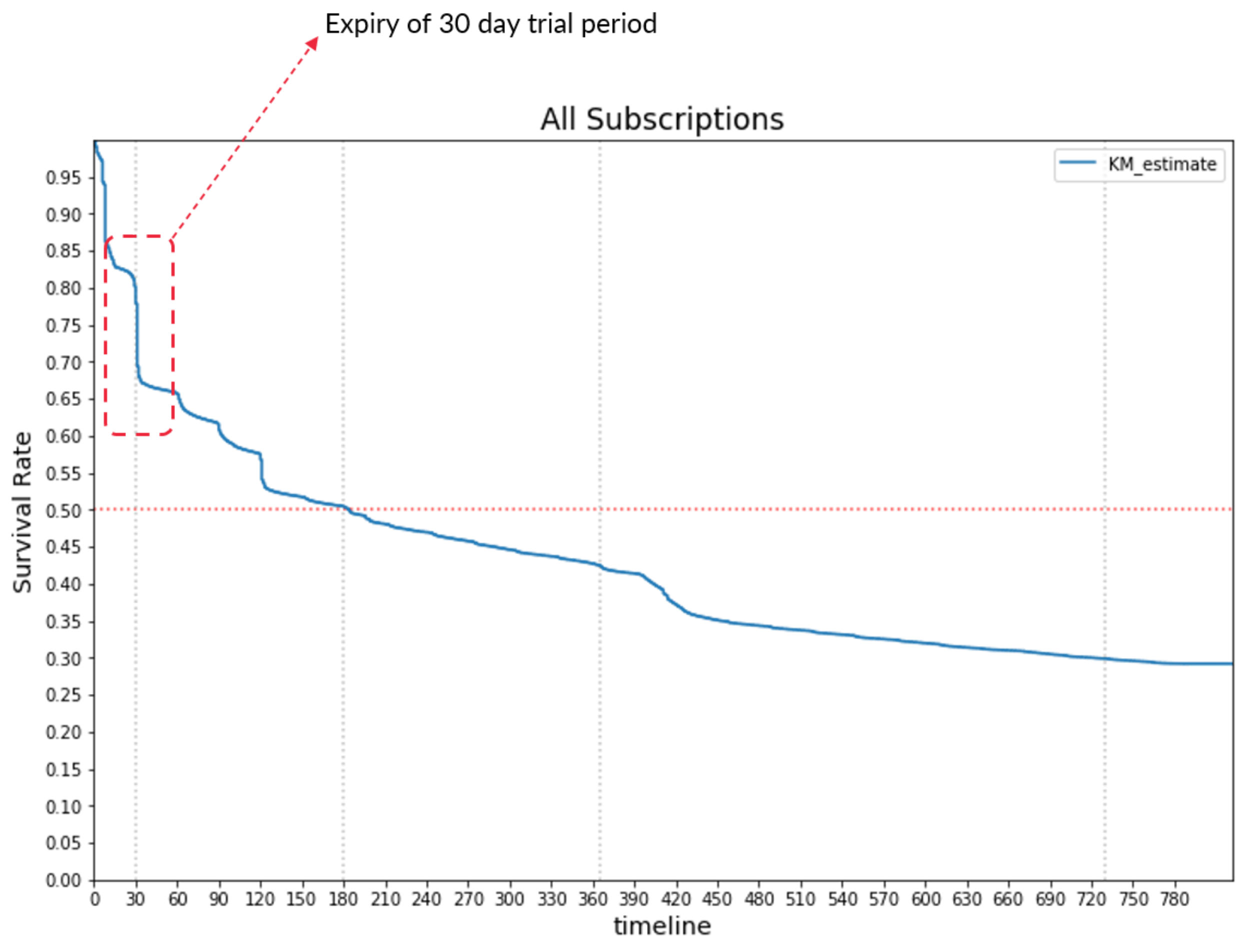
Hence, solving this user churn problem is very important for the company. But before you dive deep into the steps that the company can take to solve this problem, let's first take a step back and understand how to calculate the churn rate.

A churned customer is one who, after the expiry of their subscription, does not renew their subscription within 30 days. Note that this definition will vary from company to company. Beatit.ai defines churn with a 30-day period as most customers purchase a monthly plan, and if one does not renew their plan in the next 30-day cycle, the customer is tagged as a churned customer.

here is the formula for calculating churn:

Monthly churn rate=Total churned users at the end of a monthTotal number of users active in that month

You also looked at a graph showing how the propensity of a customer getting churned increased over time. After around a year, only 30% of the customers remained in the system. In other words, the propensity of a customer to stay on the platform for more than a year is ~30%. Therefore, it is a significant problem for the company. The following graph shows the fraction of customers who have remained in the system over time.



You also saw that the average monthly churn rate was at 7%. But is this small value going to impact the company significantly?

When the churn percentage doubled from 5% to 10%, there were almost a 5.3 times decrease in the total customers. You then looked at the different strategies which the company could have adopted to tackle this problem:

- **Acquiring new customers**: The company would need to spend a lot of money to acquire new customers. This process takes around 10-15 months to recover the amount spent to acquire the customers, yet there is no guarantee that the new customers will remain in the system for the whole duration. Hence, this is a costly and risky solution.
- **Upselling existing customers**: The company could try to increase revenue by making the most of the time for which a customer is on the platform by selling value-added features or new types of packages to them. This is not a very effective way of increasing revenue as there are usually very few customers whom you can upsell, and you are doing nothing to retain the existing customers.
- **Analysing customer churn and making predictions**: The company could focus on retaining existing customers and taking preventive steps before these customers leave the system. Building an MLOps pipeline to predict whether a customer will churn or not and taking corrective action for the customers who are likely to churn will ensure recurring revenue in the long run.

But why do you need to build an MLOps pipeline and not just a simple ML model to predict churn users? This is because user behaviour is pretty dynamic, and thus, the patterns that will be generated will be dynamic. Hence, you'll need to keep changing your model frequently. This is why you need to build an MLOps pipeline that will automate the processes of building models based on changes detected in users' behaviour and reduce the deployment lag.

Mapping Stakeholders' Requirements

The various requirements collected from the business and marketing team are summarised in the following table:

Business Team	**Marketing Team**
The output is an ML model that can predict customer churn. Also, the results of the model	A weekly report should be sent via email containing all the details of the users who are about to churn.

should be interpretable and provide insight about the reasons for exit.	
The model should reduce churn by 7% and adapt to users' new dynamic behaviour.	The prediction report should be sent one month before the actual churn, so that corrective action can be taken in time.
The entire system needs to be completely automated, and the overall time frame is 6 months to execute.	The model should also help in selecting the optimum discounts that can be given to the users who are about to churn.

Divided all the requirements into three categories:

- **Must have**: These are crucial to deliver.
- **Good to have**: These are not essential, but the product will have better features if they are delivered.
- **Out of scope**: These are things that are not possible. These features are not feasible and can be picked only when the 'must have' features have been delivered.

You put all the different stakeholders' requirements into these three categories. The following table shows how the different requirements have been split into these three categories:

Must Have	**Good to Have**	**Out of Scope**
The output is a machine learning model that can predict customer churn.	The model results should be interpretable for analysing the reasons for customers' exit.	The model should be capable of reducing churn by 7%.
The model should be capable of reducing the churn by 2-3%.	A weekly report should be sent via email containing the details of users who are about to churn.	An entirely automated (no manual intervention) system should be developed in 6 months.

The model should be able to adapt to the new dynamic behaviour of the user.

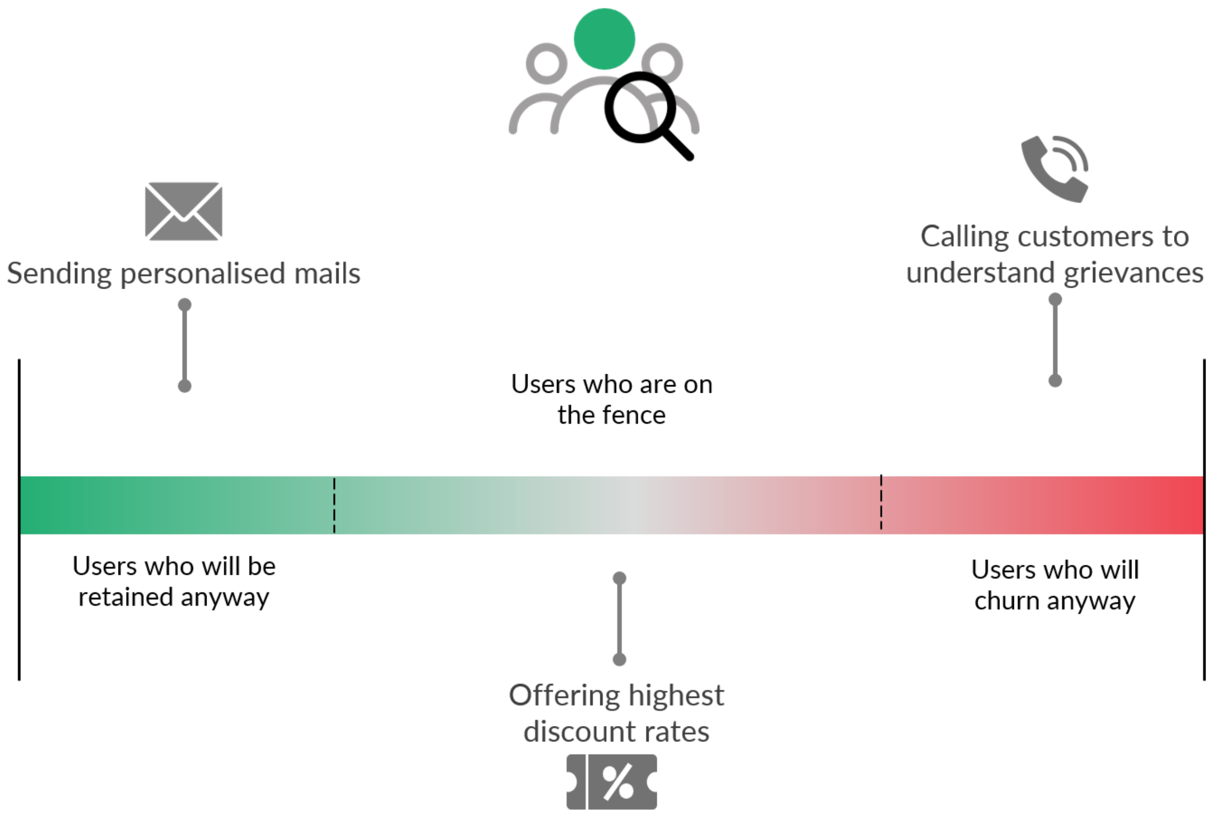
The model should also help select the optimum discounts that can be sent to a user who is about to churn.

The prediction report should be sent one month prior to the actual churn to enable corrective action to be taken.

The customers were divided into three broad categories. Let's understand the whole process in detail:

1. Calculate the probability of a user getting churned in the next 30 days using an ML model.
2. All the probable churned users should be assigned an action category (bucket) **A, B or C** based on their probability.
    - 'A' is the bucket with a very high probability of churning and 'C' has the lowest probability of churning.
    - The marketing team can then use this bucket to assign the discount to the respective customer.
        1. A (users who will churn anyway): To retain them, the company might need to call them and understand their pain points and try to address them.
        2. B (users who are on the fence): To retain them, the company can provide discounts on their subscriptions.
        3. C (users who will be retained anyway): The company does not need to focus much on them as they will not churn. However, the marketing team can send personalised emails to show them appreciation.

The following scale broadly categories the customers into three categories:


Sending personalised mails
Calling customers to understand grievances
Users who are on the fence
Users who will be retained anyway
Users who will churn anyway
Offering highest discount rates

‘Next Action’ → Propensity to churn

In addition to these, you also defined the success criteria for this project: If the churn rate is reduced by 2-3% for 3 successive months, this project is successful.

You looked at some of the elements that the data science team would need to consider when working on this project:

- Understanding the policies around sharing and sourcing data: They would need to check that it is technically feasible to access the data and that the team has all the required access from stakeholders.
- Checking if, historically, labels are present in data: If not, then the building of that mapping should be adjusted in the timeline.
- Getting a data quality evaluation from the data stakeholders: The team needs to be aware of the correctness of the data since everything will be built around those inputs.
- Evaluating thoroughly that the problem being solved is an ML problem and can be solved by building a model.

Let's quickly summarise the entire process of creating the PRD:

- Identified the problem that the company is facing and how it is affecting their business and growth.
- Proposed multiple solutions to the problem and concluded that building an ML model will help predict whether a customer will churn or not. You also looked at the steps required to retain customers.
- Gathered requirements from different stakeholders, checked the feasibility of these requirements and defined the scope of the project.
- Defined the success metrics for the project.

You can refer to this link for the PRD created for the given business problem.

So, in this segment, you looked at the summary of the steps the company followed to create the PRD.

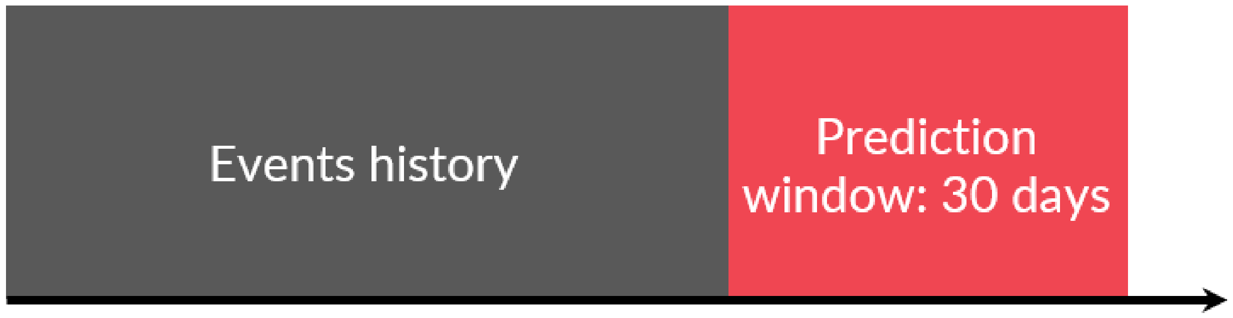
Note: There could be many more such checks. Additionally, you can refer to this link for a comprehensive machine learning checklist while performing the feasibility study.

Understanding And Evaluating Data

Until now, you have understood the business problem and mapped all the stakeholders' requirements. Now, let's jump right into the most exciting part of our use case: **system design**.

The project aims to build a binary classifier that can accurately predict the users who are at risk of cancelling the service one month after their current membership expiration date.

The train data that you'll work on consists of users whose subscriptions have expired within the month of February 2017. Hence, we are looking to predict the churn for these users who will churn or renew roughly in the month of March 2017 (one month after the subscription expiration).

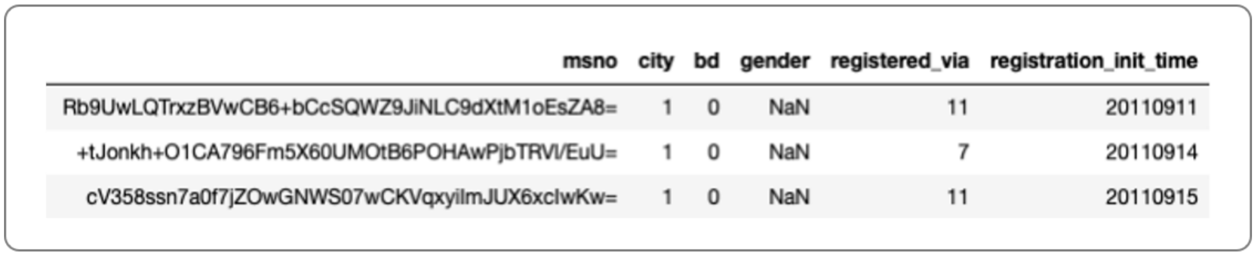


The objective of the case study is to predict whether a user will 'churn', i.e., leave their subscription-based service, based on their behaviour on the platform. Note that the churned user may either use the free version of beatiit.ai or drop off the platform.

The first task is to access and evaluate the data. The data comes from several sources and contains information about each user's subscription and streaming activities.

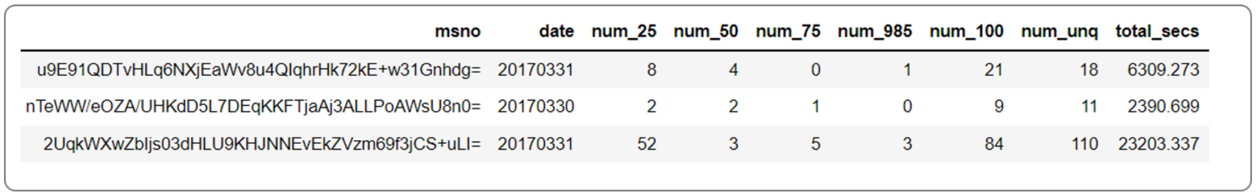
The data used in this case study is taken from the kaggle competition. It is recommended that you go through the dataset and perform some exploratory analysis to develop insights that'll help you in model building. Since the data is huge, we have selected part of the data that is useful for our building our MLOps system. You'll learn about this in the next session. For now, let's understand the data that we have.

- User profile data (**members_profile.csv**): This data includes the user's personal information like their age, city and registration time.



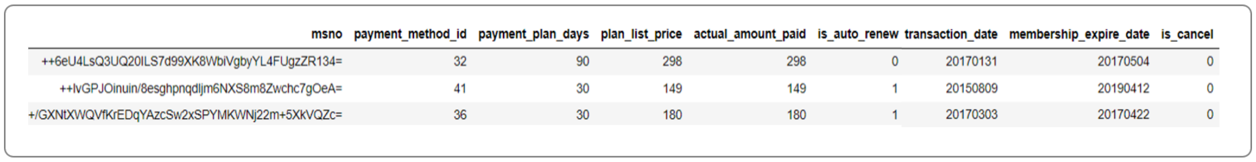
msno	city	bd	gender	registered_via	registration_init_time
Rb9UwLQTrxzBVwCB6+bCcSQWZ9JiNLC9dXtM1oEsZA8=	1	0	NaN	11	20110911
+tJonkh+O1CA796Fm5X60UMOtB6POHAwPjbTRVl/EuU=	1	0	NaN	7	20110914
cV358ssn7a0f7jZOwGNWS07wCKVqxyilmJUX6xclwKw=	1	0	NaN	11	20110915

- User logs data (**userlogs.csv**): This data consists of each user's listening behaviour in terms of songs played each day.



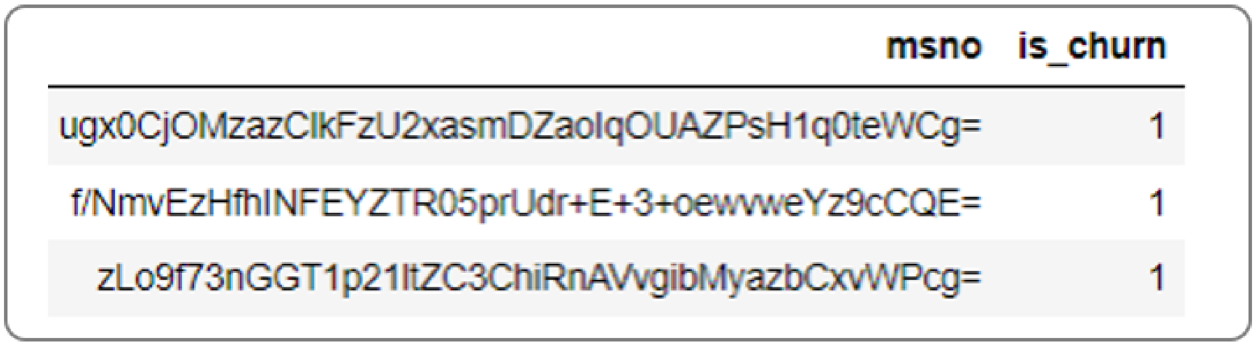
msno	date	num_25	num_50	num_75	num_985	num_100	num_unq	total_secs
u9E91QDTvHLq6NXjEaWv8u4QlqhrHk72kE+w31Gnhdg=	20170331	8	4	0	1	21	18	6309.273
nTeWW/eOZA/UHKdD5L7DEqKKFTjaAj3ALLPoAWsU8n0=	20170330	2	2	1	0	9	11	2390.699
2UqkWXwZbIjs03dHLU9KHJNNEvEkZVzm69f3jCS+uLI=	20170331	52	3	5	3	84	110	23203.337

- User transaction data (**transations_logs.csv**): This data consists of details like payment method or whether the subscription was cancelled.



msno	payment_method_id	payment_plan_days	plan_list_price	actual_amount_paid	is_auto_renew	transaction_date	membership_expire_date	is_cancel
++6eU4LsQ3UQ20ILS7d99XK8WbiVgbyYL4FUgzZR134=	32	90	298	298	0	20170131	20170504	0
++lvGPJOinuin/8esghpnqdljm6NXS8m8Zwchc7gOeA=	41	30	149	149	1	20150809	20190412	0
+/GXNtXWQVfKrEDqYAzcSw2xSPYMKWNj22m+5XkVQZc=	36	30	180	180	1	20170303	20170422	0

- Historical data (**churn_logs.csv**): This data consists of user IDs and whether these users have churned or not.



msno	is_churn
ugx0CjOMzazClkFzU2xasmDZaolqOUAZPsH1q0teWCg=	1
f/NmvEzHfhINFEYZTR05prUdr+E+3+oewweYz9cCQE=	1
zLo9f73nGGT1p21ltZC3ChiRnAVvgibMyazbCxvWPcg=	1

To create the final data, you will merge the above four data sources using the user's membership ID (msno) as the primary key. The resulting data will be stored in a feature repository containing all the necessary user behaviour features for model training.

The features created/engineered from the data are the 'fuel' for a model. While creating an ML system, it is essential to have a feature store/repository that can act as an interface between the models and the data.

Feature Store

The feature store is a central repository where you can either create or update groups of features developed from multiple data sources. Using the feature store, you can quickly train models or just retrieve the features whenever the model needs to make predictions.

Therefore having a feature store helps you in

Providing consistent features for model training and prediction: the features, once saved in the feature store, can be used in model prediction/inference. This helps in avoiding the repetition of the same steps while performing model prediction.

Sharing and Reusing of features across teams: Feature store enables re-use and discoverability on ML use cases across organizations. Data scientists can save/register the final features in the feature store, which can be easily accessed by different teams for any other purpose.

Isolation between data and model: ML workflow consists of complicated systems and components. Feature stores act as a bridge between different ML pipeline and data engineering pipeline and thus yields efficiency and avoid any complexities.

Development And Production Environment

You defined the ML objectives and looked at the different data sources. You are now ready to draft the architecture required for the entire ML system.

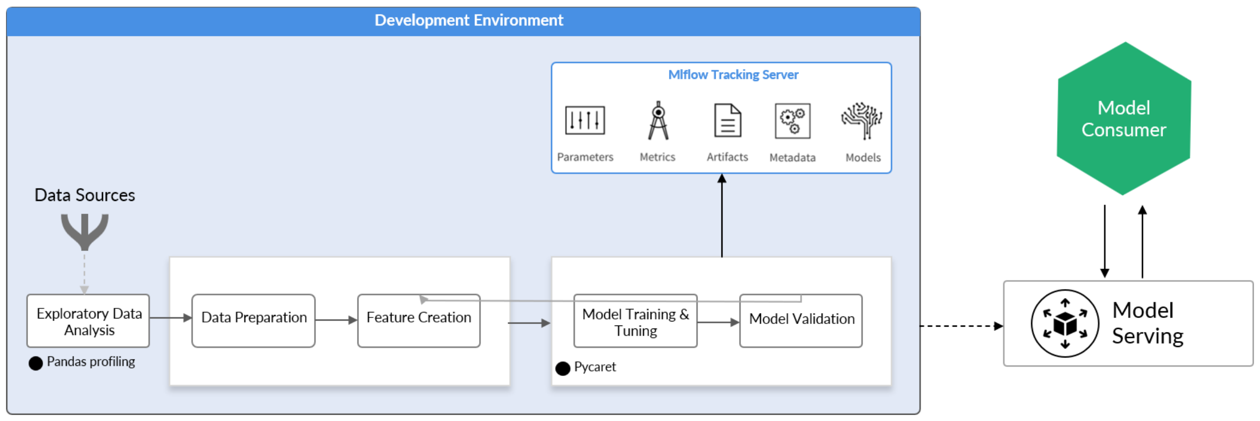
The system architecture comprises multiple layers for the different stages of the ML life cycle. The ML system is designed considering all the components, tools, methods and procedures to help deliver the overall requirements drafted earlier in the PRD.

The system architecture is divided into two environments:

1. **Development environment**: This environment is structured to help you understand which model will perform the best for the given problem statement and data. This is an environment where you can perform rapid experimentation around data and models.
2. **Production environment**: This environment is where you operationalise the best model identified from the development environment/stage.

Development Environment

Most of you will consider this stage of the MLOps life cycle the most fun as it gives you opportunities to experiment and play with different models.



As you learnt earlier, model development in the development environment is an interactive process and broadly consists of the following steps:

1. **Exploratory data analysis**: In this step, you will understand the patterns and infer insights from the given data using data visualisations. Some of the tools used in this stage will be Pandas Profiling, matplotlib, Seaborn etc.
2. **Data preparation**: Once you have understood the data and its distribution, you can move to the other operations to perform cleaning, pre-processing and feature engineering on the given data. This step is crucial for performing experimentation using the features extracted from the data.
3. **Model training and tuning**: You can start building the model using the features extracted from the previous step. As you learned in the previous module, ***rapid experimentation*** is essential to selecting the baseline model in this stage. Here, you can experiment with multiple models/algorithms; however, it is essential to strategise which models you want to select. For your use case, you will majorly work with classical ML models owing to their simplicity and explainability.

For business, you can use 'a simple model is always better than complex models' as a baseline to compare with later complex models. Also, having a simple model as a baseline helps rapidly deploy to the end user. To accelerate the development, you will be using tools like PyCaret and MLflow for rapid prototyping and experiment tracking.

- PyCaret helps with code automation and lets you build different modelling experiments quickly without worrying about building code.
- MLflow will help you track every experiment and log parameters, models, artifacts, metadata, accuracy numbers, timestamps, execution time, etc.

1. **Model validation**: In this step, based on the performance of all the candidates (models used for experimentation), you will select the best one that meets the evaluation criteria

(recall or lower false Negatives). If it doesn't meet the standards, you must go back and experiment until you reach acceptable numbers.

2. **Model serving**: Once you have finalised the best-performing model, you can directly serve the model as an API for end users/stakeholders to consume.

Production Environment

Once the model has been served to the end users/stakeholders, it will continuously keep delivering predictions on live data. Since our ML system's success criterion is an average reduction of 3-4%, it is essential that it keeps accurately predicting on the live data for a prolonged duration, let's say for three months.

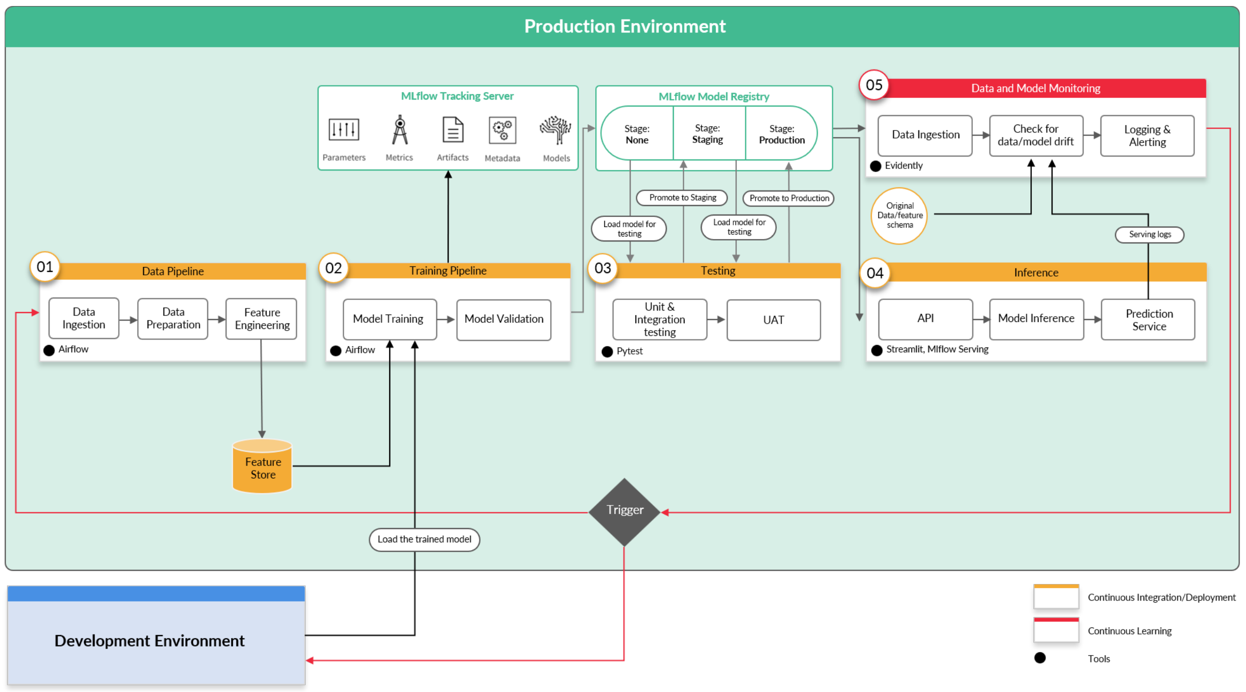
But here lies the problem! The data coming through the system changes month by month. Every month, you'll discover new usage patterns among the users as well as new transactions that they may have performed. Do you think that in such a case, where the usage pattern of the users may change, it is wise to use the same model for predictions that you created in the development environment? Well, no! You'll need to keep updating your model. On the contrary, you will not use rapid experimentation every time the data is changed since that is time-consuming. So, how frequently should you change your model via rapid experimentation?

Well, the answer to this is that you should not change your model every time a new batch of data comes in. Let's consider the case of linear regression. As soon as new data comes in, you should retrain your old model (linear regression) and update its parameters by training it on the most recent data so as to capture the evolving and emerging patterns. This is a characteristic of the **continuous learning/training** that you'll build as part of your ML system.

The continuous training pipeline can re-train your model whenever there is a change in the live data. Therefore, this helps create a robust system that can account for dynamic user behaviours. But what if your retrained model is not good enough? Well, then you go back to the development environment and perform rapid experimentation to come up with the best model. You'll get to know how this is done in the next segment.

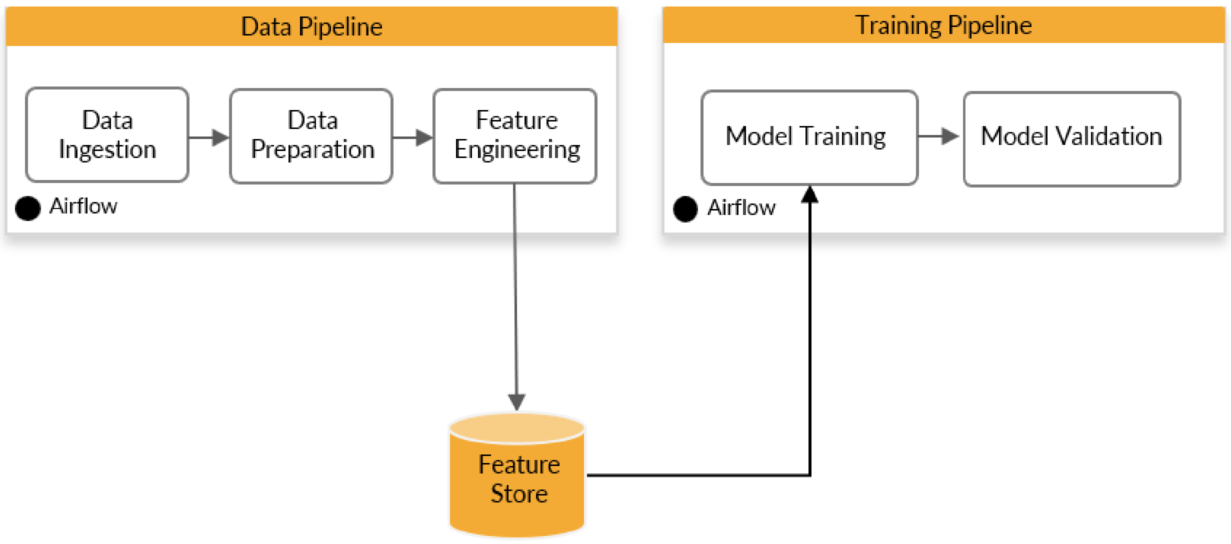
Thus, a development environment by itself isn't enough to create an ML system. You need to move away from your conventional ML workflow and bring in the production environment, which can help manage the dynamic patterns of live data.

Broadly, in the production environment, you will work with multiple pipelines. Here, the concept of the pipeline is introduced to split up your machine learning workflows into independent, reusable, modular parts that can then be integrated together to serve any task like feature creation, model training, prediction serving, etc. An ML pipeline is made up of multiple **components** which need to be reusable, composable, and potentially shareable across the entire ML pipeline.

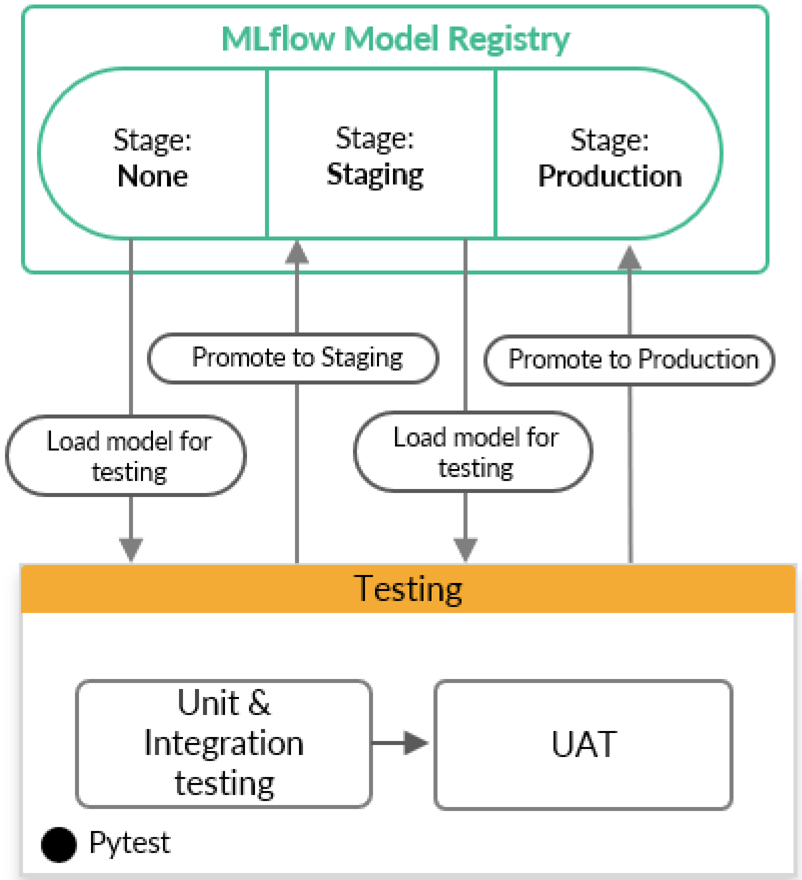


The components covered in the production environment are as follows:

- **Data and training pipeline**: This component focuses on automating model training by converting the code developed in notebooks to Python scripts. With automation coupled with using a feature store, the data and training pipelines can run whenever there is any change in the live data. This helps in the continuous delivery of existing deployed models after it is re-trained on the newly transformed data stored in the feature store. The tool used for automation in this stage is Airflow.

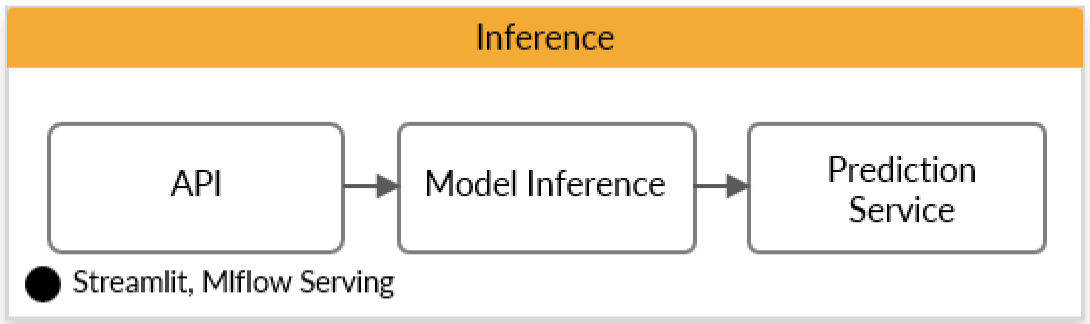


Testing: In this step, you will test the different methods used in data preparation, feature extraction and model validation to effectively track whether all the components are working in the desired manner. The tests applied here are unit tests, integration tests, and user acceptance testing (UAT). If the model passes all these tests, it can be moved to production, that is, it can be used for making inferences/predictions. Therefore, testing helps in the continuous integration of models trained on new data.

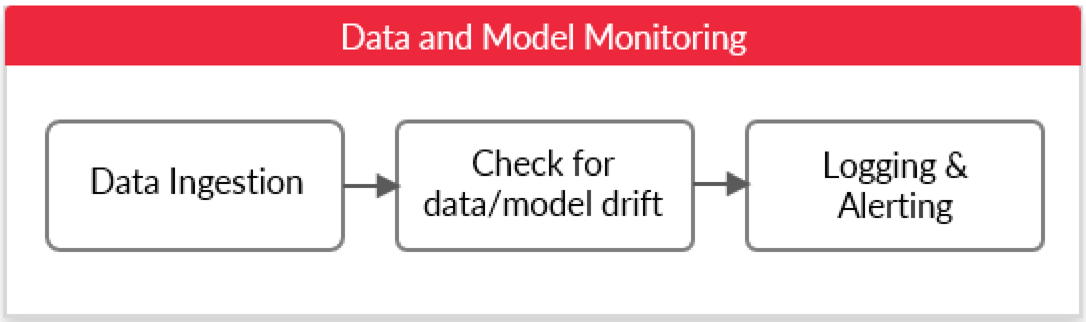


Note: This component is built manually for our use case as you usually perform testing and, based on the results, manually move the model to production. The moment the model moves to production, the end users start using it. Hence, it is a benefit to have a manual check before such an event.

- **Inference**: In this stage, once the model/code passes all the tests, you will go ahead and deploy the model for serving predictions. The tool used in this stage is Streamlit, with MLflow Serving for quickly deploying the models.



Data and model monitoring: Keeping a continuous check on the deployed model is essential for tracking the model performance and ensuring that the model doesn't go stale. It signals what action needs to be performed based on any changes in the live data. The 'trigger' connected to this component decides what action to take: model experimentation or model retraining. The tool used in this stage for monitoring any data drifts is called Evidently.



Automated Data And Training Pipelines

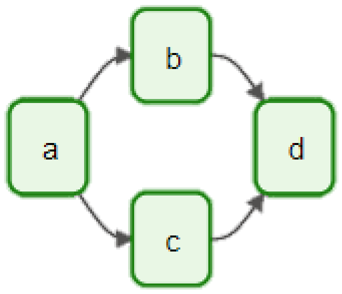
You have seen that to put a model into production, you convert the Jupyter notebooks used in the development environment into Python scripts and automate them using Airflow using directed acyclic graphs (DAGs).

A DAG or a Direct Acyclic Graph in Airflow describes a sequence of execution steps or tasks in the complex non-recurring computation. It is acyclic because the graph or the sequence of steps cannot form a loop. Simply put, they cannot be cyclic. A typical machine learning workflow consists of a sequence of transformative steps which are applied to the data. While working in an enterprise, it is important to orchestrate and schedule all these steps to reduce the manual intervention in your workflow. Thus comes DAG to the rescue! Every component you see in the workflow for our use case, the data pipeline, the training pipeline or the inference pipeline, are in themselves smaller DAGs.

Building Automated Data and Training Pipelines

- You will primarily build Airflow DAGs (directed acyclic graphs), which are acyclic graphs for executing tasks.
- All you have to do at this stage is to convert your raw code for data preparation, feature engineering, model training and validation from your notebooks into Python functions.

These can then be converted into Airflow DAGs, such that they can be executed automatically based on some time or action triggers.



- Note: At this stage, you will not be automating the EDA part because of two main reasons:
    - EDA requires manual intervention to understand and interpret the results.
    - EDA is not required in the data pipeline because you perform the data preparation steps in the same way as you did in the development phase/environment. Hence, you do not need EDA to figure out the data transformations and preparation as you already have them from the development environment. Remember, this is something you did in the 'MLOps - Prerequisites' module when you converted the Heart Disease notebook to Python scripts.
- In the production environment, you will constantly keep track of your model in the model training pipeline using MLflow as you would have done in the development environment.
- In the training pipeline, you won't perform the rapid experimentation again as you did in the development environment. As Lavi mentioned, you simply use the best model that you identified earlier in the development stage. You load this model with all its parameters and hyperparameters. Now, there are two ways of training this loaded model in the model pipeline whenever new data comes in:
    - You can fit the model on the new data by keeping the hyperparameters as they are, that is, retrain the model without changing the hyperparameters.
    - You can both fit and tune the hyperparameters on the new data, that is, retrain the model by exploring the hyperparameter space.
- Whatever method you use to retrain the model on new data, all the experiments get logged on the MLflow server as the training pipeline is connected to the MLfLow tracking server that keeps logs of every model that runs in the training pipeline.
- Therefore, this part is also called 'continuous training', as the same model is continuously trained with the new data.

In this stage, you will constantly validate the working of the data and training pipeline and its integration as a whole using tools like Pytest and MLflow Registry.

Let's now try to understand the whole process of testing.

Using the MLflow registry, which is the model-management feature of MLflow, you can manage different versions of your model and push them into various stages as and when needed. Let’s understand this better.

There are three stages in the model registry: 'None', 'Staging', and 'Production', and each of these helps you with the maintenance and tracking of different versions of models. Labelling the different versions of a model as ‘None’, ‘Staging’, or ‘Production’ helps you decide which model or version of a model should be put into production, that is, for making inferences in the inference pipeline.

- For example, once the final model is received from the model training pipeline, the model and its details are logged in the model registry. This model can then be tested on the validation data, completely unseen by the model. It can then go through unit and integration tests using Pytest.
- Once all the tests are successfully validated (indicating the successful integration of data and training pipeline), the model will be pushed to 'staging', where it will be used to predict churn on real-world data. Here, you perform the UAT (user acceptance testing), that is, you provide the model with sample data that mimics live data. Once the model results on this data are as expected and satisfactory, you can finally push the model artifact to ‘Production’. This means that the model can be used for making predictions/ inferences in the inference pipeline.

To summarise, once the model is loaded from the development environment, is retrained on the new data, and has passed all the tests in the testing stage, it gets pushed from the production stage to the inference stage for serving predictions.

In the Inference stage, you expose the final model from the production stage as an API using a simple FastAPI interface with MLFlow Serve. You’ll learn more about this in the upcoming modules.

Note: Considering the limitation of our use case, you are deploying this in a limited capacity. However, in the real world, a whole infrastructure needs to be built for this since you might have to scale up and down and perhaps integrate it as part of a larger software or product.

Once the model is served and can continuously predict on the live data, you can move to the next component in the production environment: data and model monitoring.

You have seen that continuous data and model monitoring is the most crucial component of an ML system since you never know when your model is failing silently.

A typical continuous monitoring process consists of the following steps:

- In the inference stage, a sample of the request-response (user-model) interaction during prediction serving is captured as serving logs
- A comparison is made between the original data/ feature schema or distribution and the generated schema or distribution from the serving logs.
- If there is any deviation between the two schemas/ distribution, an alert is raised to the respective stakeholders to detect data drift.

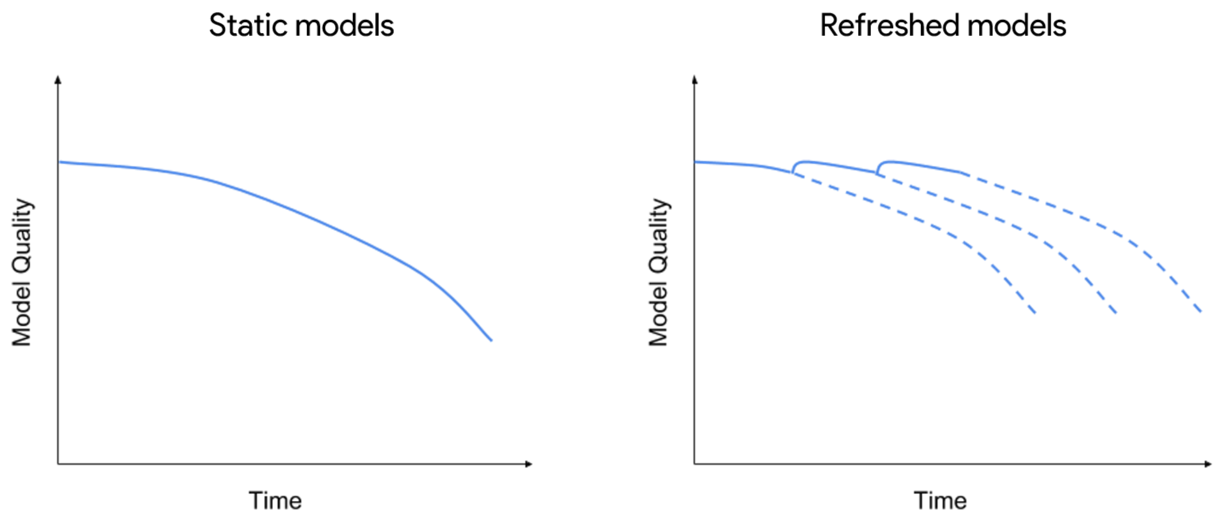
The next action can be taken based on the deviation detected. This is configured in the component called ‘trigger’. To understand this, let’s imagine three different scenarios of measuring data drift by comparing the distribution of the Gender column in the new and the old data.

- 1-2% deviation: No action is needed as the deviation is not significant.
- 5-8% deviation: The data and model training pipelines are triggered for retraining the model with the new data.
- > 10% deviation: This can occur when the user behaviour changes drastically. In such scenarios, you must go back to the development environment to perform data and model experimentation.

Note: The breakdown of deviation shown above depends on the use case and can be modified as per the business requirement.

Data And Concept Drifts

No model can live forever. Over time, it will lose its predictive power. Very rarely, a deployed model can serve continuously without any degradation in its performance. It has to be replaced with a new model trained on new data.



The effectiveness of 'continuous monitoring' is measured by the ability to detect any model decay, which is usually defined in terms of data and concept drift.

Data drift is a type of drift that occurs when the properties of the independent variable(s) change(s) through time. Because of this, the previously trained model is no longer relevant on the new data and hence, will perform poorly.

For example, for a lead-scoring model in a company, users sign in to the system through various channels. While training the model, the most common channel was Facebook, and the least common was Google. However, once this model was put into production, the marketing team launched a new ad campaign due to which, most of the users came through Google.

The current model had not been trained on the new pattern in the data (more leads from Google) since there were very few samples to learn from. Therefore, the model will not be able to accurately predict which leads have the highest conversion rate.Concept drift occurs when the relationship between the input and target variables changes over time.

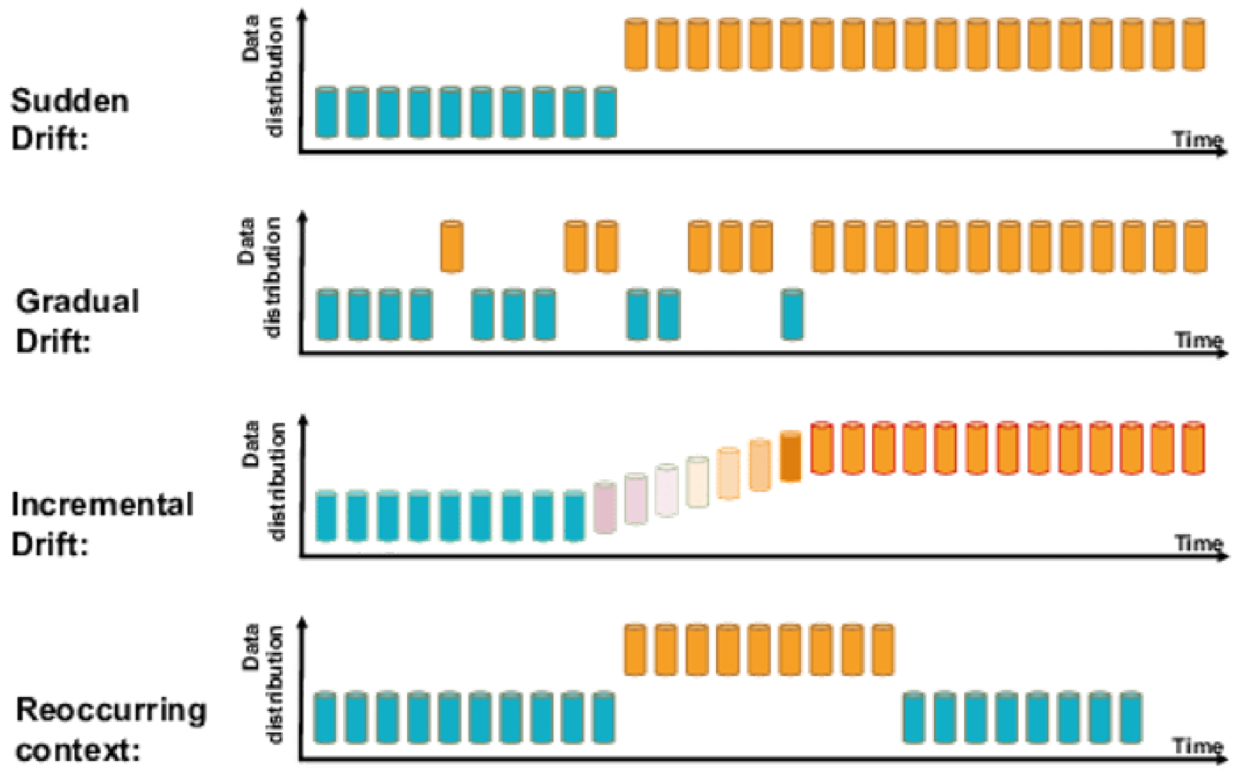
The model trains by understanding the relation/ function mapping between the independent variables, or predictors and the target variable. In a static or perfect environment, where neither of these predictors nor the target evolves, the model should perform as it did on day 1 because there's no change in the relationship at all. However, in a dynamic or live environment, the statistical properties of the target variable can change, and so does its meaning. In such a scenario, the relation/ function mapping that the model learned will not work, leading to model decay.

For example, in our previous example of lead scoring, while training the model, it understood some patterns from a pool of features and successfully defined who is a high-scoring lead. However, if the relationship between these features and the target changes, it will lead to a new definition of a high-scoring lead, which the model will not understand.

Concept drift generally doesn't occur suddenly, the drift slowly grows over time. However, due to some unforeseen situations, they can occur suddenly as well. For example, a nationwide lockdown can heavily impact user spending/buying behaviour.

The same can occur in a seasonal pattern as observed during a new year's sale. To handle this, companies can employ different models to handle seasonal changes.

So broadly, concept drift can be gradual, incremental, sudden and seasonal.



System Design Summary

Let's quickly summarise the entire process of creating the system design:

- Create a development environment for performing rapid data and model experimentation. This will help you in identifying which model performs the best for the given data.
- Once you have finalised the model, you need to automate the entire process by converting the codes in the notebook to a production-level script. The entire automation process can be done by building DAGs through Airflow.
- To bring in the characteristics of ‘Continuous Integration & Deployment (CI/CD)', you need to add different protocols for testing the different components of data and training pipelines and it working as a whole.
- Finally, you need to build a continuous monitoring system for active health checks and tracking any model staleness.

Experimenting Data and Model With MLflow

In the coding parts of the segments, you performed the following tasks:

- Defined the folder/directory structure for the entire project
- Understood the problem statement and loaded the datasets
- Explored the different available datasets in great depth
- Performed EDA and applied various data transformation functions for memory conversion, datetime fixing etc.
- Learnt pandas profiling and used it for analysing the data
- Applied basic feature engineering to the datasets
- Combined all the datasets into a single dataset and stored the result
- You loaded the resultant data set that was saved at the /home/data/interim path.
- You divided this data set into two sets: the training set and the validation set (also called the unseen data).
- You understood what MLflow is and how it is useful in tracking different machine learning experiments.
- You learnt that MLflow uses two components for storage: backend store(database) and artifact store(mlruns). While the backend store persists MLflow entities (runs, parameters, metrics, tags, notes, metadata, etc), the artifact store persists artifacts (files, models, images, in-memory objects, or model summary, etc).
- You deep-divided into PyCaret and understood how it is useful in running and comparing different models with only a few lines of code.
- You learnt about and used PyCaret’s compare_model function to build different machine learning models and check different model metrics like accuracy, F1 score and recall.
- You learnt about and used PyCaret’s create_model function to build the lightbgm model because it came out as the best candidate model.
- You also learnt about and used PyCaret’s interpret_model function to understand the contribution of different features in the prediction of a model.